Oracle Fusion Middleware 11*g*: Java Programming

Volume I • Student Guide

D53983GC11

Edition 1.1

May 2009

D60390

Author

Kate Heap

Technical Contributors and Reviewers

Ken Cooper

Clay Fuller

Taj Islam

Peter Laseau

Yvonne Price

Editors

Daniel Milne

Joyce Raftery

Graphic Designer

Satish Bettegowda

Publishers

Pavithran Adka

Nita Brozowski

Disclaimer

This document contains proprietary information and is protected by copyright and other intellectual property laws. You may copy and print this document solely for your own use in an Oracle training course. The document may not be modified or altered in any way. Except where your use constitutes "fair use" under copyright law, you may not use, share, download, upload, copy, print, display, perform, reproduce, publish, license, post, transmit, or distribute this document in whole or in part without the express authorization of Oracle.

The information contained in this document is subject to change without notice. If you find any problems in the document, please report them in writing to: Oracle University, 500 Oracle Parkway, Redwood Shores, California 94065 USA. This document is not warranted to be error-free.

Restricted Rights Notice

If this documentation is delivered to the United States Government or anyone using the documentation on behalf of the United States Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS
The U.S. Government's rights to use, modify, reproduce, release, perform, display, or disclose these training materials are restricted by the terms of the applicable Oracle license agreement and/or the applicable U.S. Government contract.

Trademark Notice

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

Contents

I Introduction

Objectives I-2

Course Overview I-3

1 Introducing the Java and Oracle Platforms

Objectives 1-2

What Is Java? 1-3

Key Benefits of Java 1-5

Object-Oriented Approach 1-7

Design Patterns 1-8

The MVC Design Pattern 1-9

Platform Independence 1-10

Using Java with Enterprise Internet Computing 1-11

Using the Java Virtual Machine 1-13

How Does the JVM Work? 1-15

Benefits of JIT Compilers 1-17

Implementing Security in the Java Environment 1-19

Deployment of Java Applications 1-21

Using Java with Oracle 11g 1-22

Java Software Development Kit 1-23

Using the Appropriate Development Kit 1-24

Java SE 6 1-25

Integrated Development Environment 1-26

Summary 1-27

2 Basic Java Syntax and Coding Conventions

Objectives 2-2

Toolkit Components 2-4

Java Packages 2-5

Documenting Using Java SE 2-6

Contents of a Java Source File 2-8

Naming Conventions 2-9

More About Naming Conventions 2-11

Defining a Class 2-13

Rental Class: Example 2-14

Creating Code Blocks 2-16

Defining Java Methods 2-17
Example of a Method 2-18
Declaring Variables 2-19
Examples of Variables in the Context of a Method 2-20
Rules for Creating Statements 2-21
Compiling and Running a Java Application 2-22
Debugging a Java Program 2-23
CLASSPATH Variable 2-24
Classpath Use Examples 2-25
Summary 2-26
Practice 2 Overview: Basic Java Syntax and Coding Conventions 2-27
UML Diagram for OrderEntry 2-28

3 Exploring Primitive Data Types and Operators
Objectives 3-2
Keywords and Reserved Words 3-3
Variable Types 3-4
Primitive Data Types 3-5
Variables 3-7
Declaring Variables 3-8
Local Variables 3-9
Defining Variable Names 3-10
Numeric Literals 3-11
Nonnumeric Literals 3-13
Operators 3-15
Categories of Operators 3-16
Using the Assignment Operator 3-17
Arithmetic Operators 3-18
More on Arithmetic Operators 3-19
Guided Practice: Declaring Variables 3-20
Examining Conversions and Casts 3-22
Incrementing and Decrementing Values 3-24
Relational and Equality Operators 3-25
Conditional Operator (?:) 3-26
Logical Operators 3-27
Compound Assignment Operators 3-28
Operator Precedence 3-29
More on Operator Precedence 3-30
Concatenating Strings 3-31
Summary 3-32
Practice 3 Overview: Exploring Primitive Data Types and Operators 3-33

4 Controlling Program Flow

Objectives 4-2

Basic Flow Control Types 4-3

Using Flow Control in Java 4-4

`if` Statement 4-5

Nested `if` Statements 4-6

Guided Practice: Spot the Mistakes 4-7

`switch` Statement 4-8

More About the `switch` Statement 4-9

Looping in Java 4-10

`while` Loop 4-12

`do…while` Loop 4-13

`for` Loop 4-14

More About the for Loop 4-15

Guided Practice: Spot the Mistakes 4-16

`break` Statement 4-17

`continue` Statement 4-18

Summary 4-19

Practice 4 Overview: Controlling Program Flow 4-20

5 Building Applications with Oracle JDeveloper (11*g*)

Objectives 5-2

Oracle JDeveloper (11*g*) 5-3

Oracle JDeveloper (11*g*) Environment 5-4

Application Navigator 5-6

Projects 5-8

Creating JDeveloper Items 5-9

Creating an Application 5-10

Project Properties: Specifying Project Details 5-12

Project Properties: Selecting Additional Libraries 5-13

Adding a New Java SE 5-14

Directory Structure 5-15

Exploring the Skeleton Java Application 5-16

Finding Methods and Fields 5-17

Supporting Code Development with Profiler and Code Coach 5-18

Code Editor Features 5-19

Refactoring 5-21

Using Javadoc 5-24

JDeveloper Help System 5-25

Obtaining Help on a Topic 5-26
Oracle JDeveloper Debugger 5-27
Breakpoints 5-29
Debugger Windows 5-31
Stepping Through a Program 5-32
Watching Data and Variables 5-33
Summary 5-34
Practice 5 Overview: Building Java with Oracle JDeveloper 11*g* 5-35

6 Creating Classes and Objects
Objectives 6-2
Object-Oriented Programming 6-3
Classes and Objects 6-5
Classes Versus Objects 6-6
Objects Are Modeled as Abstractions 6-7
Encapsulation 6-8
Inheritance 6-9
Polymorphism 6-10
Guided Practice: Spot the Operations and Attributes 6-11
Java Classes 6-12
Comparing Classes and Objects 6-13
Creating Objects 6-14
`new` Operator 6-15
Primitive Variables and Object Variables 6-16
`null` Reference 6-17
Assigning References 6-18
Declaring Instance Variables 6-19
Accessing `public` Instance Variables 6-20
Defining Methods 6-21
Calling a Method 6-22
Specifying Method Arguments: Examples 6-23
Returning a Value from a Method 6-25
Calling Instance Methods 6-26
Encapsulation in Java 6-27
Passing Primitives to Methods 6-28
Passing Object References to Methods 6-29
Java Packages 6-30
Grouping Classes in a Package 6-31
Setting the `CLASSPATH` with Packages 6-32
Access Modifiers 6-34
Practice 6 Overview: Creating Classes and Objects 6-36

JavaBeans 6-37
More About JavaBeans 6-38
Managing Bean Properties 6-39
Exposing Properties and Methods 6-40
Building and Using a JavaBean in JDeveloper 6-41
Summary 6-42

7 Object Life Cycle and Inner Classes
Objectives 7-2
Overloading Methods 7-3
Using the `this` Reference 7-4
Initializing Instance Variables 7-5
Class Variables 7-6
Initializing Class Variables 7-7
Class Methods 7-8
Guided Practice: Class Methods or Instance Methods 7-9
Examples of Static Methods in Java 7-10
Constructors 7-11
Defining and Overloading Constructors 7-12
Sharing Code Between Constructors 7-13
`final` Variables, Methods, and Classes 7-14
Reclaiming Memory 7-15
`finalize()`Method 7-16
Inner Classes 7-18
Anonymous Inner Classes 7-20
`Calendar` Class 7-22
Performing Calculations with the `Calendar` Class 7-23
Summary 7-26
Practice 7 Overview: Object Life Cycle Classes 7-27

8 Using Strings
Objectives 8-2
Strings in Java 8-3
Creating Strings 8-4
Concatenating Strings 8-5
Performing Operations on Strings 8-6
Performing More Operations on Strings 8-7
Comparing `String` Objects 8-8
Producing Strings from Other Objects 8-10
Producing Strings from Primitives 8-11
Producing Primitives from Strings 8-12

Wrapper Class Conversion Methods 8-13
Changing the Contents of a String 8-14
Formatting Classes 8-16
Formatting Dates 8-17
DecimalFormat Subclass 8-19
Using DecimalFormat for Localization 8-21
Guided Practice 8-23
A Regular Expression 8-25
Matching Strings 8-26
Replacing and Splitting Strings 8-27
Pattern Matching 8-28
Regular Expression Syntax 8-29
Steps Involved in Matching 8-31
Guided Practice 8-33
Summary 8-35
Practice 8 Overview: Using Strings and the StringBuffer, Wrapper, and Text- Formatting Classes 8-36

9 Using Streams for I/O
Objectives 9-2
Streams 9-3
Sets of I/O Classes 9-4
How to Do I/O 9-5
Why Java I/O Is Hard 9-6
Byte I/O Streams 9-7
InputStream 9-9
OutputStream 9-10
Using Byte Streams 9-11
Character I/O Streams 9-13
Using Character Streams 9-15
The InputStreamReader Class 9-17
The OutputStreamWriter Class 9-18
The Basics: Standard Output 9-19
PrintStream and PrintWriter 9-20
Formatted Output 9-22
Format Specifiers 9-23
Guided Practice 9-25
The Basics: Standard Input 9-26
Scanner API 9-28
Remote I/O 9-29

Data Streams 9-31
Object Streams 9-32
Object Serialization 9-33
Serialization Streams, Interfaces, and Modifiers 9-36
`IOException` Class 9-37
Summary 9-39
Practice 9 Overview: Using Streams for I/O 9-40

10 Inheritance and Polymorphism
Objectives 10-2
Key Object-Oriented Components 10-3
Example of Inheritance 10-5
Specifying Inheritance in Java 10-6
Defining Inheritance with Oracle JDeveloper 10-8
Subclass and Superclass Variables 10-9
Default Initialization 10-10
`Super()` Reference 10-11
`Super()` Reference: Example 10-12
Using Superclass Constructors 10-13
Specifying Additional Methods 10-15
Overriding Superclass Methods 10-17
Invoking Superclass Methods 10-19
Example of Polymorphism in Java 10-20
Treating a Subclass as Its Superclass 10-21
Browsing Superclass References with Oracle JDeveloper 10-22
Hierarchy Browser 10-23
Acme Video and Polymorphism 10-24
Using Polymorphism for Acme Video 10-25
`instanceof` Operator 10-27
Limiting Methods and Classes with `final` 10-29
Ensuring Genuine Inheritance 10-31
Summary 10-32
Practice 10 Overview: Inheritance and Polymorphism 10-33

11 Arrays and Collections
Objectives 11-2
Arrays 11-3
Creating an Array of Primitives 11-4
Declaring an Array 11-5
Creating an Array Object 11-6
Initializing Array Elements 11-8

Creating an Array of Object References 11-9
Initializing the Objects in an Array 11-10
Using an Array of Object References 11-11
Going Through the Array Elements 11-12
Arrays and Exceptions 11-13
Multidimensional Arrays 11-14
Passing Command-Line Parameters to `main()` 11-15
Java Collections Framework 11-16
Framework Interface Hierarchy 11-17
Collections Framework Components 11-18
The `Collection` Interface and the `AbstractCollection` Class 11-19
`Iterator` Interface 11-20
Sets 11-22
`HashSet` 11-23
`LinkedHashSet` 11-25
`TreeSet` 11-27
Lists 11-29
`ArrayList` 11-30
Modifying an `ArrayList` 11-31
Accessing an `ArrayList` 11-32
`LinkedList` 11-33
Maps 11-35
Types of Maps 11-36
Example of Using Maps 11-37
Summary 11-39
Practice 11 Overview: Using Arrays and Collections 11-40

12 Using Generic Types
Objectives 12-2
Generics 12-3
Declaring Generic Classes 12-5
Using Generic Classes 12-6
Generic Methods 12-7
Wildcards 12-9
Raw Types 12-11
Type Erasure 12-12
Summary 12-13

13 Structuring Code Using Abstract Classes and Interfaces
Objectives 13-2
Abstract Classes 13-3
Creating Abstract Classes 13-5
Abstract Methods 13-6
Defining Abstract Methods 13-7
Defining and Using Interfaces 13-8
Examples of Interfaces 13-9
Creating Interfaces 13-10
Interfaces Versus Abstract Classes 13-11
Implementing Interfaces 13-12
Sort: A Real-World Example 13-13
Overview of the Classes 13-14
How the Sort Works 13-15
`Sortable` Interface 13-16
`Sort` Class 13-17
`Movie` Class 13-18
Using the Sort 13-19
Using `instanceof` with Interfaces 13-20
Summary 13-21
Practice 13 Overview: Structuring Code Using Abstract Classes and Interfaces 13-22

14 Throwing and Catching Exceptions
Objectives 14-2
What Is an Exception? 14-3
Exception Handling in Java 14-4
Advantages of Java Exceptions: Separating Error-Handling Code 14-5
Advantages of Java Exceptions: Passing Errors Up the Call Stack 14-7
Advantages of Java Exceptions: Exceptions Cannot Be Ignored 14-8
Checked Exceptions, Unchecked Exceptions, and Errors 14-9
Handling Exceptions 14-11
Catching and Handling Exceptions 14-12
Catching a Single Exception 14-14
Catching Multiple Exceptions 14-15
Cleaning Up with a `finally` Block 14-16
Guided Practice: Catching and Handling Exceptions 14-17
Allowing an Exception to Pass to the Calling Method 14-19
Throwing Exceptions 14-20
Creating Exceptions 14-21

Catching an Exception and Throwing a Different Exception 14-22
Summary 14-23
Practice 14 Overview: Throwing and Catching Exceptions 14-24

15 Using JDBC to Access the Database
Objectives 15-2
Java, Java EE, and Oracle 11*g* 15-3
Connecting to a Database with Java 15-4
Java Database Connectivity (JDBC) 15-5
Preparing the Environment 15-6
Steps for Using JDBC to Execute SQL Statements 15-7
Step 1: Register the Driver 15-8
Connecting to the Database 15-9
Oracle JDBC Drivers: Thin-Client Driver 15-10
Oracle JDBC Drivers: OCI Client Driver 15-11
Choosing the Right Driver 15-12
Step 2: Obtain a Database Connection 15-13
JDBC URLs 15-14
JDBC URLs with Oracle Drivers 15-15
Step 3: Create a Statement 15-16
Using the `Statement` Interface 15-17
Step 4a: Code the Query Statement 15-18
`ResultSet` Object 15-19
Step 4b: Submit DML Statements 15-20
Step 4b: Submit DDL Statements 15-21
Step 5: Process the Query Results 15-22
Mapping Database Types to Java Types 15-23
Step 6: Clean Up 15-25
Basic Query Example 15-26
Handling an Unknown SQL Statement 15-27
Handling Exceptions 15-28
Transactions with JDBC 15-29
`PreparedStatement` Object 15-31
Creating a `PreparedStatement` Object 15-32
Executing a `PreparedStatement` Object 15-33
What Is a `DataSource`? 15-34
Advantages of Using a `DataSource` 15-35
Using an `OracleDataSource` to Connect to a Database 15-36
Maximizing Database Access with Connection Pooling 15-38
Connection Pooling 15-40

Summary 15-43
Practice 15 Overview: Using JDBC to Access the Database 15-44

16 User Interface Design: Swing Basics for Planning the Application Layout
Objectives 16-2
AWT, Swing, and JFC 16-3
Swing Features 16-5
Lightweight and Heavyweight Components 16-6
Planning the UI Layout 16-7
Swing Containment Hierarchy 16-8
Top-Level Containers 16-10
Intermediate Containers 16-12
Atomic Components 16-14
Layout Management Overview 16-15
Border Layout 16-17
GridBag Layout 16-18
GridBag Constraints 16-19
Using Layout Managers 16-21
Combining Layout Managers 16-23
Java Frame Classes 16-24
`JPanel` Containers 16-26
Internal Frames 16-28
Adding Components with Oracle JDeveloper 16-29
Creating a Frame 16-30
Adding Components 16-31
Pluggable Look and Feel 16-33
Summary 16-35
Practice 16 Overview: Swing Basics for Planning the Application Layout 16-36

17 Adding User Interface Components and Event Handling
Objectives 17-2
Swing Components 17-3
Swing Components in JDeveloper 17-5
Invoking the UI Editor 17-7
Adding a Component to a Form 17-8
Editing the Properties of a Component 17-9
Code Generated by JDeveloper 17-10
Creating a Menu 17-12
Using the JDeveloper Menu Editor 17-13
Practice 17-1 Overview: Adding User Interface Components 17-14
UI for the Order Entry Application 17-15

Java Event Handling Model 17-16
Event Listener Handling Code Basics 17-17
Event Handling Process: Registration 17-18
Event Handling Process: The Event Occurs 17-20
Event Handling Process: Running the Event Handler 17-21
Using Adapter Classes for Listeners 17-22
Swing Model-View-Controller Architecture 17-23
Basic Text Component Methods 17-26
Basic `JList` Component Methods 17-27
What Events Can a Component Generate? 17-28
Defining an Event Handler in JDeveloper 17-29
Default Event Handling Code Style Generated by JDeveloper 17-30
Completing the Event Handler Method 17-31
Summary 17-32
Practice 17-2 Overview: Adding Event Handling 17-33

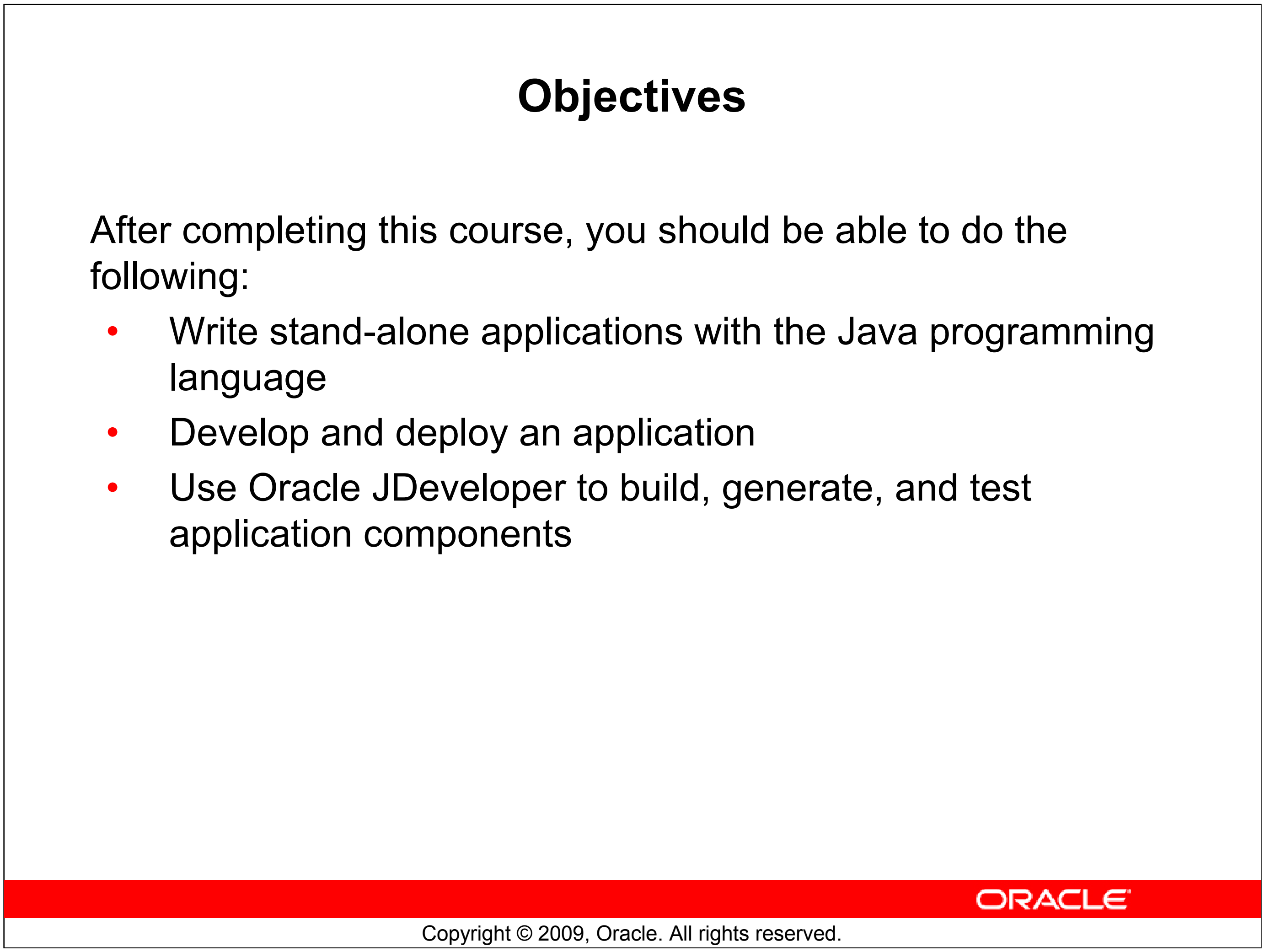
18 Deploying Java Applications
Objectives 18-2
Packaging and Deploying Java Projects 18-3
Deploying a `.jar` File 18-4
Deploying Applications with JDeveloper 18-5
Creating the Deployment Profile 18-6
Selecting Files to Deploy 18-8
Creating and Deploying the Archive File 18-10
Creating an Executable `.jar` File 18-11
Java Web Start 18-13
Advantages of Web Start 18-14
Running a Web Start Application 18-15
Examining the JNLP File 18-16
Using JDeveloper to Deploy an Application for Java Web Start 18-17
Step 1: Generate Deployment Profiles and Application Archive 18-18
Step 2a: Start the Server 18-19
Step 2b: Test the Connection 18-20
Step 3: Use the Web Start Wizard to Create a JNLP File 18-21
Step 4: Archive and Deploy the Application to the WebLogic Server 18-22
Summary 18-23

Appendix A: Practices

Appendix B: Java Language Quick-Reference Guide

I

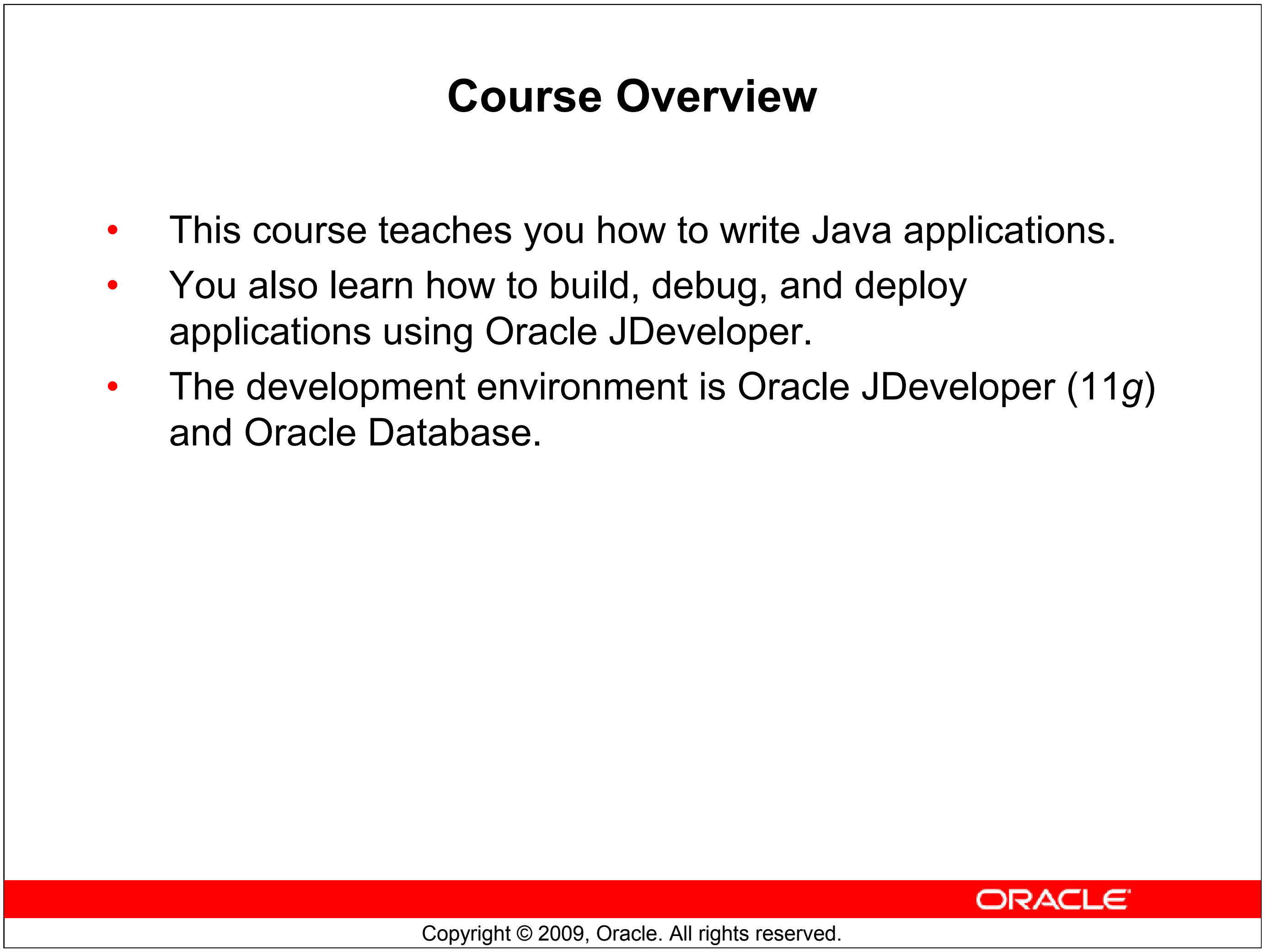
Introduction



Objectives

After completing this course, you should be able to do the following:

- Write stand-alone applications with the Java programming language
- Develop and deploy an application
- Use Oracle JDeveloper to build, generate, and test application components



Course Overview

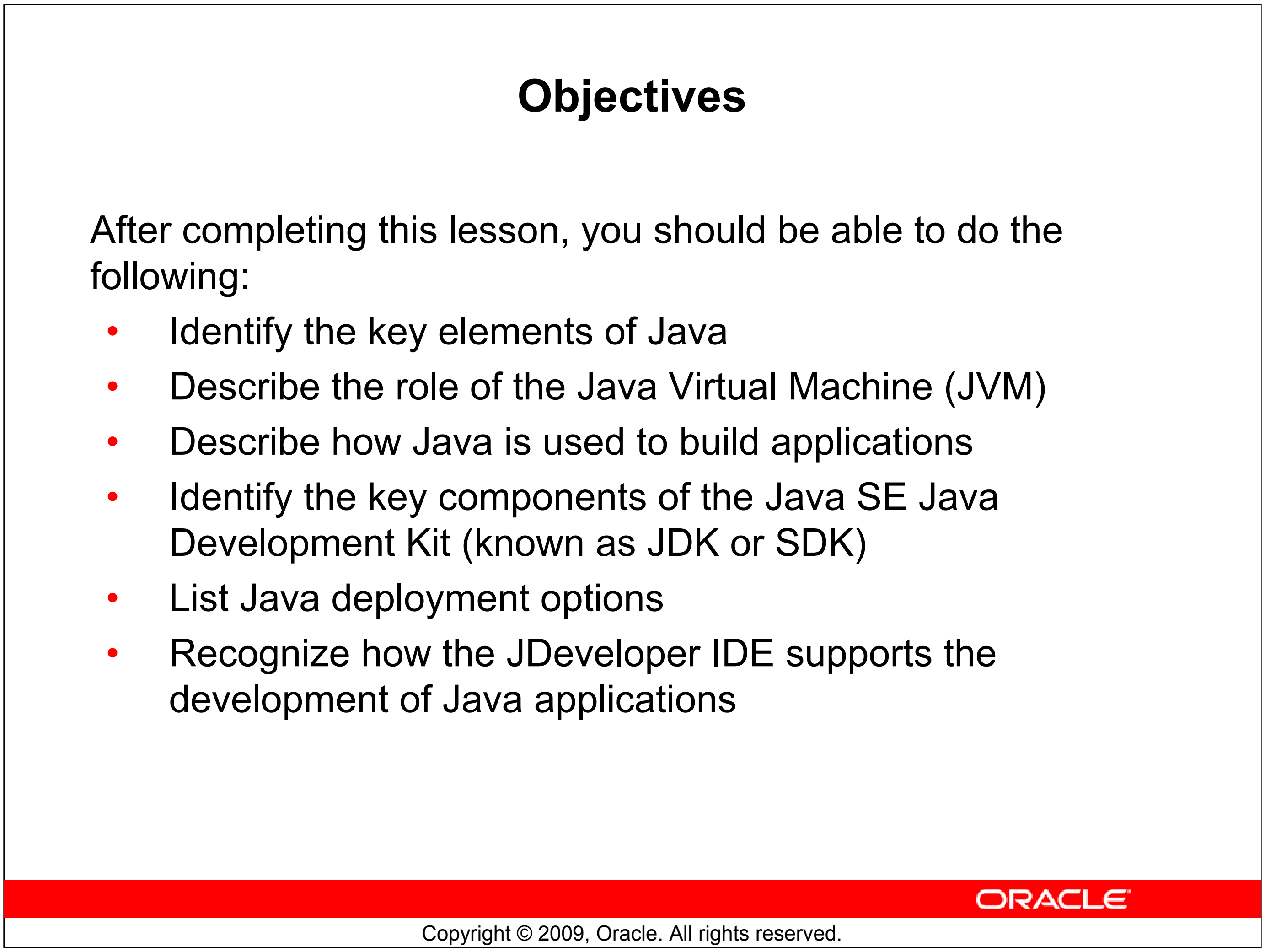
- This course teaches you how to write Java applications.
- You also learn how to build, debug, and deploy applications using Oracle JDeveloper.
- The development environment is Oracle JDeveloper (11*g*) and Oracle Database.

Development Environment

The version of JDeveloper used in the course is Release 11*g*.

1

Introducing the Java and Oracle Platforms



Objectives

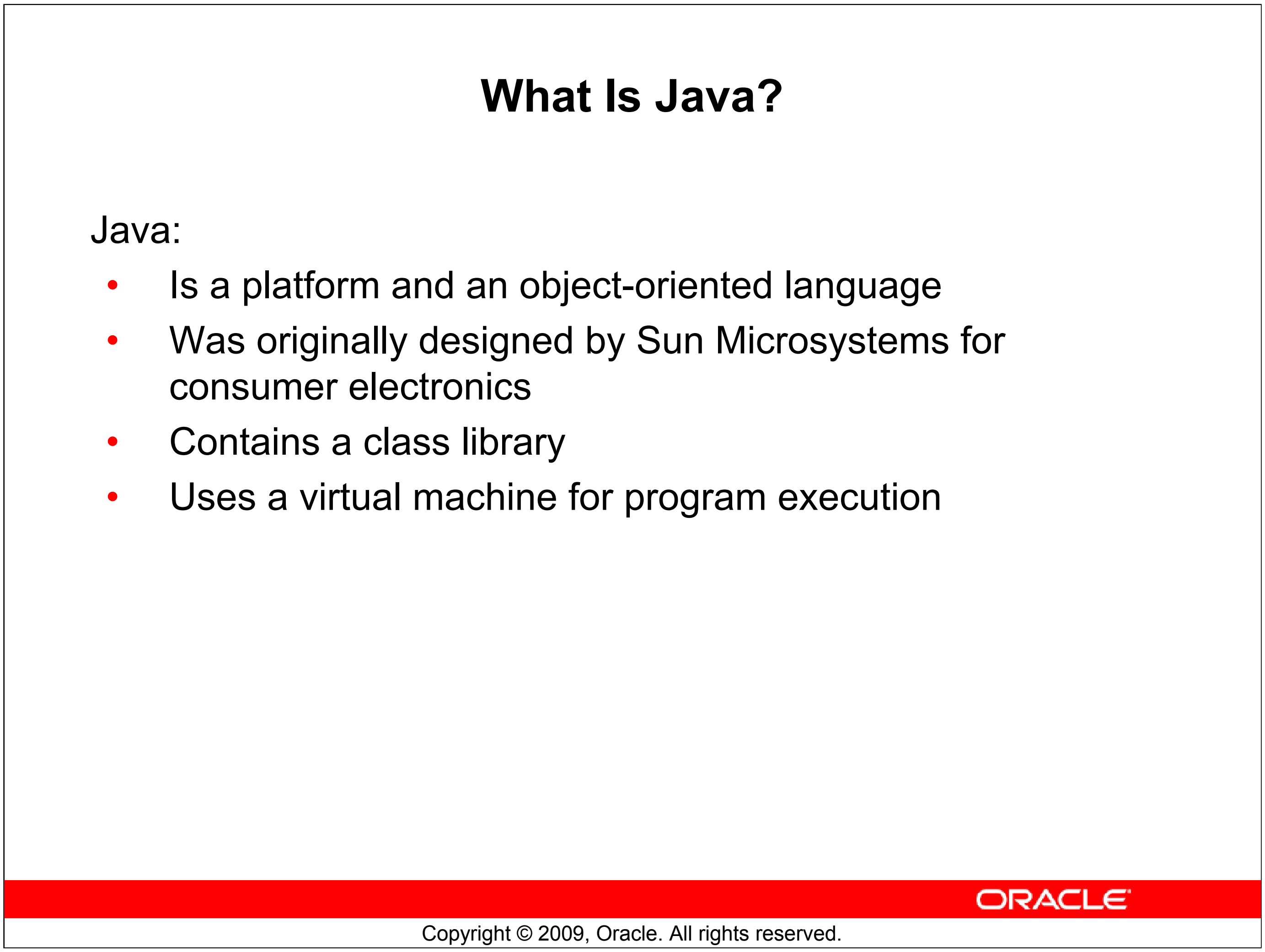
After completing this lesson, you should be able to do the following:

- Identify the key elements of Java
- Describe the role of the Java Virtual Machine (JVM)
- Describe how Java is used to build applications
- Identify the key components of the Java SE Java Development Kit (known as JDK or SDK)
- List Java deployment options
- Recognize how the JDeveloper IDE supports the development of Java applications

Lesson Objectives

This lesson provides some background information about the Java language and discusses the role of Java in the Oracle11*g* platform.

Java is the programming language of choice for Internet applications. It has gained this status because of its robustness, its object-oriented nature, the depth of its predefined classes, and its “write once, run anywhere” deployment model. In this lesson, you learn how Java supports object-oriented programming and architecture-neutral deployment.



What Is Java?

Java:

- Is a platform and an object-oriented language
- Was originally designed by Sun Microsystems for consumer electronics
- Contains a class library
- Uses a virtual machine for program execution

What Is Java?

Designed by Sun Microsystems

Originally developed by Sun Microsystems, Inc., Java is a platform and an object-oriented programming language. It was created by James Gosling for use in consumer electronics. Because of the robustness and platform-independent nature of the language, Java soon moved beyond the consumer electronics industry and found a home on the World Wide Web. Java is a platform, which means that it is a complete development and deployment environment.

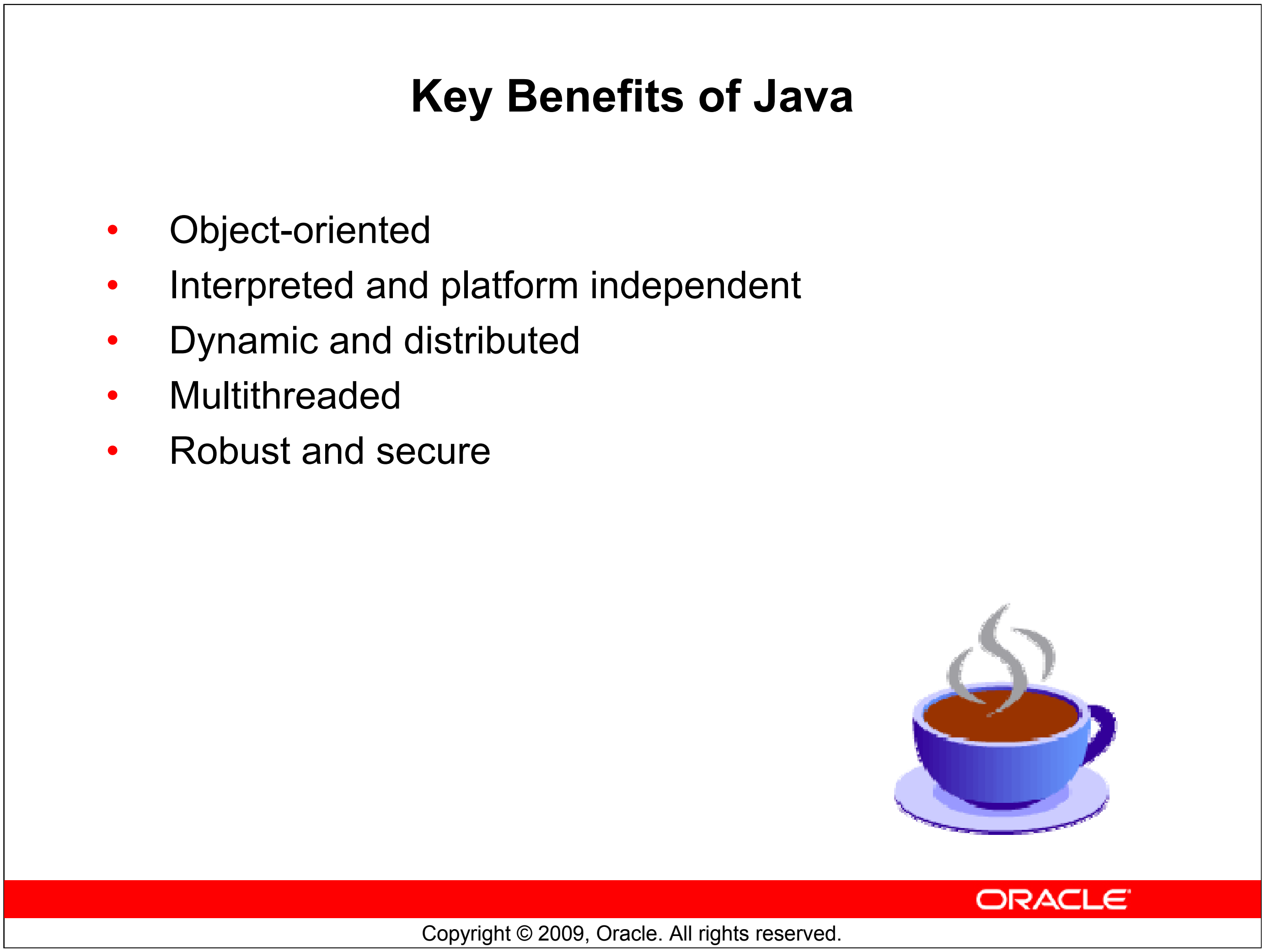
Class Libraries

Java comes with a broad set of predefined classes that contain attributes and methods that handle most of the fundamental requirements of programs. Window management, input/output, and network communication classes are included in the Java Development Kit (JDK). The class library makes Java programming significantly easier and faster to develop when compared with other languages. JDK also contains several utilities to facilitate development processes. These utilities handle operations such as debugging, deployment, and documentation.

What Is Java? (continued)

Java Uses a Virtual Machine

One of the key elements of Java is platform independence. A Java program that is written on one platform can be deployed on any other platform. This is usually referred to as “write once, run anywhere” (WORA) and is accomplished through the use of the Java Virtual Machine (JVM). The JVM runs on a local machine, interprets the Java bytecode, and converts it into platform-specific machine code.



Key Benefits of Java

- Object-oriented
- Interpreted and platform independent
- Dynamic and distributed
- Multithreaded
- Robust and secure

Key Benefits of Java

Object-Oriented

An object is an entity that has data attributes, plus a set of functions that are used to manipulate the object. Java is a strongly typed language, which means that everything in Java must have a data type—there are no variables that assume data types (as in Visual Basic). The main exceptions are primitive data types, such as integers and characters.

Interpreted and Platform Independent

Java programs are interpreted to the native machine’s instruction set at run time. Because Java executes under the control of a JVM, Java programs can run on any operating system that provides a JVM.

Dynamic and Distributed

Java classes can be downloaded dynamically over the network when required. In addition, Java provides extensive support for client/server and distributed programming.

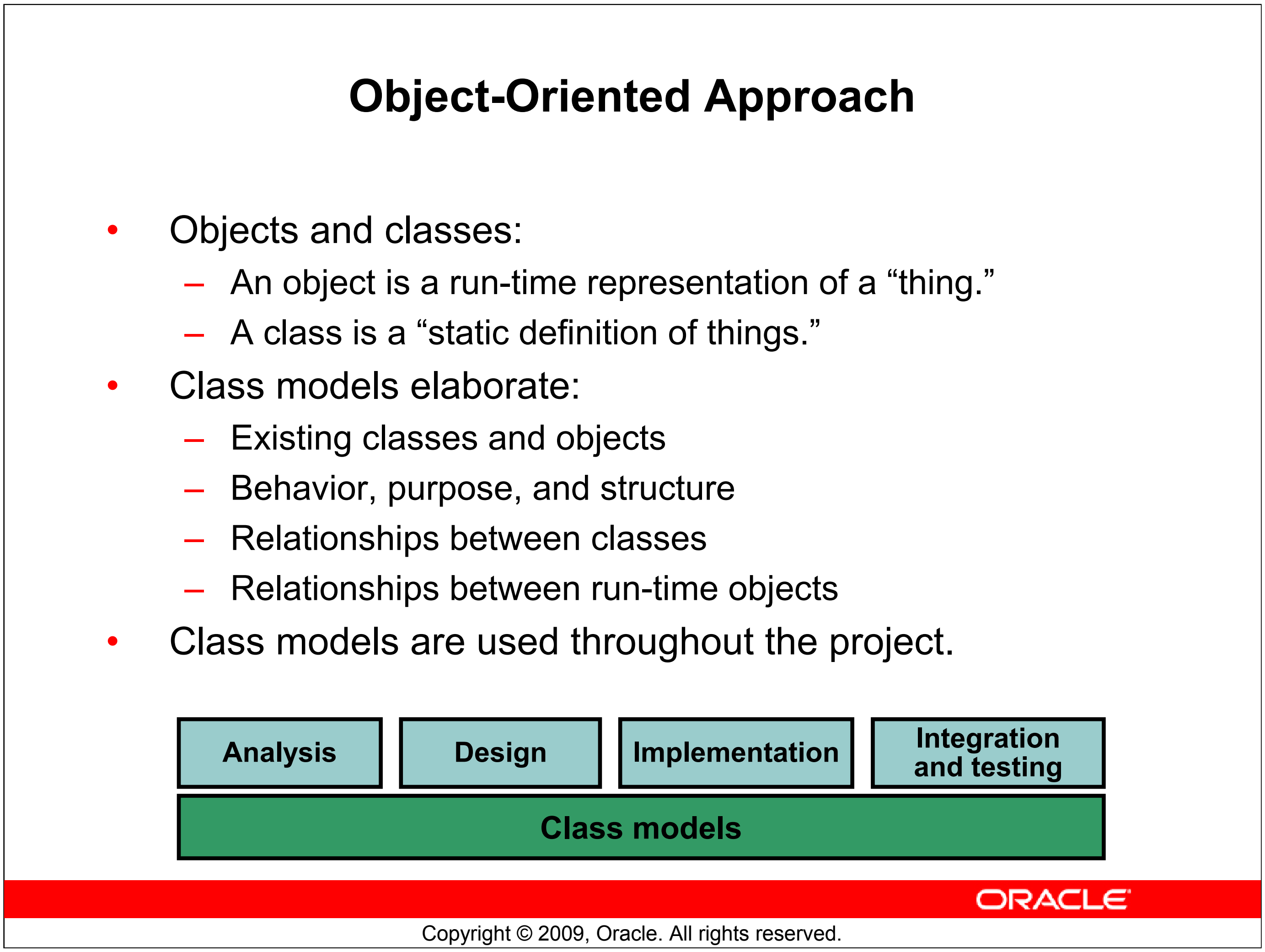
Key Benefits of Java (continued)

Multithreaded

Java programs can contain multiple threads to carry out many tasks in parallel. The multithreading capability of Java is built in and is under the control of the platform-dependent JVM.

Robust and Secure

Java has built-in capabilities to prevent memory corruption. Java manages the processes of memory allocation and array-bounds checking. It prohibits pointer arithmetic and restricts objects to named spaces in memory.



Object-Oriented Approach

- Objects and classes:
  - An object is a run-time representation of a “thing.”
  - A class is a “static definition of things.”
- Class models elaborate:
  - Existing classes and objects
  - Behavior, purpose, and structure
  - Relationships between classes
  - Relationships between run-time objects
- Class models are used throughout the project.

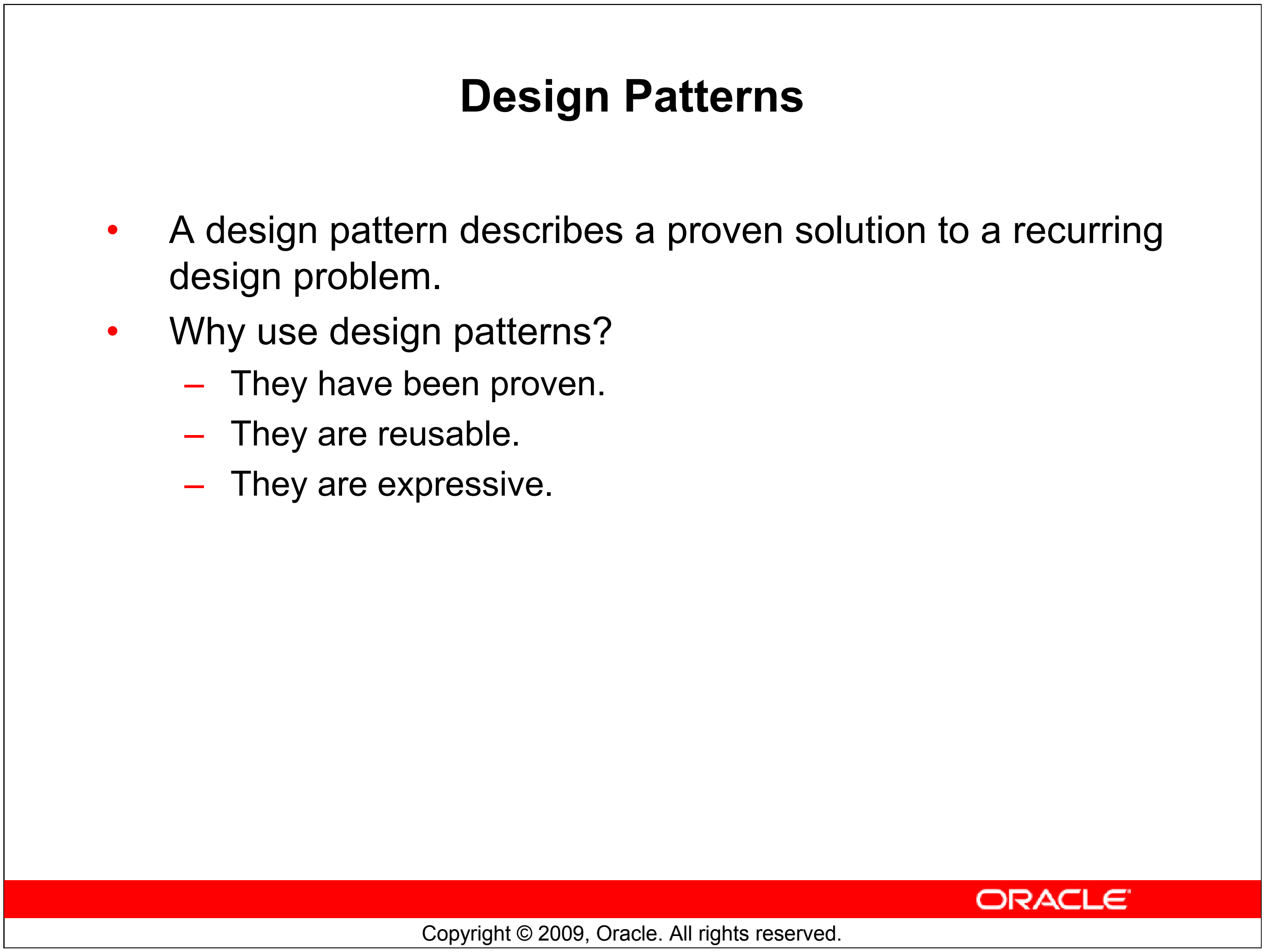
Analysis | Design | Implementation | Integration and testing

Class models

Object-Oriented Approach

In procedural programming, data and operations on the data are separate and this approach requires sending data to methods. Object-oriented programming places data and the operations that pertain to them within a single entity called an *object*; this approach solves many of the problems inherent in procedural programming. The object-oriented programming approach organizes programs in a way that mirrors the real world, in that all objects are associated with both attributes and activities. Using objects improves software reusability and makes programs easier to develop and easier to maintain. Programming in Java involves thinking in terms of objects—a Java program can be viewed as a collection of cooperating objects.

Classes provide a means to capture the structure and behavior of a real-world person, place, or thing; classes represent a one-to-one mapping between the real-world object and its implementation. This one-to-one mapping tends to eliminate the typical transformations that are found in design approaches that are not object-oriented.



Design Patterns

- A design pattern describes a proven solution to a recurring design problem.
- Why use design patterns?
  - They have been proven.
  - They are reusable.
  - They are expressive.

Design Patterns

As you design and build different applications, you continually come across the same, or similar, problems. Design patterns describe proven solutions, from experienced hands, for recurring design problems.

Why use design patterns?

- They have been proven. Patterns reflect the experience, knowledge, and insight of developers who have successfully used these patterns in their own work.
- They are reusable. Patterns provide a ready-made solution that can be adapted to different problems as necessary.
- They are expressive. Patterns provide a common vocabulary of solutions that can express complex solutions succinctly.

The MVC Design Pattern

The Model-View-Controller (MVC) design pattern:

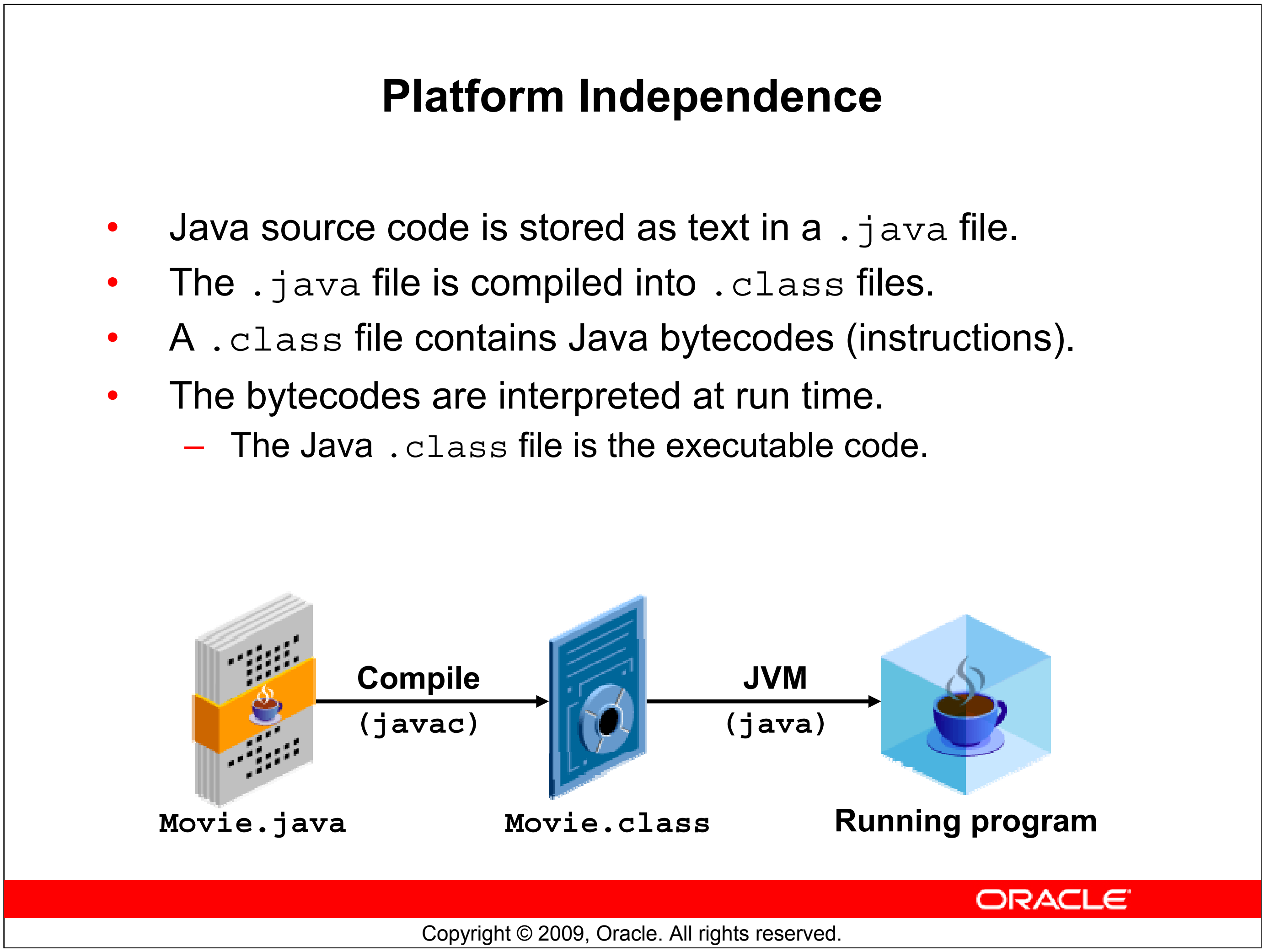
- Is one of the most popular architectural patterns for object-oriented design
- Splits an application into three distinct parts:
  - The Model handles data and logic.
  - The View handles output.
  - The Controller handles input.

The MVC Design Pattern

MVC originated in the late 1970s in the object-oriented language Smalltalk. It remains one of the most popular architectural patterns for object-oriented design.

The MVC design pattern separates user input logic and data presentation from data access and business logic.

- The Model handles data and logic. Model code accesses and represents data and handles business operations. “Business operations” include changes to both persistent business data and to things such as shopping cart contents.
- The View handles output. View code displays data to the user.
- The Controller handles input. Controller code manipulates the Model or changes the view accordingly in response to user input (the Controller can also do both).
- You can find out more about MVC in the following resources:
  - *Design Patterns: Elements of Reusable Object-Oriented Software*. Gamma, Helm, Johnson, and Vlissides. Addison-Wesley, 1995. ISBN 0201633612.
  - Java BluePrints MVC Design Pattern (http://java.sun.com/blueprints/patterns/MVC.html) (http://java.sun.com/blueprints/patterns/MVC-detailed.html)



Platform Independence

- Java source code is stored as text in a `.java` file.
- The `.java` file is compiled into `.class` files.
- A `.class` file contains Java bytecodes (instructions).
- The bytecodes are interpreted at run time.
  - The Java `.class` file is the executable code.

Compile
(javac)
JVM
(java)
Movie.java
Movie.class
Running program

Platform Independence

Java Is an Interpreted Language

Java program source code is stored in `.java` files. For example, a Java program dealing with movies in a video rental company may have files called `Movie.java`, `Customer.java`, and `Rental.java`. Each `.java` file is compiled into a corresponding `.class` file with the same name. For example, a `Movie.java` file compiles to at least one class file. (Inner classes are quite common.) However, in this case the public `Movie.java` file compiles to just one `Movie.class`. These `.class` files contain Java bytecodes, which are platform-independent machine instructions.

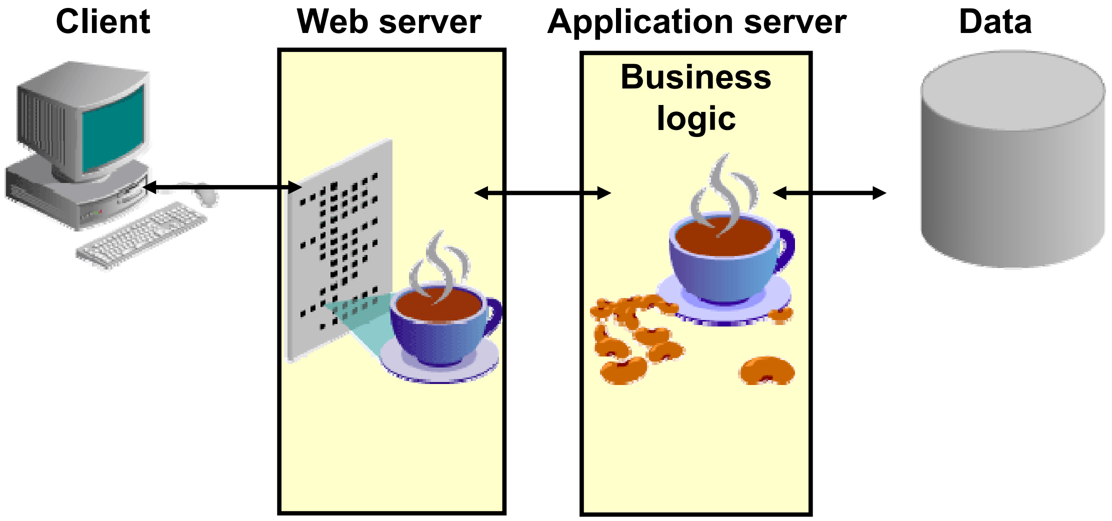
JVM

The JVM provides the environment for running Java programs. The JVM interprets Java bytecodes into the native instruction set for the machine on which the program currently runs. The same `.class` files can be executed unaltered on any platform for which a JVM is provided. For this reason, JVM is sometimes referred to as a virtual processor.

Traditional Compiled Languages

When a programmer compiles a traditional language such as C, the code written is converted into machine instructions for the platform on which the compilation takes place. This compiled program can then run only on machines that have the same processor as that on which it was compiled, such as Intel, SPARC, or Alpha.

Using Java with Enterprise Internet Computing



- HTTP requests

- Enterprise JavaBeans (EJB)
- CORBA
- Servlets
- JavaServer Pages/JavaServer Faces

Using Java with Enterprise Internet Computing

You can design Java programs as server-based components that form scalable Internet applications.

The currently accepted model for Java Internet computing divides the end-to-end application process into several logical tiers. To make use of this model, JavaSoft defined Java Platform, Enterprise Edition (Java EE). There are four logical tiers:

Client Tier

When Java executes on client machines, it is typically implemented as a browser-based application. But a thin client can be simply Web pages that are delivered from a server as HTML.

Web (Presentation) Tier

This is executed on a Web server. Code in this tier handles the application's presentation to the client. The Web server handles the HTTP protocol; when processing an HTTP request a Web server may respond with a static HTML page or image, send a redirect, or delegate the dynamic response generation to some other program such as JavaServer Pages (JSP), servlets, server-side JavaScripts or some other server-side technology. The Web server's delegation model is fairly simple. When a request comes into the Web server, it simply passes the request to the program best able to handle it. The Web server does not provide any functionality beyond simply providing an environment in which the server-side program can execute and pass back the generated responses.

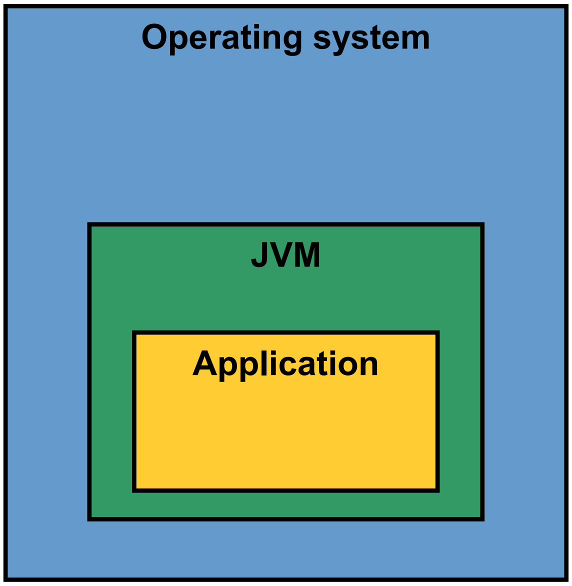
Application (Business Logic) Tier

You can use Java on an application server to implement shareable, reusable business logic as application components. A common way to implement this is to use component models, such as Enterprise JavaBeans (EJB) and Common Object Request Broker Architecture (CORBA) objects. When a distributed environment is required, you also consider these two components during design time. A Java EE application server runs servlets and JSPs that are used to create HTML pages dynamically – in fact a part of the application server called the Web container is responsible for running servlets and JSPs.

Data Tier

The data server not only stores data but also stores and executes Java code, particularly where this code is data intensive or enforces validation rules pertaining to the data. You can also use Business Components, from the Oracle Application Development Framework (ADF), to support your application's data access.

Using the Java Virtual Machine



Operating system

JVM

Application

Running Java Applications

All Java applications run in a JVM. The JVM is invoked differently depending on whether the Java program is an application or an applet.

Applications

You can run stand-alone applications by invoking a local JVM directly from the operating system command line and supplying the name of the main class for the application. After loading the applications main class file, the JVM runs the program by calling a known entry point in the class, that is, a public static method called `main(...)`. The JVM runs the code by interpreting the bytecodes in the Java program and converting them into platform-specific machine instructions.

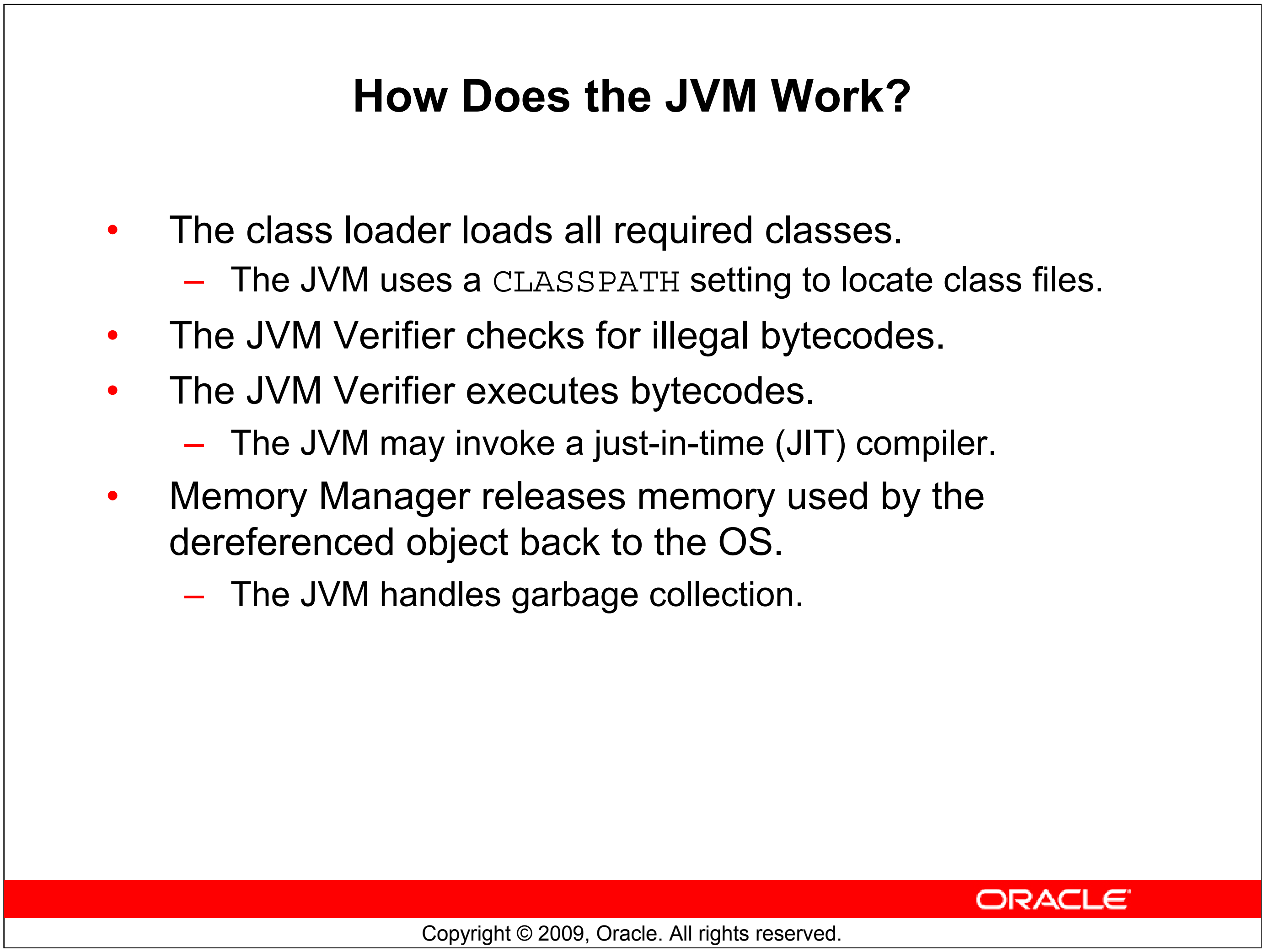
Running Java Applications (continued)

Running Java Applets

A Java applet is a special type of Java program that is used in Web pages. When a Web browser reads an HTML page with an applet tag, it downloads the applet over the network to the local system and runs the applet in a JVM that is built into the browser. The browser invokes a specific call sequence of known methods in the Java applet class to execute the Java code in the context of the browser's JVM. The applet entry points differ from the entry point that is used by JVM to run stand-alone applications.

In the case of an applet, the presentation server is not necessarily used. A Java application is quite able (and typically is so configured) to connect directly to the business logic.

Applets are not covered in this course and are mentioned here only for completeness. In this course, you use the Java Web Start product to deploy your application.



How Does the JVM Work?

- The class loader loads all required classes.
  - The JVM uses a `CLASSPATH` setting to locate class files.
- The JVM Verifier checks for illegal bytecodes.
- The JVM Verifier executes bytecodes.
  - The JVM may invoke a just-in-time (JIT) compiler.
- Memory Manager releases memory used by the dereferenced object back to the OS.
  - The JVM handles garbage collection.

How Does the JVM Work?

JVM Class Loader

When a `.class` file is run, it may require other classes to help perform its task. These classes are loaded by the class loader in the JVM. The required classes may reside on the local disk or on another system across the network. The JVM uses the `CLASSPATH` environment variable to determine the location of local `.class` files. The classpath can be added in run time by using the `java -cp` or `-classpath` option.

Classes that are loaded from the network are kept in a separate namespace from those on the local system. This prevents name clashes and the replacement or overriding of standard classes, malicious or otherwise.

JVM Verifier

It is the job of the verifier to make sure that the Java code that is being interpreted does not violate any of the basic rules of the Java language and that the code is from a trusted source. A trusted source is an option; if a trusted source is used, the check is not performed.

This validation ensures that there are no memory access violations or other illegal actions performed.

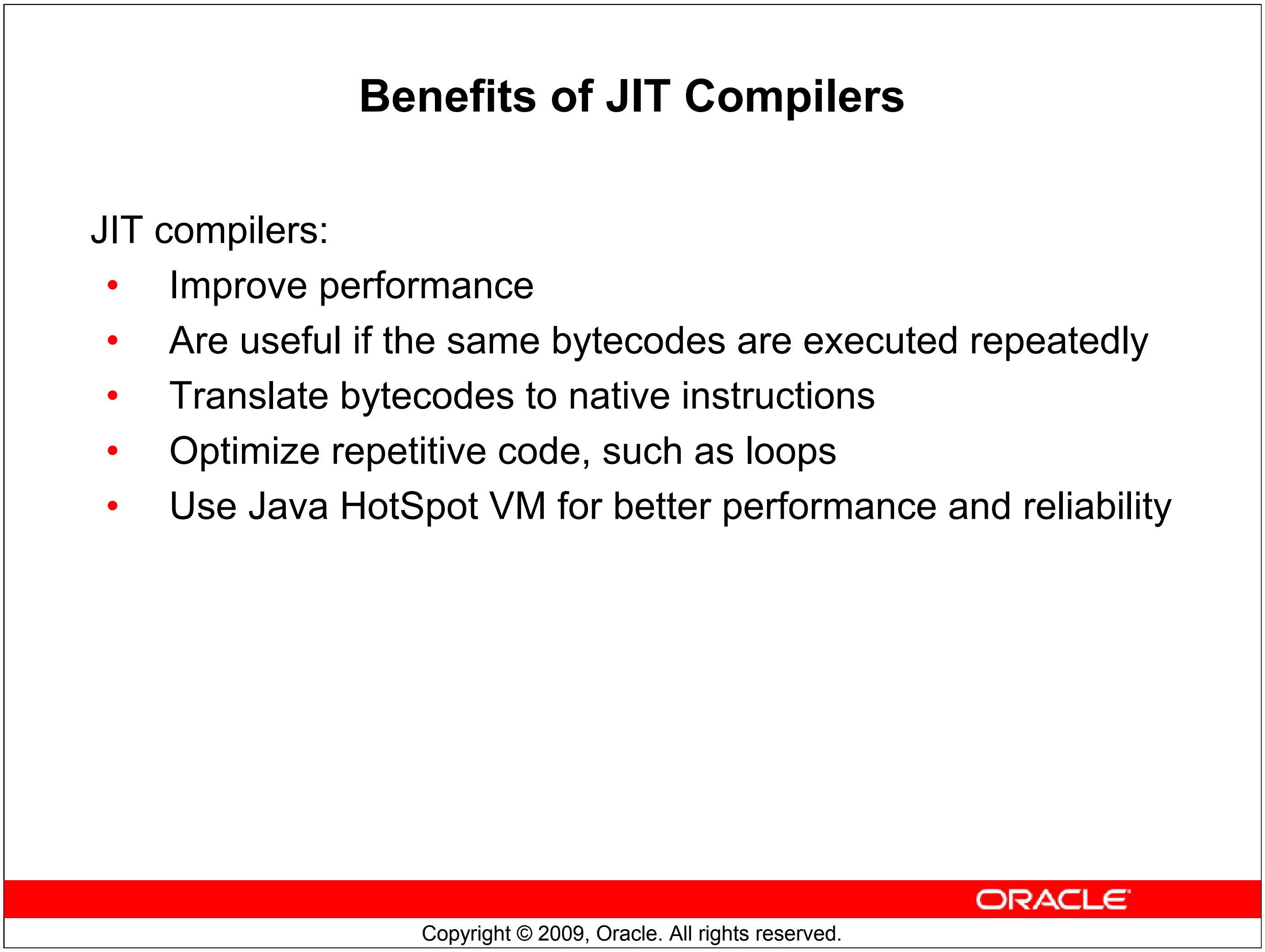
How Does the JVM Work? (continued)

Bytecode Interpreter

The JVM is the bytecode interpreter that executes the bytecodes for the loaded class. If enabled, the JVM can use JIT technology to translate Java bytecodes into native machine instructions.

Memory Management

The JVM keeps track of all instances in use. After an instance is no longer in use, the JVM is allowed to release the memory that is used by that object. It performs the release of memory after the object is no longer needed, but not necessarily immediately. The process (thread) that the JVM uses to manage dereferenced objects is called *garbage collection*.



Benefits of JIT Compilers

JIT compilers:

- Improve performance
- Are useful if the same bytecodes are executed repeatedly
- Translate bytecodes to native instructions
- Optimize repetitive code, such as loops
- Use Java HotSpot VM for better performance and reliability

Benefits of JIT Compilers

JVMs translate Java bytecodes into native machine instructions. What happens if the same code is executed again later in the program? In an environment without JIT compilers, the code is interpreted every time it is encountered, even if it has already been interpreted earlier in the program.

The compilers are designed to easily translate bytecode into machine code, which is optimized to run on the target platform.

Most JVMs now support JIT compilation. JIT compilers translate bytecodes only the first time that they are encountered; if the same code is executed later, it is mapped to the corresponding native machine instruction.

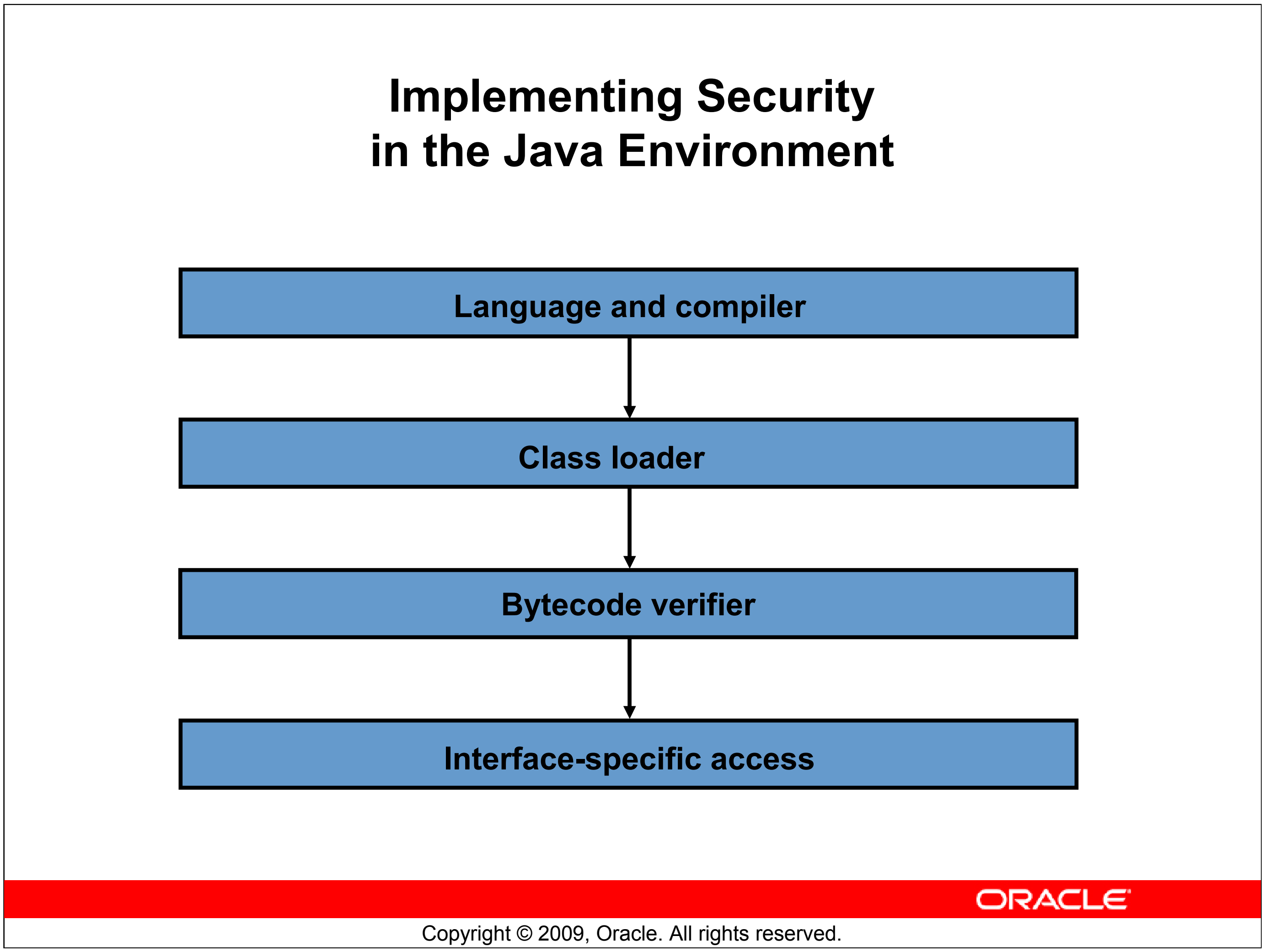
JIT compilers enable Java programs to run more quickly because they obviate the need for the repeated translation of bytecodes to native machine instructions. This is especially effective in repetitive code, such as loops or recursive functions. Some JIT compilers are intelligent enough to optimize groups of related bytecodes into more efficient native machine instructions.

Benefits of JIT Compilers (continued)

Java HotSpot

The Java HotSpot virtual machine (VM) is a key component in maximizing the deployment of enterprise applications. It is a core component of Java 2, Standard Edition (Jave SE) software and is Sun's answer to the limitations of conventional JIT compilers. It uses a wide variety of techniques to increase performance, including adaptive optimization technology, which detects and accelerates performance-critical code. It provides ultrafast thread synchronization for maximum performance of thread-safe programs based on Java technology. It provides a garbage collector (GC) that is not only extremely fast but also fully accurate (and therefore more reliable). Finally, at a source-code level, the Java HotSpot performance engine is written in a clean, high-level, object-oriented design style that provides major improvements in maintainability and extensibility.

The Java HotSpot VM supports virtually all aspects of the development, deployment, and management of corporate applications.



Implementing Security in the Java Environment

Language and compiler

Class loader

Bytecode verifier

Interface-specific access

Java Security Layers

Language and Compiler

Java was designed to be a safe language. The constructs that enable direct manipulation of memory pointers have been eliminated, thereby reducing or even eliminating run-time program crashes and, as a consequence, memory leaks.

Class Loader

The class loader ensures that each class coming from a local source (built-ins) and the classes from each network source are stored separately. During execution, the run-time system first looks up the built-ins for referenced classes; if they are not found, it consults the referencing class. This ensures that built-in classes are not overridden by network-loaded classes. This prevents “spoofing,” or overriding the expected and trusted behavior of a built-in class. Inside a JVM, there can be several classloaders controlling each application’s namespace.

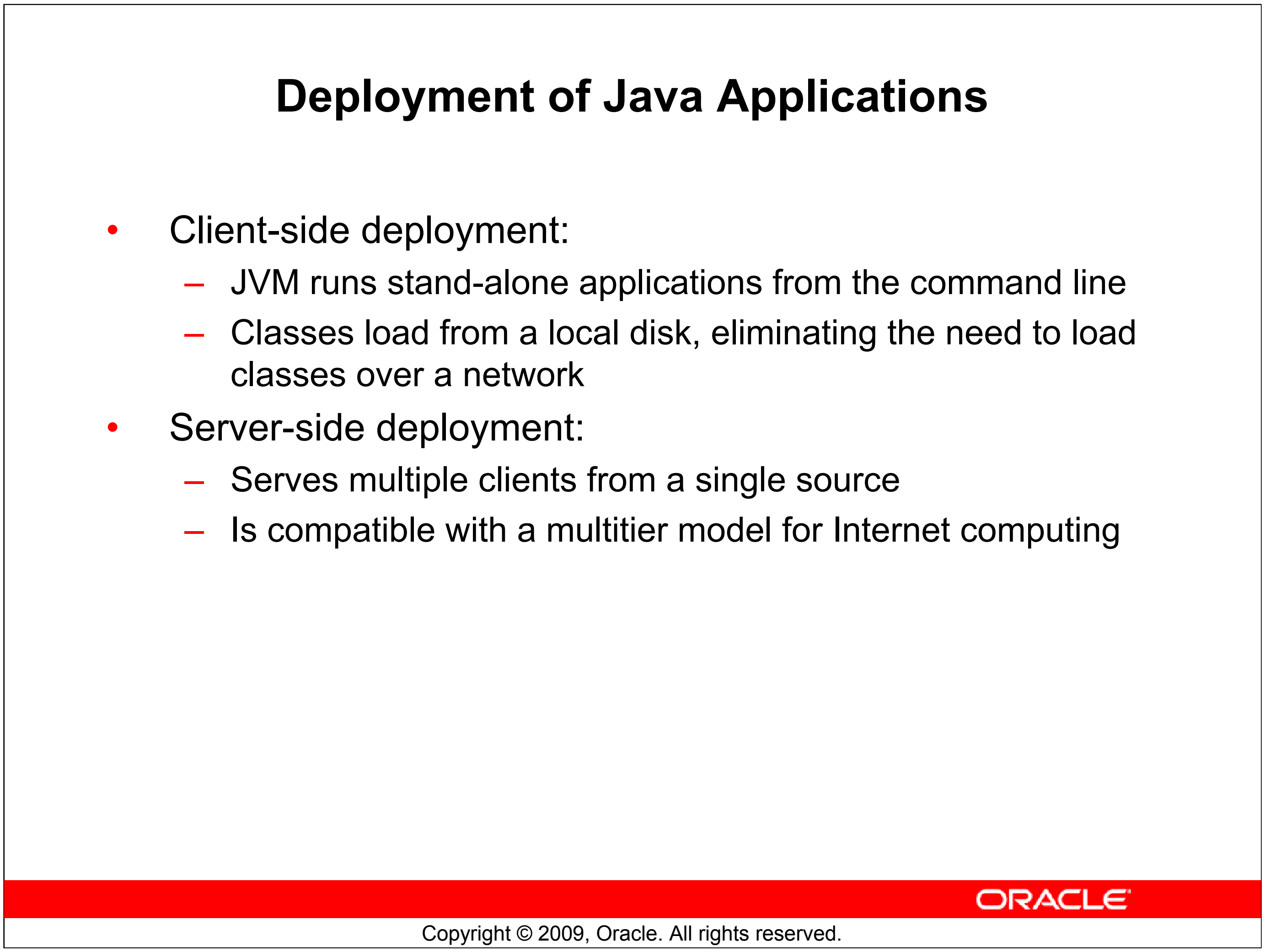
Java Security Layers (continued)

Bytecode Verifier

During the execution of a Java program, the JVM can import code from anywhere. Java must make sure that the imported code is from a trustworthy source. To accomplish this task, the run-time system performs a series of checks (called *bytecode verification*).

Interface-Specific Access

Built-in classes and methods control access to the local file system and network resources. These classes are restrictive by default. If imported code tries to access the local file system, the security mechanism prompts the user.



Deployment of Java Applications

- Client-side deployment:
  - JVM runs stand-alone applications from the command line
  - Classes load from a local disk, eliminating the need to load classes over a network
- Server-side deployment:
  - Serves multiple clients from a single source
  - Is compatible with a multitier model for Internet computing

Deployment of Java Applications

Java originally gained popular acceptance because of the success of its applets. Today, however, it is also possible to write stand-alone applications in Java. A Java application is invoked by using a JVM and is not run from a browser.

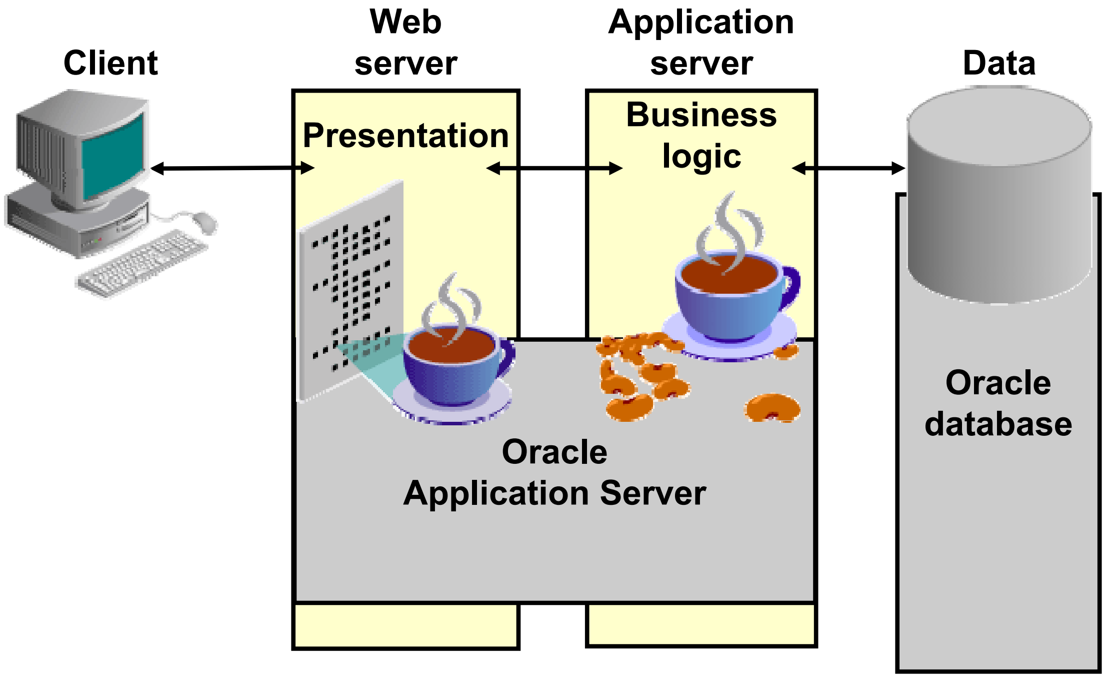
Client-Side Deployment

Java applications can be deployed to run stand-alone applications in a local operating system from the command line. For example, Java applications can access the local file system or establish connections with other machines on the network.

Server-Side Deployment

Java applications can also execute on the server machine, as long as a JVM is available on that platform. The use of server-side Java applications is compatible with the multitier model for Internet computing.

Using Java with Oracle 11*g*



Java and Oracle 11*g*

Oracle 11*g* is a complete and integrated platform that supports all the server-side requirements for Java applications. The Oracle 11*g* platform comprises the following:

Oracle Database 11*g*

In addition to its database management features, Oracle Database (currently version 11*g*) provides support for a variety of Java-based structures, including Java components and Java stored procedures. These Java structures are executed in the database by its built-in Java Virtual Machine, called the Enterprise Java Engine (EJE).

Oracle Application Server 11*g*

Oracle Application Server 11*g* maintains and executes all your application logic, including Enterprise JavaBeans, through its own built-in JVM. Oracle Application Server 11*g* uses the WebLogic server to execute servlets and JSPs. Oracle Application Server 11*g* Enterprise Manager is the tool that is used to manage and distribute applications for ease of use.

Java Software Development Kit

Sun Java SE (known as JDK and Java SDK) provides:

- Compiler (javac)
- Core class library
  - `rt.jar`
- Debugger (`jdb`)
- Bytecode interpreter: JVM (java)
- Documentation generator (javadoc)
- Java Archive utility (`jar`)
- Others

Java SE

Java Software Development Kit

Sun provides Java Platform, Standard Edition (Java SE), which is also known as the Java Software Development Kit (Java SDK) or the Java Development Kit (JDK). The components that are provided by Java SE include the following:

- The Java compiler (javac) compiles Java source code into Java bytecodes.
- The Java bytecode interpreter (java) is the engine that runs Java applications.
- The program that generates documentation in HTML from Java source code comments is javadoc.

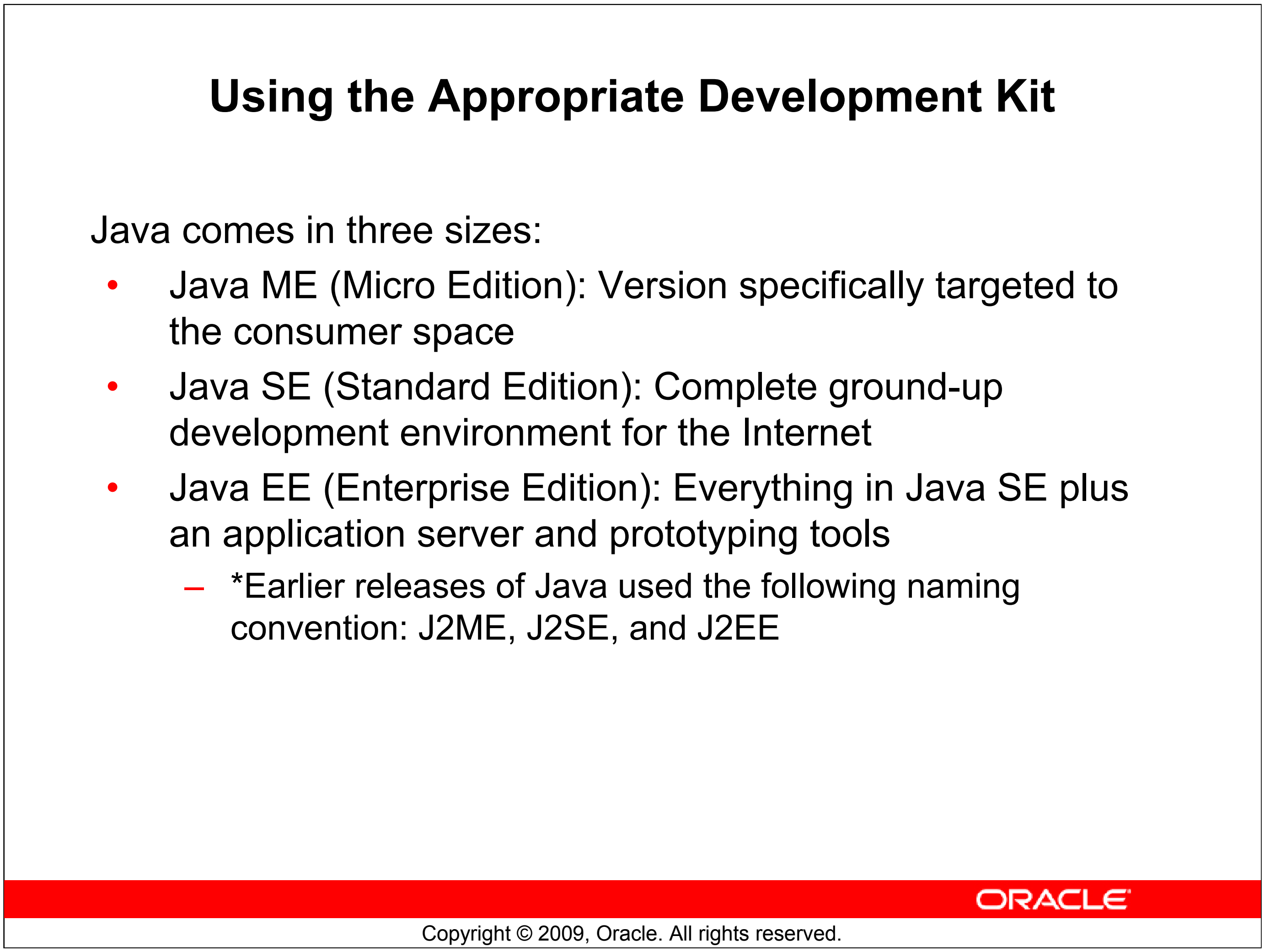
Core Class Library

Java SE provides core Java classes in the following class library files:

- `rt.jar` located in the `jdk_home\jre\lib` for Java SDK 1.2.*x* or later

Other Java SE Tools

- `jdb` is the Java class debugger. It is similar to the `dbx` and `gdb` debuggers in UNIX.
- `jar` is used to create Java Archive (JAR) files, which are zipped Java programs.
- `javah` is used to generate C files for native methods.
- `javakey` supports the generation of certification keys for trusted Java code.
- `javap` is used to disassemble Java bytecodes into human-readable format.
- `native2ascii` converts Java source code to Latin 1 characters.
- `serialver` is used to generate version numbers for classes.



Using the Appropriate Development Kit

Java comes in three sizes:

- Java ME (Micro Edition): Version specifically targeted to the consumer space
- Java SE (Standard Edition): Complete ground-up development environment for the Internet
- Java EE (Enterprise Edition): Everything in Java SE plus an application server and prototyping tools
  - *Earlier releases of Java used the following naming convention: J2ME, J2SE, and J2EE

A Size for Every Need

Java Micro Edition

The technology that Java Micro Edition (Java ME) uses covers a range from extremely tiny commodities (such as smart cards and pagers) all the way up to the set-top television box, which is an appliance that is almost as powerful as a computer. Like the other editions, the Java ME platform maintains the qualities for which Java technology has become famous.

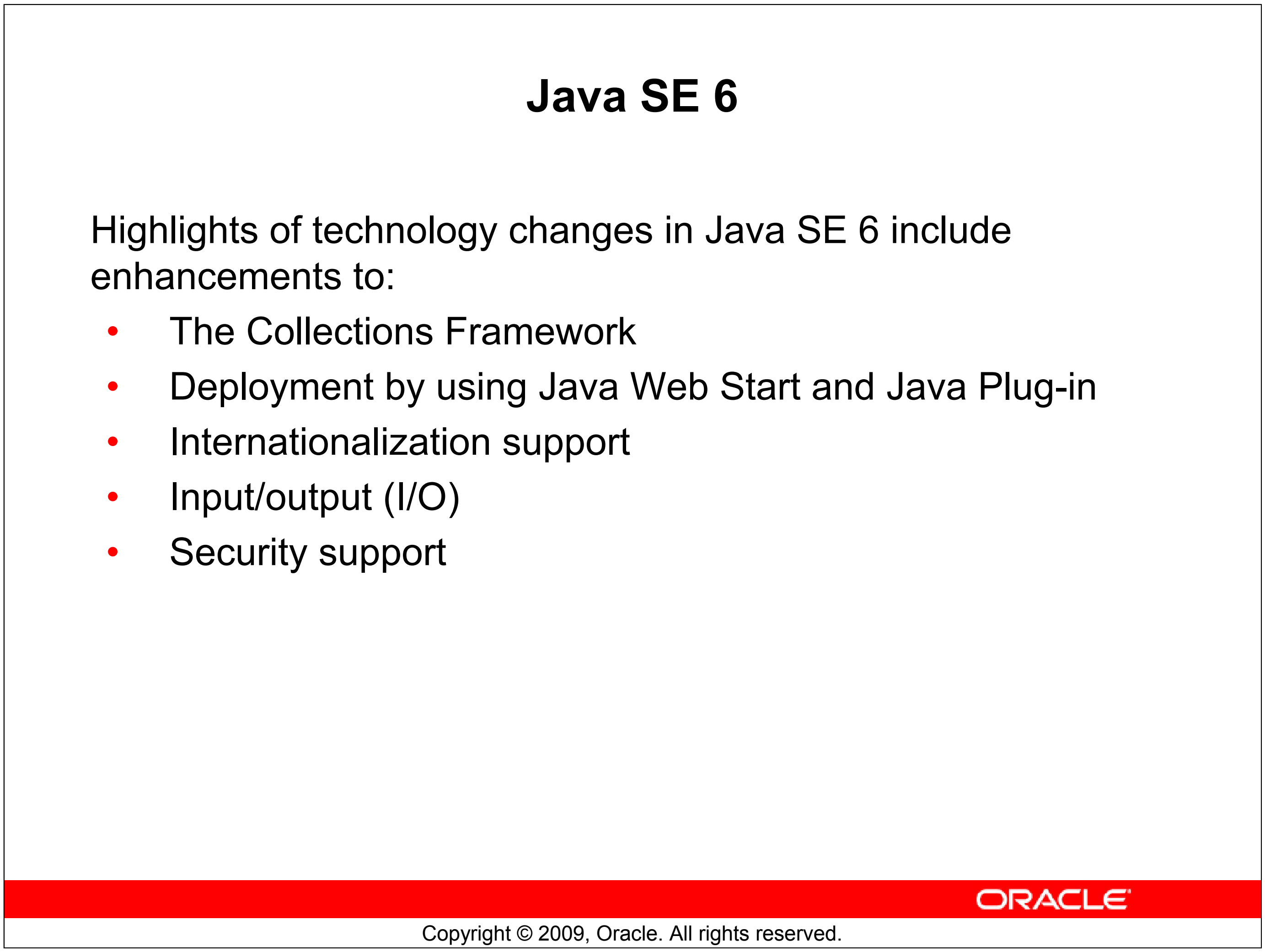
Java Platform, Standard Edition

Java SE technology has revolutionized computing with the introduction of a stable, secure, and feature-complete development and deployment environment that is designed from the ground up for the Web. It provides cross-platform compatibility, safe network delivery, and smart card–to–supercomputer scalability. It provides software developers with a platform for rapid application development.

Java Platform, Enterprise Edition

The Java EE platform is a guide for implementations in the application-server marketplace. The Java EE SDK includes a Java EE application server and various tools to help developers prototype Java EE applications.

There are many versions of Java SE. The latest version is Java SE 6.0. Sun releases each version of Java SE with a *Java Development Toolkit* (JDK). For Java SE 6.0 the Java Development Toolkit is called JDK 6.



Java SE 6

Highlights of technology changes in Java SE 6 include enhancements to:

- The Collections Framework
- Deployment by using Java Web Start and Java Plug-in
- Internationalization support
- Input/output (I/O)
- Security support

Java SE 6

Java Platform Standard Edition 6 is a major feature release.

Collections Framework

Some new collection interfaces and some new concrete implementation classes have been provided. In addition, some existing classes have been retrofitted to implement new interfaces.

Deployment

The caching mechanism and download engine have been redesigned and consolidated between Java Web Start and Java Plug-in.

I/O

One new class, Console, containing methods to access a character-based console device, has been added.

You can learn more about the enhancements contained in Java SE 6 at: http://java.sun.com/javase/6/webnotes/features.html.

Integrated Development Environment

ORACLE 11*g* JDEVELOPER

Development

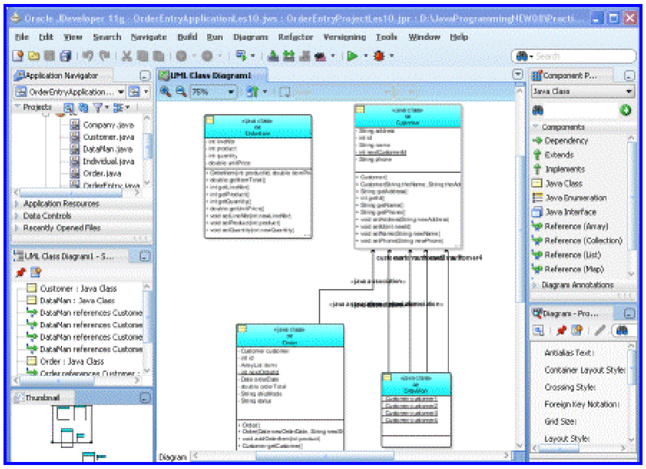
UML

ADF

JSF

EJB

XML



Debug

Database

HTML

Deployment

Source control

Web services

SOA

WebCenter

Integrated Development Environment

The add-in API architecture of the Oracle JDeveloper integrated development environment (IDE) means that all the tool components (for example, navigator, editor, and modeler) share memory models and event systems. In this way, an update in one tool is communicated to another tool so that the latter can refresh its image or take other appropriate actions.

In Oracle 11*g*, the JDeveloper IDE was developed in pure Java. Synchronization between model and code can be set so that you can decide to work using one or the other user interface.

Customizable Environment

You can customize the JDeveloper IDE and arrange its look to better suit your project needs and programming style. To modify the JDeveloper IDE to your individual taste, you can:

- Change its look and feel
- Create and manipulate custom navigators
- Customize the Component Palette
- Customize the IDE environment
- Select JDeveloper’s embedded Java EE server
- Arrange the windows in the IDE

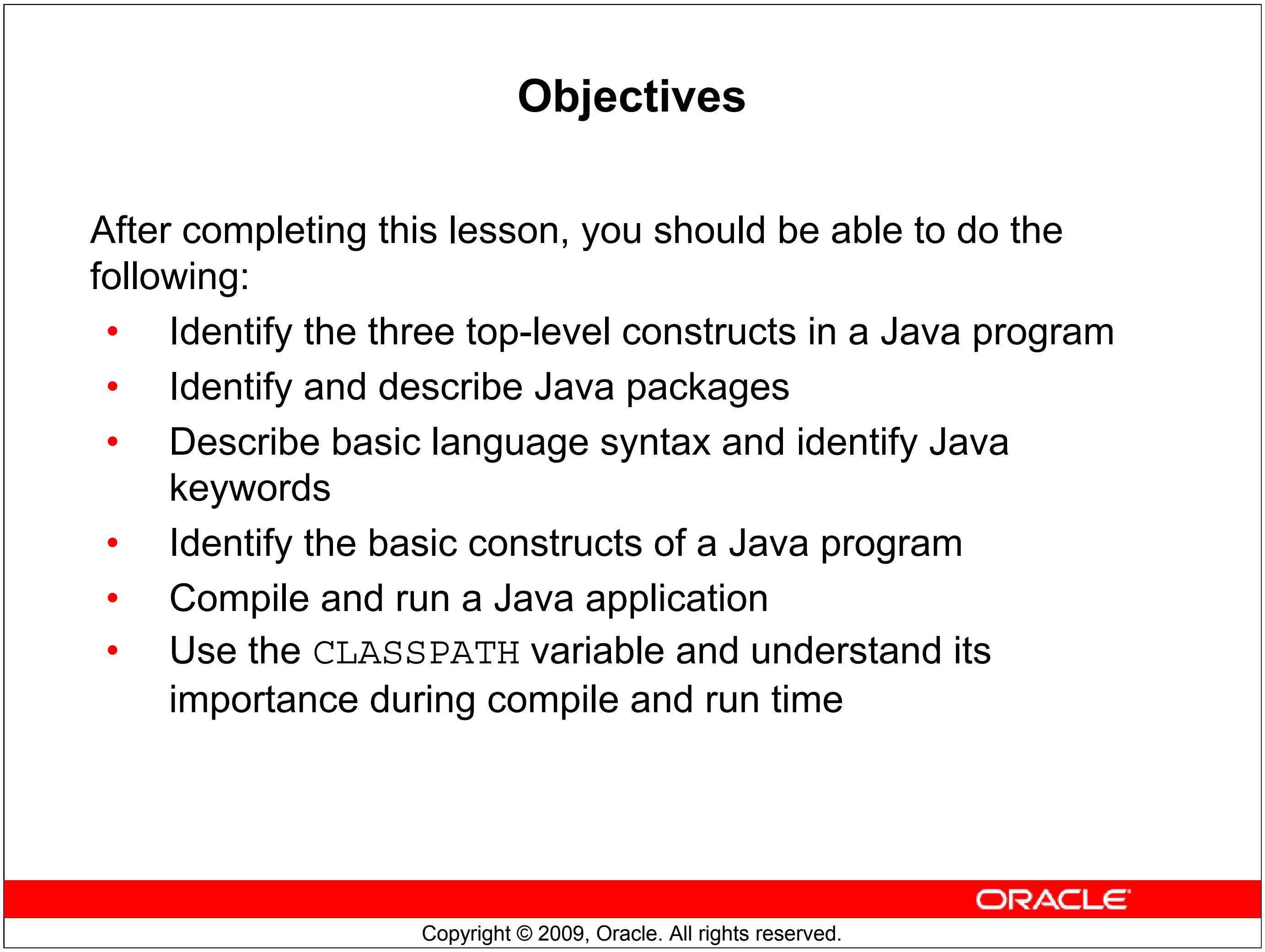
Summary

In this lesson, you should have learned how to:

- Identify the key elements of Java
- Describe the key benefits of Java
- List the key components of the Java SE Java Development Kit (known as JDK or SDK)
- List the Java deployment options
- Recognize how the JDeveloper IDE supports the development of Java applications

2

Basic Java Syntax and Coding Conventions



Objectives

After completing this lesson, you should be able to do the following:

- Identify the three top-level constructs in a Java program
- Identify and describe Java packages
- Describe basic language syntax and identify Java keywords
- Identify the basic constructs of a Java program
- Compile and run a Java application
- Use the `CLASSPATH` variable and understand its importance during compile and run time

Lesson Objectives

Before you can start writing Java code, you need to know the basic constructs of the Java language: conventions, standards (for example, capitalization, spacing, and so on), and the way to compile and run applications. This lesson introduces you to these constructs so that you can recognize and manipulate them with or without using Oracle JDeveloper.

Overview

Java Components

The Java environment comprises a run-time engine, documentation tools, debugging utilities, and predefined classes and methods that are designed to decrease development time.

Conventions

When programming in Java, you must use established naming, capitalization, indenting, and commenting conventions.

Classes, Objects, and Methods

In Java, almost everything is an object. Objects are created from blueprints called classes. The objects contain attributes (data) that can be accessed by functions that are contained within the object. Functions that act on the data are called methods.

Using Javadoc

Javadoc is a facility that is provided within Java SE and produces HTML documentation from your program. It reads the source code and parses specially formatted and positioned comments into documentation.

Compiling and Running Java

Java is an interpretive language, which means that the code is interpreted to machine code only at run time. This is what makes the "write once, run anywhere" (WORA) concept work. There are several steps in the process of interpreting program source code into a running program. Java code is first compiled into bytecodes by the Java compiler. Bytecodes are an interpretable, intermediate representation of the Java program. The resulting bytecodes are interpreted and converted into machine-specific instructions by the Java Virtual Machine (JVM) at run time.

Security Concerns

Using `jad.exe`, you can make the code so confused or opaque as to be difficult to perceive or understand. Obfuscation of your code can make field debugging more difficult. For example, stack traces are often very useful in isolating bugs. After compression or obfuscation by one of these tools, however, the stack trace may no longer contain the original method names. In general, refrain from using obfuscation unless you really want to make it difficult to modify your code.

Toolkit Components

Java SE and Java EE provide:

- Compiler
- Bytecode interpreter
- Documentation generator

Java SE

Toolkit Components

Sun Java SE Components

- The Java compiler is javac; it compiles Java source code into Java bytecodes.
- The Java bytecode interpreter is the engine that runs Java applications.
- The program that generates documentation in HTML from Java source code comments is Javadoc.

Other Java SE Tools

- `jdb`: Used as a Java class debugger; similar to the `dbx` or `gdb` debuggers on UNIX
- `jar`: Used to create Java Archive (JAR) files, which are zipped Java programs
- `javah`: Used to generate C files for native methods
- `javakey`: Provides support for generating certification keys for trusted Java code
- `javap`: Used to disassemble Java bytecodes into human-readable format
- `native2ascii`: Used to convert Java source code to Latin 1 characters

Note: Java Standard Edition provides a complete environment for application development and deployment on desktops and servers. Java Enterprise Edition builds on the foundation of Java SE and provides Web services, component models, management, and communications APIs.

Java Packages

Java SE and Java EE provide standard packages for:

- Language
- Windowing
- Input/output
- Network communication
- Other packages

Java SE

Java Packages

Java includes a series of classes that are organized into packages, depending on functional groups. For example, there is a set of classes that help create and use network connections; these classes are contained in the `java.net` package. The basic package for Java is named `classes.zip` in version 1.1.*x* and `rt.jar` in versions 1.2.*x* and later.

Standard Java Packages

These packages contain the classes that form the foundation for all Java applications.

Built-in Classes that Are Distributed with Java

- `java.lang`: Basic language functions
- `javax.swing`: Parent package to all Swing-related packages
- `java.util`: Facility supporting interfacing, implementing, sorting, and searching on collections
- `java.awt`: Utility to managing layout, handling events, and rendering graphics for AWT
- `java.io`: General interface for all I/O operations

Documenting Using Java SE

Java SE and Java EE provide documentation support for:

- Comments
  - Implementation
  - Documentation
- Documentation generator
- Annotations complement javadoc tags

Java SE

Documentation

There are two types of documentation methods in Java that you can specify in your source code. One is for the internal documentation, and the other is for external documentation.

Comments

- Implementation comments are included in the source code. They are useful for programmers who are examining the code. When writing implementation comments, you must explain the code and any special processes that may need detailed explanation. Use the following conventions:
  - `//` to start comments up to the end of the line
  - `/*` to start comments across multiple lines, ending with `*/`
- Documentation comments are created using Javadoc. Javadoc is a utility that is provided with Java SE and creates an HTML document from the documentation comments in Java source code. You can use the Javadoc utility to document classes and methods so that they can be better understood when used by other programmers. Use the following conventions:
  - `/**` to start documentation comments across multiple lines, ending with `*/`

Documentation Generator

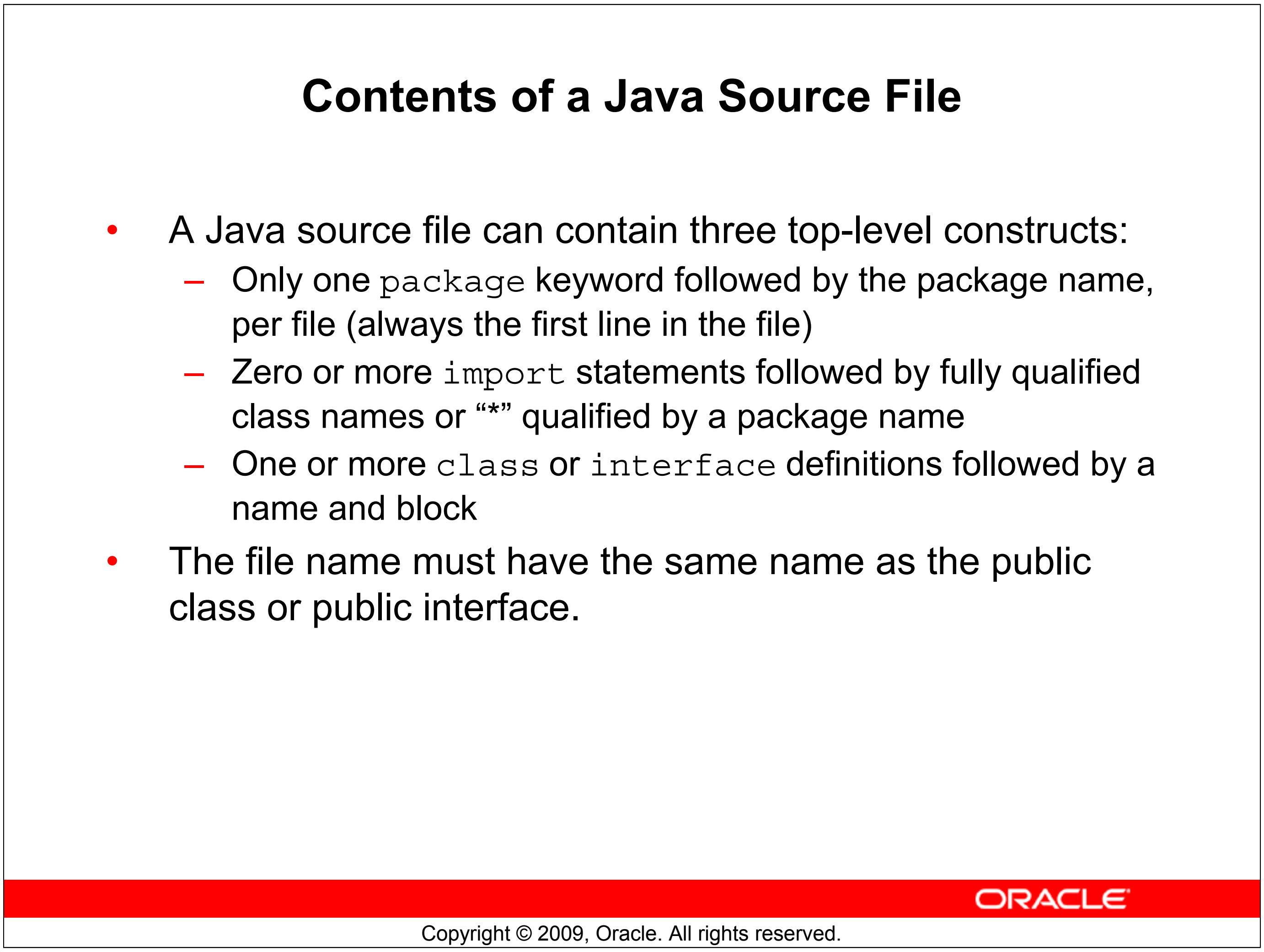
Javadoc is a documentation generator that is part of Java SE.

Documentation (continued)

Annotations

Annotations are a language feature that was introduced in Java SE 5.0. They complement javadoc tags. Developers can avoid writing boilerplate code in many circumstances by generating it from annotations in the source code. This leads to a “declarative” programming style in which the programmer says what should be done and tools create the code to do it. Annotations also eliminate the need for maintaining “side files” that must be kept up-to-date with changes in source files. Instead, the information can be maintained *in* the source file.

In general, if the markup is intended to affect or produce documentation, it should probably be a javadoc tag; otherwise, it should be an annotation.



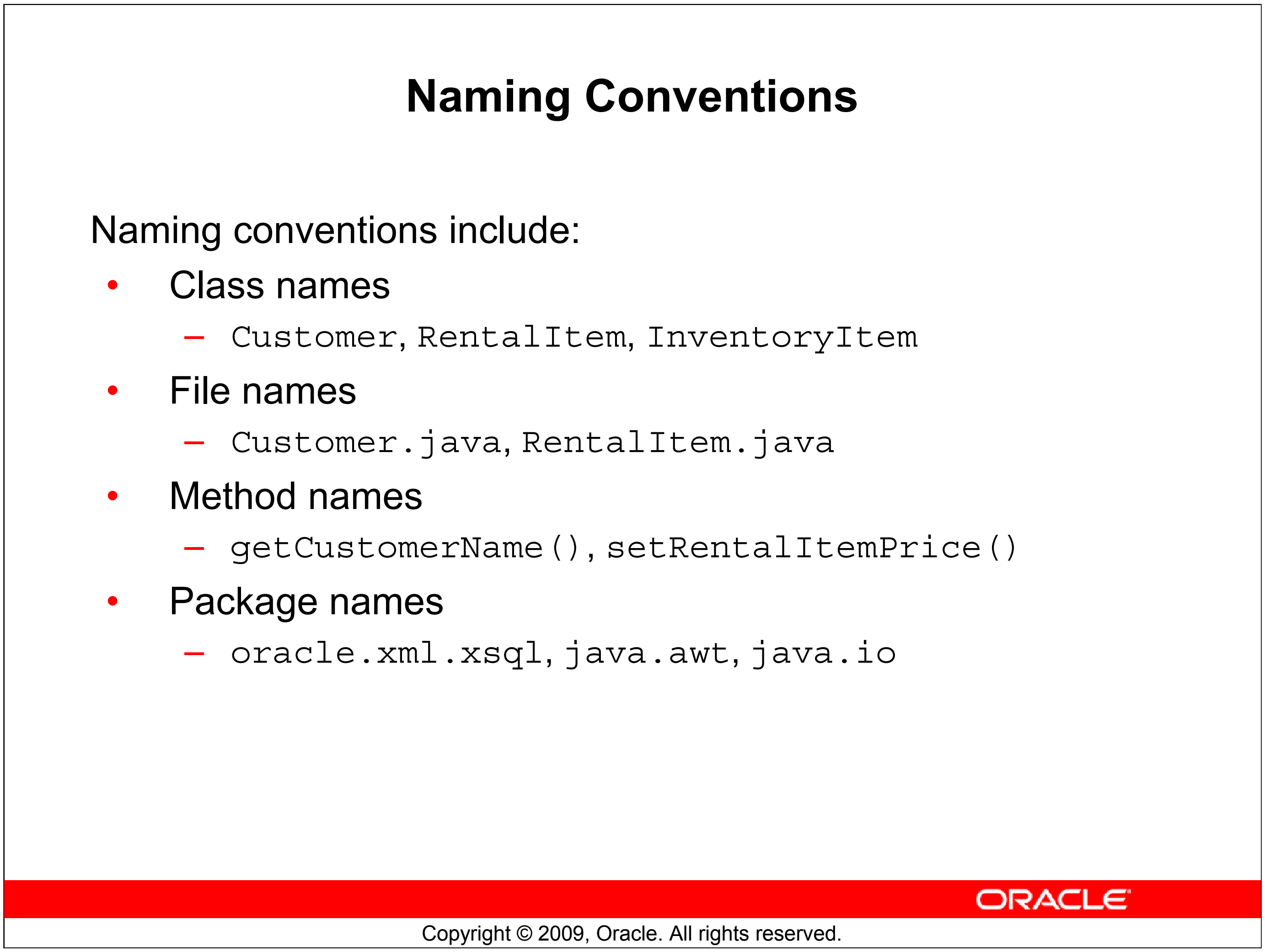
Contents of a Java Source File

- A Java source file can contain three top-level constructs:
  – Only one `package` keyword followed by the package name, per file (always the first line in the file)
  – Zero or more `import` statements followed by fully qualified class names or “*” qualified by a package name
  – One or more `class` or `interface` definitions followed by a name and block
- The file name must have the same name as the public class or public interface.

ORACLE

RentalItem.java

```
package practice16; // collection of classes of similar
functionality
import java.util.*;   // import for Date class
public class RentalItem {
  private InventoryItem inventoryItem;
  private Date dueDate;
  private Date dateReturned;
  public RentalItem(int aInventoryItemId) {
    try {
      inventoryItem =
DataMan.fetchInventoryItemById(aInventoryItemId);
    }
    catch (ItemNotFound e2) {
      System.out.println("Invalid Item ID");
    }
    dueDate = calcDueDate();
  } // end constructor
```



Naming Conventions

Naming conventions include:

- Class names
  - `Customer`, `RentalItem`, `InventoryItem`
- File names
  - `Customer.java`, `RentalItem.java`
- Method names
  - `getCustomerName()`, `setRentalItemPrice()`
- Package names
  - `oracle.xml.xsql`, `java.awt`, `java.io`

Naming Conventions

File Names

Java source code is stored in files with the `.java` extension. Use the name of the class held within the file as the file name. Remember that Java is case-sensitive, and the names must match exactly, including case. Compiled Java code, or bytecode, is stored in files with the `.class` extension.

The name of the Java source file *must* be the name of the *public* class in the file; otherwise, the code does not compile. You may have source code for more than one class in the file, but *only one* can be public, and the file name must be the same name as the public class.

Class Names

Use descriptive nouns or noun phrases for class names. Capitalize the first letter of each word in the class name, including the first word (for example, `MyFirstClassName`).

Naming Conventions (continued)

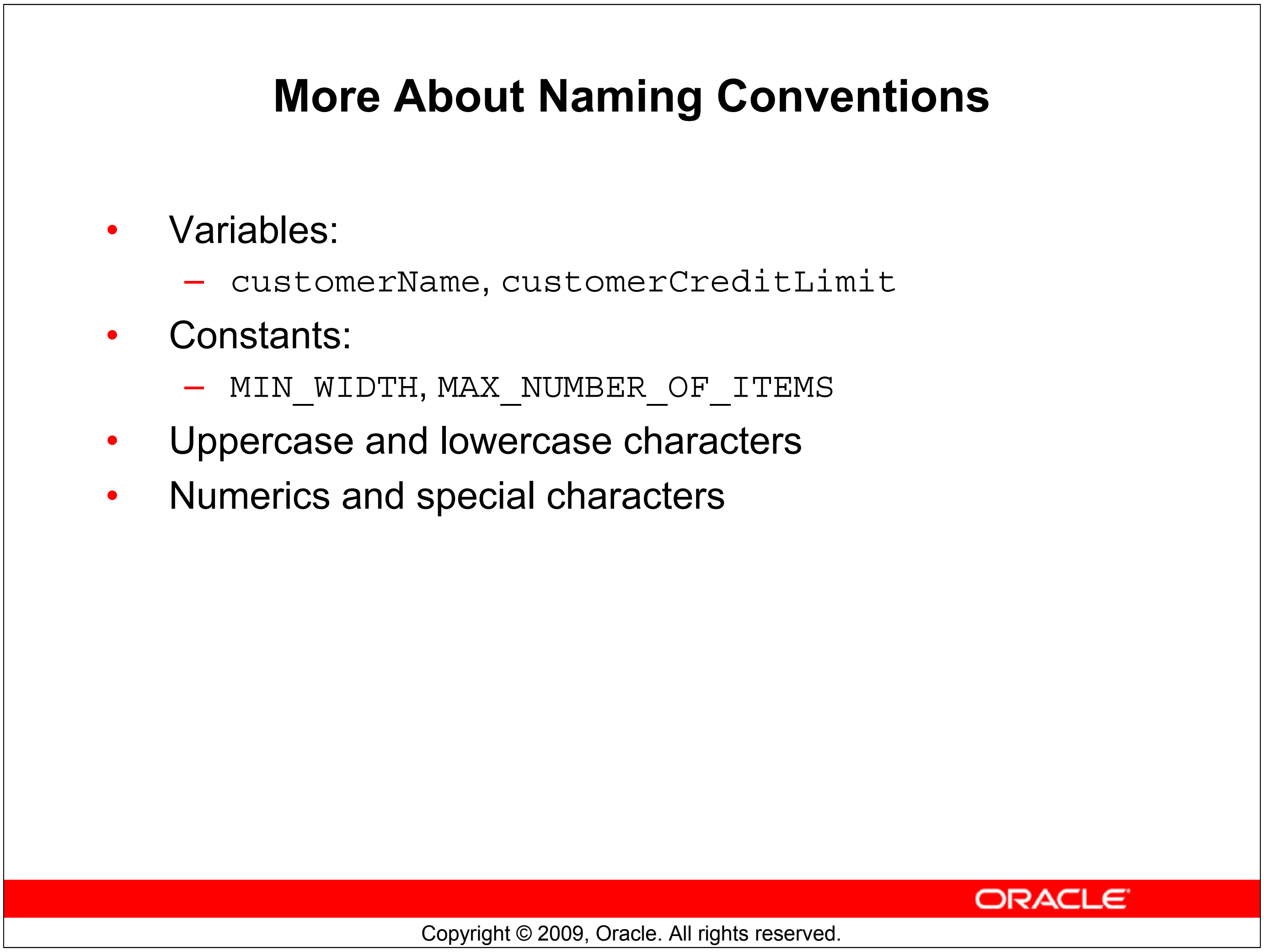
Method Names

Use verbs or verb clauses for method names. Lowercase the first letter of the method name, and capitalize the first letter of each internal word, as in the following example:

```
getSomeInformation()
```

Packages

The Java documentation states that package names must nearly always be in lowercase and must resemble a “domain name” in reverse. For example, when Oracle Corporation develops class libraries, package names begin with “`oracle`.” followed by a descriptive name. The naming convention ensures that a package name is unique to avoid conflicts when using APIs from multiple vendors in the same application.



More About Naming Conventions

- Variables:
  - `customerName`, `customerCreditLimit`
- Constants:
  - `MIN_WIDTH`, `MAX_NUMBER_OF_ITEMS`
- Uppercase and lowercase characters
- Numerics and special characters

Naming Conventions (continued)

Variables

Use short, meaningful names for variables. Use mixed-case letters with the first letter lowercase, and begin all internal words with uppercase letters (for example, `int squareFootage`). Choose names that indicate the intended use of the variable. Avoid using single-character variable names except for temporary variables. Common names for temporary variables are *c*, *d*, and *e* for character fields and *i*, *j*, *k*, *m*, and *n* for integers.

```
String customerName;        // string variable
int customerCreditLimit;    // integer variable
```

Constants

Declare constants with descriptive names written in all uppercase letters. Separate internal words with underscores (for example, `int MIN_WIDTH`).

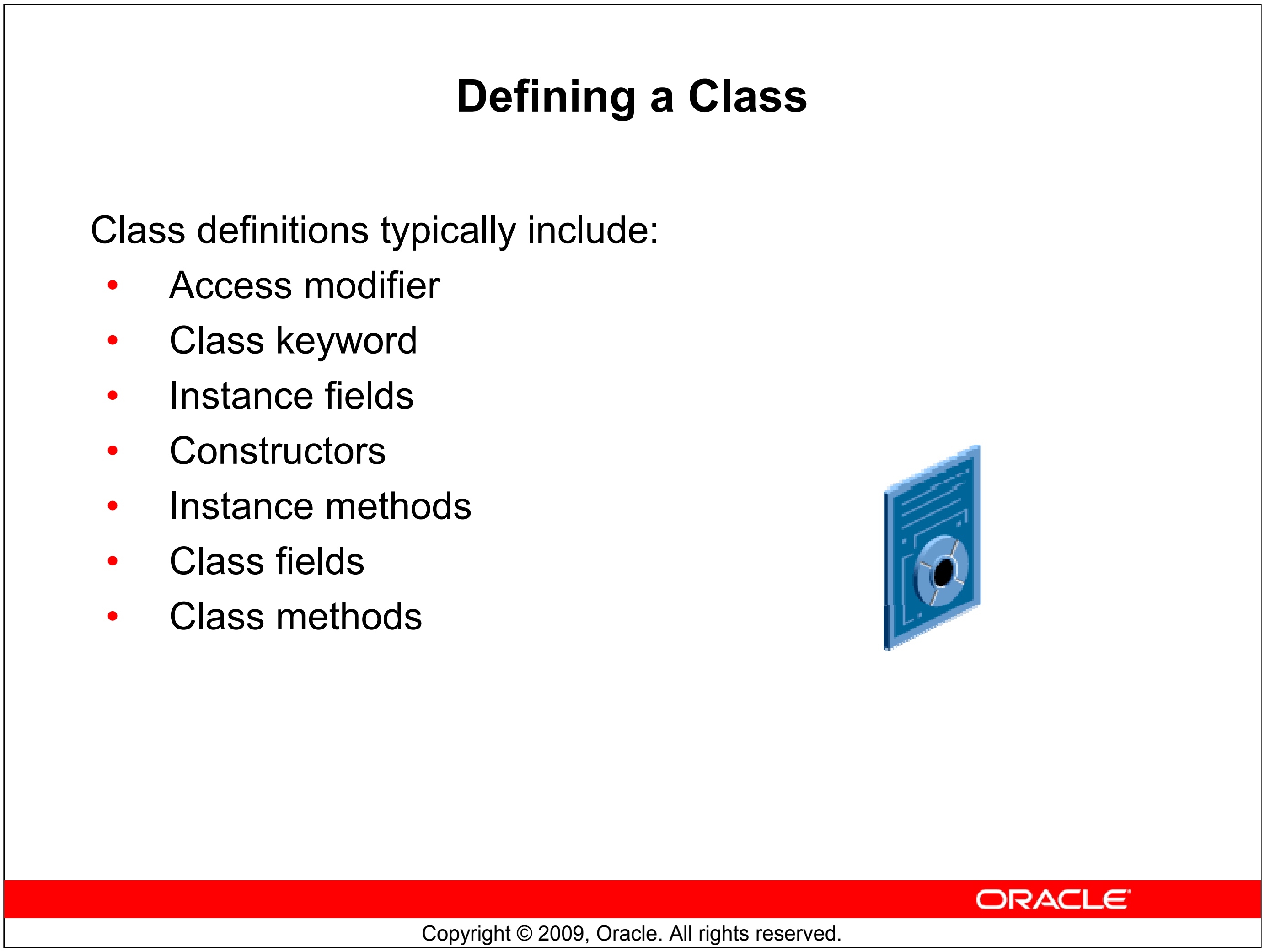
Naming Conventions (continued)

Uppercase and Lowercase

Java is case-sensitive. You must adopt and follow a strict capitalization scheme. The scheme that is presented in the slide is a generally accepted practice.

Numerics and Special Characters

In addition to uppercase and lowercase letters, you can use numerals, underscores, and dollar signs. The only syntactic restriction is that identifiers must not begin with a number. This rule prevents them from being confused with numeric literals. Underscores are not generally used except for private and local variables.



Defining a Class

Class definitions typically include:

- Access modifier
- Class keyword
- Instance fields
- Constructors
- Instance methods
- Class fields
- Class methods

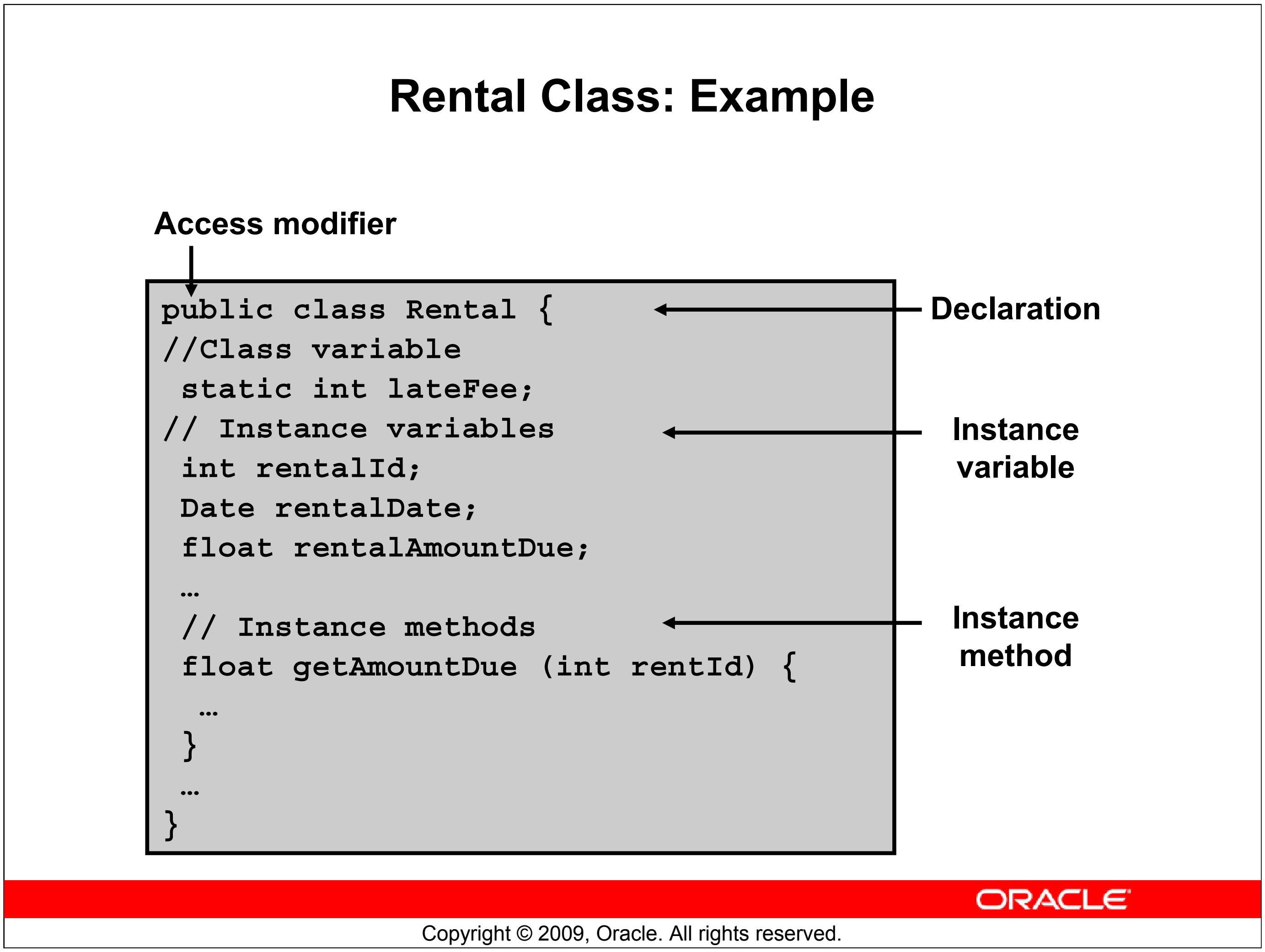
Defining a Class

A class is an encapsulated collection of data and the methods to operate on that data. A class definition—together with the data and methods—serves as a blueprint for the creation of new objects of the class.

A class definition typically consists of:

- **Access modifier:** Specifies the availability of the class from other classes
- **Class keyword:** Indicates to Java that the following code block defines a class
- **Instance fields:** Contain variables and constants that are used by objects of the class
- **Constructors:** Are methods having the same name as the class, which are used to control the initial state of any class object that is created
- **Instance methods:** Define the functions that can act upon data in this class
- **Class fields:** Contain variables and constants that belong to the class and are shared by all objects of that class
- **Class methods:** Are methods that are used to control the values of class fields

The order of the fields, constructors, and methods does not matter in Java. However, consistent ordering of the parts of a Java program makes your code easier to use, debug, and share. The order shown in the slide is generally accepted.



Rental Class: Example

Access modifier

```
public class Rental {
//Class variable
 static int lateFee;
// Instance variables
 int rentalId;
 Date rentalDate;
 float rentalAmountDue;
 …
 // Instance methods
 float getAmountDue (int rentId) {
  …
 }
 …
}
```

Declaration

Instance variable

Instance method

ORACLE

Rental Class: Example

This slide shows the syntax of a class definition. Every Java program needs at least one class definition, even if the class contains only a static `main()` method.

First Line

The class definition usually starts with the `public` access modifier. If you omit the `public` modifier, the class is visible only to other classes in the same package. The access modifier is followed by the `class` keyword, followed by the class body enclosed in braces.

Instance Variables

The class body contains declarations of instance variables and instance methods. Instance variables constitute the state of an object. Instance variables are usually declared `private`. If no access level is specified, they can then be accessed by any class in the same package. The access modifier is followed by the class keyword, followed by the class name and then the class body enclosed within braces.

Rental Class: Example (continued)

Instance Methods

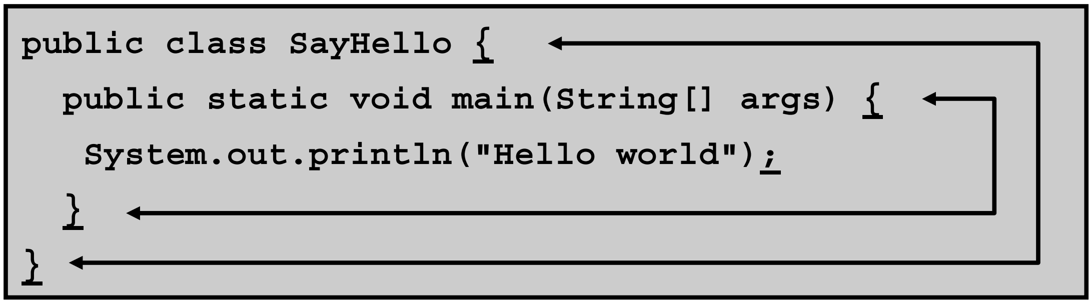
Instance methods define the operations that can be performed on objects of this class type. Each instance method has a name, an optional list of arguments, and a return type. If no access level is specified, instance methods can be accessed by any class in the same package.

Class Variables and Class Methods

Class fields and class methods are defined by using the static keyword.

Creating Code Blocks

- Enclose all class declarations.
- Enclose all method declarations.
- Group other related code segments.



```
public class SayHello {
  public static void main(String[] args) {
   System.out.println("Hello world");
  }
}
```

Creating Code Blocks

Enclose class declarations within braces. The class declaration contains variables, constants, and methods. The class body must begin on a new line following the class declaration and the opening brace. Indicate the end of the class declaration by a single closing brace on a new line. There is another coding convention that puts the opening brace on the next line aligned with the closing brace from the previous statement.

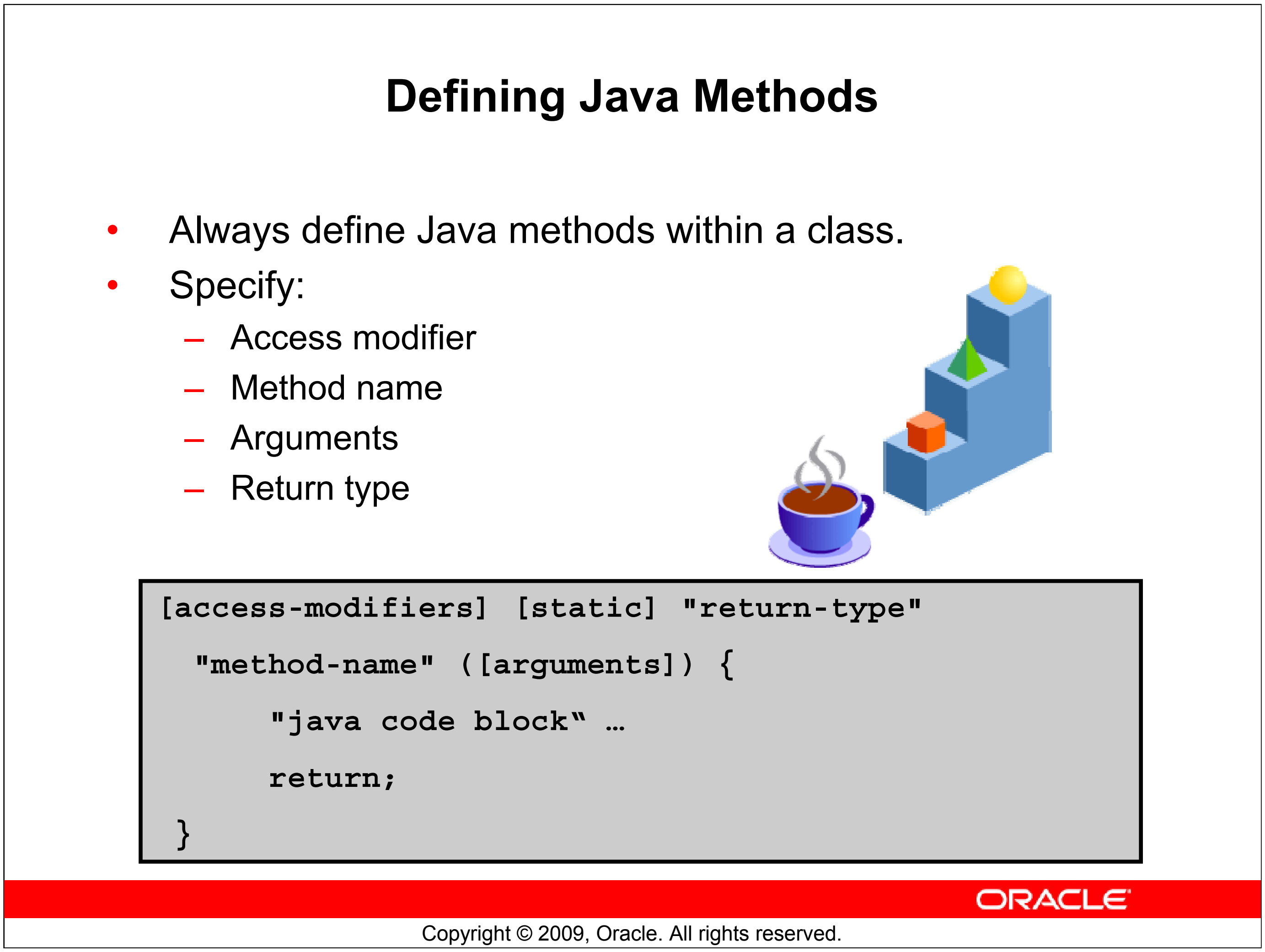
Method declarations follow the same form, with the opening brace on the same line as the method definition and the closing brace on a new line. All statements and declarations within the braces are part of the method.

You can also group code segments within a method declaration by using braces. Grouping enhances the readability of your code. If your code is easy to read and follow, it will be easier to debug and reuse.

Variables that are defined within a code block are available during the execution of that code block and are discarded at the completion of execution.

The `main()` Method

The interpreter looks for a specific method to start the program; that method is called `main()`. It is simply the starting place for the interpreter. Only one class needs to have a main method for the interpreter to begin. You need a main method only for an application. If you are writing an applet, the main method is not needed because the browser has its own way to start or bootstrap the program.



Defining Java Methods

- Always define Java methods within a class.
- Specify:
  - Access modifier
  - Method name
  - Arguments
  - Return type

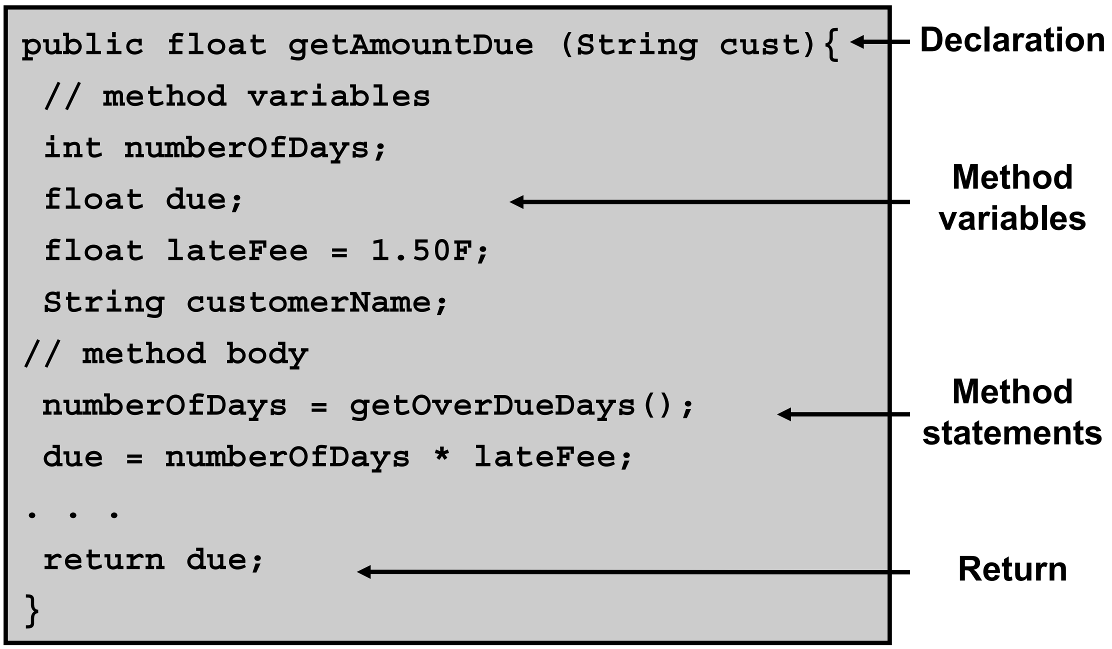
```
[access-modifiers] [static] "return-type"
 "method-name" ([arguments]) {
     "java code block‟ …
     return;
}
```

Defining Java Methods

When you define a class for an object-oriented program, you implement all the behavior of that class in one or more methods. A Java method is equivalent to a function, procedure, or subroutine in other languages, except that it must be defined within a class definition. Methods consist of the following:

- **Access modifier:** You can specify a method as either public, private, or protected. Public methods can be known and used by external users; whereas private methods can be seen or used only by methods within the class. Protected methods are accessible from any class that extends or inherits from the class.
- **Method name:** The method name begins with a lowercase letter, and the beginning letters of each subsequent word are in uppercase letters.
- **Arguments:** The arguments that are listed for a method are those parameters that are required for the method to perform its function.
- **Return type:** A return type is required for all method definitions. There is no default return type in Java. The return type specifies the object type that is returned when the method has completed its task. It can be an integer, a string, any defined object type, or void.

Example of a Method



```
public float getAmountDue (String cust){
 // method variables
 int numberOfDays;
 float due;
 float lateFee = 1.50F;
 String customerName;
// method body
 numberOfDays = getOverDueDays();
 due = numberOfDays * lateFee;
. . .
 return due;
}
```

Declaration

Method variables

Method statements

Return

Example of a Method

Declaration

In this example, the defined method returns a float primitive type. The method name is declared as `getAmountDue`. The next item is the declaration of the expected parameter list. The parameter list consists of the argument data type and argument name.

Method Variables

The next set of statements defines any method-specific variables. These variables are used only during the execution of the method and are discarded when control is passed back to the object that called this method. The float data type is used for real numbers (numbers with decimal points).

Method Statements

The executable body of the method comes next. This section contains the Java statements that are used to act upon data. It can include conditional statements, data manipulation operations, or any other valid Java statements.

Return

The return statement accomplishes two things. The return causes the execution to branch back to the caller of the method, and then it passes back the specified value, if there is one.

Declaring Variables

- You can declare variables anywhere in a class block and outside any method.
- You must declare variables before they are used inside a method.
- It is typical to declare variables at the beginning of a class block.
- The scope or visibility of variables is determined in the code block.
- You must initialize method variables before using them.
- Class and instance variables are automatically initialized.

Declaring Variables

Declaration

Java requires that variables be declared before they can be accessed. Declare variables and constants by placing each statement on a separate line so that the code is clear and easy to understand. You can declare multiple variables of the same type within one statement.

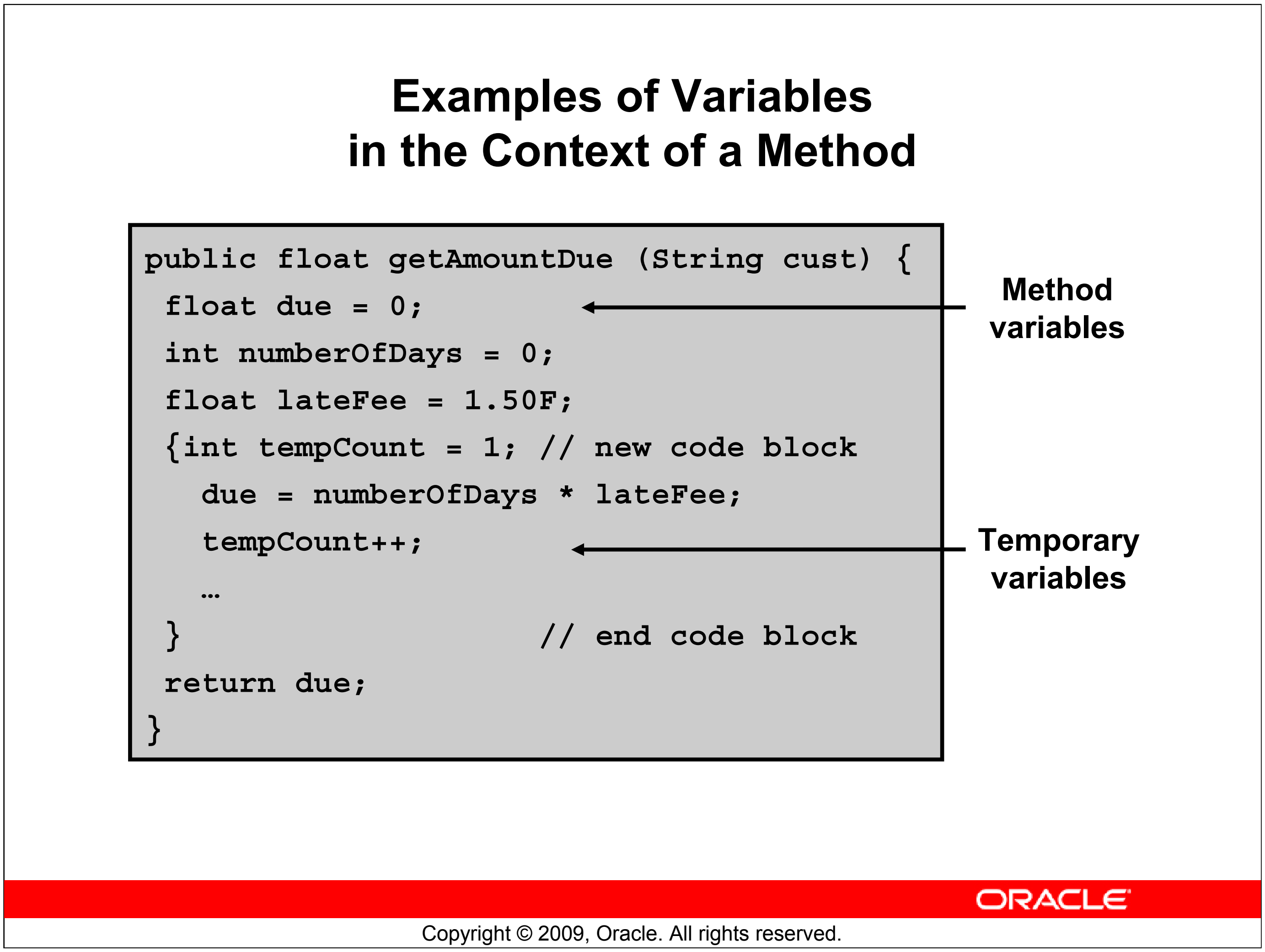
Scope

If variables or constants are required only within a code block, you declare them at the top of the code block. Variables that are defined within a code block are discarded when the execution of the code block is complete. This is useful for temporary variables and for those variables that are needed to hold specific information during a calculation or process. After the process is complete, they are no longer needed.

Initialization

Whenever possible, you should initialize variables at declaration. This provides some insight into the use or purpose of the variable. The only reason for not initializing at declaration is if the value is dependent on a calculation that has not yet been performed.

When you declare primitive instances or class variables, they are automatically initialized to a default value depending on their type.



Examples of Variables in the Context of a Method

```
public float getAmountDue (String cust) {
 float due = 0;
 int numberOfDays = 0;
 float lateFee = 1.50F;
 {int tempCount = 1; // new code block
   due = numberOfDays * lateFee;
   tempCount++;
   …
 }                    // end code block
 return due;
}
```

Method variables

Temporary variables

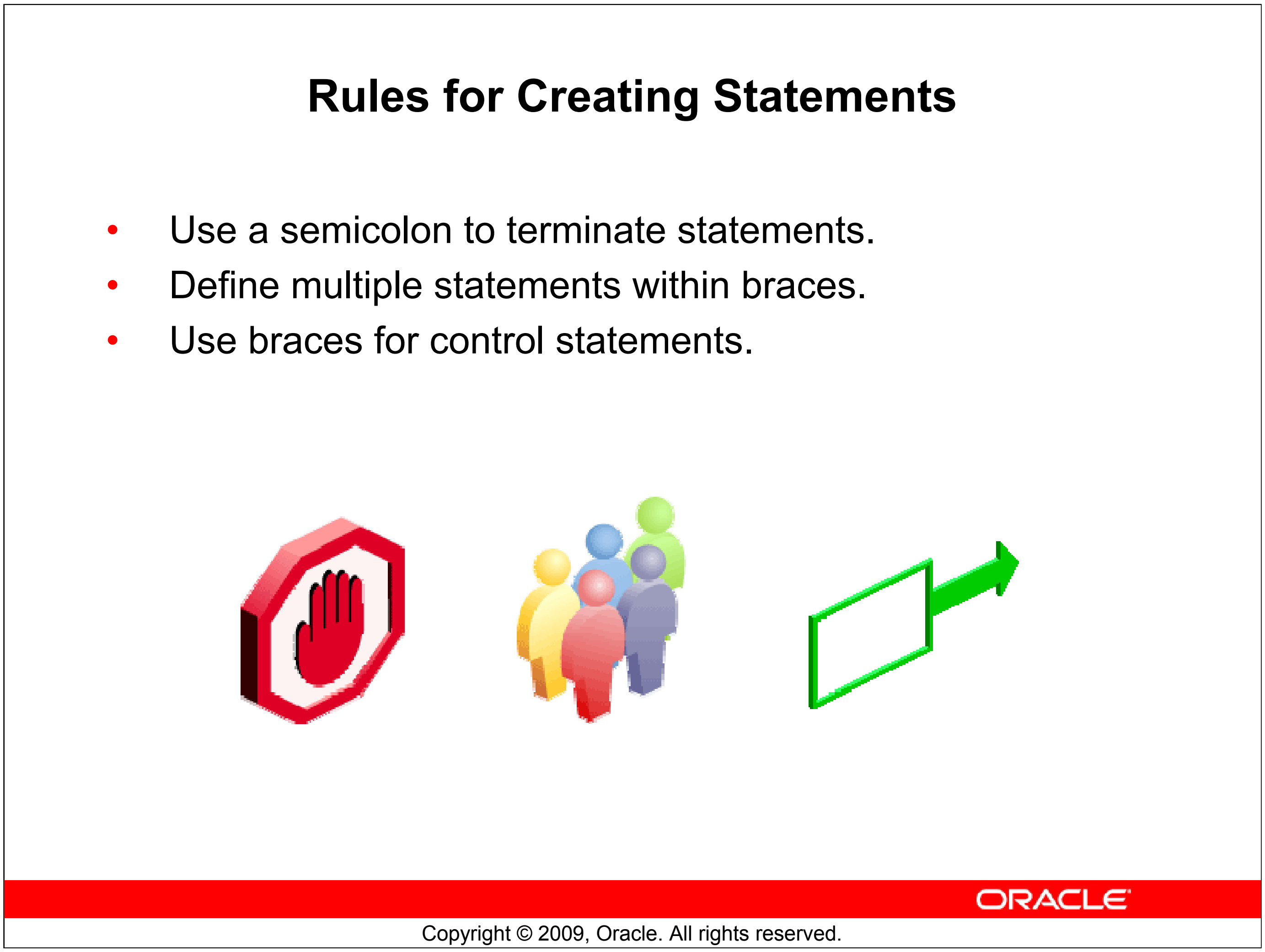
Examples of Variables in the Context of a Method

Method Variables

In this example, the `numberOfDays` and `lateFee` variables are defined at the beginning of the method. Both of these variables are method variables—that is, they are used while the method is being executed and are discarded at the return of execution control to the caller.

Temporary Variables

The `tempCount` variable is defined within a subblock of the `getAmountDue` method. The `tempCount` variable is available only during the execution of this block of code. When the block of code is complete, the variable is discarded whether or not the method maintains execution control.



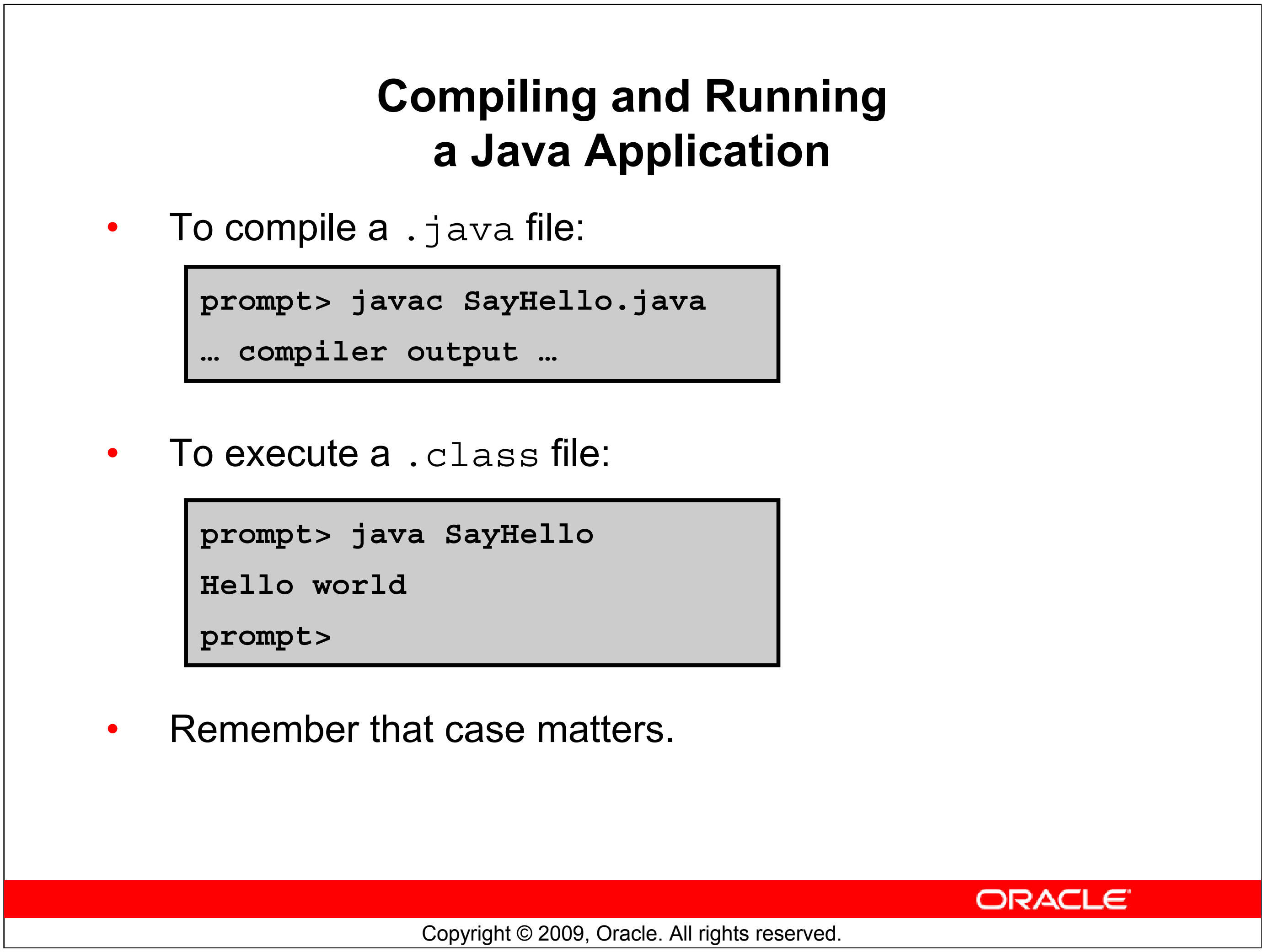
Rules for Creating Statements

- Use a semicolon to terminate statements.
- Define multiple statements within braces.
- Use braces for control statements.

Rules for Creating Statements

Java statements cause an action to occur, such as setting a value, sending output, reading a database, and so on. They always end with a semicolon. A compound statement is a list of related statements that is contained within a set of braces. Indent the statements one level below the enclosing statement for clarity. Place the opening brace on the line that begins the compound statement. Place the closing brace on a separate line, indented to the correct level. Use braces around all statements when they are part of a control (`if-else`) structure.

```
public float getAmountDue (String cust) {
    float due = 0;
    int numberOfDays = 0;
    float lateFee = 1.50F;
  If (…) { {int tempCount = 1; // new code block
  due = numberOfDays * lateFee;
  tempCount++;   //nested compound statement
     …  }
  // end code block
  } //end if
   return due;
  }
```



Compiling and Running a Java Application

- To compile a `.java` file:

```
prompt> javac SayHello.java
… compiler output …
```

- To execute a `.class` file:

```
prompt> java SayHello
Hello world
prompt>
```

- Remember that case matters.

Java Development Kit (JDK) Tools

Java SE includes `javac.exe` and `java.exe`, two executables for compiling and executing a Java program.

Compiling Java Code

Run `javac.exe` from the command prompt to compile `.java` files into `.class` files. For example, to compile `SayHello.java` into a bytecodes file named `SayHello.class`, enter the following at the command prompt:

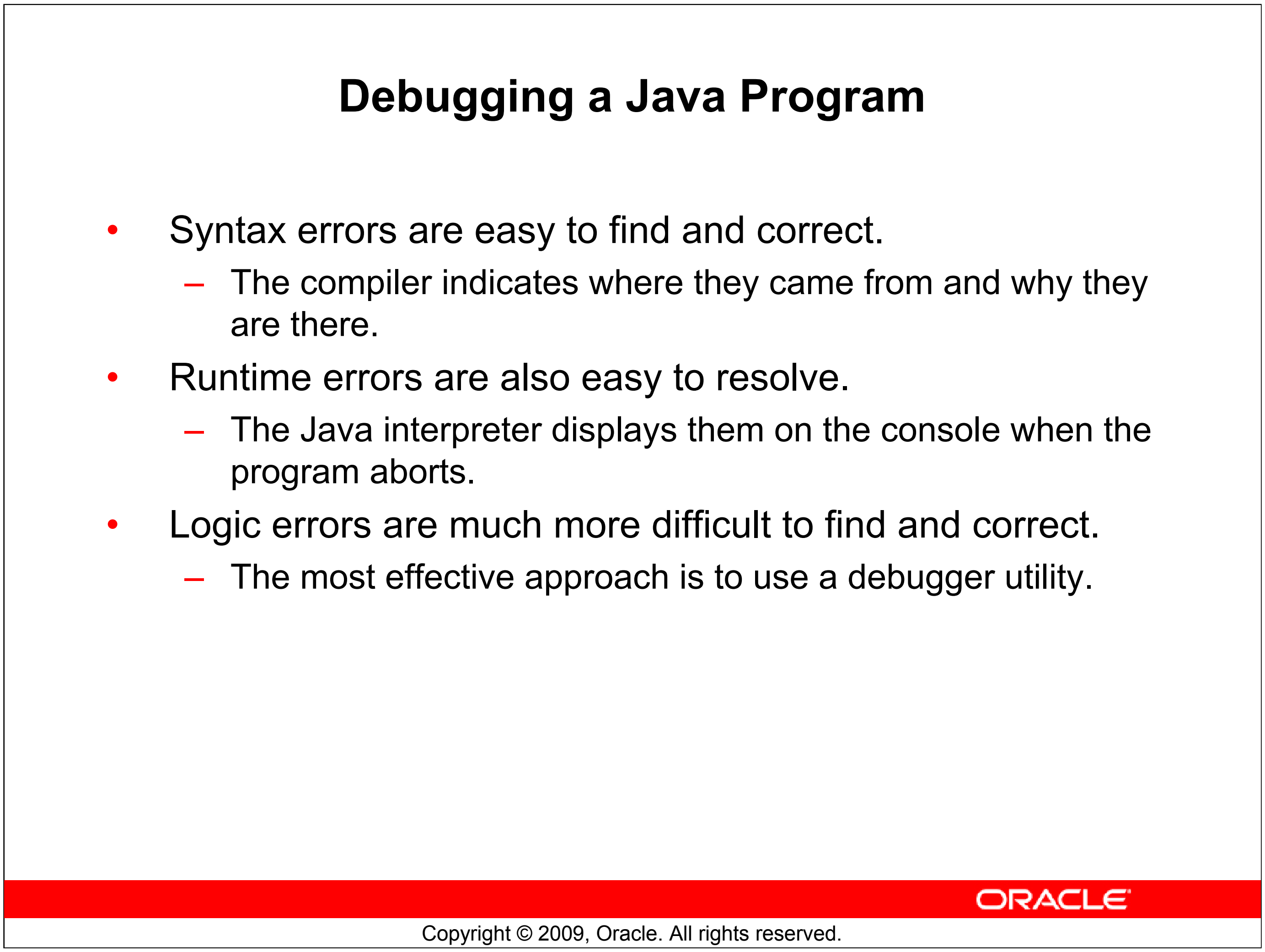
```
    javac SayHello.java
```

Running a Java Application from the Command Line

The `java.exe` executable loads the class, verifies the bytecodes, interprets them into machine language, and executes the code. Start the interpreter by entering the following command at the command prompt:

```
    java SayHello
```

This starts the JVM, which loads `SayHello.class` and tries to call its `main()` method. The JVM expects `SayHello.class` to have a `main()` method where program execution starts. If `SayHello.class` calls methods in other classes, the JVM loads these other classes only when they are required.

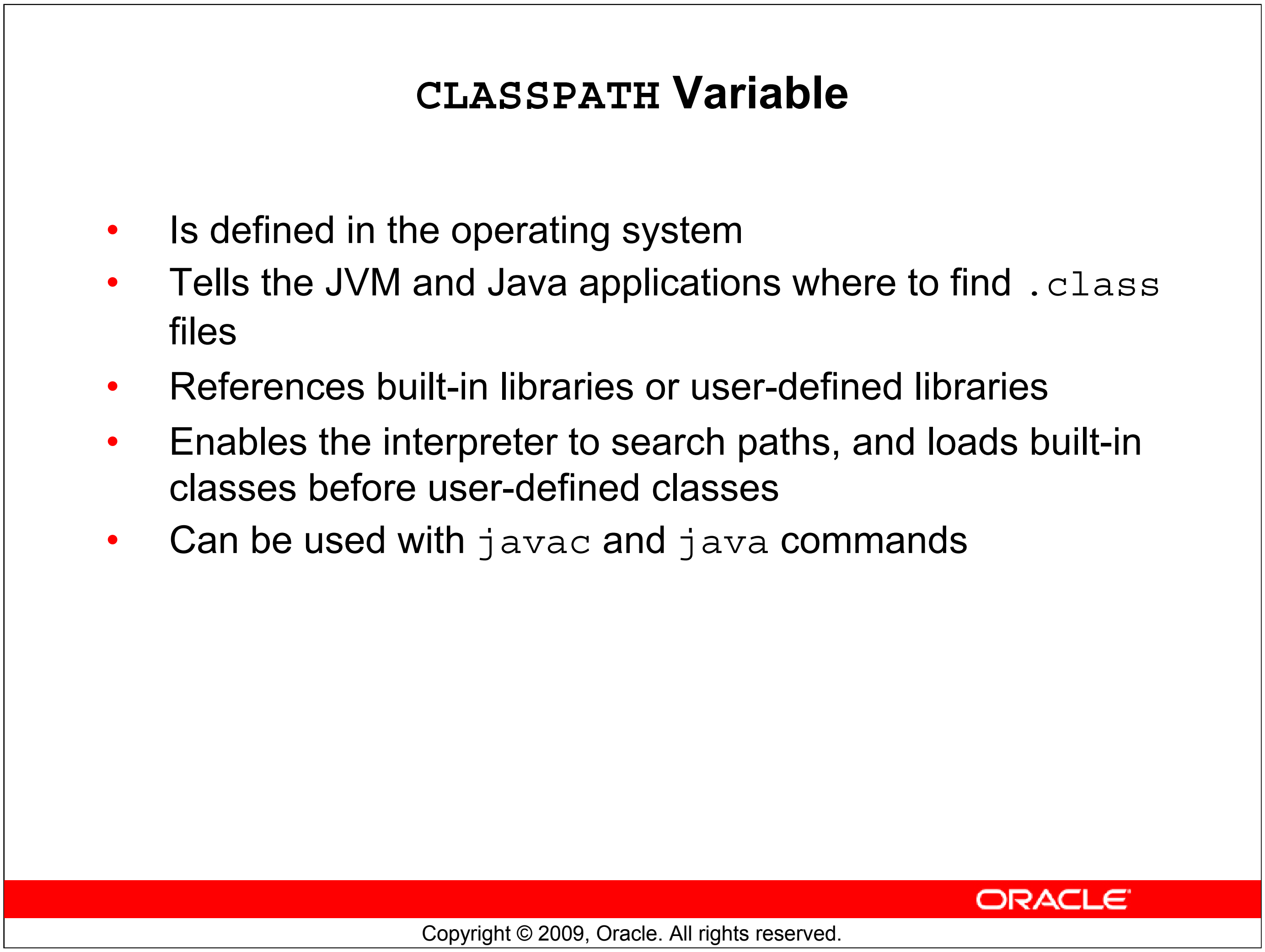


Debugging a Java Program

- Syntax errors are easy to find and correct.
  - The compiler indicates where they came from and why they are there.
- Runtime errors are also easy to resolve.
  - The Java interpreter displays them on the console when the program aborts.
- Logic errors are much more difficult to find and correct.
  - The most effective approach is to use a debugger utility.

Debugging a Java Program

A common approach to debugging is to use a combination of methods to narrow down to the part of the program where the bug is located. You can *hand-trace* the program (catch errors by reading the program), or you can insert print statements to show the values of variables or the execution flow of the program. This approach might work for a short simple program, but for a large complex program, the most effective approach is to use a debugger utility. JDeveloper provides a very powerful debugger utility.



CLASSPATH Variable

- Is defined in the operating system
- Tells the JVM and Java applications where to find `.class` files
- References built-in libraries or user-defined libraries
- Enables the interpreter to search paths, and loads built-in classes before user-defined classes
- Can be used with `javac` and `java` commands

CLASSPATH Variable

If the `CLASSPATH` variable is not set, a default is used. The default includes the absolute path to the `jdk1.5.0_05\bin`, … `\lib\` and …`\classes\` directories, which house the Java SE packages (contained in the `rt.jar` file). You can create a …`/classes/` directory and add your class files, and then `CLASSPATH` will find them.

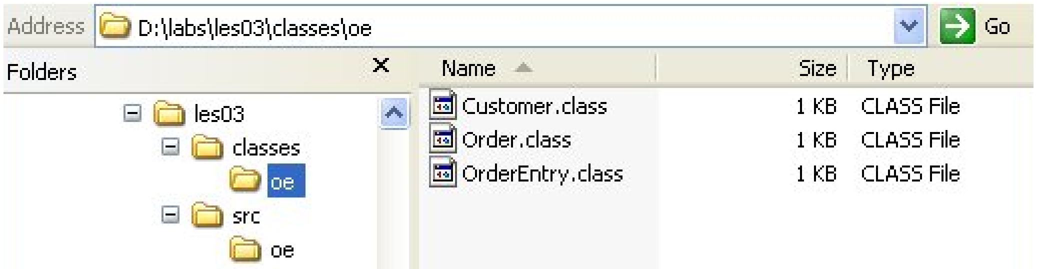
`CLASSPATH` examines individual class files or those stored in `.zip` or `.jar` files. You can set the `CLASSPATH` variable by using the commands `setenv` in UNIX (in a C-shell) and `set` in Windows NT. Separate directories with a semicolon (for example, `c:\myClasses;d:\myOtherClasses`). Set `CLASSPATH` to include the directory containing the `.class` files.

The interpreter looks for classes in the directory sequence as they are found in the `CLASSPATH` variable. If the interpreter cannot find the named class in the first directory, it searches the second and all the others in the list.

If you want the `CLASSPATH` to point to class files that belong to a package, you must specify a path name that includes the path to the directory one level above the directory that has the name of your package.

`Classpath` Use Examples

Location of `.class` files in the `oe` package



Address: D:\labs\les03\classes\oe

Name	Size	Type
Customer.class	1 KB	CLASS File
Order.class	1 KB	CLASS File
OrderEntry.class	1 KB	CLASS File

Setting `CLASSPATH` at the OS level

```
C:\>set CLASSPATH=D:labs\les03\classes
```

Using the -`classpath` option

```
C:\> javac -classpath D:\labs\les03\classes -d D:\labs\les03\classes
          D:\labs\les03\src\oe\OrderEntry.java
C:\> java -classpath D:\labs\les03\classes oe.OrderEntry
```

Setting the `CLASSPATH`

You can override the `CLASSPATH` setting in the `javac` and `java` commands by using the `–classpath` option.

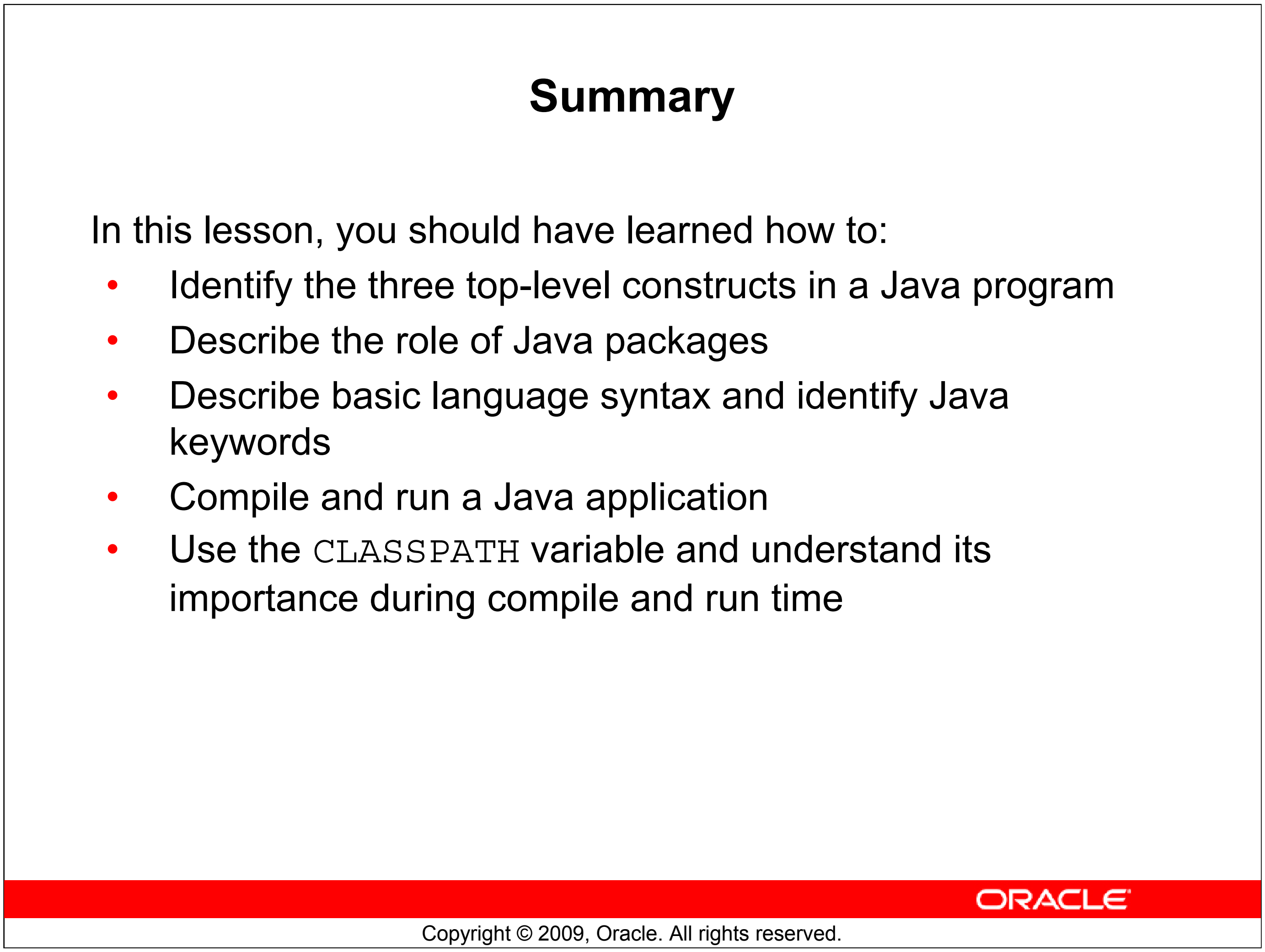
```
javac -classpath <classpath value> -d <output folder>
<path to the source>
```

Example: `javac -classpath D:\labs\les03\classes -d D:\labs\les03\classes D:\labs\les03\src\oe\OrderEntry.java`

```
java -classpath <classpath value> <package.ClassName>
```

Example: `java -classpath D:\labs\les03\classes oe.OrderEntry`

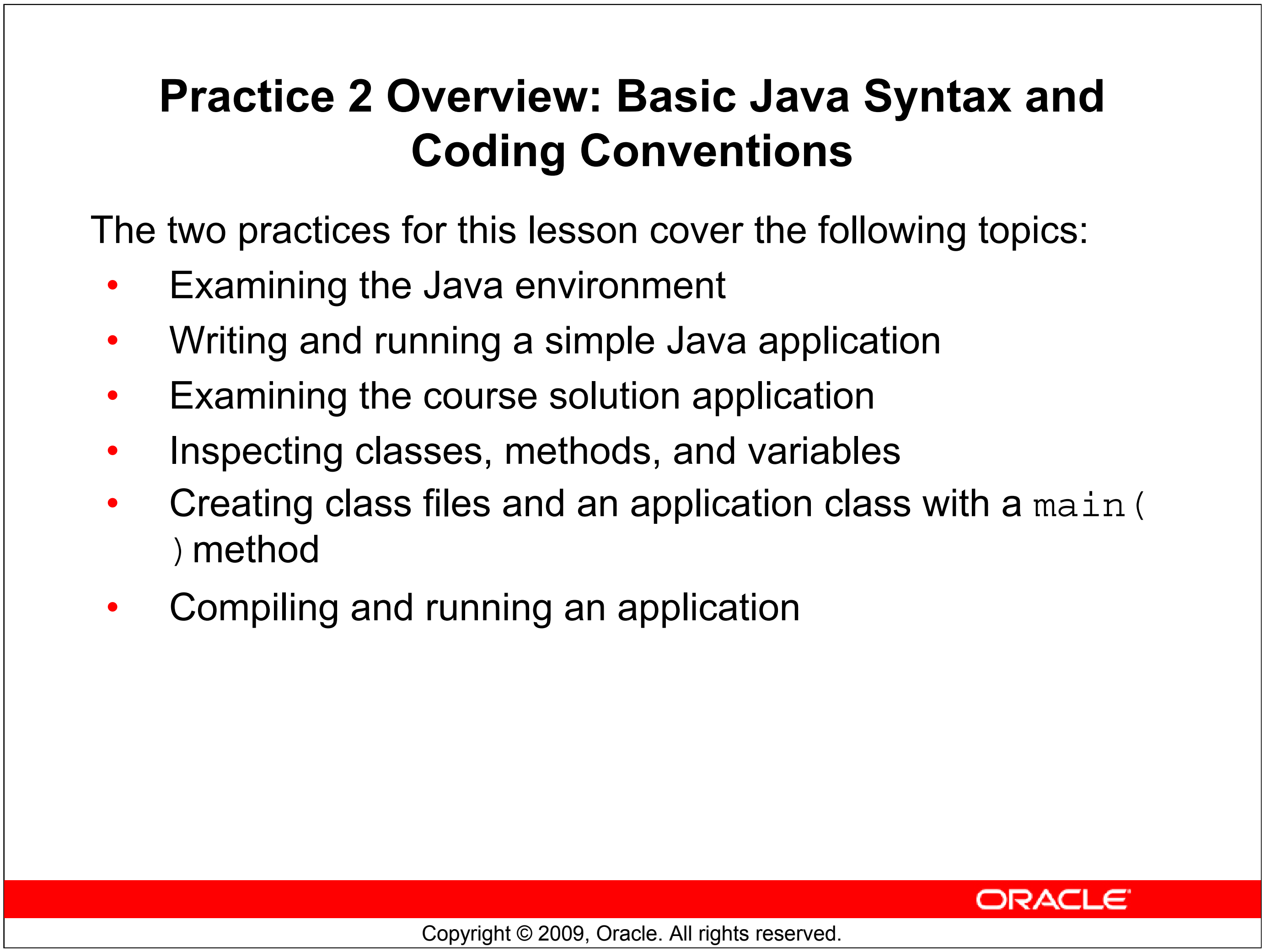
Note: For practical reasons, using the `-classpath` command line option is the preferred approach since it is very common for developers to be using more than one release of Java. This avoids problems due to an incorrect `CLASSPATH` being set in the operating system.



Summary

In this lesson, you should have learned how to:

- Identify the three top-level constructs in a Java program
- Describe the role of Java packages
- Describe basic language syntax and identify Java keywords
- Compile and run a Java application
- Use the `CLASSPATH` variable and understand its importance during compile and run time



Practice 2 Overview: Basic Java Syntax and Coding Conventions

The two practices for this lesson cover the following topics:

- Examining the Java environment
- Writing and running a simple Java application
- Examining the course solution application
- Inspecting classes, methods, and variables
- Creating class files and an application class with a `main( )` method
- Compiling and running an application

Practice 2: Overview

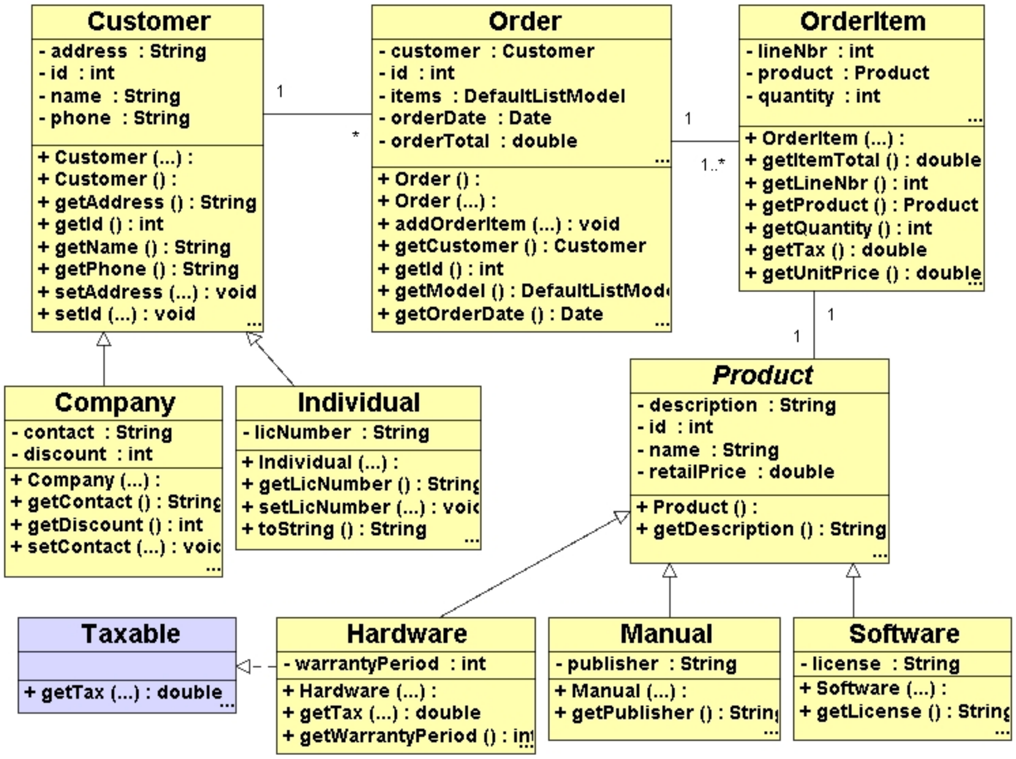
The goal of the practice is to use the Java Development Kit and examine the development environment. You write, compile, and run a simple Java application. You create a class representing a command-line application for the Order Entry system that contains the application entry point in the form of a `main()` method.

Also, you use a UML model as a guide to creating additional class files for your application. You run some simple Java applications, fixing any errors that occur.

The practices in the lessons titled "Basic Java Syntax and Coding Conventions," "Exploring Primitive Data Types and Operators," and "Controlling Program Flow" are to help you better understand the syntax and structure of Java. Their sole purpose is to instruct rather than to reflect any set of best practices for application development.

The goals of the practices from the lesson titled "Building Applications with Oracle JDeveloper" to the end of the course are different. Starting from the lesson titled "Building Applications with Oracle JDeveloper (11*g*)," you use JDeveloper to build an application by techniques you use during real-world development. The practices continue to support the technical material presented in the lesson while incorporating some best practices that you use during the development of a Java application.

UML Diagram for `OrderEntry`



Customer
- address : String
- id : int
- name : String
- phone : String

+ Customer (...) :
+ Customer () :
+ getAddress () : String
+ getId () : int
+ getName () : String
+ getPhone () : String
+ setAddress (...) : void
+ setId (...) : void
...

Order
- customer : Customer
- id : int
- items : DefaultListModel
- orderDate : Date
- orderTotal : double
...
+ Order () :
+ Order (...) :
+ addOrderItem (...) : void
+ getCustomer () : Customer
+ getId () : int
+ getModel () : DefaultListMod
+ getOrderDate () : Date
...

OrderItem
- lineNbr : int
- product : Product
- quantity : int
...
+ OrderItem (...) :
+ getItemTotal () : double
+ getLineNbr () : int
+ getProduct () : Product
+ getQuantity () : int
+ getTax () : double
+ getUnitPrice () : double
...

Company
- contact : String
- discount : int

+ Company (...) :
+ getContact () : String
+ getDiscount () : int
+ setContact (...) : void
...

Individual
- licNumber : String

+ Individual (...) :
+ getLicNumber () : String
+ setLicNumber (...) : void
+ toString () : String
...

Product
- description : String
- id : int
- name : String
- retailPrice : double

+ Product () :
+ getDescription () : String
...

Taxable

+ getTax (...) : double
...

Hardware
- warrantyPeriod : int

+ Hardware (...) :
+ getTax (...) : double
+ getWarrantyPeriod () : int
...

Manual
- publisher : String

+ Manual (...) :
+ getPublisher () : String
...

Software
- license : String

+ Software (...) :
+ getLicense () : String
...

UML Diagram for the Order Entry Application

This diagram of the classes for the Order Entry application was created using JDeveloper. You build the Order Entry application (`OrderEntry`) in the practices for the course.

How to Read the Diagram

The italicized class *Product* is an abstract class, and the lavender-colored class Taxable is an interface. The solid line with an arrow is a Generalization and is used for all dependencies between subtypes and supertypes; it generates an `extends` statement in the class.

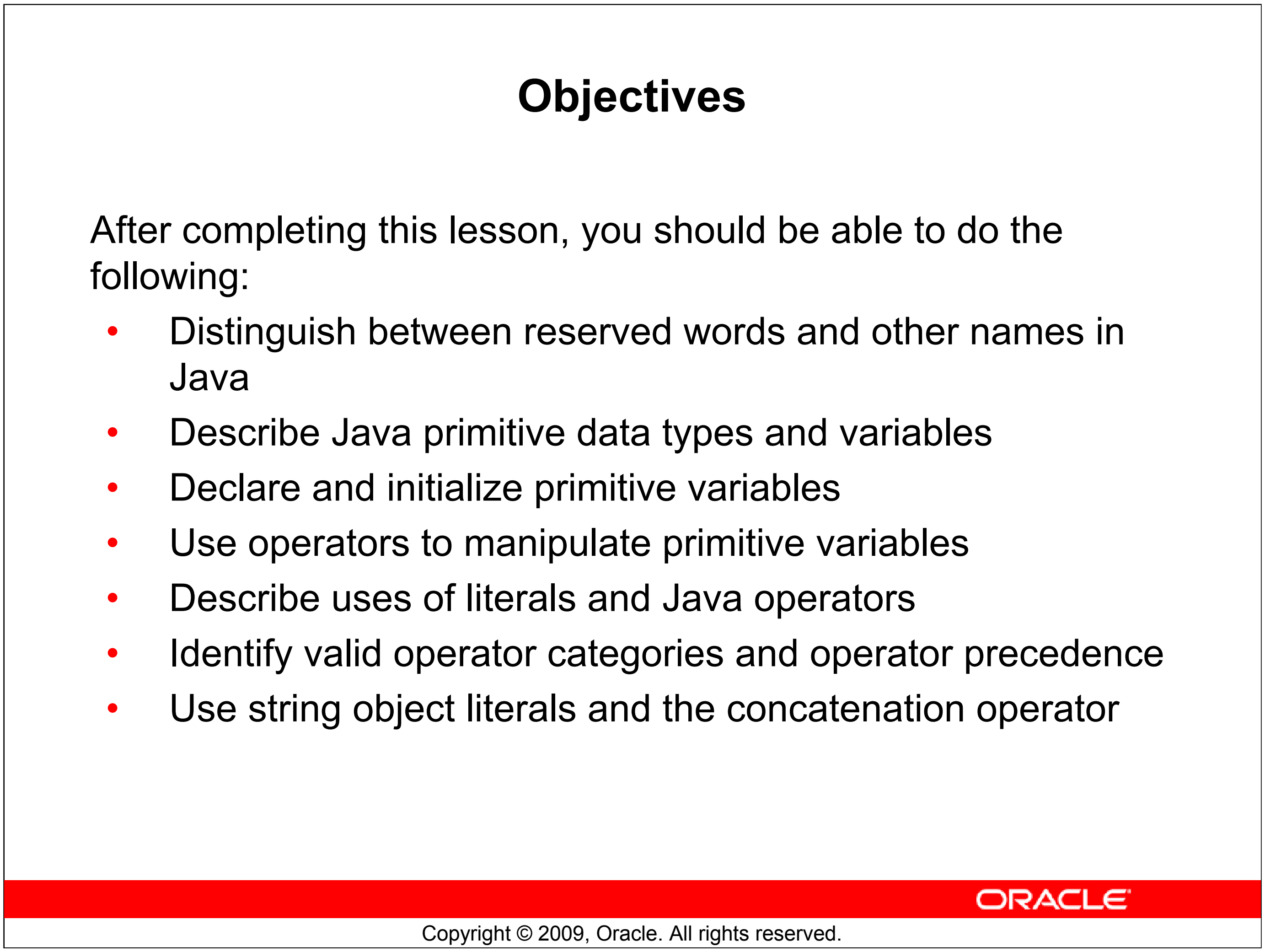
A generalization is the relationship between a more specific element and a less specific element, and defines the inheritance structure in the model. Generalization relationships can be created between two Java classes, between two Java interfaces, or between two entity objects.

The dotted line with the arrow represents a realization and is used between an interface and a class; it generates an `implements` statement in the class. A realization relationship identifies which Java class (or classes) implements a Java interface.

The solid line with no arrow is an association, which signifies a call or a reference from one class to another. The type of relationship between `OrderItem` and `Order` is composition.

The plus symbol (+) next to attributes means that they are public. The “-” symbol means that they are private.

3 Exploring Primitive Data Types and Operators



Objectives

After completing this lesson, you should be able to do the following:

- Distinguish between reserved words and other names in Java
- Describe Java primitive data types and variables
- Declare and initialize primitive variables
- Use operators to manipulate primitive variables
- Describe uses of literals and Java operators
- Identify valid operator categories and operator precedence
- Use string object literals and the concatenation operator

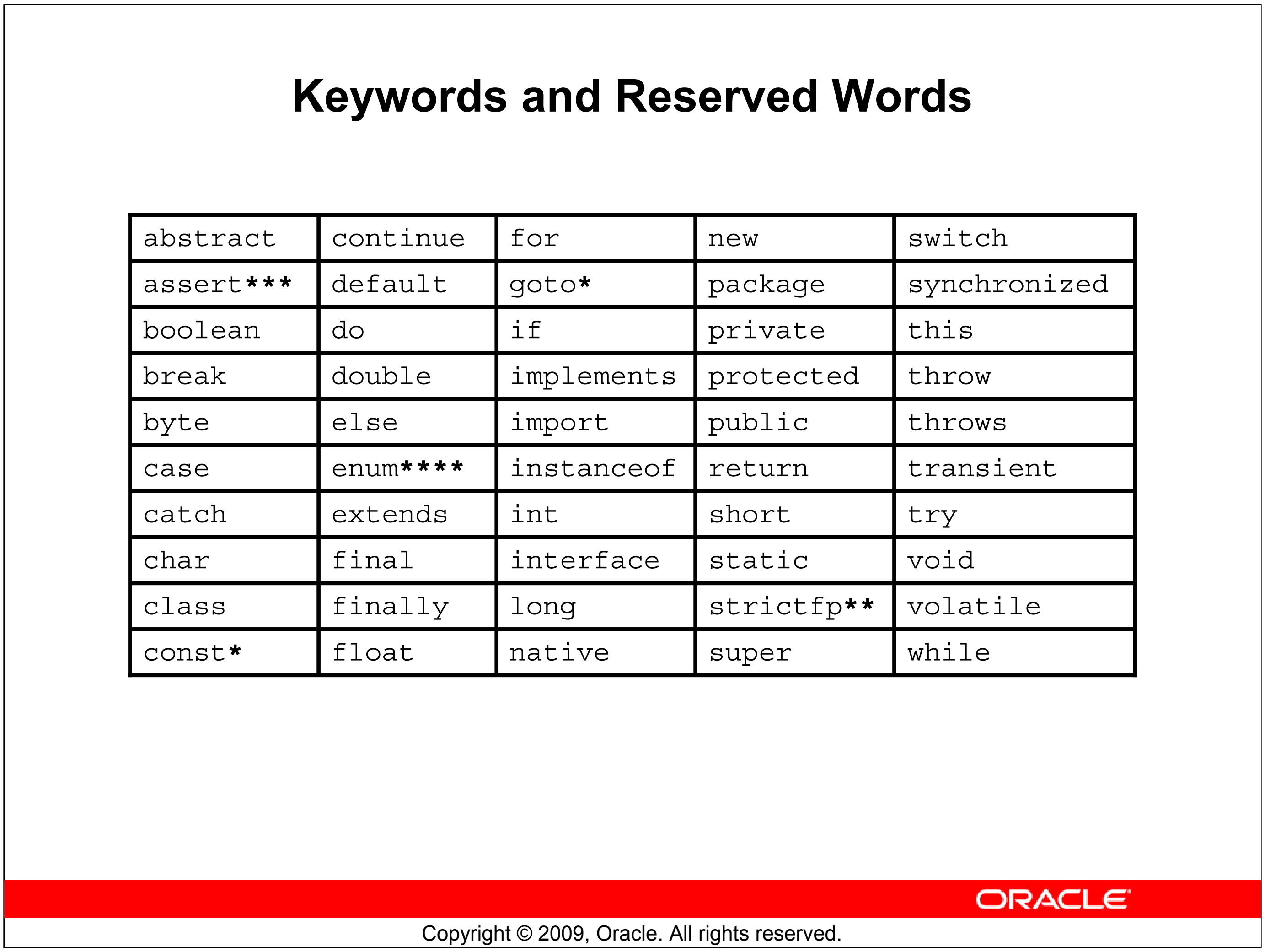
Lesson Objectives

Primitive Data Types in Java

Java is an object-oriented programming language, which means that a Java program is made up of objects. For example, a Java program dealing with video rentals may have objects representing the various videos and games that are available, the customers who have rented videos, the numerous titles that are available, and so on.

If you take a look at one of these objects in more detail, however, you see that it contains fundamental values such as whole numbers, fractions, and characters. Java provides eight predefined data types to represent these atomic entities. The Java community calls these the *primitive* types, as opposed to *object* types (user-defined types) that you can define yourself, such as `Customer` or `Title`.

This lesson describes how to declare and initialize variables of each of the eight primitive types.



Keywords and Reserved Words

abstract	continue	for	new	switch
assert***	default	goto*	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum****	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp**	volatile
const*	float	native	super	while

Keywords and Reserved Words

The table shows a list of Java keywords. These words are reserved—you cannot use any of these words as names in your programs. `true`, `false`, and `null` are not keywords but rather reserved words, so you cannot use them as names in your programs either.

*	not used
**	added in 1.2
***	added in 1.4
****	added in 1.5 (5.0)

For more information about keywords and reserved words, go to http://java.sun.com/docs/books/tutorial/java/nutsandbolts/_keywords.html.

Variable Types

- Eight primitive data types:
  - Six numeric types
  - One character type
  - One Boolean type (for truth values)
- User-defined types:
  - Classes
  - Interfaces
  - Arrays

Variable Types

A variable is the basic storage unit in Java: a symbolic name for a chunk of memory. Variables can be defined to hold primitive values (which are not real objects in Java) or to hold object references. The object references that they hold can be user-defined classes or Java-defined classes.

Primitive Data Types

A variable declaration consists of a type and a name. Consider the following example:

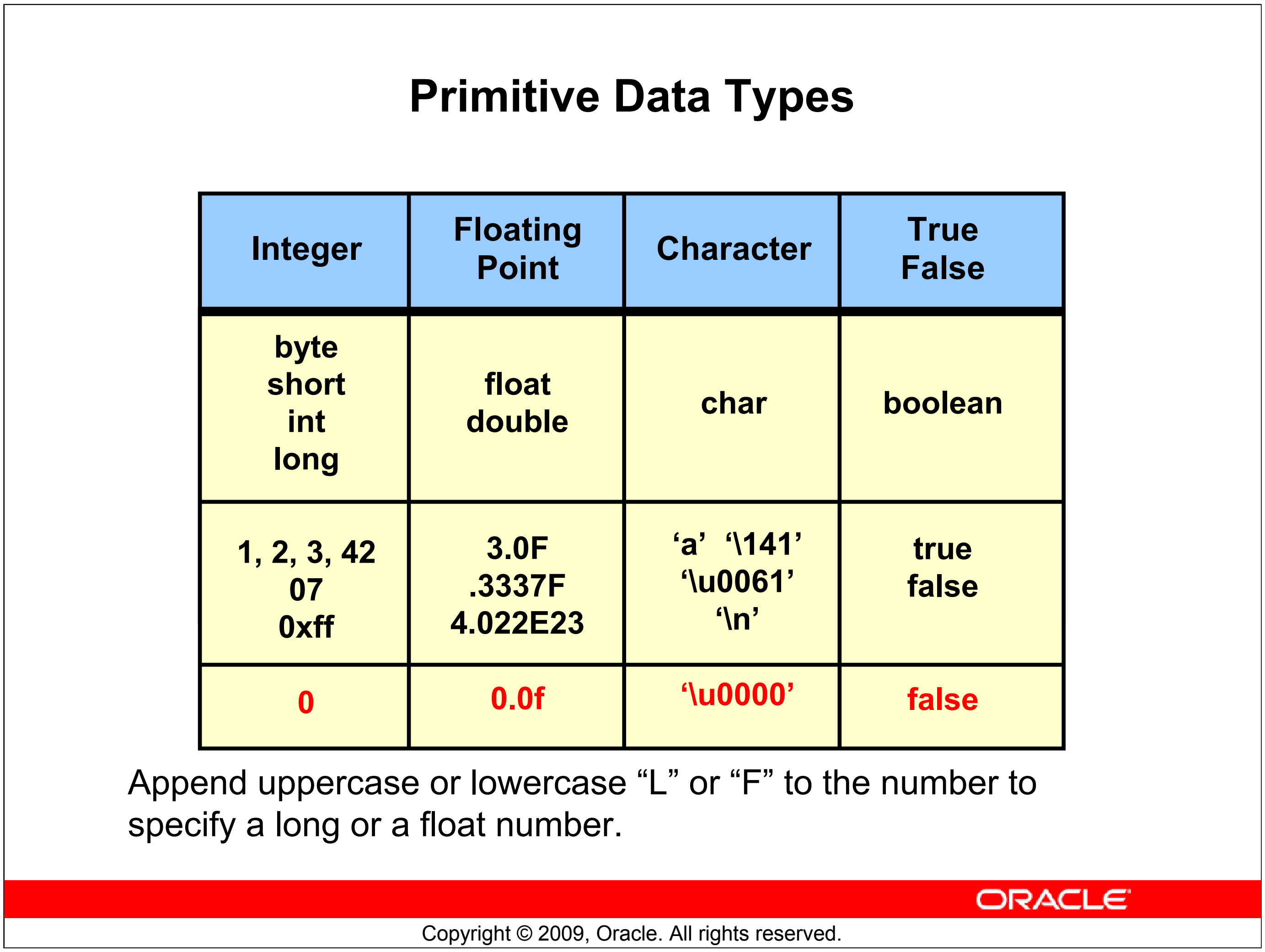
```
double balanceDue;
```

This statement declares a variable called `balanceDue`, whose type is `double`. `double` is one of the eight primitive data types that are provided in Java. These types are called *primitive* because they are not objects and are built in to Java.

Unlike similar types in other languages, such as C and C++, the size and characteristics of each primitive data type are defined by Java and are consistent across all platforms.

Classes, Interfaces, and Arrays

Classes, interfaces, and arrays are user-defined types with specific characteristics. After they are defined, you can declare variables of the new type just as you declare primitive variables.



Primitive Data Types

Integer	Floating Point	Character	True False
byte short int long	float double	char	boolean
1, 2, 3, 42 07 0xff	3.0F .3337F 4.022E23	‘a’ ‘\141’ ‘\u0061’ ‘\n’	true false
0	0.0f	‘\u0000’	false

Append uppercase or lowercase “L” or “F” to the number to specify a long or a float number.

Primitive Data Types

Integer

Java provides four different integer types to accommodate different size numbers. All the numeric types are signed, which means that they can hold positive or negative numbers.

The integer types have the following ranges:

- `byte` range is –128 to +127. Number of bits = 8.
- `short` range is –32,768 to +32,767. Number of bits = 16.
- `int` range is –2,147,483,648 to +2,147,483,647. The most common integer type is `int`. Number of bits = 32.
- `long` range is –9,223,372,036,854,775,808 to +9,223,372,036,854,775,807. Number of bits = 64.

Floating Point

The floating-point types hold numbers with a fractional part and conform to the IEEE 754 standard. There are two types: `float` and `double`. `double` is so called because it provides double the precision of `float`. `float` uses 32 bits to store data, whereas `double` uses 64 bits.

Primitive Data Types (continued)

Character

The `char` type is used for individual characters, as opposed to a string of characters (which is implemented as a `String` object). Java supports Unicode, an international standard for representing a character in any written language in the world in a single 16-bit value. The first 256 characters coincide with the ISO Latin 1 character set, part of which is ASCII.

Boolean

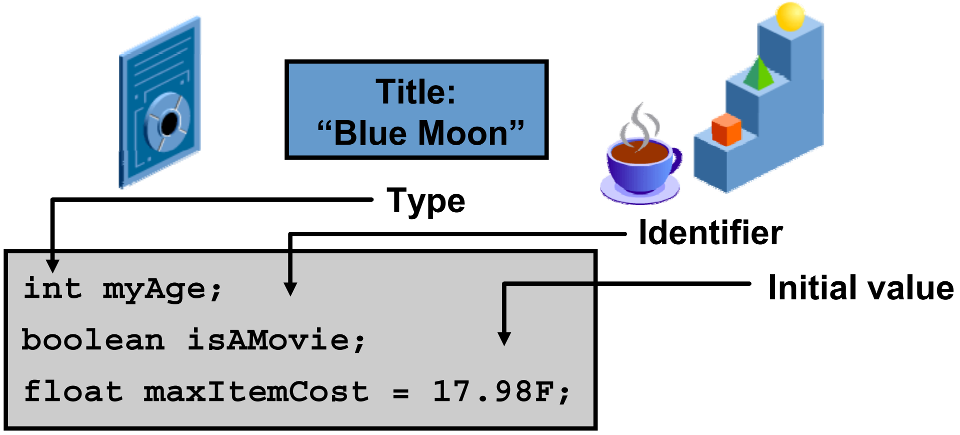
The `boolean` type can hold either `true` or `false`.

Default Values

If a value is not specified, a default value is used. The values in red in the slide are the defaults used. The default `char` value is `null` (represented as `?\u0000?`), and the default value for `boolean` is `false`.

Variables

- A variable is a basic unit of storage.
- Variables must be explicitly declared.
- Each variable has a type, an identifier, and a scope.
- There are three types of variables: class, instance, and method.



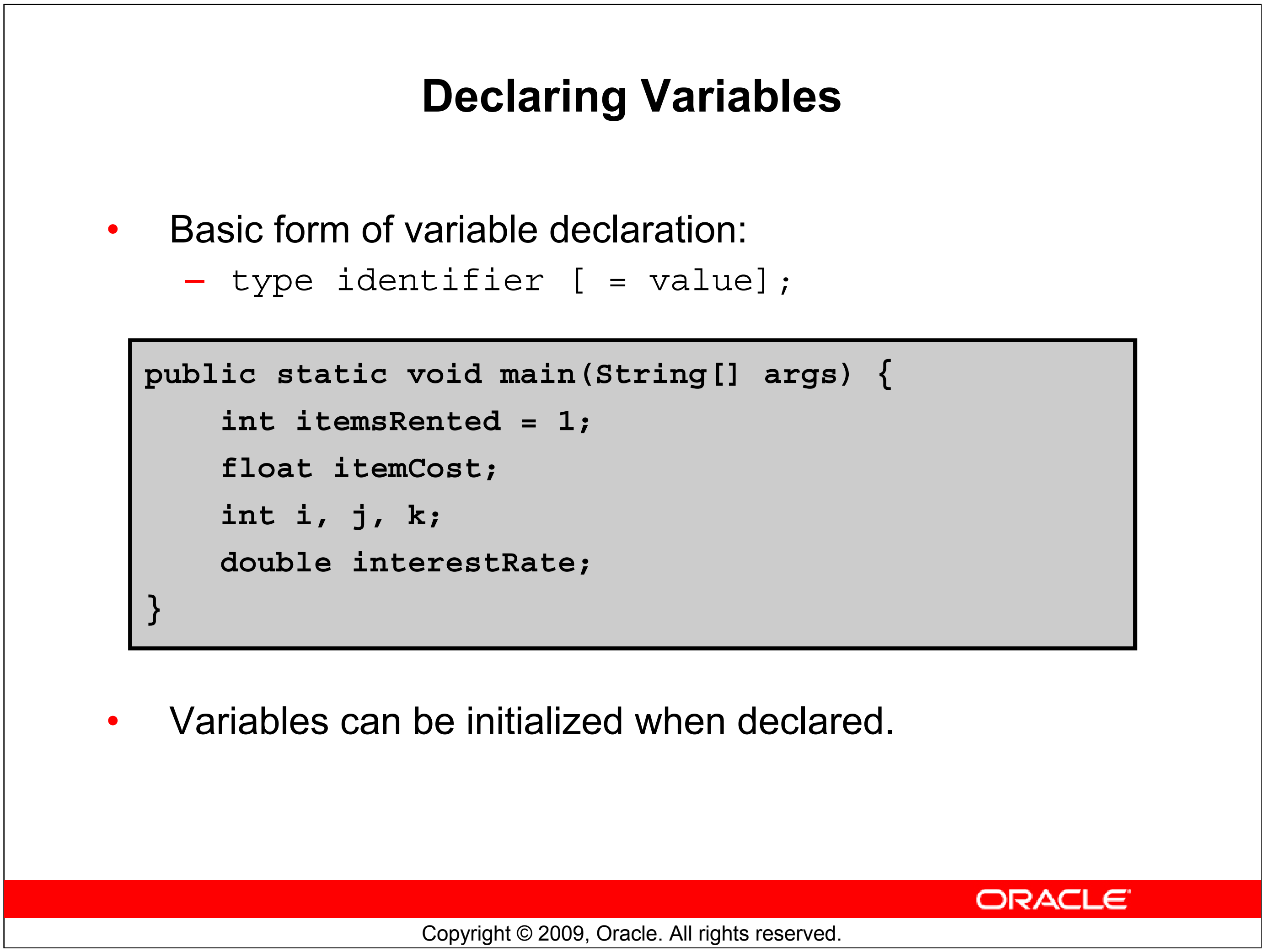
```
int myAge;
boolean isAMovie;
float maxItemCost = 17.98F;
```

Variables

A variable is a symbolic name for a chunk of memory in which a value can be stored. Because Java is a strongly typed language, all variables must be declared before they can be used.

Variables also have a *scope* that determines where you can use the variable. Scope also determines the life span of the variable. When a variable is defined in a method, the variable is available only in the execution of the method. When the method ends, the variable is released and is no longer accessible. When defining a variable with a local scope, use braces.

Variables can also be explicitly initialized (that is, they can be given a default value).



Declaring Variables

- Basic form of variable declaration:
  - `type identifier [ = value];`

```
public static void main(String[] args) {
    int itemsRented = 1;
    float itemCost;
    int i, j, k;
    double interestRate;
}
```

- Variables can be initialized when declared.

Declaring Variables

As mentioned earlier, you must declare all variables before using them. The declaration consists of the type followed by the variable name. Do not forget to terminate the statement with a semicolon. You can declare a variable anywhere within a block, although declaring it at the start is often preferable.

You can declare several variables in one statement. All the variables that are declared in one statement must be of the same type, such as `int`, `char`, `float`, and so on. The syntax is as follows:

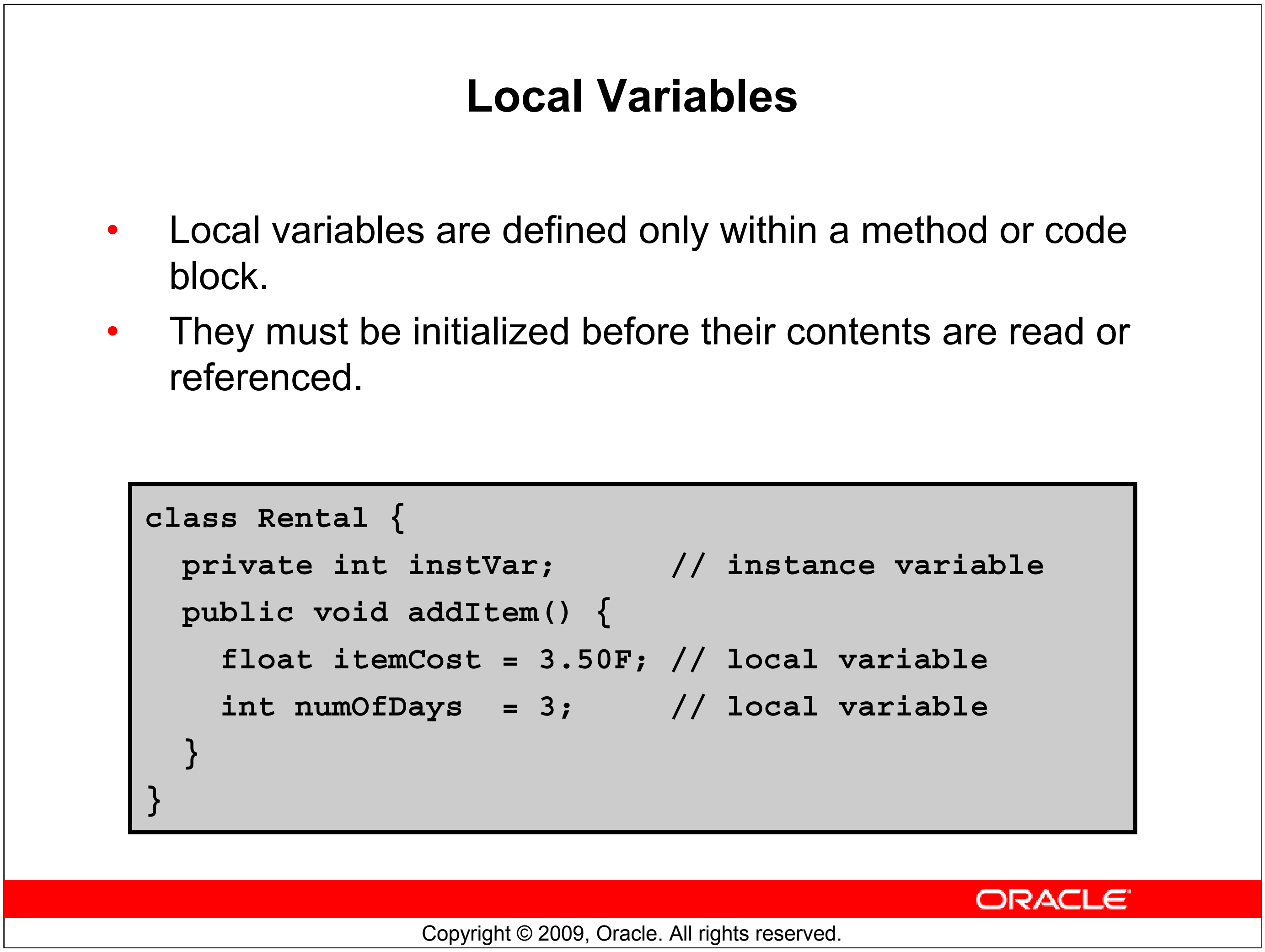
```
int itemsRented, numOfDays, itemId;
```

Initializing Variables

You can declare and initialize a variable in the same statement by using the assignment operator (=). Even though they are in the same statement, each variable is initialized independently. Each variable needs its own assignment and value:

```
int itemsRented = 0, numOfDays, itemId = 0;
```

In this case, `itemsRented` and `itemId` are initialized, whereas `numOfDays` is not. The compiler accepts this statement as valid.



Local Variables

- Local variables are defined only within a method or code block.
- They must be initialized before their contents are read or referenced.

```
class Rental {
  private int instVar;        // instance variable
  public void addItem() {
    float itemCost = 3.50F; // local variable
    int numOfDays  = 3;      // local variable
  }
}
```

Local Variables

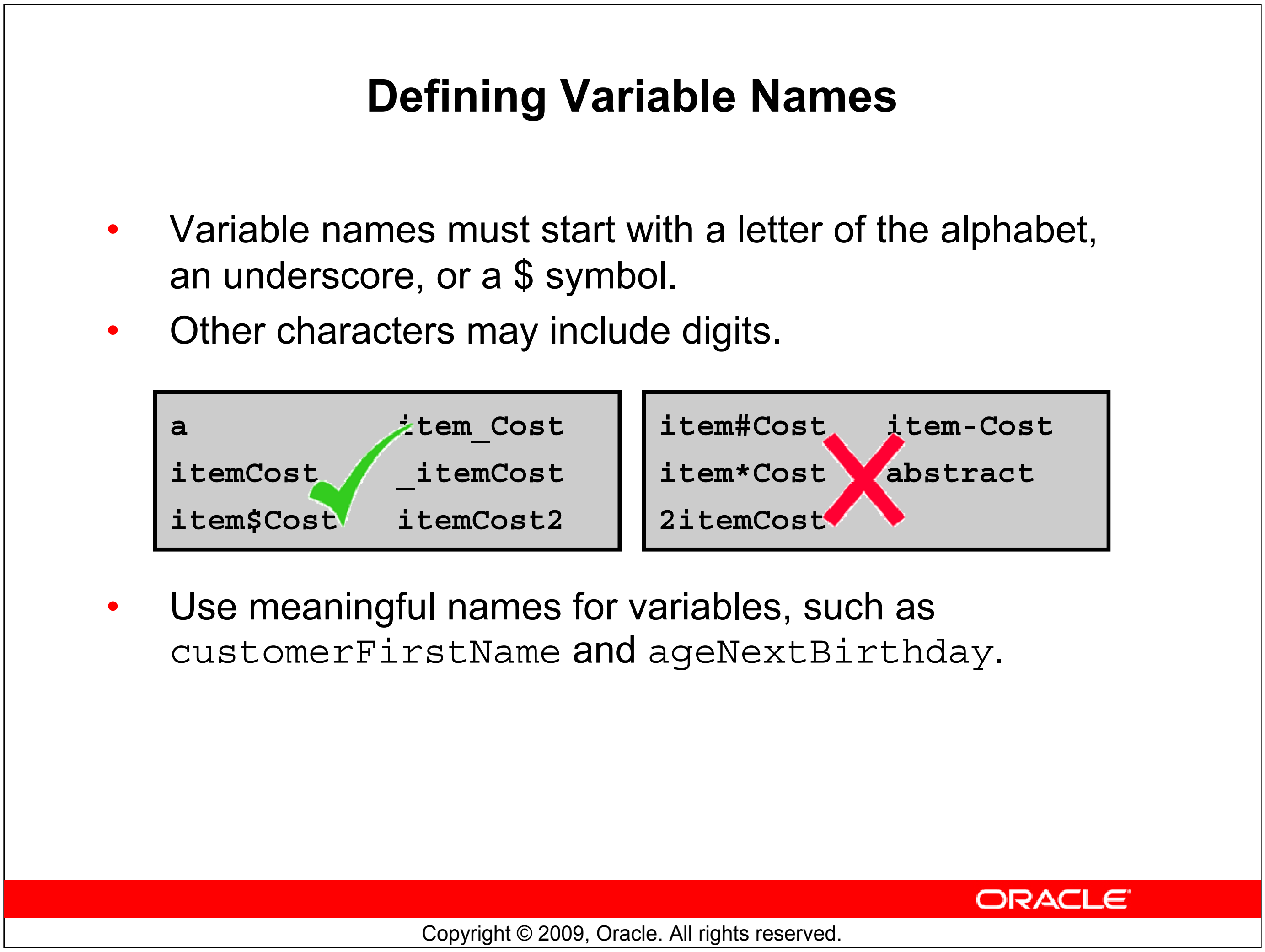
A local variable is one that is defined inside a method and can therefore be accessed only inside that method. In contrast, a variable that is defined outside a method can be accessed by any method in that class. Consider the following example:

```
class Rental {
  private int memberId;  // Use in any method in the class
  public void addItem() {
    float itemCost = 3.50F; // Accessible only in addItem()
    int numOfDays  = 3;     // Accessible only in addItem()
  }
}
```

Local Variables Must Be Assigned a Value Before They Can Be Used

A local variable that is declared inside a method must be given a value before it can be used in an expression. If you use a local variable that has not been assigned a value, the compiler issues an error.

Local variables are also known as method variables or even method local variables. In this course, they are referred to as local variables.



Defining Variable Names

- Variable names must start with a letter of the alphabet, an underscore, or a $ symbol.
- Other characters may include digits.

✓

```
a              item_Cost
itemCost       _itemCost
item$Cost      itemCost2
```

✗

```
item#Cost   item-Cost
item*Cost   abstract
2itemCost
```

- Use meaningful names for variables, such as `customerFirstName` and `ageNextBirthday`.

Defining Variable Names

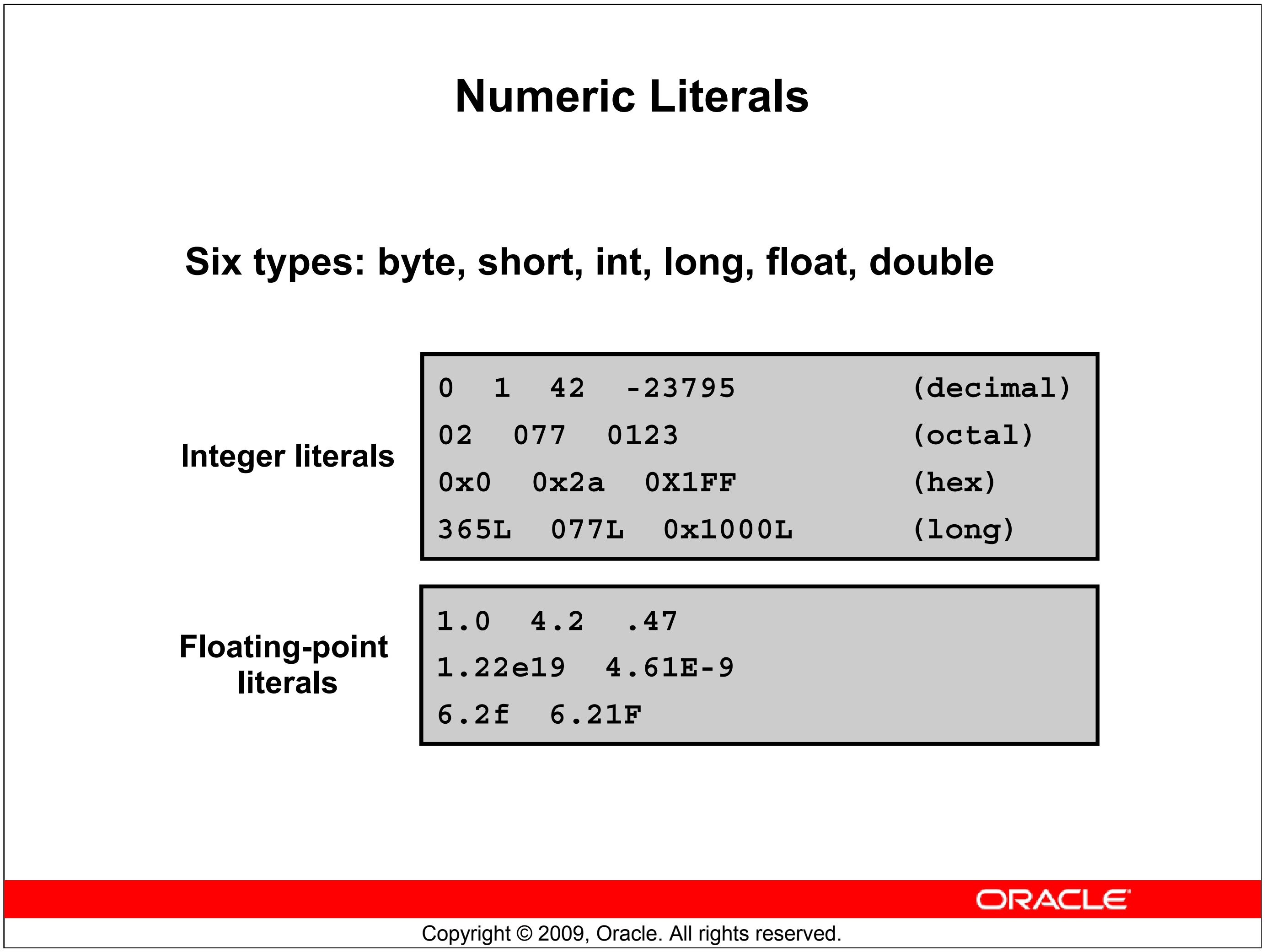
A variable name must start with a letter of the alphabet, an underscore, or a dollar sign, although most Java programmers avoid the use of underscores and dollar signs. Subsequent characters can include the digits 0 through 9. Note that Java is case-sensitive; lowercase letters are different from uppercase letters.

There is a 64 KB restriction on the length of variable names, and it is recommended that you choose meaningful names that combine several words. By convention, the first word must be set in lowercase, and subsequent words must start with uppercase letters, as in the example `customerFirstName`.

Examples of Illegal Variable Names

The variables in the box on the right in the slide are illegal for the following reasons:

- `item#Cost` is illegal because # is not allowed.
- `item-cost` is illegal because – (minus sign, not underscore) is not allowed.
- `item*Cost` is illegal because `*` is an operator.
- `abstract` is illegal because `abstract` is a keyword.
- `2itemCost` is illegal because it starts with a digit.



Numeric Literals

Six types: byte, short, int, long, float, double

Integer literals

```
0   1   42   -23795          (decimal)
02   077   0123              (octal)
0x0   0x2a   0X1FF           (hex)
365L   077L   0x1000L        (long)
```

Floating-point literals

```
1.0   4.2   .47
1.22e19   4.61E-9
6.2f   6.21F
```

Numeric Literals

Variables hold values, whereas literals are the values themselves.

```
float itemCost = 4.95F;
```

The variable is `itemCost`, and `4.95` is the literal. Literals can be used anywhere in a Java program, just like a variable. The difference is that literals cannot be stored or held without the use of variables.

Integer Literals

By default, integer literals are 32-bit signed numbers. Integer literals can be specified in decimal, octal, or hexadecimal. When assigning values to a variable, do *not* use commas to separate thousands. To specify a literal in decimal form, simply use a number (12317.98). Literals with a leading zero are in octal form. Octal numbers are in base 8 and can contain the digits 0 through 7. To specify a hexadecimal value, use a leading 0x or 0X. The valid characters are 0 through 9 and A through F (which can be set in uppercase or lowercase).

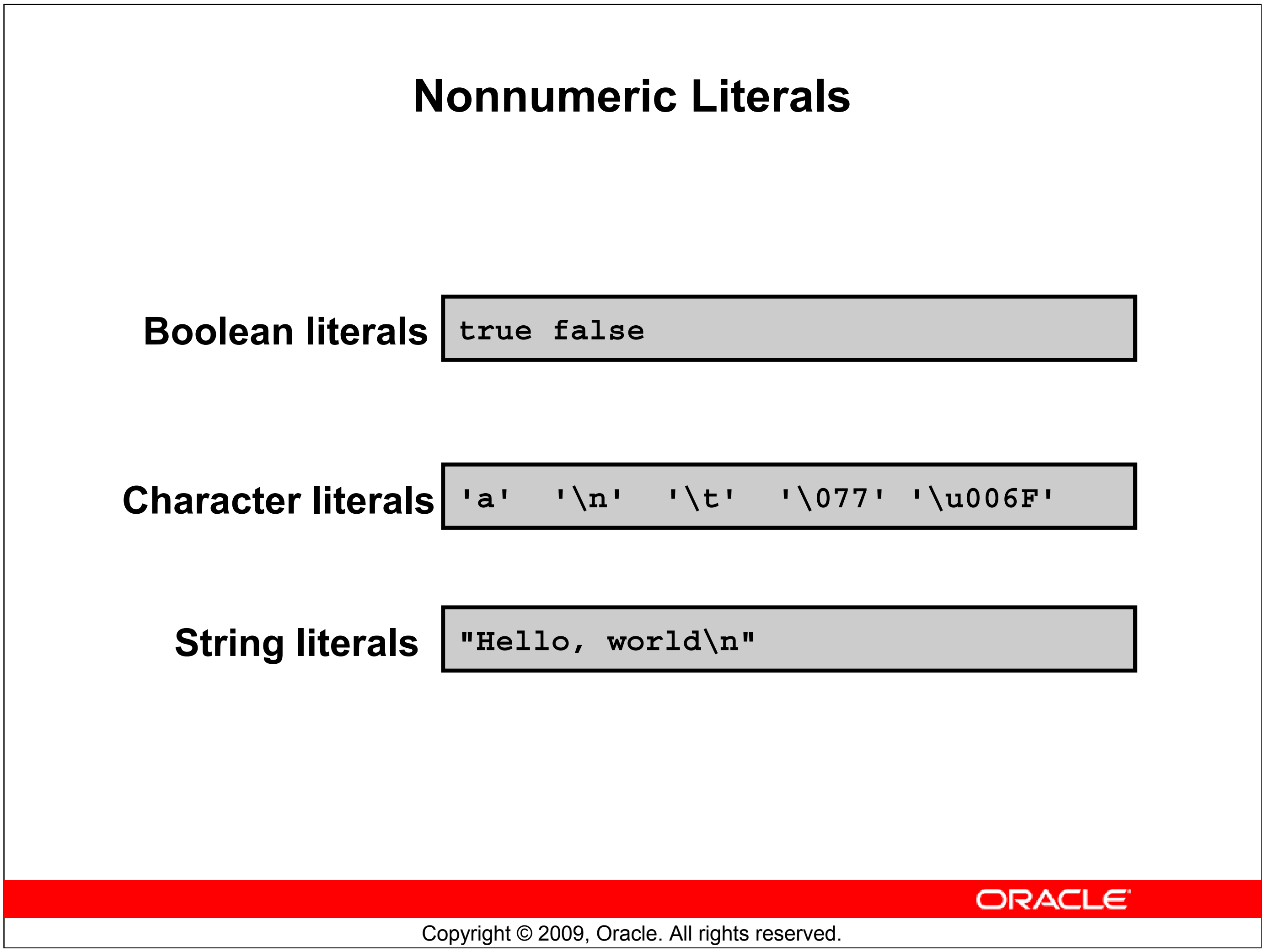
Numeric Literals (continued)

Integer Literals (continued)

A long integer is a 64-bit type. It can hold a larger number than the 32-bit version (int). Remember that the 32-bit version can range up to 2,147,483,647. If you need to store or use a number larger than that, you must use a long integer. You can force any integer to be treated as long by appending an uppercase or lowercase L.

Floating-Point Literals

Floating-point literals can be specified in standard format (123.4) or in scientific notation (1.234e2). By default, floating-point literals are taken as double precision. You can obtain single precision by appending an uppercase or lowercase F.



Nonnumeric Literals

Boolean literals
```
true false
```

Character literals
```
'a'  '\n'  '\t'  '\077' '\u006F'
```

String literals
```
"Hello, world\n"
```

Nonnumeric Literals

Boolean Literals

A Boolean literal can be either `true` or `false`, which are Java keywords. Note that `true` and `false` are not numeric values and cannot be converted to or from integers.

Character Literals

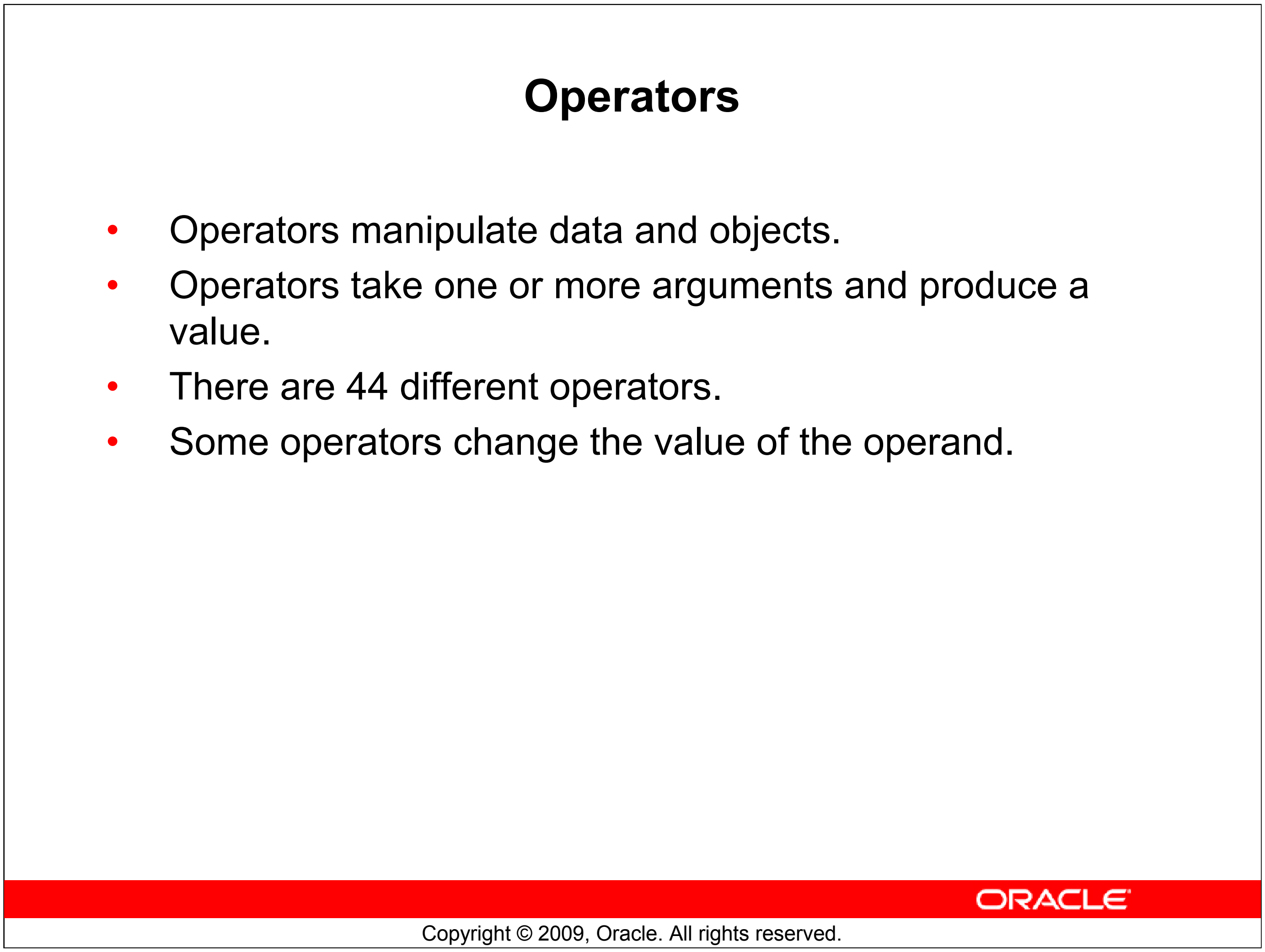
Character literals are normally printable characters that are enclosed in single quotation marks, such as `‘a’, ‘b’,` and `‘c’`. To specify a nonprintable character, such as a new line or a tab, you must specify either its octal or hexadecimal Unicode value, or use its corresponding “escape sequence.” An escape sequence consists of the backslash character followed by another character.

Character	**Escape Sequence**
New Line	`'\n'`
Tab	`'\t'`
Single quote	`'\''`
Backslash	`'\\'`
Unicode values	`'\u006F'`

Nonnumeric Literals (continued)

String Literals

String literals consist of any number of characters within double quotation marks. String literals are different from character literals because the character primitive holds only one character. String literals hold multiple characters and are delimited by double quotation marks.



Operators

- Operators manipulate data and objects.
- Operators take one or more arguments and produce a value.
- There are 44 different operators.
- Some operators change the value of the operand.

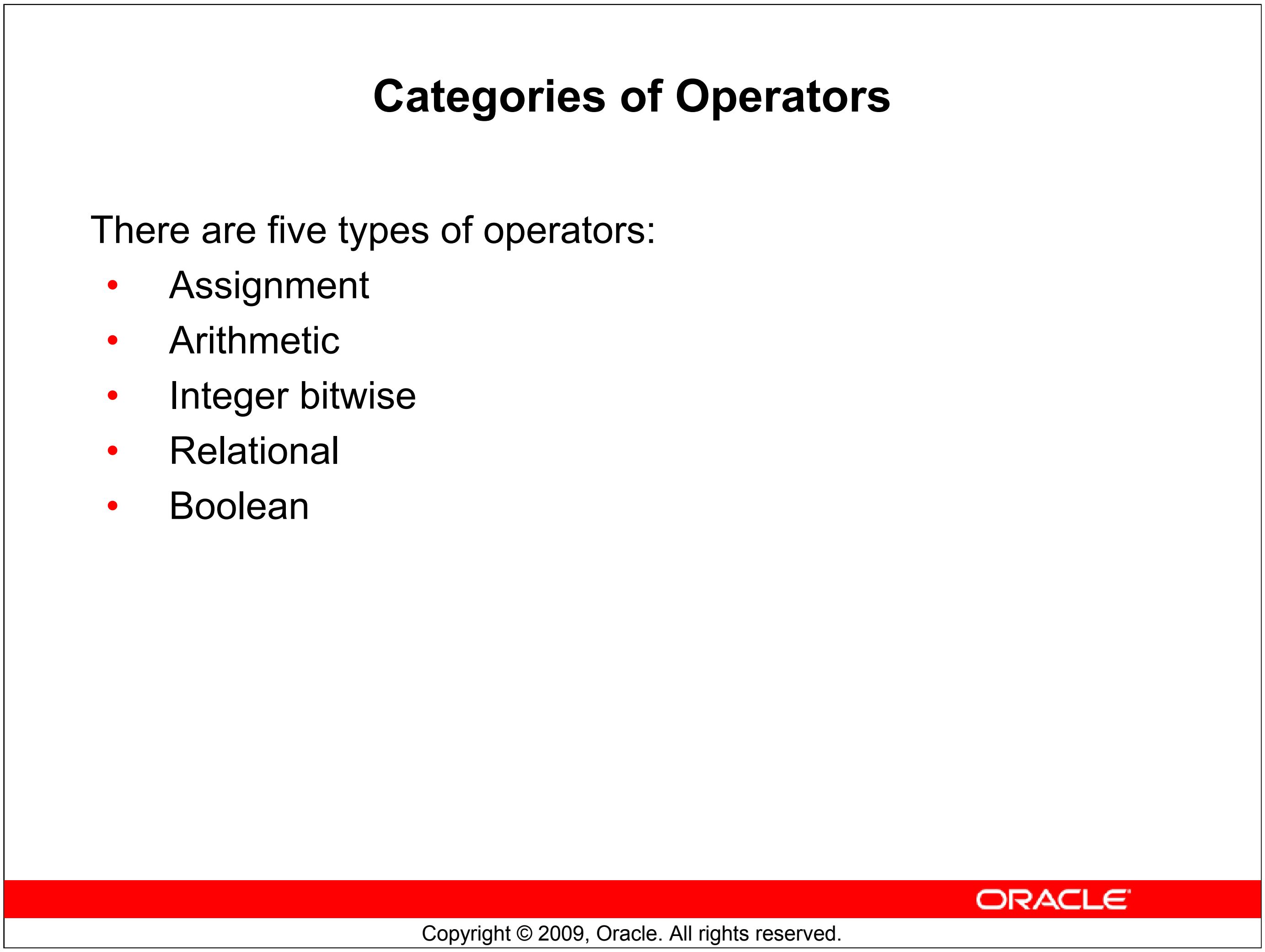
Operators

Operators are used to manipulate the values that are stored in variables; these variables can be in expressions or they may contain literal values. Most programs manipulate data for some purpose or goal. For example, to calculate the due date of a video rental, a program must take the day that the video was rented and add some number of days. This is data manipulation. It is hard to imagine a program that does not use and manipulate data in some way, with the exception of raw queries of the database.

Java operators take one or more arguments, or operands, and produce a value. Java provides 44 different operators that manipulate data in one way or another. Some of those operators are more complex than others, and some are used more often.

This lesson focuses on the more important operators.

Note: It is not possible to extend the functionality of the operators. In C++, for example, you can define your own new meaning for the "–" operator. In Java, you cannot do this.



Categories of Operators

There are five types of operators:

- Assignment
- Arithmetic
- Integer bitwise
- Relational
- Boolean

Categories of Operators

Operators are special characters that are used to instruct the Java compiler to perform an operation on an operand. Java includes a set of 44 different operators. Most programs need only some of the 44 distinct operators.

Assignment Operators

Assignment operators set the value of a variable to a literal value or to the value of another variable or expression.

Arithmetic Operators

Arithmetic operators perform mathematical computations on operands. Arithmetic operators operate on all numeric types.

Integer Bitwise Operators

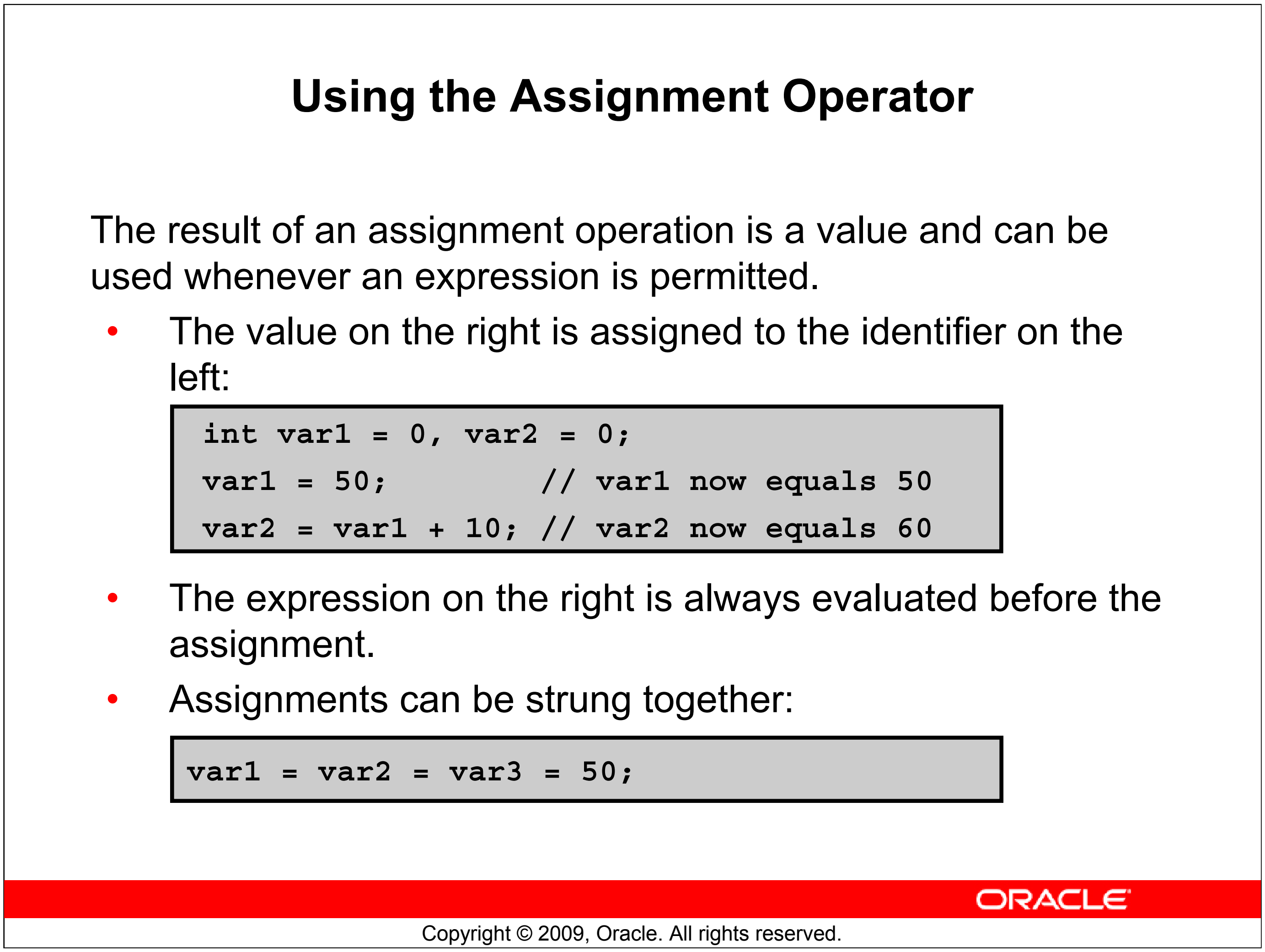
Bitwise operators are provided to inspect and modify the internal bits that make up integer numeric types, whereas arithmetic operators modify the values of a variable as a whole unit.

Relational Operators

Relational operators compare two values. You can use relational comparison to set other values or to control program flow.

Boolean Logical Operators

Boolean operators can be used only on Boolean variables or expressions. The result of a Boolean operator is always a Boolean value.



Using the Assignment Operator

The result of an assignment operation is a value and can be used whenever an expression is permitted.

- The value on the right is assigned to the identifier on the left:

```
int var1 = 0, var2 = 0;
var1 = 50;         // var1 now equals 50
var2 = var1 + 10; // var2 now equals 60
```

- The expression on the right is always evaluated before the assignment.
- Assignments can be strung together:

```
var1 = var2 = var3 = 50;
```

ORACLE

Using the Assignment Operator

After a variable has been declared, you can assign a value by using the assignment operator. The value of the expression on the right side of the assignment operator is determined or evaluated, and then the result is assigned to the variable on the left.

Note the following example:

```
var1 = 10;
var2 = var1 + 10;  // the right side is evaluated first
```

`var2` now equals 20.

Assignments Can Be Strung Together in the Same Statement

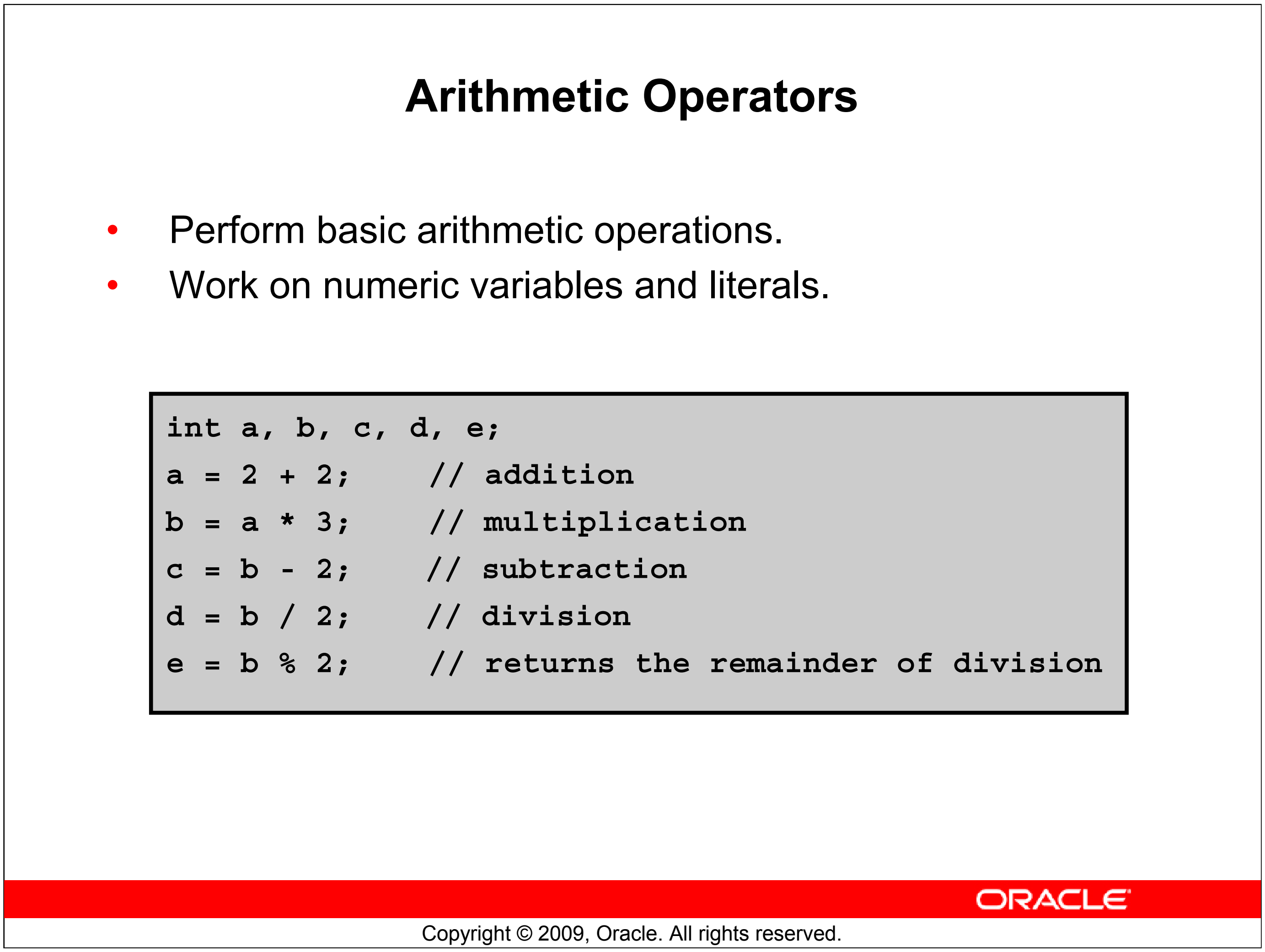
Multiple assignment operators can be used in a single statement as follows:

```
var1 = var2 = var3 = 50;
```

The assignment operator has “right associativity,” which means that the right-most assignment operator is performed first in this statement. The other assignments are evaluated moving from right to left. The statement can be rewritten as follows, to emphasize the order in which the assignments are carried out and to make the code clearer:

```
var1 = (var2 = (var3 = 50));
```

The net result is that the value `50` is assigned to all three variables.



Arithmetic Operators

- Perform basic arithmetic operations.
- Work on numeric variables and literals.

```
int a, b, c, d, e;
a = 2 + 2;     // addition
b = a * 3;     // multiplication
c = b - 2;     // subtraction
d = b / 2;     // division
e = b % 2;     // returns the remainder of division
```

Arithmetic Operators

Most arithmetic operators in Java are similar to those in other languages. Both operands are usually of numeric types, and the result of the operation is normally numeric. The important concepts to remember when using simple arithmetic operators are the following:

- Integer division results in an integer, and any remainder is ignored.
- The multiply, divide, and modulus operators have higher precedence than the add and subtract operators. In other words, multiplication, division, and modulus operations are evaluated before addition and subtraction operations.
- Arithmetic operations can be performed on variables and literals.
- Modulus (`mod`) returns the remainder of a division operation.

More on Arithmetic Operators

Most operations result in `int` or `long`:

- `byte`, `char`, and `short` values are promoted to `int` before the operation.
- If either argument is of the `long` type, the other is also promoted to `long`, and the result is of the `long` type.

```
byte b1 = 1, b2 = 2, b3;
b3 = b1 + b2;      // ERROR: result is an int
                   // b3 is byte
```

ORACLE

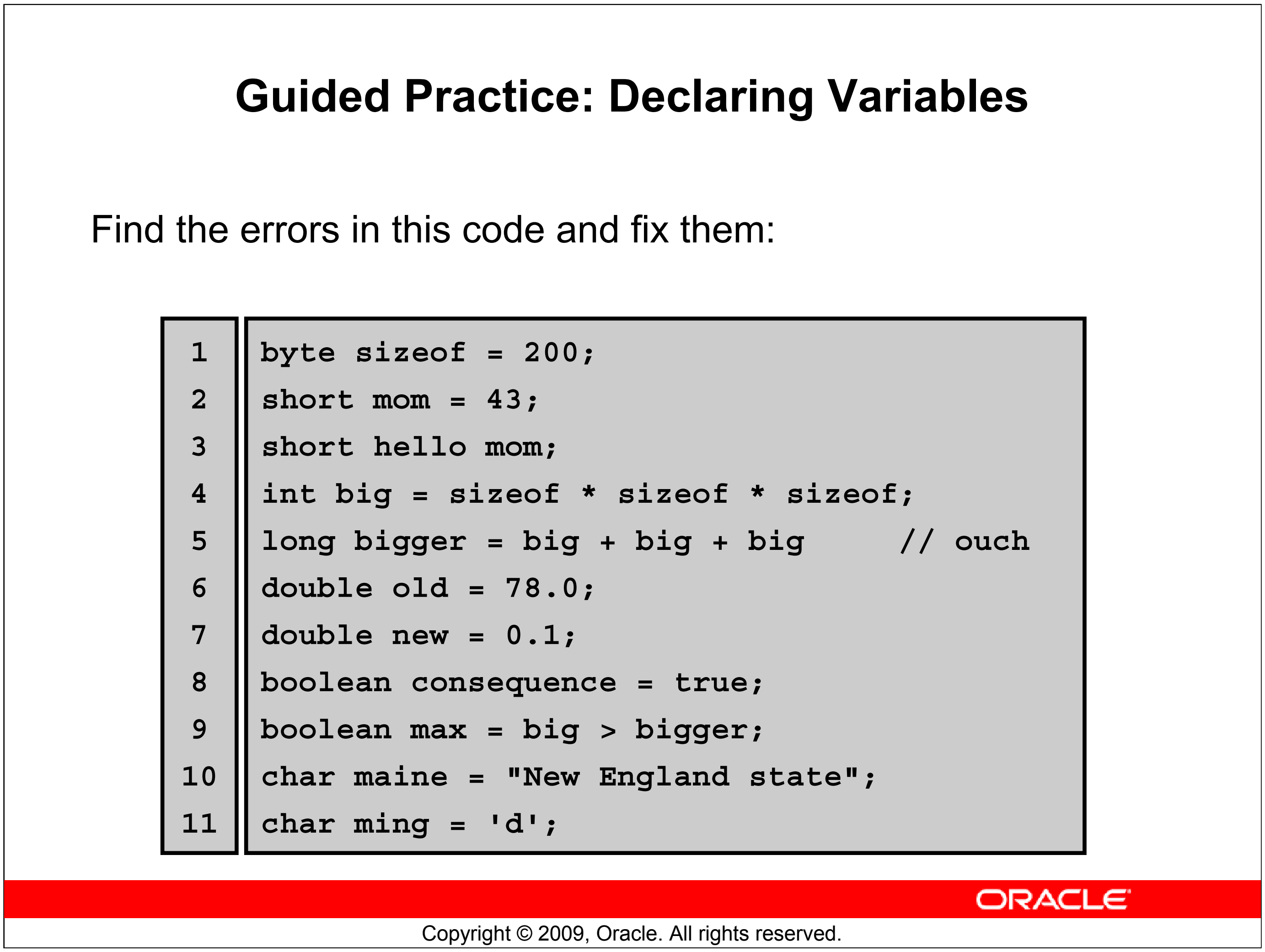
More on Arithmetic Operators

In Java, all integer arithmetic is performed with `int` or `long` values; `byte`, `char`, and `short` values are automatically widened (promoted) to `int` before an arithmetic operation begins, and the result is also `int`. Similarly, if the argument on one side of an arithmetic operator is a `long`, the argument on the other side is automatically promoted to a `long` as well, and the result is a `long`.

Consequently, if the result is to be assigned to a variable of a smaller type, the compiler signals an error.

Promoting `floats`

If an expression contains a `float`, the entire expression is promoted to `float`. All literal floating-point values are viewed as `double`.



Guided Practice: Declaring Variables

Find the errors in this code and fix them:

```
1   byte sizeof = 200;
2   short mom = 43;
3   short hello mom;
4   int big = sizeof * sizeof * sizeof;
5   long bigger = big + big + big     // ouch
6   double old = 78.0;
7   double new = 0.1;
8   boolean consequence = true;
9   boolean max = big > bigger;
10  char maine = "New England state";
11  char ming = 'd';
```

ORACLE

Guided Practice: Declaring Variables

This slide highlights some of the problems that you may encounter when declaring variables in Java. Hopefully, you will never write code like this. The following lines are illegal.

Line 1

The problem here is that `200` may not be assigned to a byte because `200` is treated as an `int`. The proper assignment is `byte b1 = (byte)200`. In this case, `b1 = –56`.

Line 3

A token is needed between `hello` and `mom`, such as an assignment operator to copy the value of `mom` into `hello`. It might be that both variables are of the same type. In that case, the only thing missing is a comma.

Guided Practice: Declaring Variables (continued)

Line 5

There is no statement terminator (;) at the end of the statement. The // ouch comment is valid.

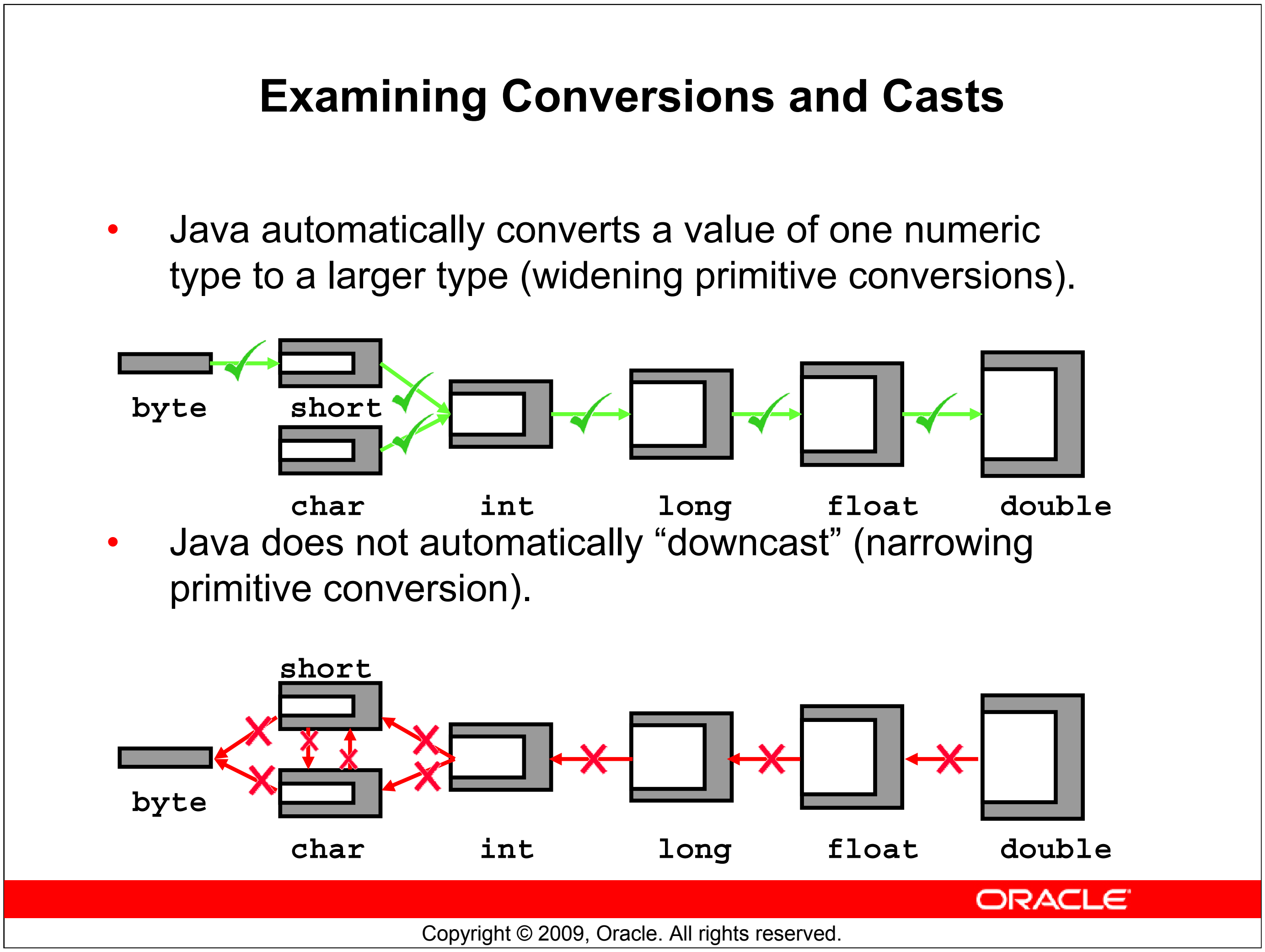
Line 7

This statement is illegal because new is a reserved word in Java.

Line 10

char variables can hold only a single character. Declare a String object instead:

```
String maine = "New England state";
```



Examining Conversions and Casts

- Java automatically converts a value of one numeric type to a larger type (widening primitive conversions).

- Java does not automatically “downcast” (narrowing primitive conversion).

Examining Conversions and Casts

The way to force one variable type to be stored as another variable type is called *casting*.

Java Can Convert Small Numeric Types into Wider Types Automatically

Java converts or casts a variable or expression from one numeric type to a wider type if necessary. For example, if you try to assign a `byte` to a `short`, the compiler converts the `byte` to a `short` before making the assignment.

Java Does Not Convert Large Numeric Types into Narrower Types Automatically

Java does not provide an automatic conversion from large numeric types to narrower types because this may result in loss of information. If you assign the value of a larger type to a smaller type, the compiler issues an error. However, you can force the compiler to convert a variable of one numeric type to a narrower type by using an explicit cast.

Examining Conversions and Casts (continued)

Casting takes the internal bit pattern of the source variable and places it in the target variable's type. This can result in a loss of data and unpredictable results. If you need to cast a variable, you must be aware of the possibility of data loss.

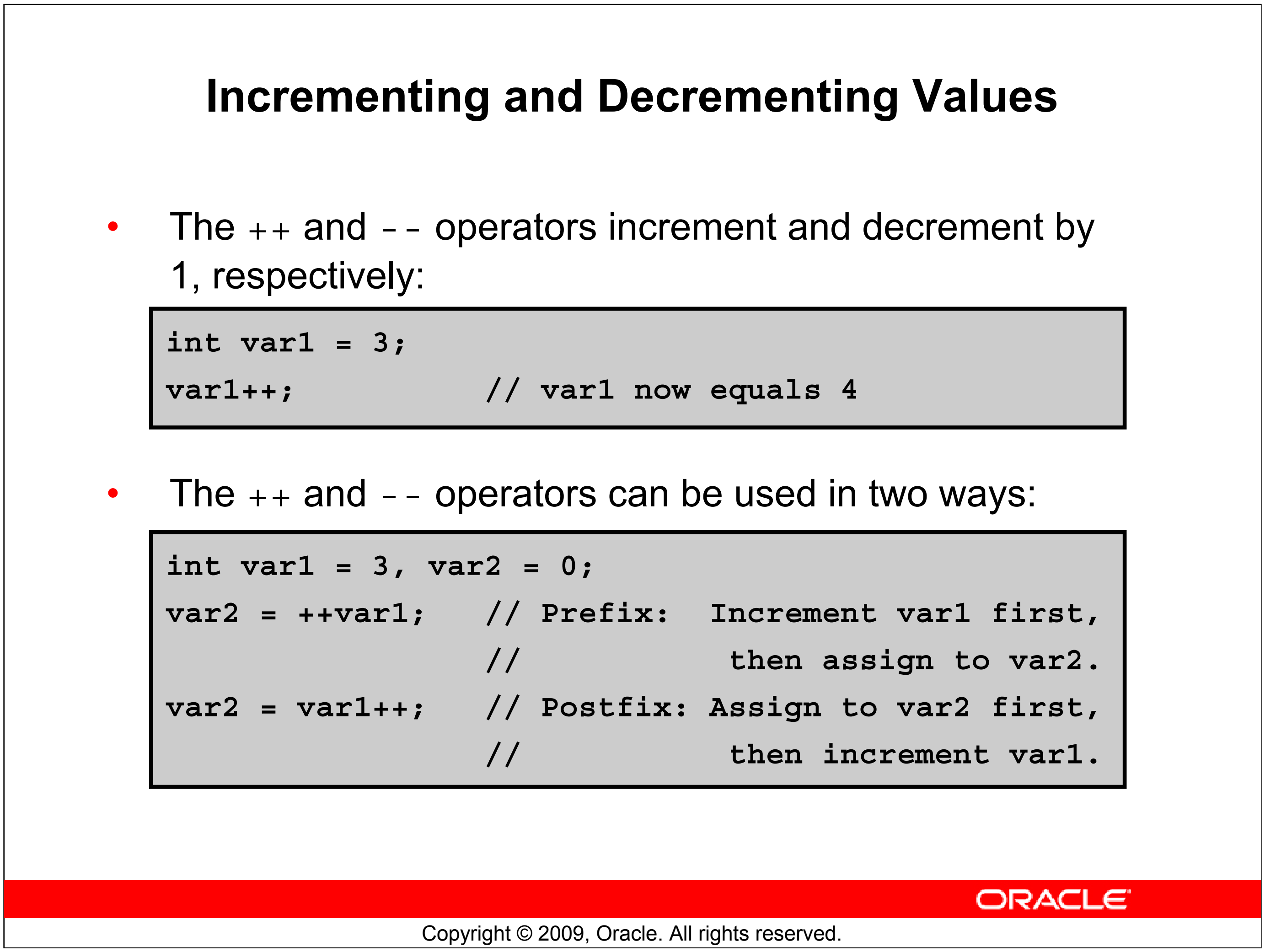
The syntax for an explicit cast is to put the target type in parentheses in front of the expression or variable, as in the following example:

```
byte b1 = 1, b2 = 2, b3;
b3 = b1 + b2; // error: the result is automatically int
b3 = (byte) (b1 + b2);
             // this corrects it but may
             // result in data loss
             // if the added value is
             // greater than positive 127
```

Remember that casts stop the compiler from performing useful checks on the validity of your code.

Casting and Arithmetic Operations

Be careful when performing arithmetic operations while casting to a smaller type. Note that performing a narrowing conversion, such as `int` to `byte`, may produce the wrong arithmetic result. For example, if you add a `byte` with the value `100` to another `byte` with the value `100`, the `int` result is `200`. However, if this result is cast to a `byte`, the value becomes `-56` because this is how the bit pattern for `200` is treated in a `byte`.



Incrementing and Decrementing Values

- The ++ and -- operators increment and decrement by 1, respectively:

```
int var1 = 3;
var1++;           // var1 now equals 4
```

- The ++ and -- operators can be used in two ways:

```
int var1 = 3, var2 = 0;
var2 = ++var1;    // Prefix:  Increment var1 first,
                  //           then assign to var2.
var2 = var1++;    // Postfix: Assign to var2 first,
                  //           then increment var1.
```

Incrementing and Decrementing Values

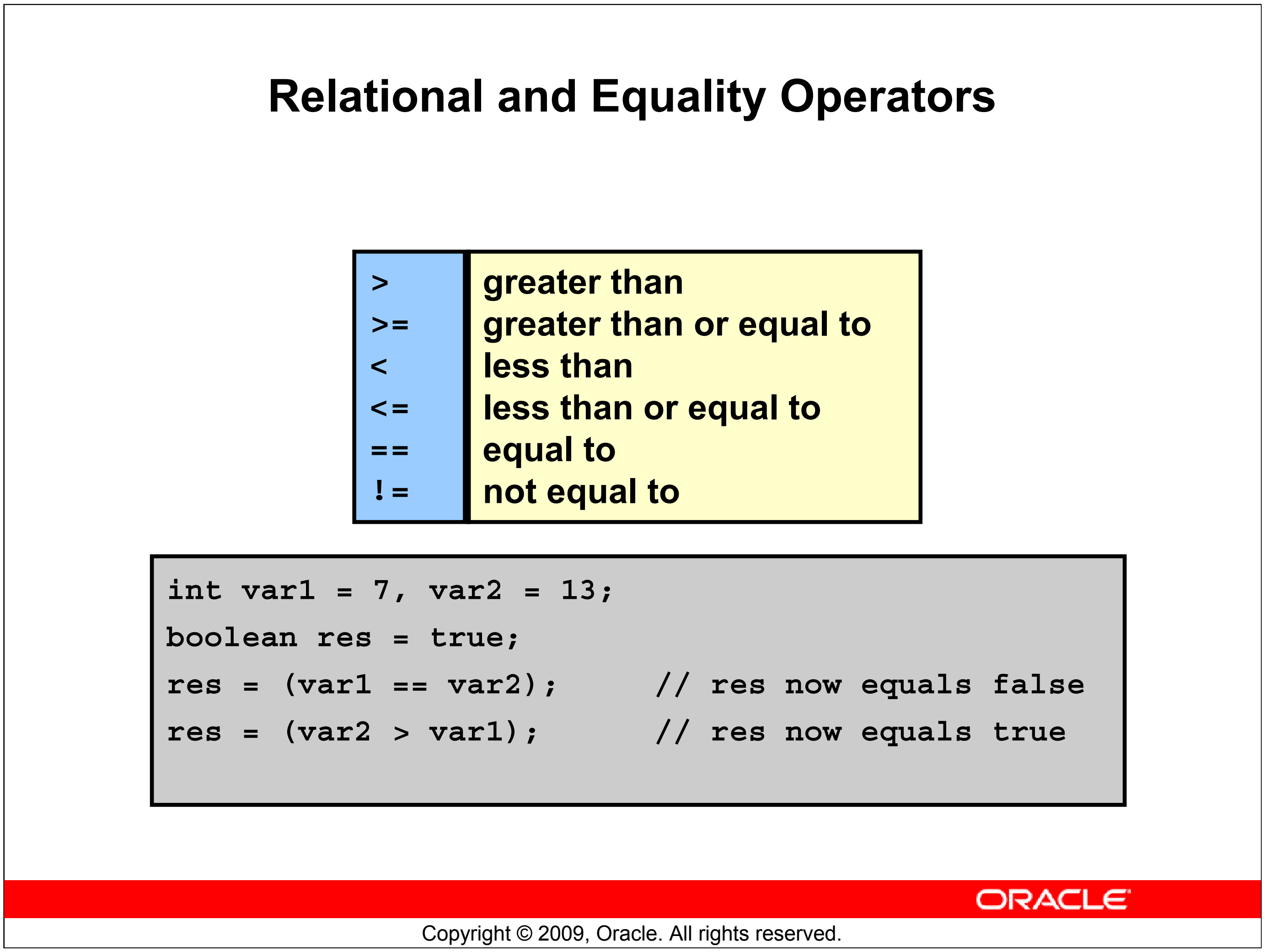
Incrementing or decrementing a value by one is a very common operation in any language. Like C and C++, Java provides special operators for this purpose. Both the increment (++) and decrement (--) operators can be prefixed or postfixed—that is, they can be placed before or after the operand. The placement of the operators affects when the operation takes place.

Prefixed Operator

When an increment or decrement operator is placed in front of a variable, the variable is incremented before any assignment operation. In other words, the value in the variable is adjusted by 1 (either up or down), and then it is used in the assignment.

Postfixed Operator

When the operator is placed after the variable, the variable is adjusted after the assignment operation. The value that is assigned is the value of the variable before it is incremented or decremented.



Relational and Equality Operators

>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
==	equal to
!=	not equal to

```
int var1 = 7, var2 = 13;
boolean res = true;
res = (var1 == var2);     // res now equals false
res = (var2 > var1);      // res now equals true
```

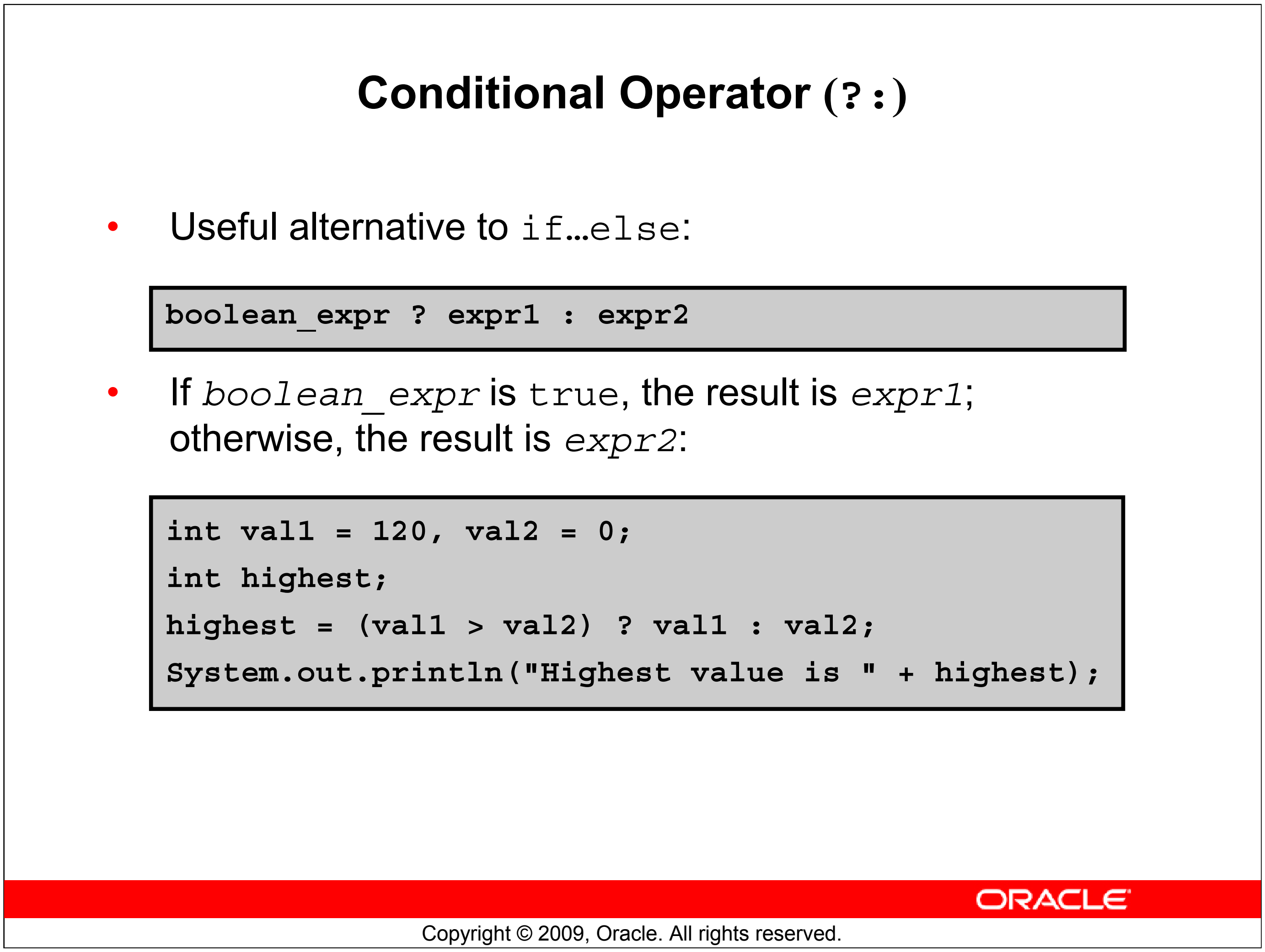
Relational and Equality Operators

Java provides a set of relational and equality operators for comparing the values of two variables or expressions. Unlike many languages (other than C and C++), the equality operator is the double equal sign ==. The inequality operator is `!=`.

Remember that the assignment operator (the equal sign =) sets the value of the variable on the left of the operator to the value of the expression or variable on the right of the operator.
By contrast, the double equal sign (==) tests for the equality of both sides, as in the following example:

```
int goodCreditRating = 3;
boolean goodCredit = false;
goodCredit = (custRating == goodCreditRating);
```

In the example, if the customer's rating is not `3`, then `goodCredit` is assigned `false`, which is the result of comparing the value of `goodCreditRating` with the customer's actual rating. If the actual rating was `3`, the result is `true`.



Conditional Operator (`? :`)

- Useful alternative to `if…else`:

```
boolean_expr ? expr1 : expr2
```

- If *`boolean_expr`* is `true`, the result is *`expr1`*; otherwise, the result is *`expr2`*:

```
int val1 = 120, val2 = 0;
int highest;
highest = (val1 > val2) ? val1 : val2;
System.out.println("Highest value is " + highest);
```

ORACLE

Conditional Operator (`? :`)

The conditional operator (`? :`) is the only ternary operator in Java, which means it has three operands. The conditional operator is an expression that returns a value and is a useful alternative to `if…else`.

The Boolean expression evaluates first and, if `true`, returns the value of the first expression (`expr1`) or, if `false`, returns the value of the second expression (`expr2`).

For example, consider the following example:

```
max = (10 > 8) ? 100: 200;
```

Because 10 is greater than 8, `max` is set to `100`. The values of the expressions (`100` and `200`) can be any valid Java expressions, including literals.

A similar effect can be achieved by using an `if…else` statement:

```
if (10 > 8)
  max = 100;
else
  max = 200;
```

Use whichever approach you find more intuitive.

Logical Operators

Results of Boolean expressions can be combined by using logical operators:

&&	&	AND (with or without short-circuit evaluation)
||	|	OR (with or without short-circuit evaluation)
^		exclusive OR
!		NOT

```
int var0 = 0, var1 = 1, var2 = 2;
boolean res = true;
highest = (val1 > val2)? val1 : val2;
res = !res;
```

ORACLE

Logical Operators

Boolean values and expressions that result in Boolean values can be combined by using the logical operators &, |, and !, which represent AND, OR, and NOT operations, respectively.

Short-Circuit Evaluation

The && and ||operators provide support for “short-circuit evaluation”; if the expression on the left of the operator has already determined the outcome of the whole logical expression, the expression on the right of the operator is not performed.

Consider the following example using the && operator:

```
        if (test1() && test2())
```

If test1 returns false, then there is no need to carry out test2 because a logical AND requires both tests to yield true; therefore, test2 is not performed.

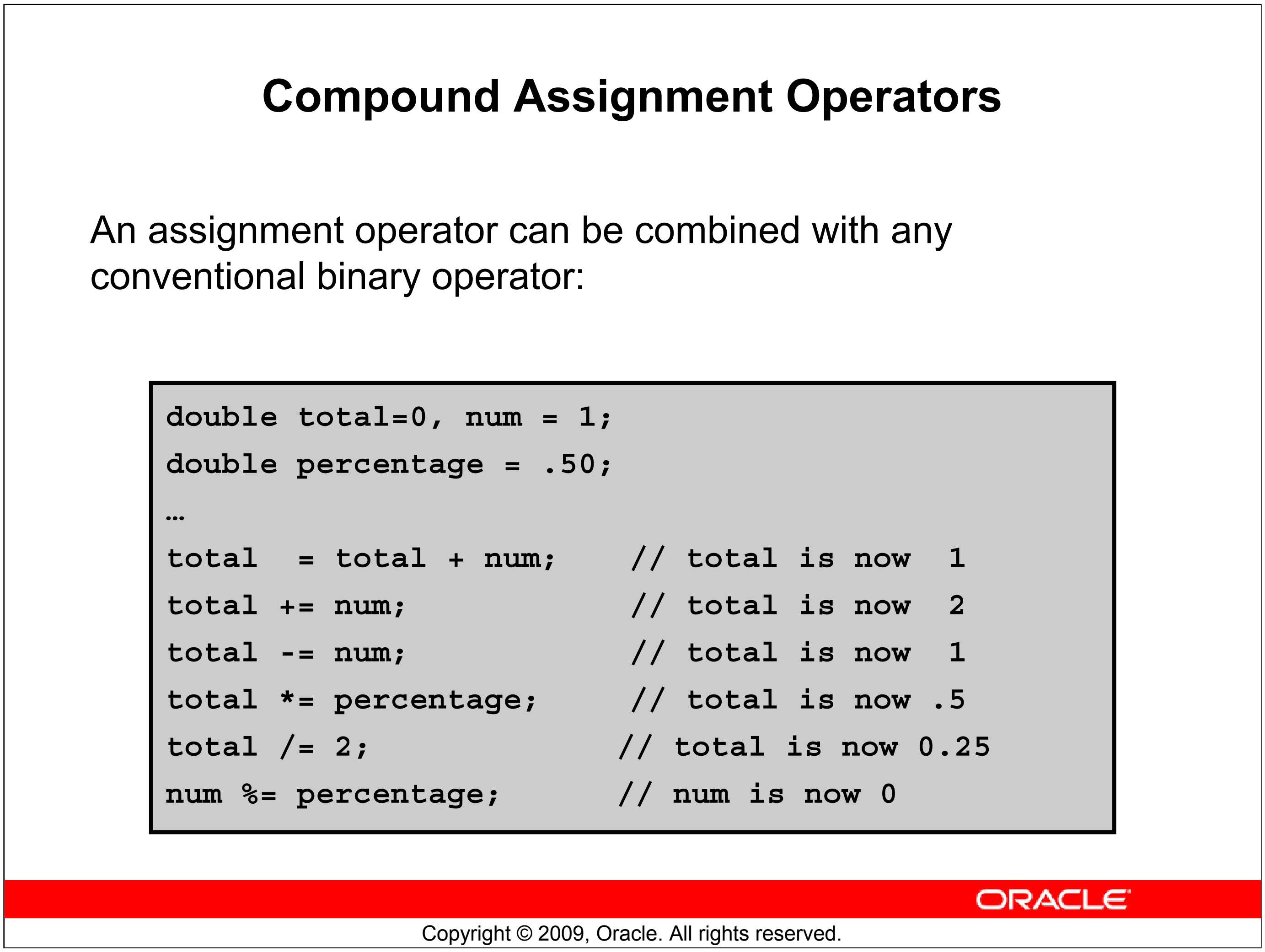
Likewise, consider the following example using the || operator:

```
        if (test1() || test1())
```

If test1 returns true, there is no need to carry out test2 because a logical OR requires only one of the tests to yield true; therefore, test2 is not performed.

Non-Short-Circuit Evaluation

If you have a Boolean expression involving two tests and you want the second test to be performed regardless of the outcome of the first test, use the & operator instead of &&, and the | operator instead of ||. Also, note that there is a ^ operator that performs the exclusive OR operation.



Compound Assignment Operators

An assignment operator can be combined with any conventional binary operator:

```
double total=0, num = 1;
double percentage = .50;
…
total  = total + num;     // total is now  1
total += num;             // total is now  2
total -= num;             // total is now  1
total *= percentage;      // total is now .5
total /= 2;              // total is now 0.25
num %= percentage;       // num is now 0
```

Compound Assignment Operators

Expressions such as `var1 = var1 + 20` are so common that Java provides compound assignment operators as a shorthand equivalent. You can form these compound operators by combining the assignment operator with any of the conventional *binary* operators. Binary operators are those operators that use two arguments, such as +, -, *, /, and so on.

For example, the expression:

```
    rentalDueDate = rentalDueDate + 3;
```

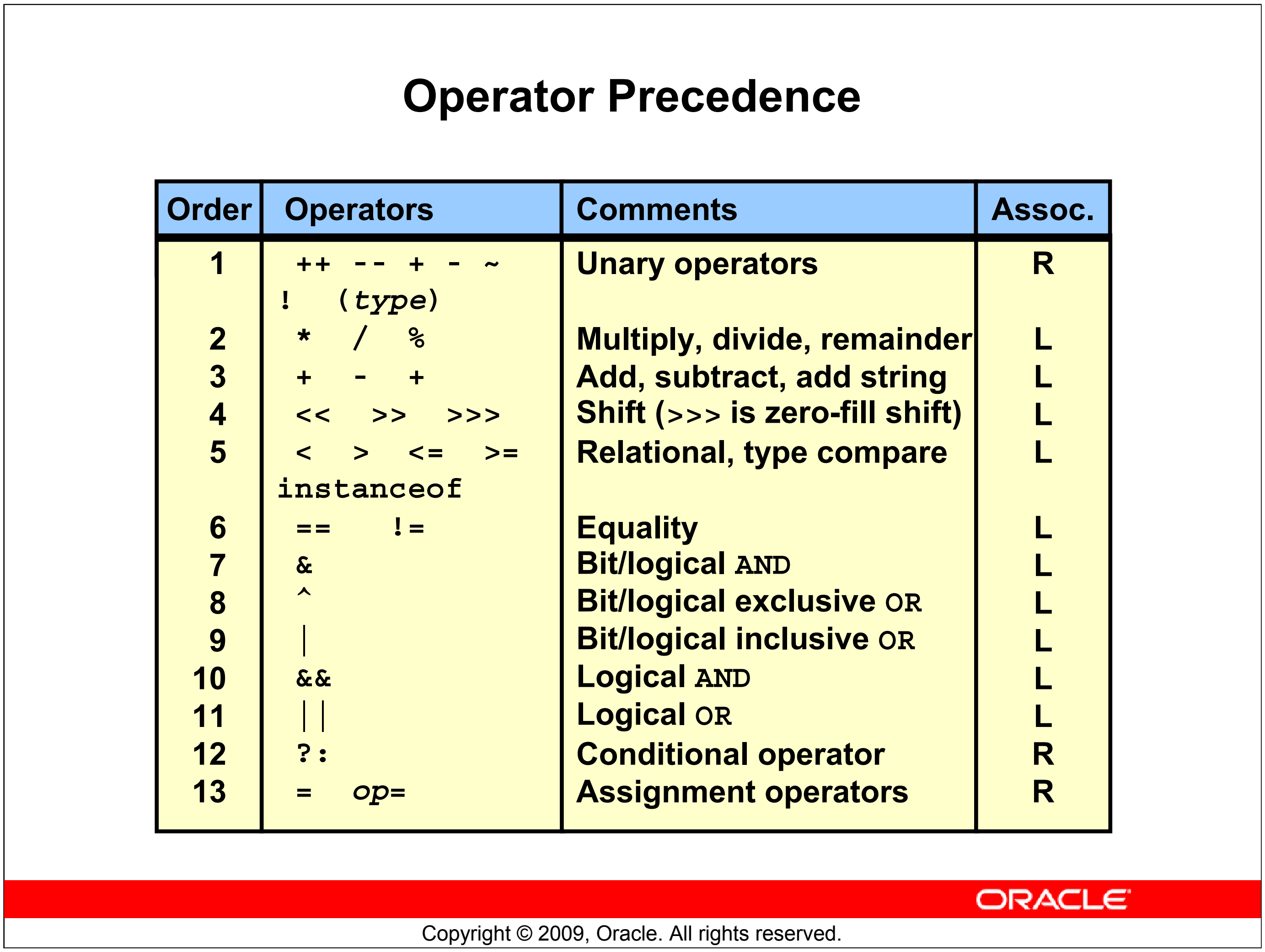
can be rewritten as follows:

```
    rentalDueDate += 3;
```

This expression takes the current value of `rentalDueDate` and adds `3`. It then places the result into `rentalDueDate`.

The `%=` operator computes the remainder of dividing the first variable by the second, and then assigns it to the first variable.

Note, however, that compound assignment operators may also be used with shift and bitwise operators.



Operator Precedence

Order	Operators	Comments	Assoc.
1	`++ -- + - ~ ! (type)`	Unary operators	R
2	`* / %`	Multiply, divide, remainder	L
3	`+ - +`	Add, subtract, add string	L
4	`<< >> >>>`	Shift (`>>>` is zero-fill shift)	L
5	`< > <= >= instanceof`	Relational, type compare	L
6	`== !=`	Equality	L
7	`&`	Bit/logical AND	L
8	`^`	Bit/logical exclusive OR	L
9	`|`	Bit/logical inclusive OR	L
10	`&&`	Logical AND	L
11	`||`	Logical OR	L
12	`?:`	Conditional operator	R
13	`=` *op=*	Assignment operators	R

Operator Precedence

Precedence refers to the order in which operators are executed. For example, multiplication is always performed before addition or subtraction. The table in the slide shows the Java operators in order of precedence, where row 1 has the highest precedence. With the exception of the unary, conditional, and assignment operators, which are right associative, operators with the same precedence are executed from left to right.

Associativity

Operators with the same precedence are performed in order according to their associativity. In the slide, the final column in the table shows the associativity for each operator:

- L indicates left-to-right associativity. Most operators are in this category.
- R indicates right-to-left associativity. Only the unary operators (row 1), the conditional operator (row 12), and the assignment operators (row 13) are in this category.

For example, consider the following statement:

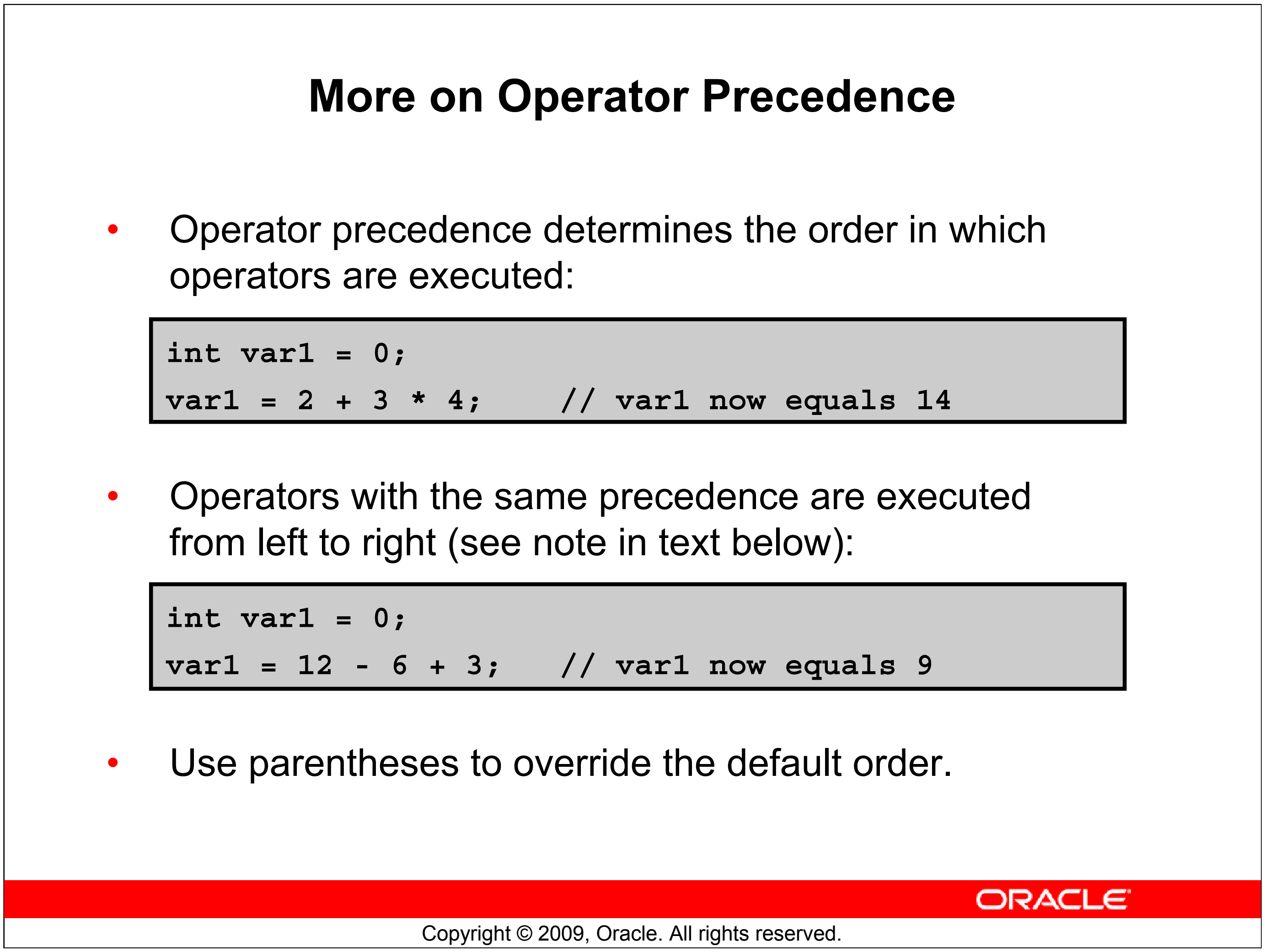
```
int j = 3 * 10 % 7;
```

The `*` and `%` operators have the same precedence but have left-to-right associativity. The `*` operation is thus performed first, as if parentheses had been used as follows:

```
int j = (3 * 10) % 7;  // Same result, 30%7, which is 2
```

If you choose to place parentheses in a different place, you obtain a different result:

```
int j = 3 * (10 % 7);// Different result, 3*3, which is 9
```



More on Operator Precedence

- Operator precedence determines the order in which operators are executed:

```
int var1 = 0;
var1 = 2 + 3 * 4;     // var1 now equals 14
```

- Operators with the same precedence are executed from left to right (see note in text below):

```
int var1 = 0;
var1 = 12 - 6 + 3;    // var1 now equals 9
```

- Use parentheses to override the default order.

Using the Precedence Table

The precedence table defines the order in which operators are evaluated. If you have a complex expression containing many operators, refer to the precedence table to make sure that the operators are being performed in the order that you expect.

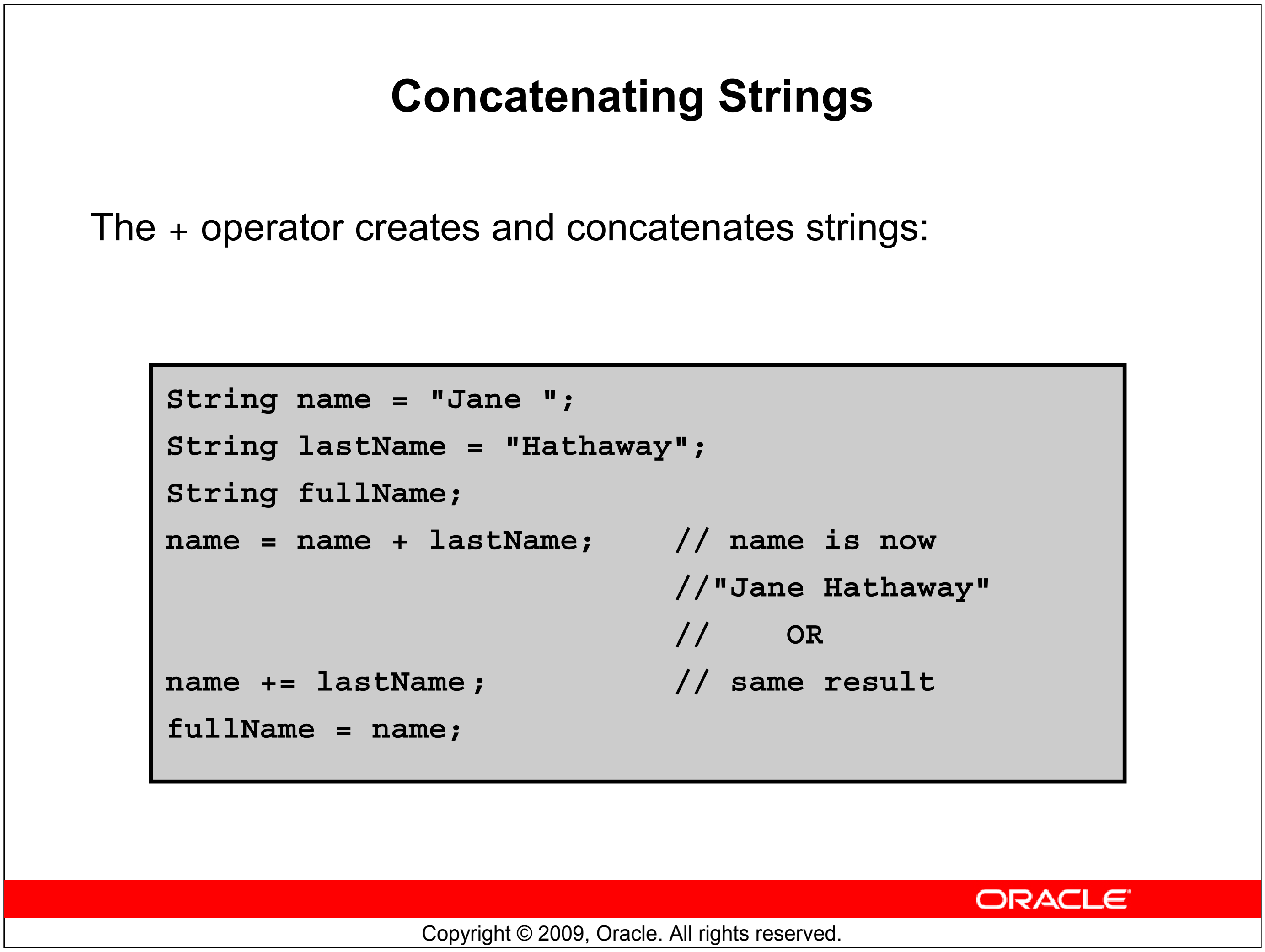
To make things simpler, you can use parentheses to override the default order in which operators are executed. Many programmers also use parentheses, not because the parentheses are strictly necessary but because they make the code more readable.

Note: The second bullet in the slide states that operators with the same precedence are evaluated from left to right. This is true for all operators except the unary operators, the conditional operator, and the assignment operators, which are evaluated from right to left as described on the previous page.

Using Parentheses

Use parentheses to control the order of statement evaluation exactly. Consider the following variations:

```
var1 = 12 - 6 + 3; // var1 now equals 9
var1 = 12 - (6 + 3);    // var1 now equals 3
```



Concatenating Strings

The + operator creates and concatenates strings:

```
String name = "Jane ";
String lastName = "Hathaway";
String fullName;
name = name + lastName;     // name is now
                            //"Jane Hathaway"
                            //    OR
name += lastName ;          // same result
fullName = name;
```

ORACLE

Concatenating Strings

Only three operators can be used with strings: the assignment operator (=), the addition operator (+), and the compound addition assignment operator (+=).

Using the Assignment Operator

You can use the assignment operator to create a new `String` object or to set the value of an existing string reference to refer to a `String` object:

```
String firstName = "John";
firstName = "John";
```

Using the Addition Operator

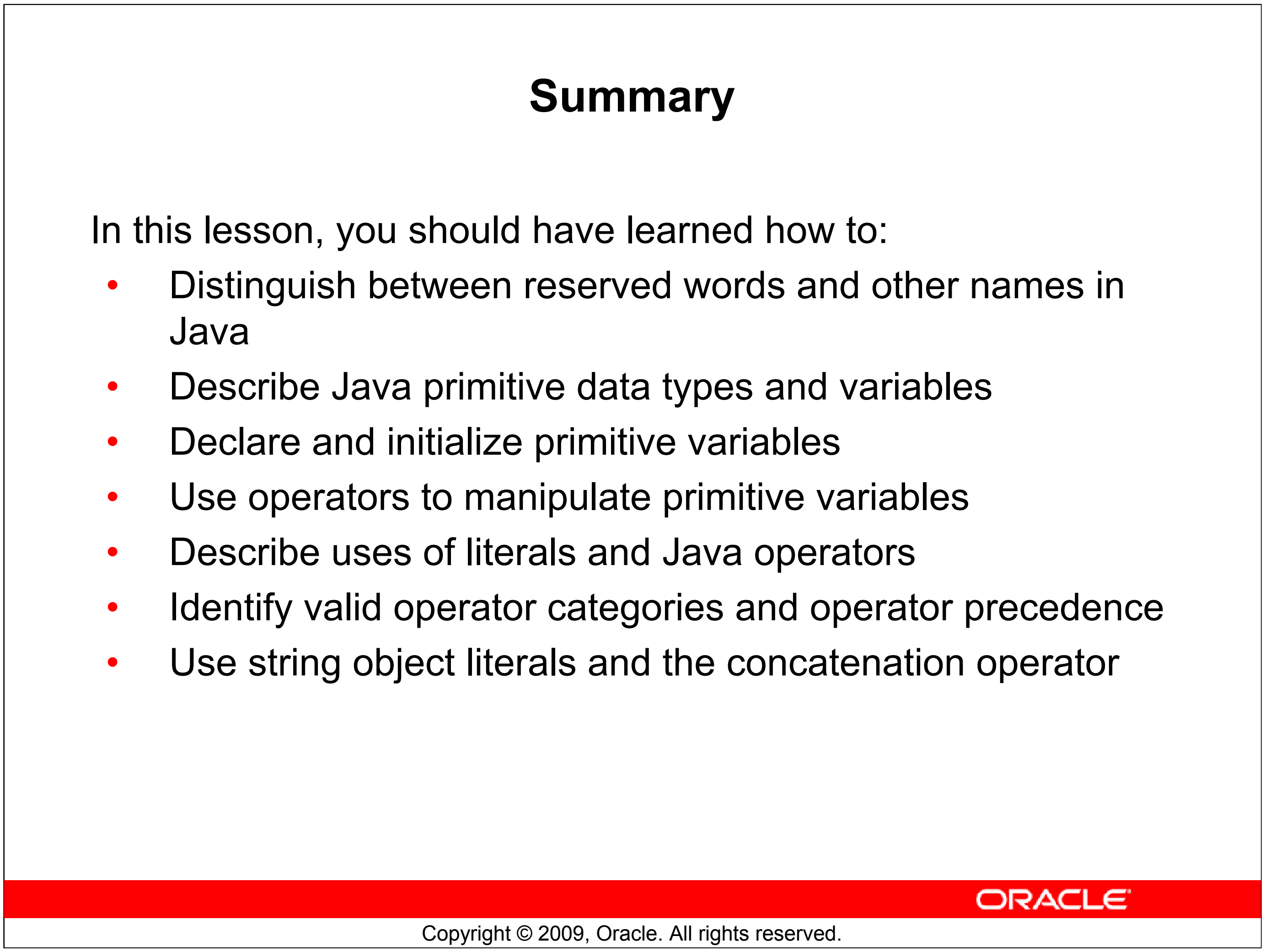
The addition operator is very useful with strings because it creates a new `String` object by concatenating the contents of two `String` objects. Consider the following code:

```
String firstName = "John";
String lastName = "Doe";
fullName = firstName + " " + lastName
fullName is now "John Doe".
```

If part of a `String` expression is not a `String`, Java converts that part into a `String` object and then concatenates its contents with the other `String` object to form a new `String` as a result.

Using the Compound Assignment Operator

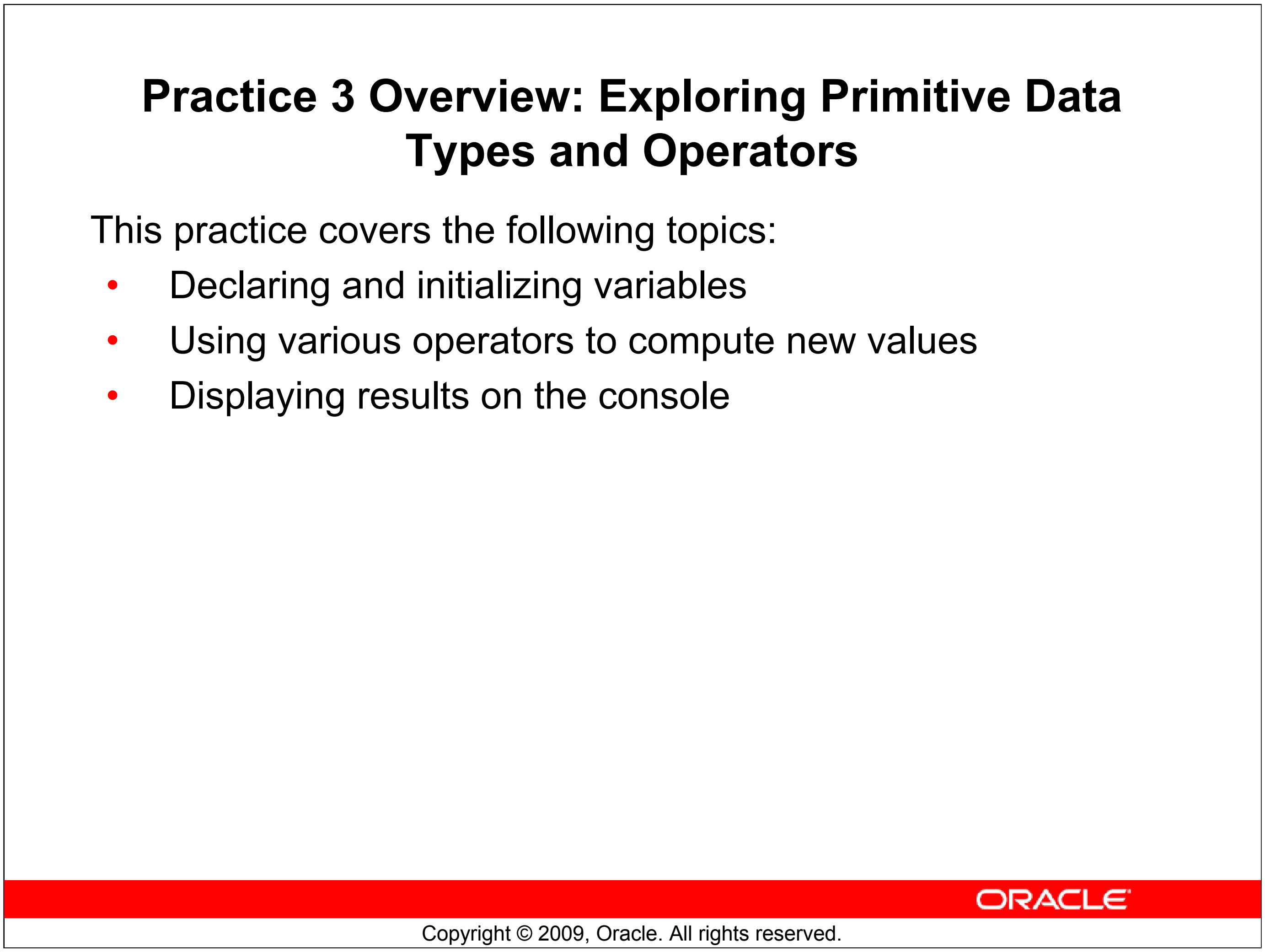
You can also use the += operator to concatenate a `String` to an existing `String`.



Summary

In this lesson, you should have learned how to:

- Distinguish between reserved words and other names in Java
- Describe Java primitive data types and variables
- Declare and initialize primitive variables
- Use operators to manipulate primitive variables
- Describe uses of literals and Java operators
- Identify valid operator categories and operator precedence
- Use string object literals and the concatenation operator



Practice 3 Overview: Exploring Primitive Data Types and Operators

This practice covers the following topics:

- Declaring and initializing variables
- Using various operators to compute new values
- Displaying results on the console

Practice 3 Overview: Exploring Primitive Data Types and Operators

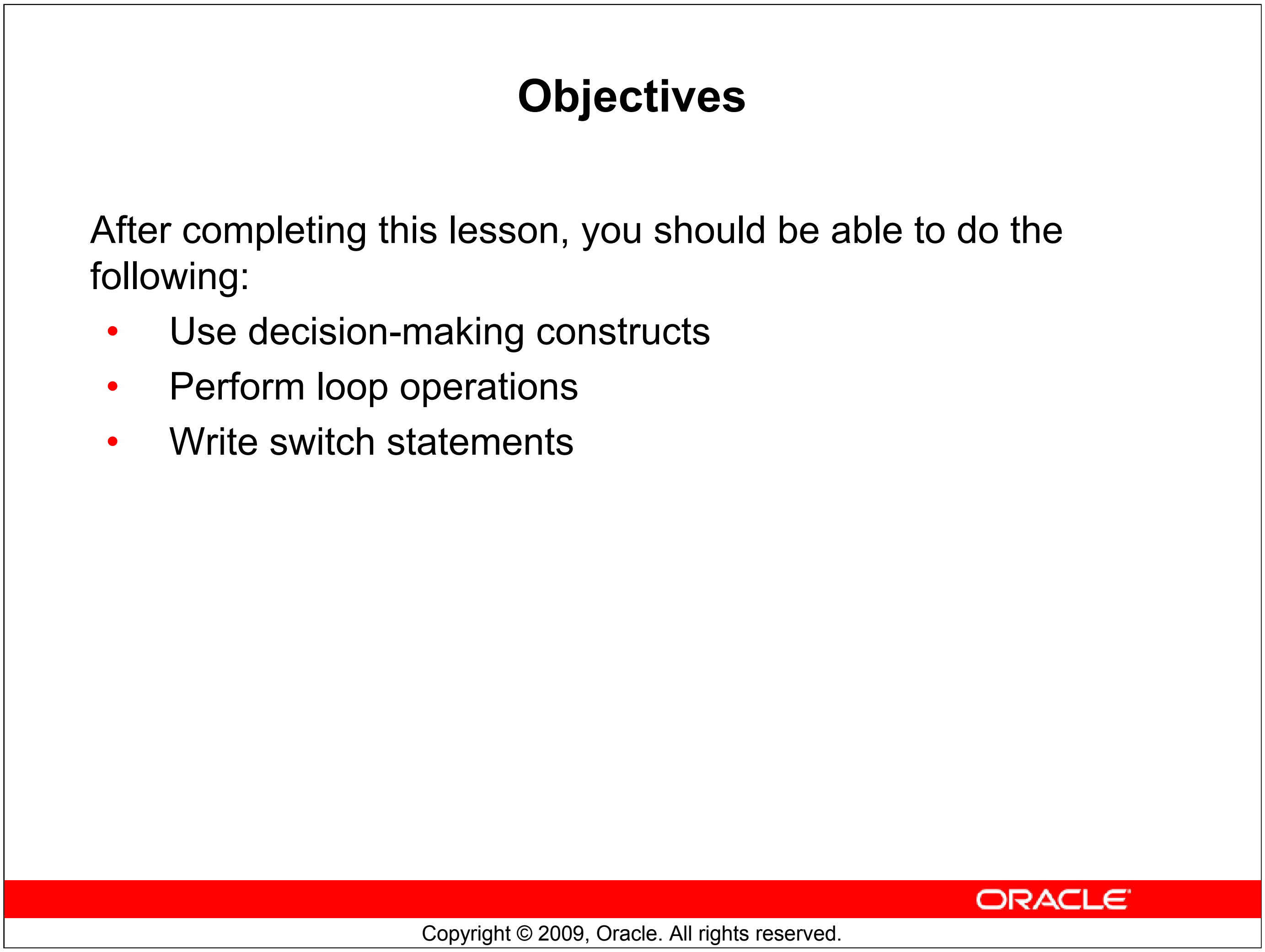
The goal of this practice is to declare and initialize variables and use them with operators to calculate new values. You also categorize the primitive data types and use them in code.

Note: If you have successfully completed the previous practice, you should continue using the same directory and files. If the compilation from the previous practice was unsuccessful and you want to move on to this practice, change to the `les03` directory and continue with this practice.

Remember that if you close a DOS window or change the location of the `.class` files, you must set the `CLASSPATH` variable again.

4

Controlling Program Flow



Objectives

After completing this lesson, you should be able to do the following:

- Use decision-making constructs
- Perform loop operations
- Write switch statements

Lesson Objectives

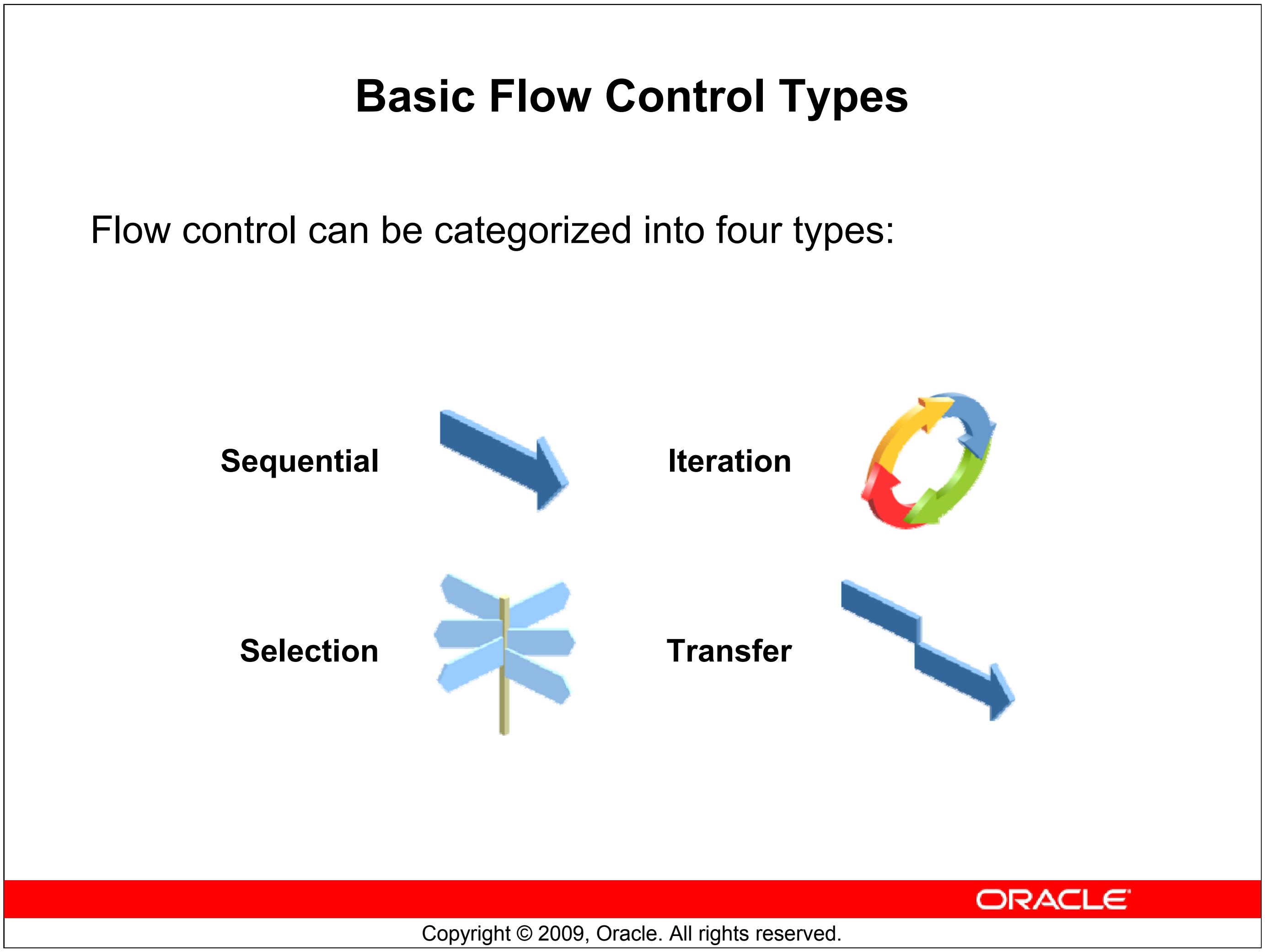
This lesson introduces Java decision-making and repetition constructs. You learn how to use these constructs to build nonlinear applications.

What Is Program Flow?

By default, computer programs start execution at, or near, the beginning of the code and move down the code until the end. This is fine if your program does one thing, does it always in the same order, and never needs to deviate from this path. Unfortunately, this does not happen very often in most businesses.

Program control constructs are designed so that the programmer can design and build programs that perform certain parts of code conditionally. There are also constructs so that code can be executed repetitively. These coding structures give you infinite control of what, when, and how many times your program performs a particular task.

By providing standard control mechanisms, Java gives programmers control over the exact execution order of their program code.



Basic Flow Control Types

Flow control can be categorized into four types:

Sequential

Iteration

Selection

Transfer

Basic Flow Control Types

Sequential

Sequential is the flow control type in which the program flow follows a simple sequential path, executing one statement after another. The primary sequential structure is a compound block statement, which is a series of statements inside braces.

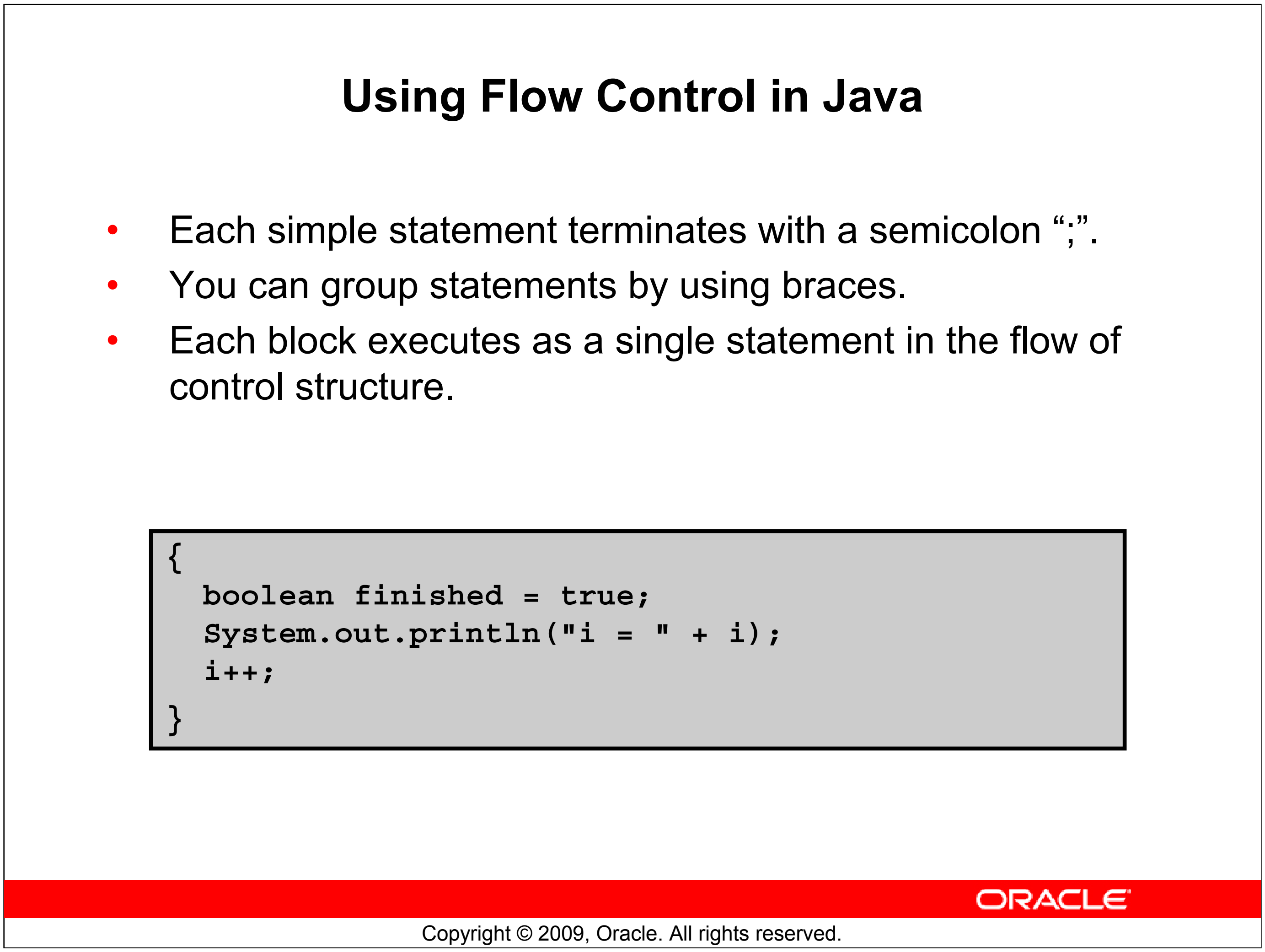
Selection

Selection is the flow control type in which only one path out of a number of possibilities is taken.

Simple selection involves the conditional execution of a statement or block of code, which is guarded by an expression that has the value `true` if the guarded code is to be executed. This is the `if` statement.

An `if…else` statement provides an alternate path of execution: the true or false evaluation of a control expression determines which branch is taken.

A `switch` statement supports a multiway branch based on the value of a control expression.



Using Flow Control in Java

- Each simple statement terminates with a semicolon “;”.
- You can group statements by using braces.
- Each block executes as a single statement in the flow of control structure.

```
{
  boolean finished = true;
  System.out.println("i = " + i);
  i++;
}
```

Using Flow Control in Java

Simple Statements

A simple statement is any expression that terminates with a semicolon, as in the following examples:

```
    var1 = var2 + var3;
    var3 = var1++;
    var3++;
```

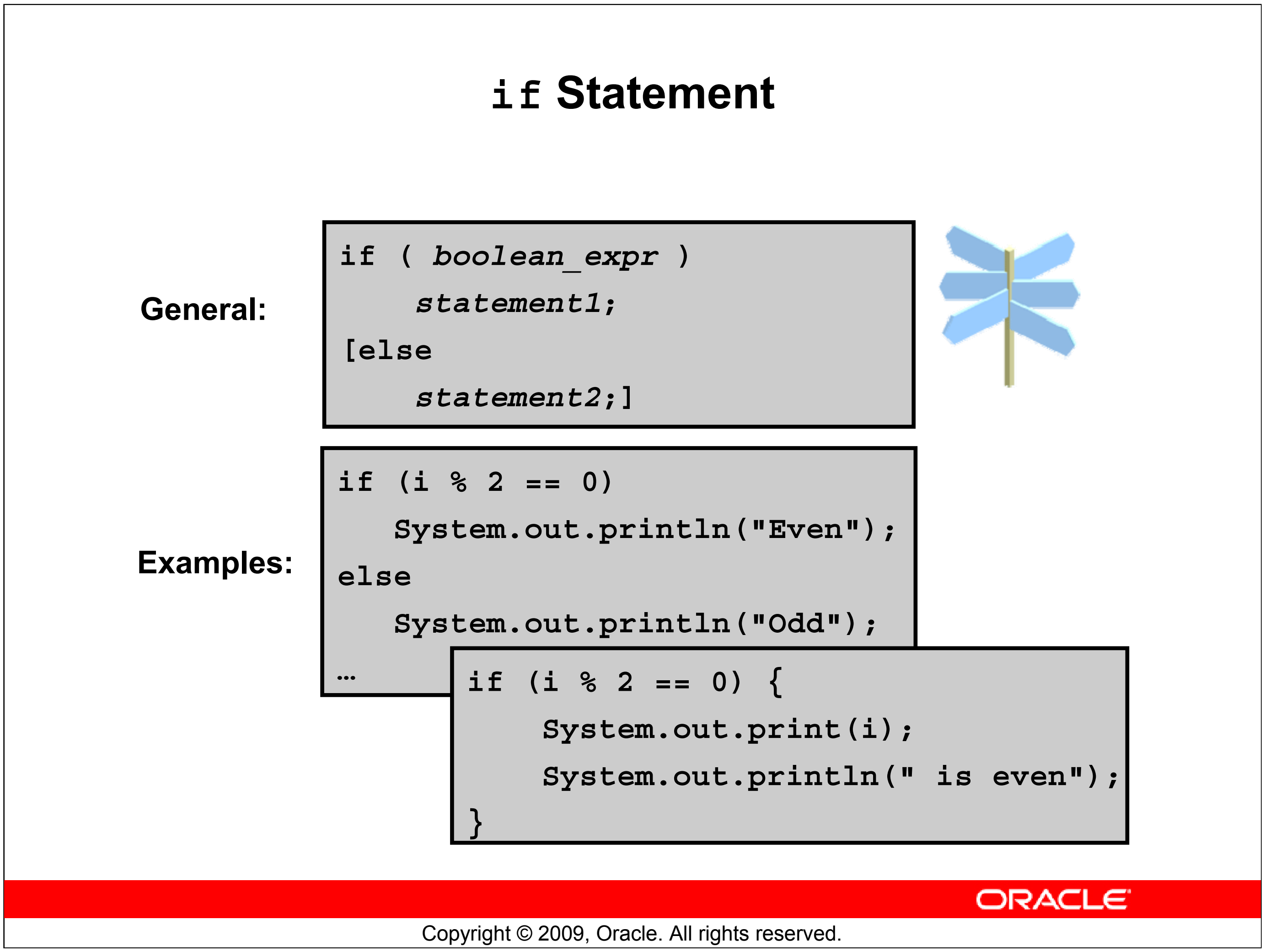
Compound Statements (Blocks)

Related statements can be grouped in braces to form a compound statement or block:

```
     {
         int i;
        boolean finished = true;
        System.out.println("i = " + i);
        i++;
      }
```

Semantically, a block behaves like a single statement and can be used anywhere a single statement is allowed. There is no semicolon after the closing brace. Unlike PL/SQL, Java uses no matched block delimiters, such as `if` and `end if`. Any variables that are declared in a block remain in scope until the closing brace. After the block is exited, the block variables cease to exist.

Blocking improves the readability of program code and can help make your program easier to control and debug.



if Statement

General:

```
if ( boolean_expr )
     statement1;
[else
     statement2;]
```

Examples:

```
if (i % 2 == 0)
   System.out.println("Even");
else
   System.out.println("Odd");
…
```

```
if (i % 2 == 0) {
    System.out.print(i);
    System.out.println(" is even");
}
```

if Statement

The if statement provides basic selection processing. A Boolean control expression determines which branch is taken, as follows:

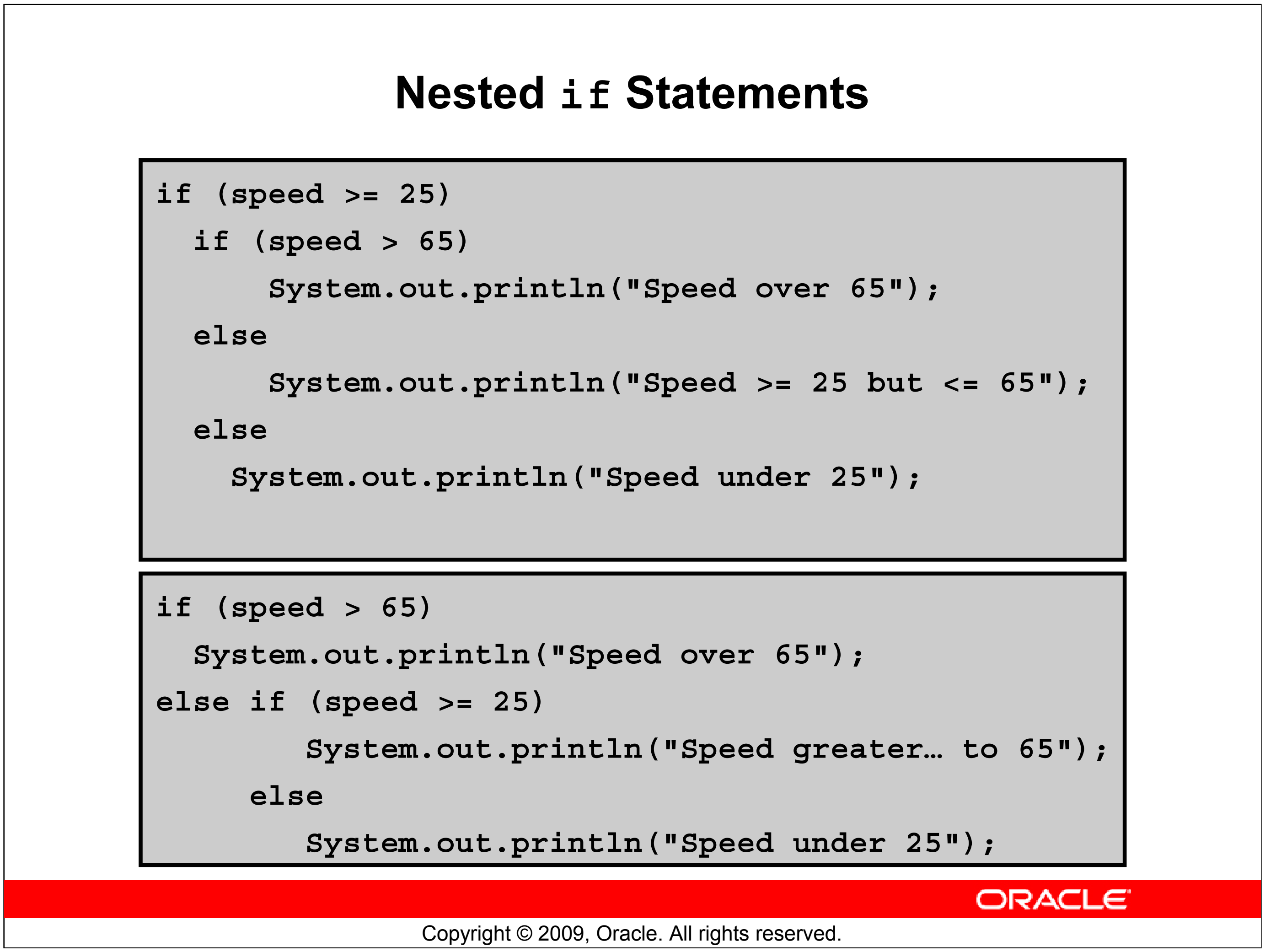
- If the expression evaluates to true, the first branch is taken—that is, the if body is executed.
- If the expression evaluates to false, the second branch is taken—that is, the else body is executed. The else clause is optional; if it is omitted, nothing is executed if the control expression evaluates to false.

Example

```
if (orderIsPaid) {
     System.out.println("send with receipt");
}
else {
          System.out.println("collect funds");
}
```

Common Mistake When Using if Statements

Use the equality operator (==) rather than the assignment operator (=) in the control expression.



Nested `if` Statements

```
if (speed >= 25)
  if (speed > 65)
      System.out.println("Speed over 65");
  else
      System.out.println("Speed >= 25 but <= 65");
  else
    System.out.println("Speed under 25");
```

```
if (speed > 65)
  System.out.println("Speed over 65");
else if (speed >= 25)
        System.out.println("Speed greater… to 65");
     else
        System.out.println("Speed under 25");
```

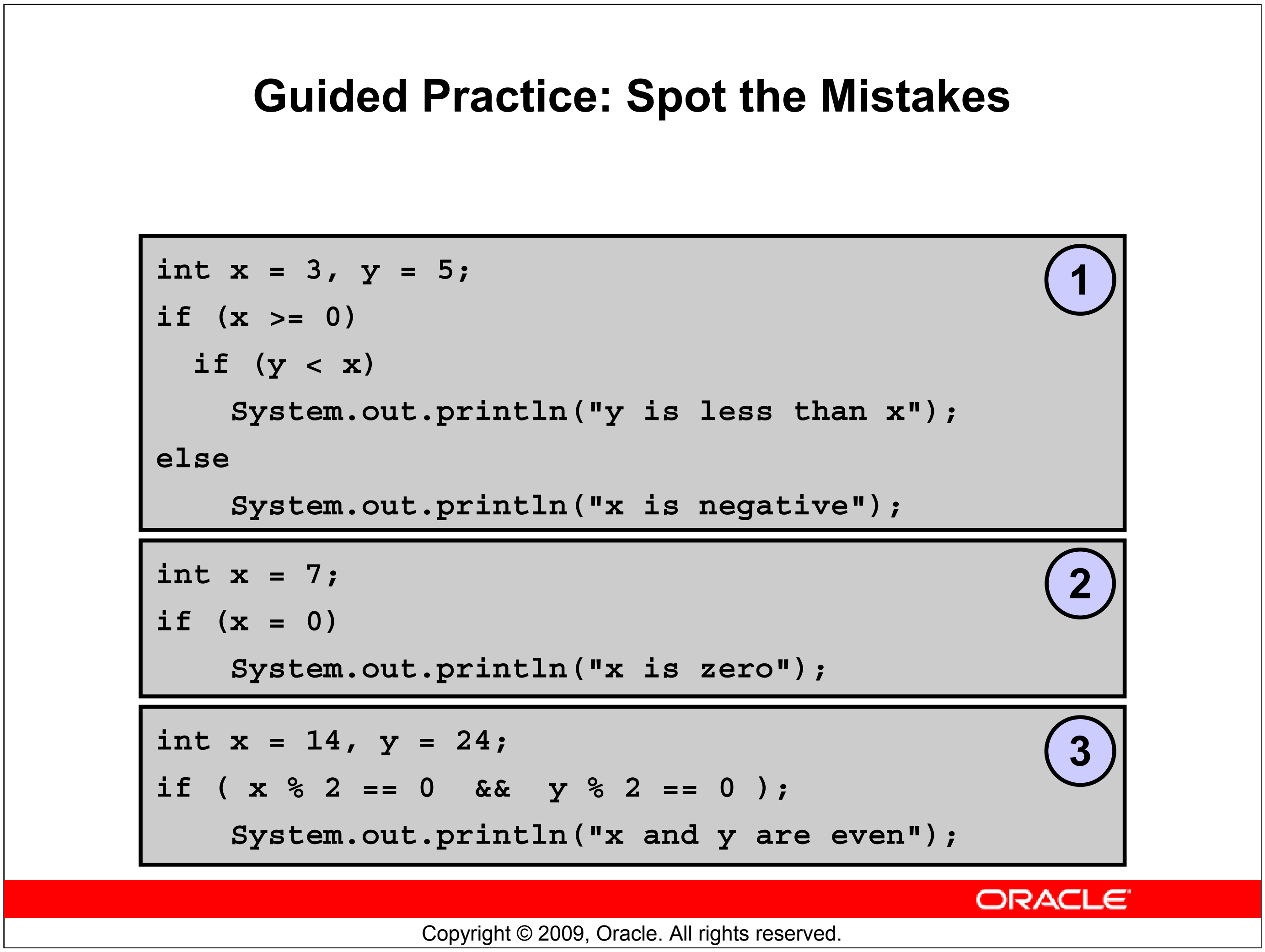
Nested `if` Statements

If multiple tests are necessary, `if` statements can be nested. However, this approach is not recommended because you have to maintain a mental stack of the decisions that are being made. This becomes difficult if you have more than three levels of nested `if` statements.

Also, it is very easy to forget that an `else` clause always binds to the nearest `if` statement above it that is not already matched with an `else`, even if the indentation suggests otherwise. This is sometimes referred to as the “dangling else” problem.

`if…else…if` Construct

The “dangling else” problem can be solved with a prudent use of braces, but a cleaner approach is to use the `if…else… if` construct, as shown in the second example in the slide. Note that these are two separate keywords; unlike some languages, Java does not have an `elseif` keyword.



Guided Practice: Spot the Mistakes

1

```
int x = 3, y = 5;
if (x >= 0)
  if (y < x)
    System.out.println("y is less than x");
else
    System.out.println("x is negative");
```

2

```
int x = 7;
if (x = 0)
    System.out.println("x is zero");
```

3

```
int x = 14, y = 24;
if ( x % 2 == 0  &&  y % 2 == 0 );
    System.out.println("x and y are even");
```

Guided Practice: Spot the Mistakes

Example 1

Use braces to associate or bind statements and make the code easier to follow:

```
if (x >= 0) {
   if (y < x)
      System.out.println("y is less than x");}
   else
      System.out.println("x is negative");
```

It is the last `if` that pairs with an `else`. Although it does not become a compiler error, it becomes an error in logic.

Example 2

The second example uses an assignment operator (=) rather than an equality operator (==) in the `if` test. Fortunately, Java compilers detect this mistake and indicate a compiler error because the expression in the `if` test must evaluate to a Boolean.

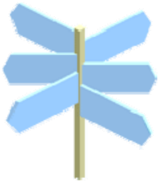
Example 3

The third example has an extra semicolon at the end of the `if` test. This is not a compiler error. The compiler treats the semicolon as an empty `if` body, as follows:

```
if ( x % 2 == 0  &&  y % 2 == 0 )
   ;    // Null "if" body
```

switch Statement

```
switch ( integer_expr ) {

    case constant_expr1:
        statement1;
        break;
    case constant_expr2:
        statement2;
        break;
    [default:
        statement3;]
}
```



- The switch statement is useful when selecting an action from several alternative integer values.
- Integer_expr must be byte, int, char, or short.

switch Statement

The switch statement provides a clean way to dispatch to different sections of your code, depending on predefined values. It can be used to choose among many alternative actions based on the value.

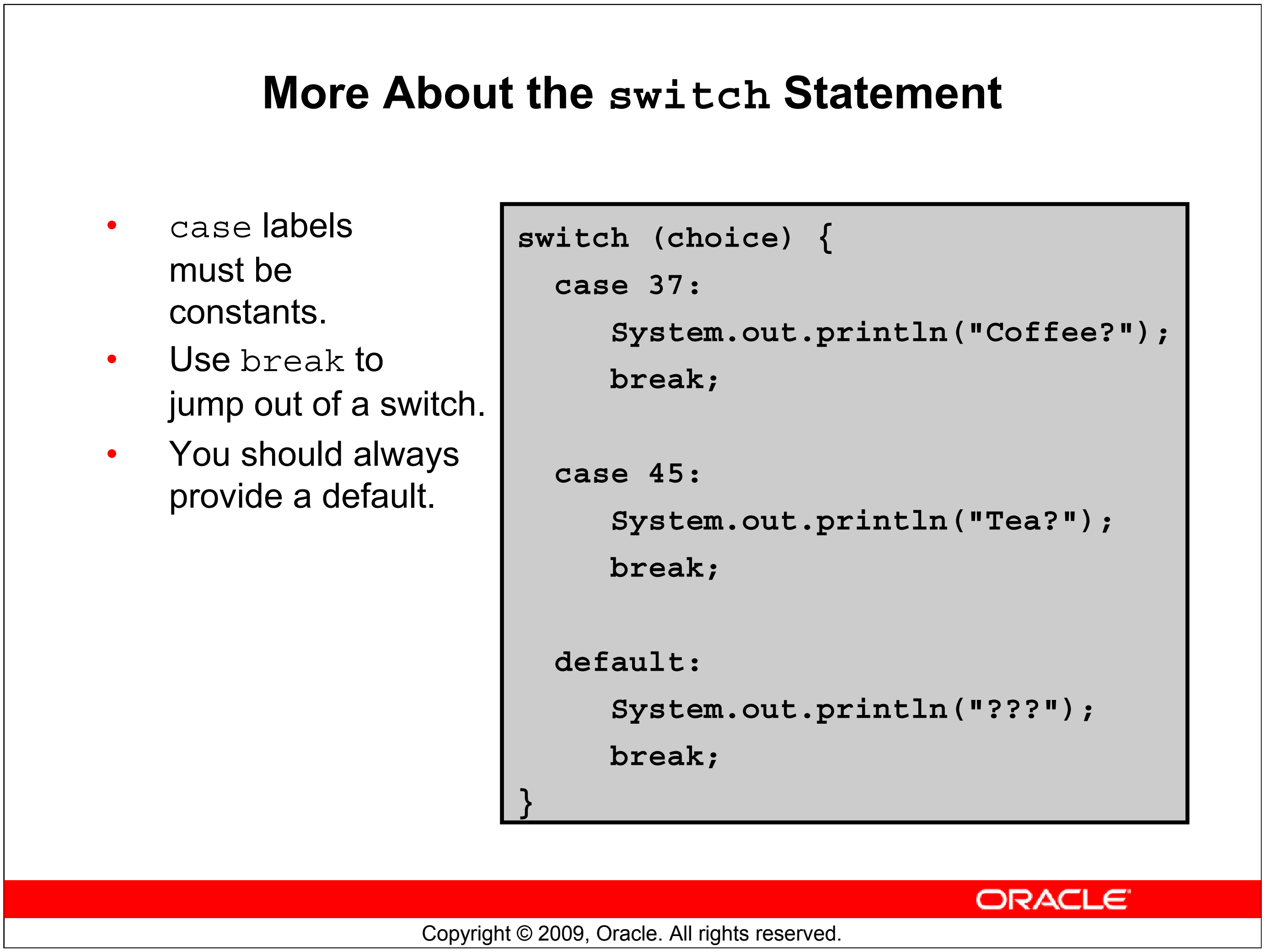
Anatomy of the switch Statement

The switch statement is useful when selecting an action from several alternatives. The value inside the test expression must be a byte, char, short, or int. It cannot be a boolean, long, double, float, String, or any other kind of object.
The value inside the test expression is compared with the case labels, which are constant expressions.

- If a match is found, the statements following the label are executed. Execution continues until a break is encountered, which transfers control to the statement following the switch statement.
- If no match is found, control passes to the statements following the default label. The default label is optional; if no default label is provided, the switch statement does nothing when no match is found. It is a good practice always to provide a default label, even if no action is required.

case Labels

The case labels must be constant expressions that are known at compile time. You can use either literal numbers or final variables (constants).



More About the `switch` Statement

- `case` labels must be constants.
- Use `break` to jump out of a switch.
- You should always provide a default.

```
switch (choice) {
  case 37:
     System.out.println("Coffee?");
     break;

  case 45:
     System.out.println("Tea?");
     break;

  default:
     System.out.println("???");
     break;
}
```

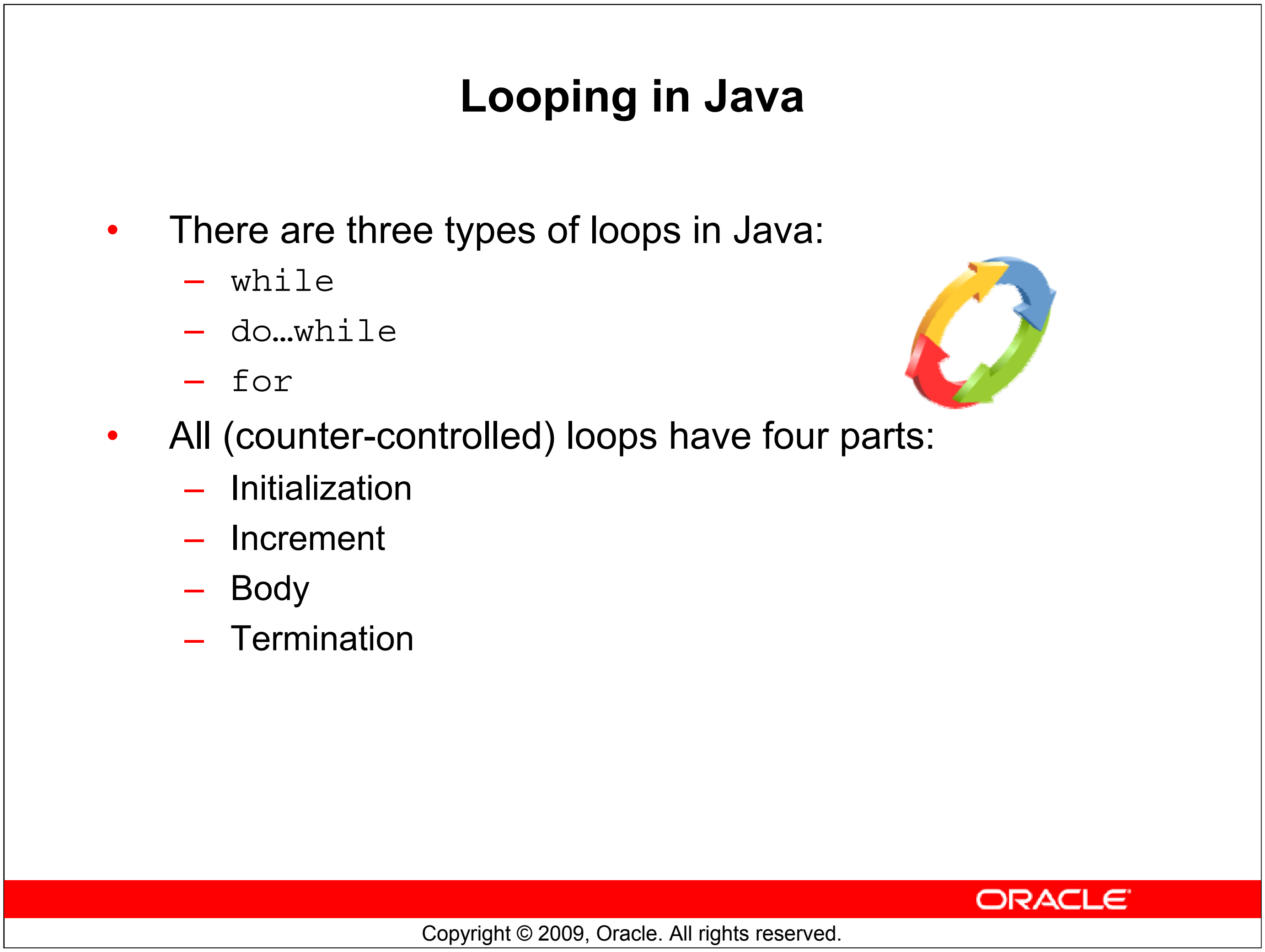
More About the `switch` Statement

The slide shows a simple example of a `switch` statement. There are situations in which falling through can be useful. To fall through, simply do not include a break, as in the following example:

```
char c = 'b';
switch (c) {
  case 'a': System.out.println("First letter in alphabet");
         break;
  case 'b': System.out.println("Second letter in alphabet");
  case 'c': System.out.println("Third letter in alphabet");
  case 'd': System.out.println("Fourth letter in alphabet");
}
```

In this example, the printed result displays the following lines because of the absence of a `break` statement between each `case` test for `'b'`, `'c'`, and `'d'`:

```
Second letter in alphabet
Third letter in alphabet
Fourth letter in alphabet
```



Looping in Java

- There are three types of loops in Java:
  - `while`
  - `do…while`
  - `for`
- All (counter-controlled) loops have four parts:
  - Initialization
  - Increment
  - Body
  - Termination

Looping in Java

What Is Looping?

Looping in any programming language refers to repeatedly executing a block of code until a specified condition is met. Java provides three standard loop constructs: `while`, `do…while`, and `for`.

All counter-controlled loops have four parts: initialization, increment, statement body, and termination.

Initialization: Initialization sets the initial conditions of the loop, including any variable that may be incremented to control the execution of the loop.

Increment: The increment expression is invoked at each iteration and is used to control the loop execution. It is possible to increment or decrement a value.

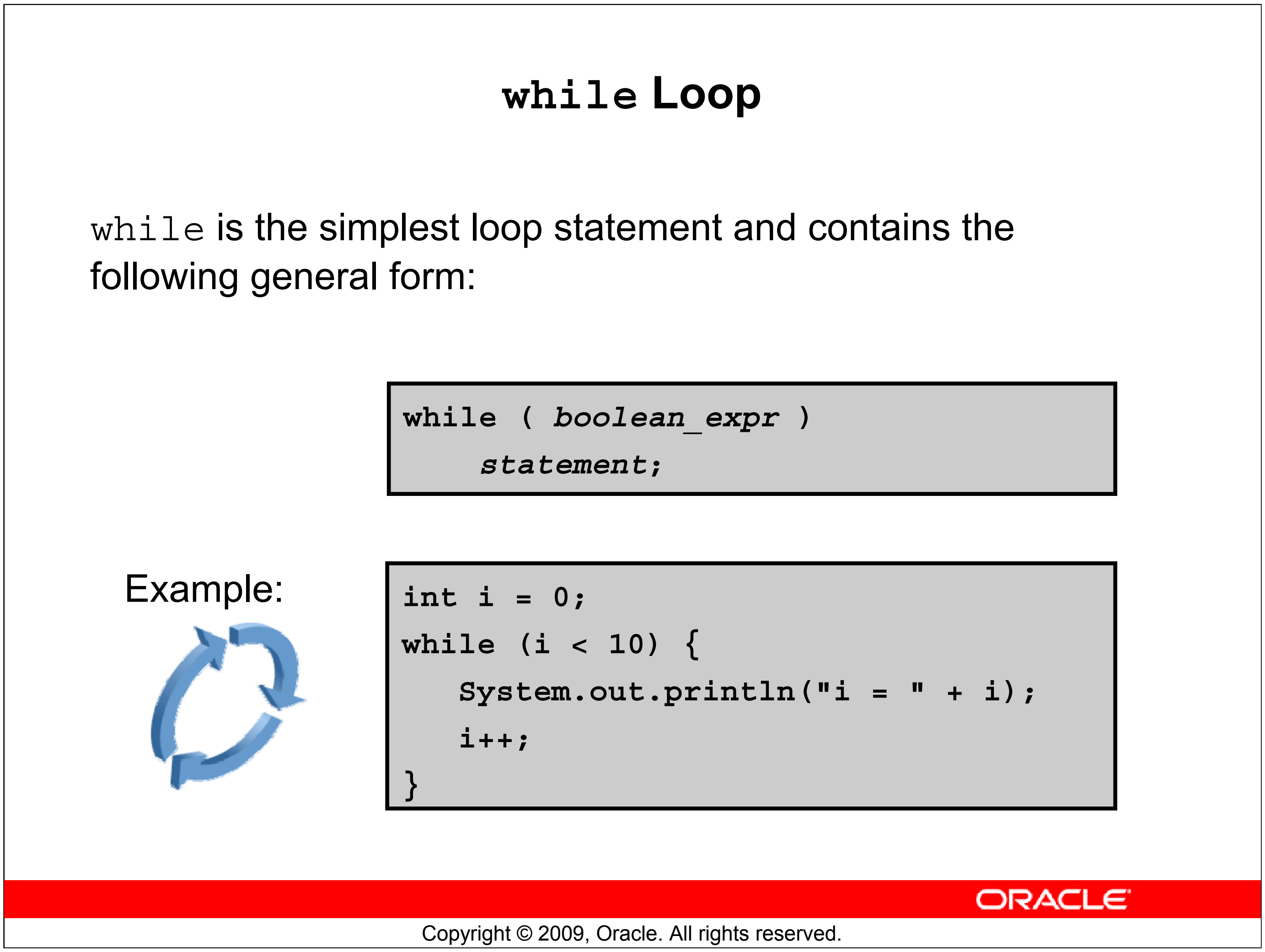
Looping in Java (continued)

Body: The body is executed if the termination condition is `true`.

Termination: Termination is the expression that is evaluated to determine whether the body must be executed. The expression must be a Boolean expression. If the expression evaluates to `true`, the body is executed; if it is `false`, the body is not executed.

Choosing the appropriate loop

Use the `while` loop to ensure that the termination condition is tested before executing the body of the loop. Use `do`…`while` to ensure that the body executes once before the termination condition is made. The `for` loop is similar to the `while` loop.



while Loop

while is the simplest loop statement and contains the following general form:

```
while ( boolean_expr )
    statement;
```

Example:

```
int i = 0;
while (i < 10) {
   System.out.println("i = " + i);
   i++;
}
```

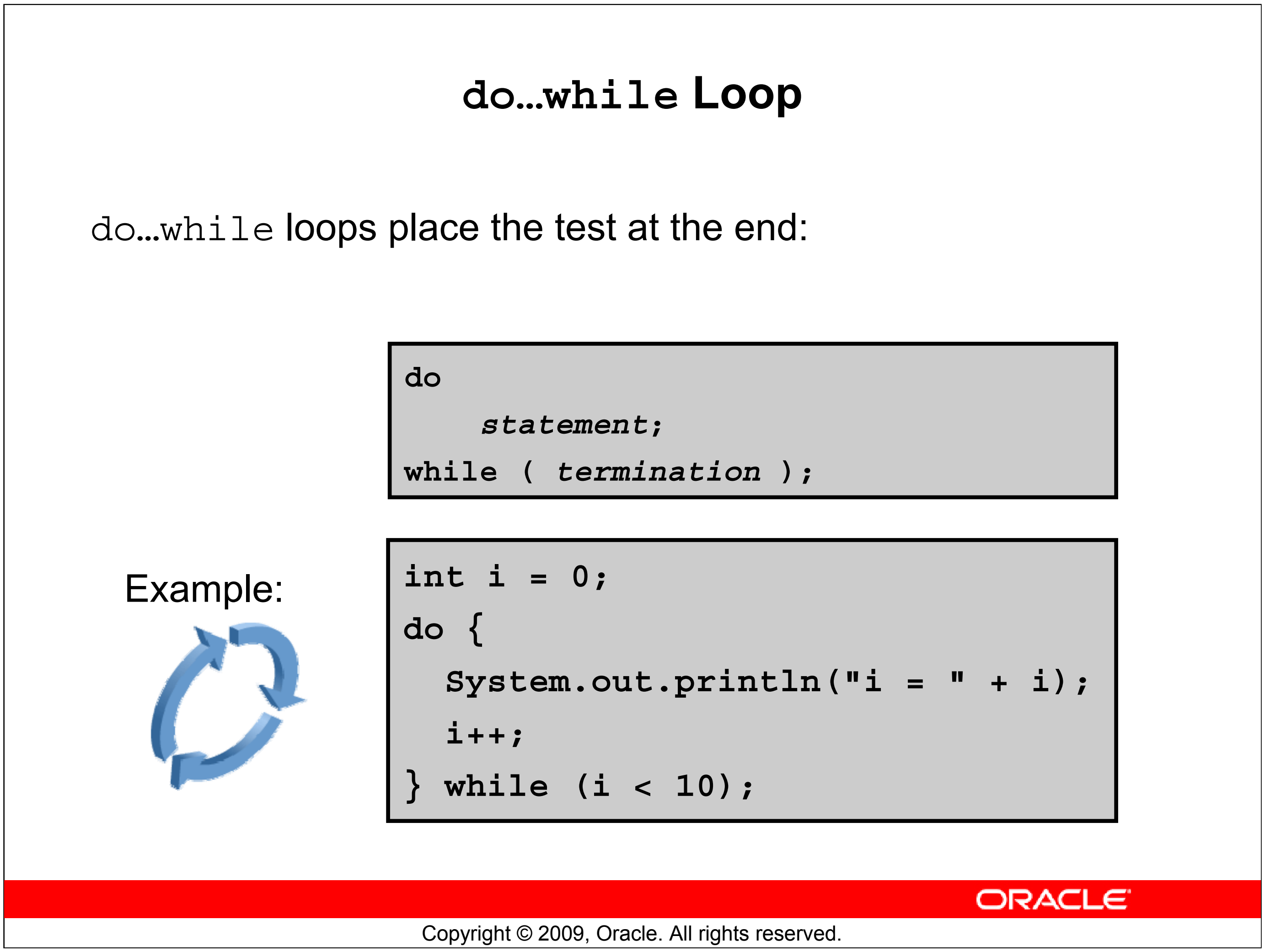
while Loop

The simplest loop construct in Java is the `while` loop. In the `while` loop, the loop body is executed repeatedly while a `boolean` control expression evaluates to `true`. When the control expression evaluates to `false`, the loop terminates and control passes to the first statement after the closing brace of the loop body.

Note that the control expression is evaluated before the loop body is executed. If the control expression evaluates to `false` the first time, it does not enter the loop body.

Common Mistakes When Using while Loops

A common mistake when using `while` loops is to forget to increment the variable that is used in the control expression. This results in an infinite loop. Another common mistake is to put a semicolon at the end of the first line, between the control expression and the loop body. This places the incrementing of the counter outside the loop and results in an infinite loop.



do…while Loop

do…while loops place the test at the end:

```
do
    statement;
while ( termination );
```

Example:

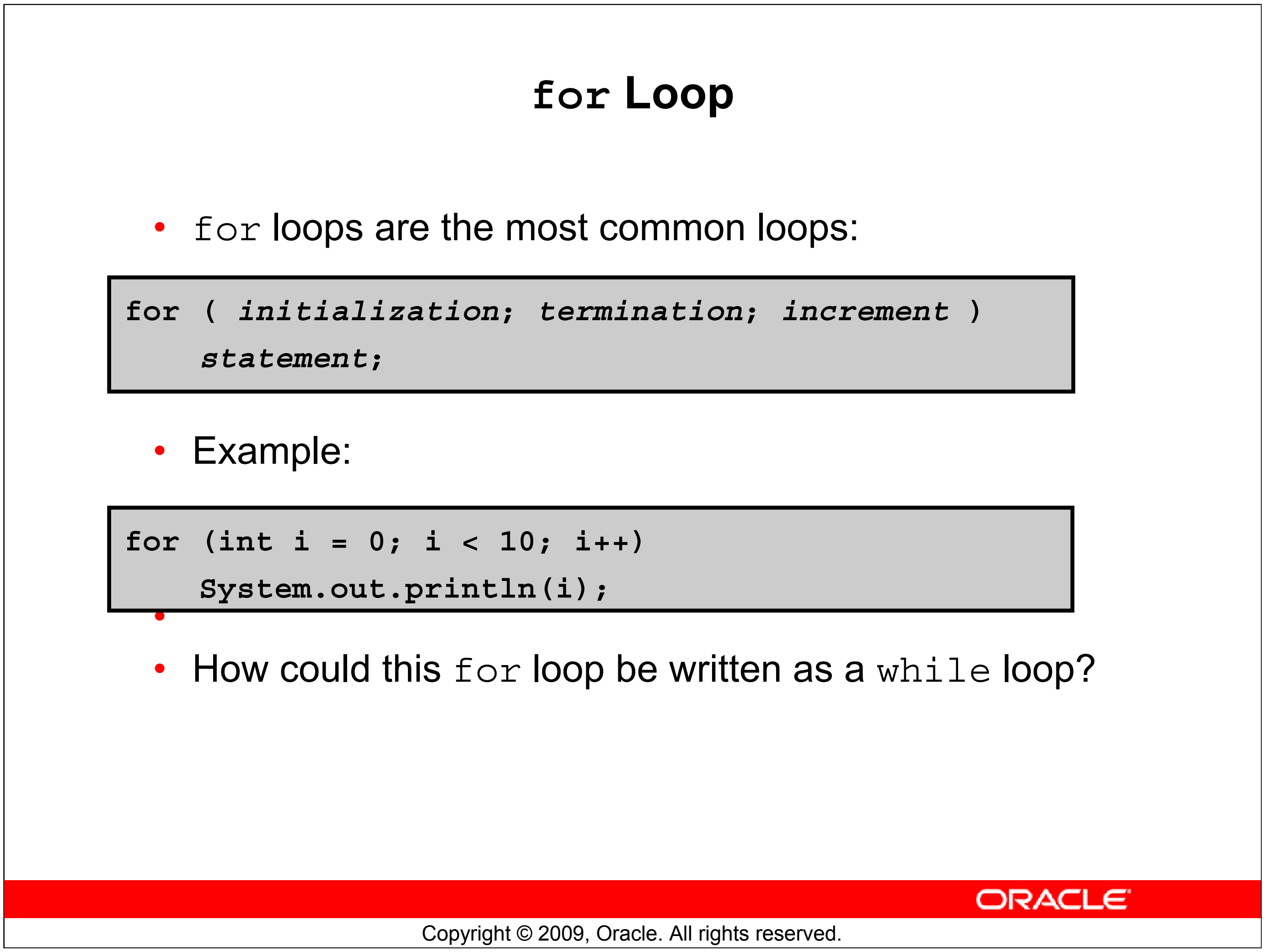
```
int i = 0;
do {
  System.out.println("i = " + i);
  i++;
} while (i < 10);
```

do…while Loop

The do…while loop is similar to the while loop except that the control expression is evaluated after the loop body has been executed. This guarantees that the loop body executes at least once. This construct is useful if you want to execute the statement at least once without regard to the while condition.

Common Mistakes When Using do…while Loops

A common mistake when using do…while loops is to forget one of the braces of a compound statement or to forget the final semicolon. The compiler detects both types of errors.



for Loop

- for loops are the most common loops:

```
for ( initialization; termination; increment )
    statement;
```

- Example:

```
for (int i = 0; i < 10; i++)
    System.out.println(i);
```

- How could this for loop be written as a while loop?

for Loop

The for loop combines components of a loop into a single construct with positions for each component. The three components that are enclosed in parentheses are separated by semicolons.

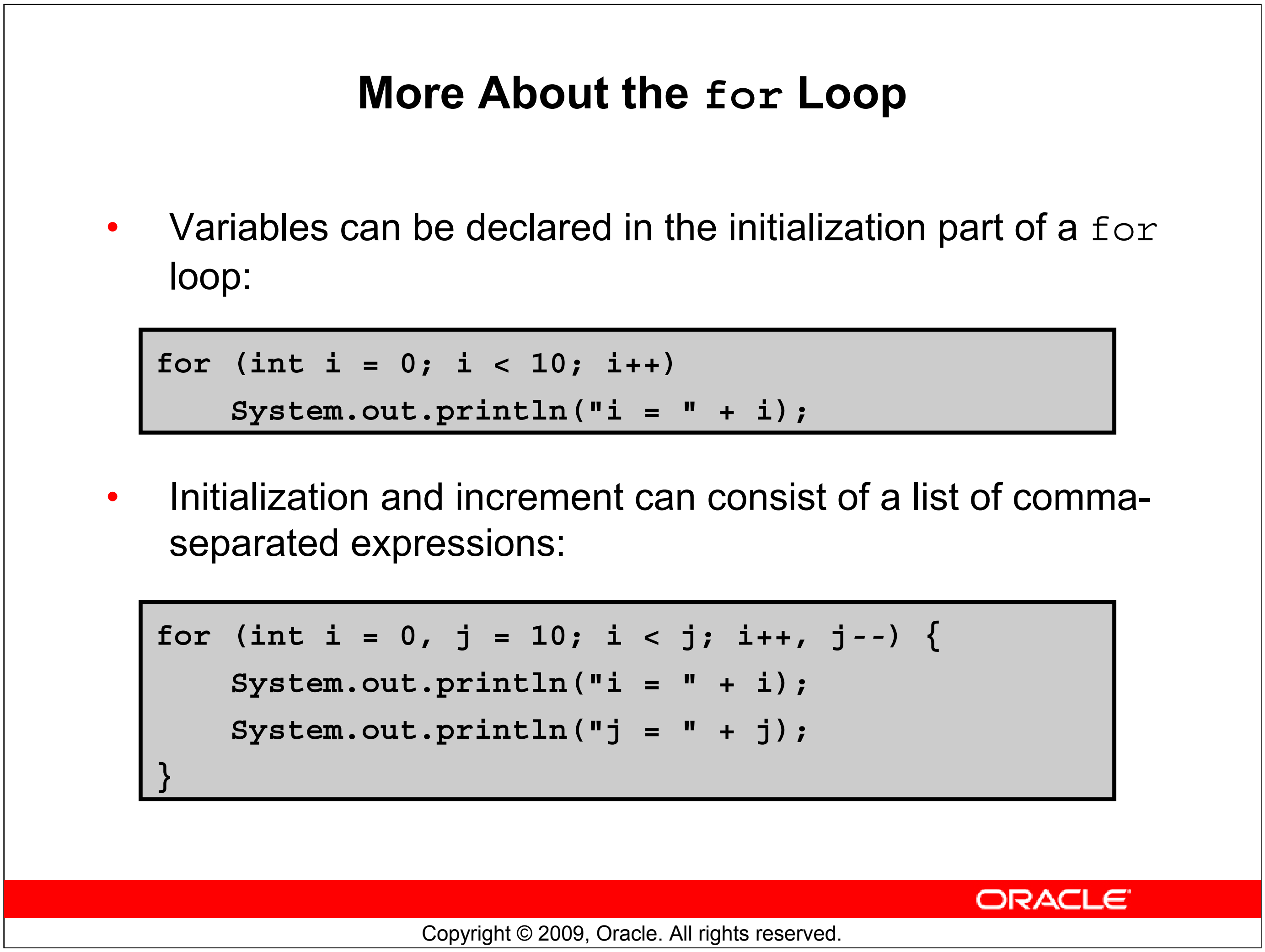
Common Mistakes When Using for Loops

A common coding error with for loops is to put a semicolon after the closing parenthesis. This is equivalent to an empty loop body that does nothing on each iteration.

for Loops and while Loops Compared

The for loop shown in the slide is equivalent to the following while loop:

```
int i = 0;
while (i < 10) {
  System.out.println(i);
  i++;
}
```



More About the `for` Loop

- Variables can be declared in the initialization part of a `for` loop:

```
for (int i = 0; i < 10; i++)
    System.out.println("i = " + i);
```

- Initialization and increment can consist of a list of comma-separated expressions:

```
for (int i = 0, j = 10; i < j; i++, j--) {
    System.out.println("i = " + i);
    System.out.println("j = " + j);
}
```

More About the `for` Loop

Declaring Variables in the Initialization Part of a `for` Loop

The initialization expression in a `for` loop can include a declaration, as shown in the first example in the slide. Any variable that is declared here is local to the `for` loop itself. After the execution leaves the loop, the variable is no longer available.

Complex Initialization and Increment Components

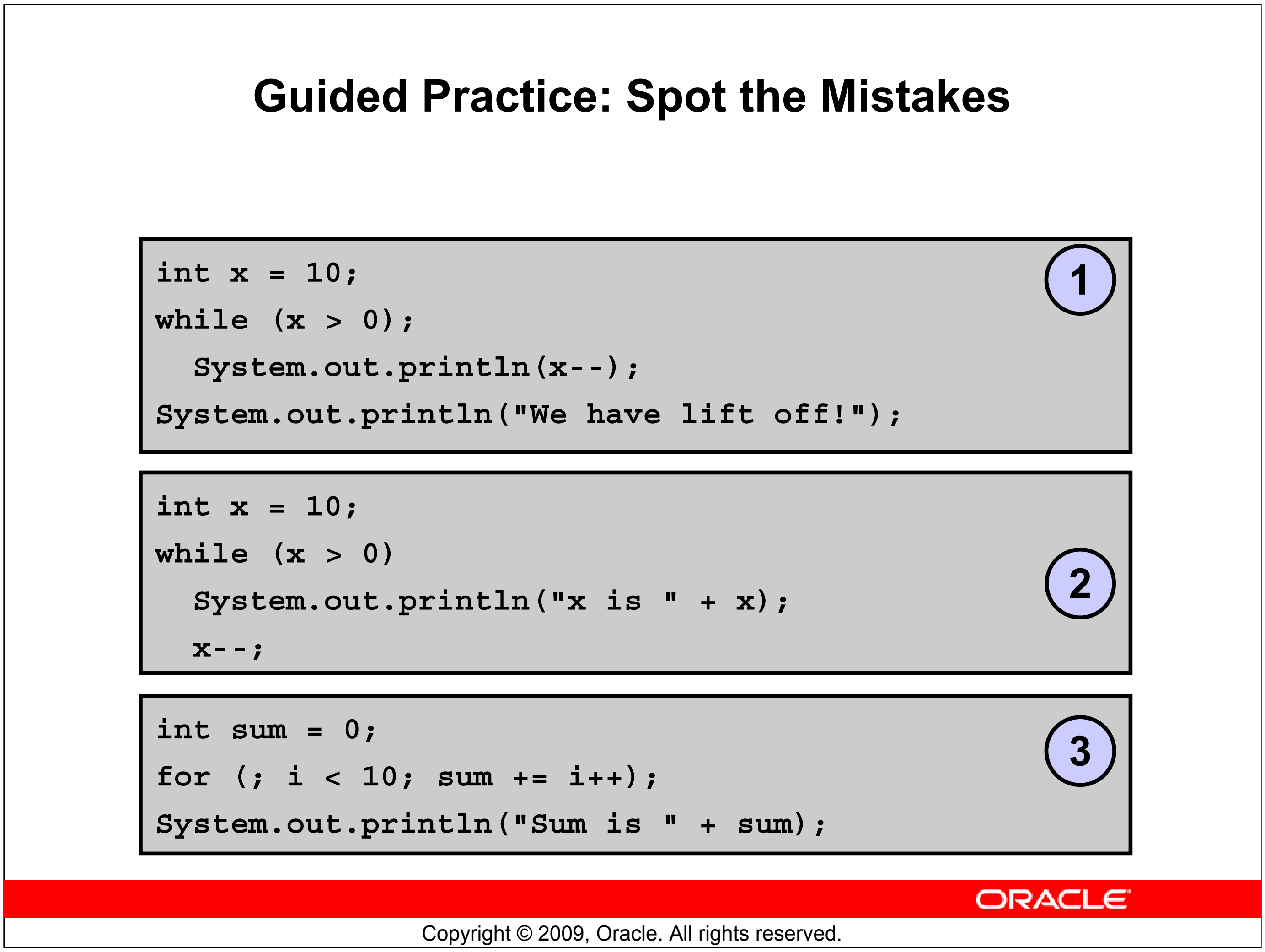
The initialization and increment components can consist of several comma-separated expressions, as shown in the second example in the slide, where two indexes are incremented or decremented in opposite directions.

Empty Initialization, Termination, or Increment Expressions

Any of the components of a `for` loop can be omitted. For example, a common way to implement a “do forever” loop is as follows:

```
for (;;) {   // loop forever
   …
}
```

This kind of loop is common in multithreaded programs in which one of the threads loops continually while doing some dedicated task. For example, a server program may have a thread that loops forever while listening for connections from client programs.



Guided Practice: Spot the Mistakes

1

```
int x = 10;
while (x > 0);
  System.out.println(x--);
System.out.println("We have lift off!");
```

2

```
int x = 10;
while (x > 0)
  System.out.println("x is " + x);
  x--;
```

3

```
int sum = 0;
for (; i < 10; sum += i++);
System.out.println("Sum is " + sum);
```

Guided Practice: Spot the Mistakes

Example 1

The first example contains an extra semicolon at the end of the `while` loop. This is not a compiler error; the compiler treats the semicolon as an empty loop body:

```
while (x > 0)
  ;    // Null loop body
```

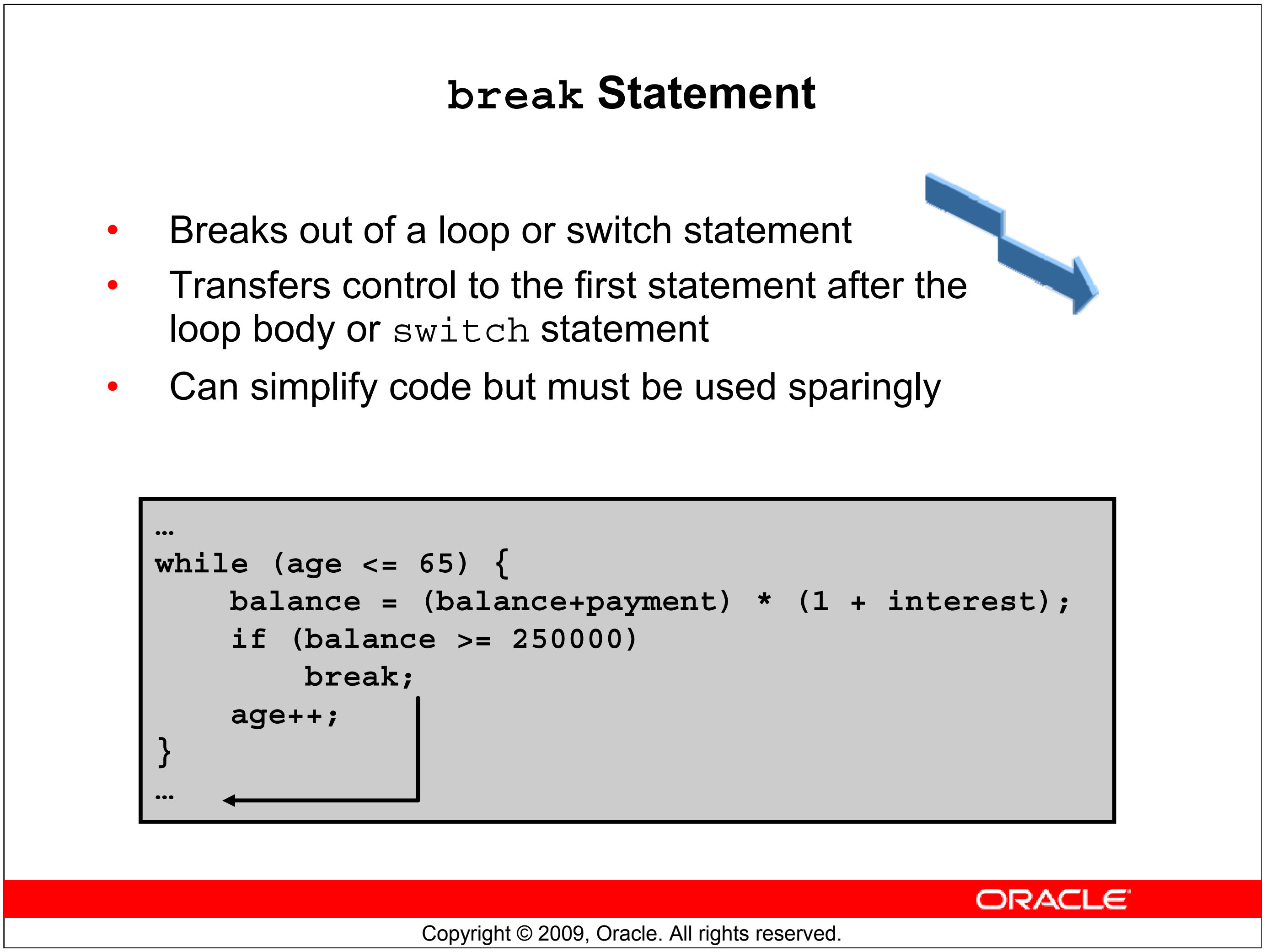
Example 2

The problem with the second example is that `x` is not changed inside the loop. The `x--` term is deemed to be outside the loop because there are no braces. Therefore, if `x` is greater than zero the first time through the loop, it will always be greater than zero, and the loop never terminates.

Example 3

The problem with this example is that `i` is not initialized anywhere. The rest of the loop is fine. Here is a description of what is happening:

- The loop keeps iterating while `i` is less than 10.
- The semicolon at the end of the `for` line indicates a null loop body.
- The increment expression in the `for` loop adds `i` to `sum` and then increments `i` ready for the next loop iteration.



break Statement

- Breaks out of a loop or switch statement
- Transfers control to the first statement after the loop body or `switch` statement
- Can simplify code but must be used sparingly

```
…
while (age <= 65) {
    balance = (balance+payment) * (1 + interest);
    if (balance >= 250000)
        break;
    age++;
}
…
```

break Statement

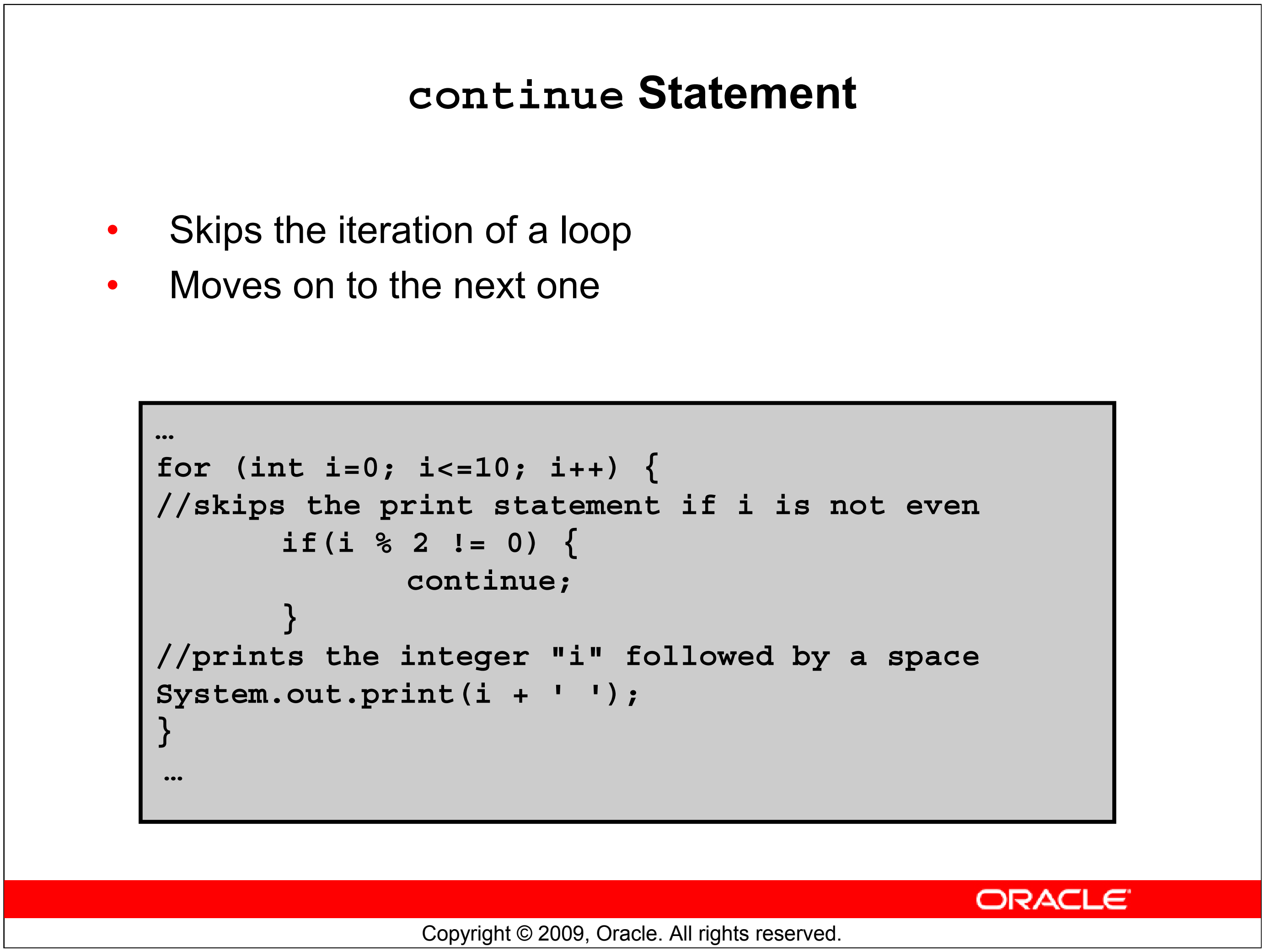
You can use a `break` statement to exit any kind of loop when a specific condition is met. Control is transferred immediately to the first statement following the closing brace of the loop body.

Should `break` Be Used?

Many programmers frown upon the use of `break` because it is not strictly necessary. For example, in the code fragment shown in the slide, you can avoid using `break` by expanding the control expression as follows:

```
while (age <=65 && balance < 250000) {
  …
}
```

Nevertheless, the use of `break` statements is useful when you need to abort a loop if some event occurs while executing the loop body. As you have already seen, `break` statements are also used in `switch` statements to prevent falling through to a subsequent `case` label.



`continue` Statement

- Skips the iteration of a loop
- Moves on to the next one

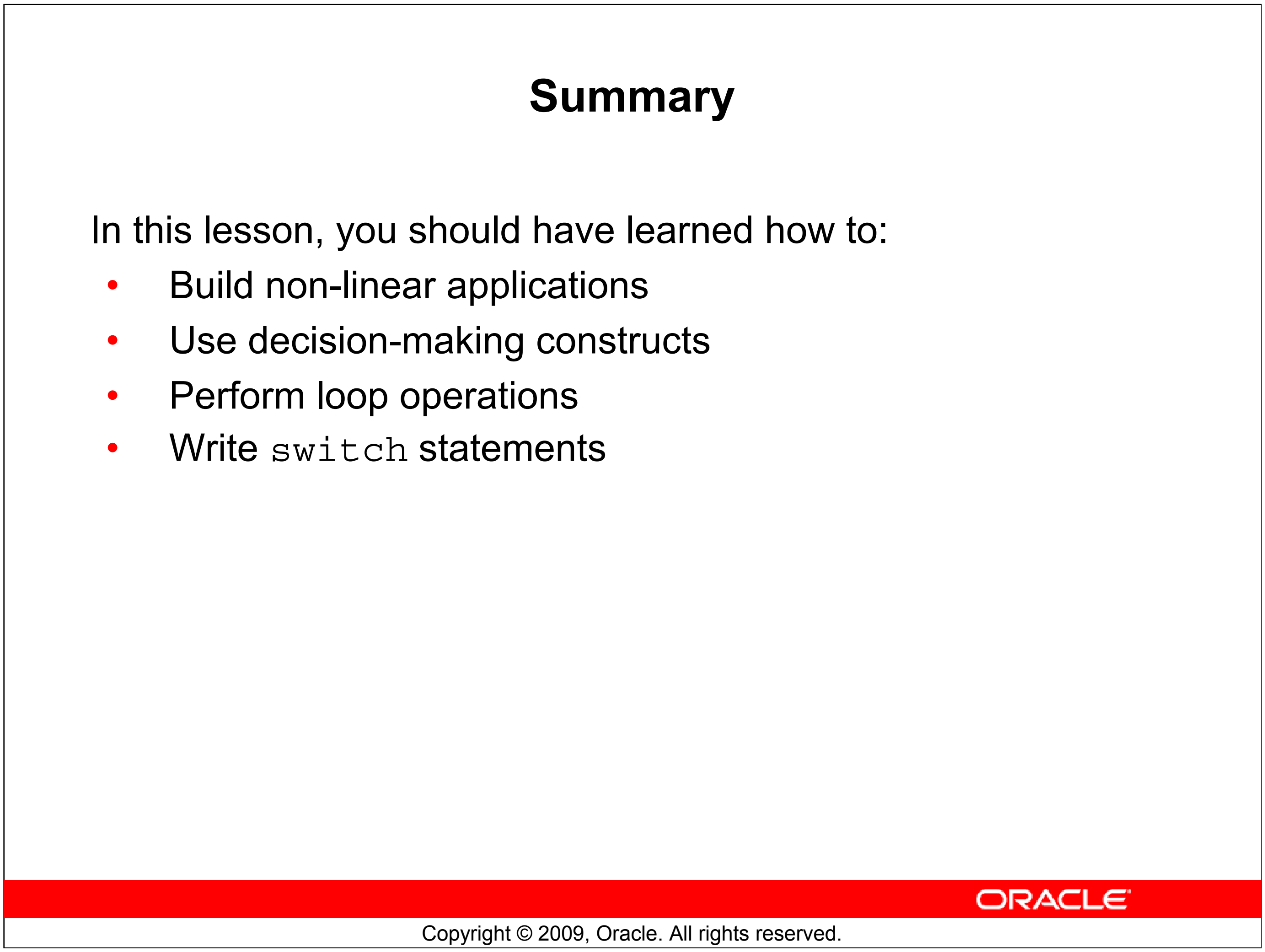
```
…
for (int i=0; i<=10; i++) {
//skips the print statement if i is not even
       if(i % 2 != 0) {
              continue;
       }
//prints the integer "i" followed by a space
System.out.print(i + ' ');
}
 …
```

`continue` Statement

You can use a `continue` statement to skip to the next iteration of the loop without performing any more code of the current iteration.

This example illustrates how to print even numbers:

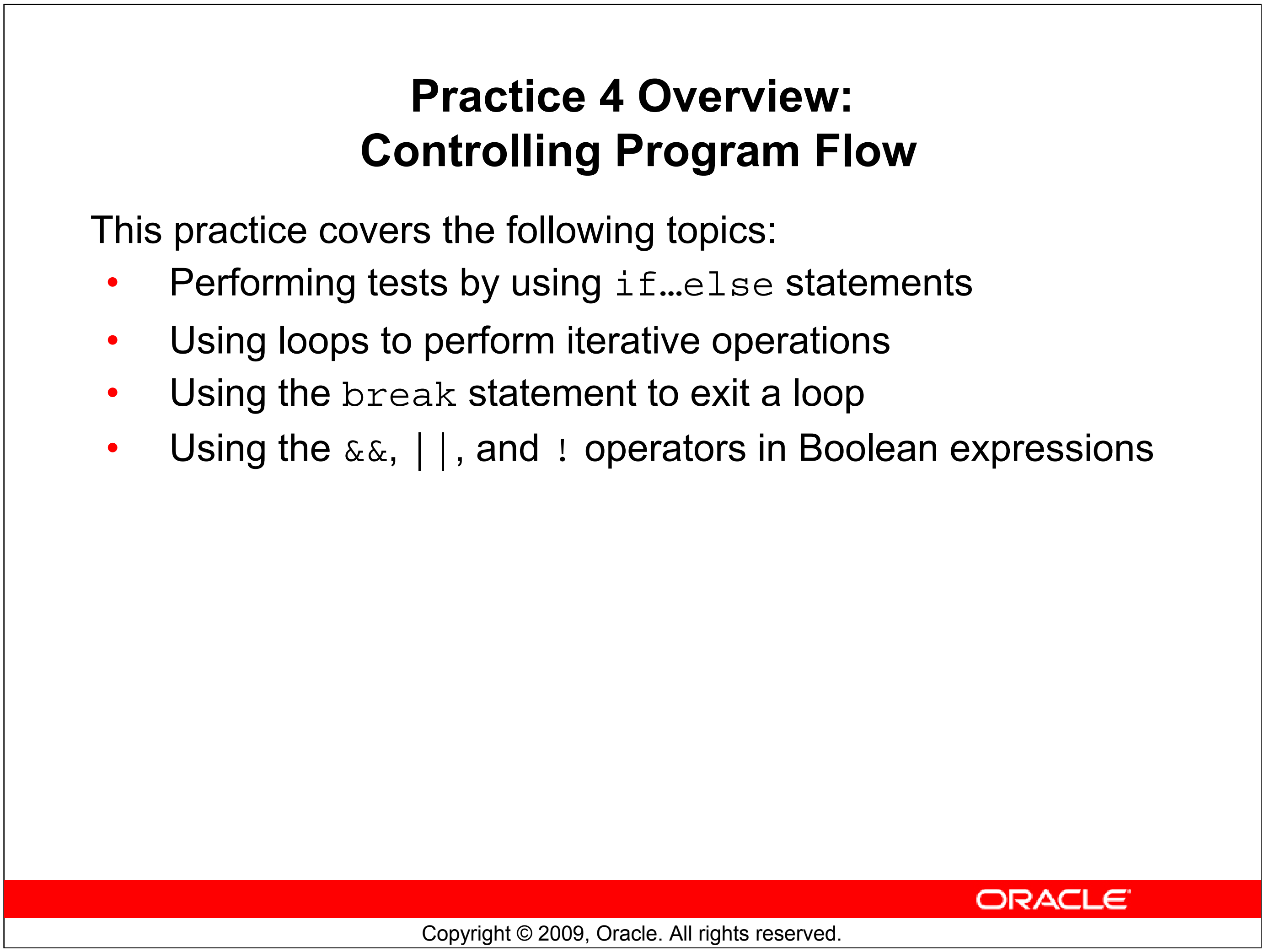
```
…
for (int i=0; i<=10; i++) {
//skips the print statement if i is not even
    if(i % 2 != 0) {
         continue;
    }
//prints the integer "i" followed by a space System.out.print(i
+ ' ');
}
```



Summary

In this lesson, you should have learned how to:

- Build non-linear applications
- Use decision-making constructs
- Perform loop operations
- Write `switch` statements



Practice 4 Overview: Controlling Program Flow

This practice covers the following topics:

- Performing tests by using `if…else` statements
- Using loops to perform iterative operations
- Using the `break` statement to exit a loop
- Using the `&&`, `||`, and `!` operators in Boolean expressions

Practice 4 Overview: Controlling Program Flow

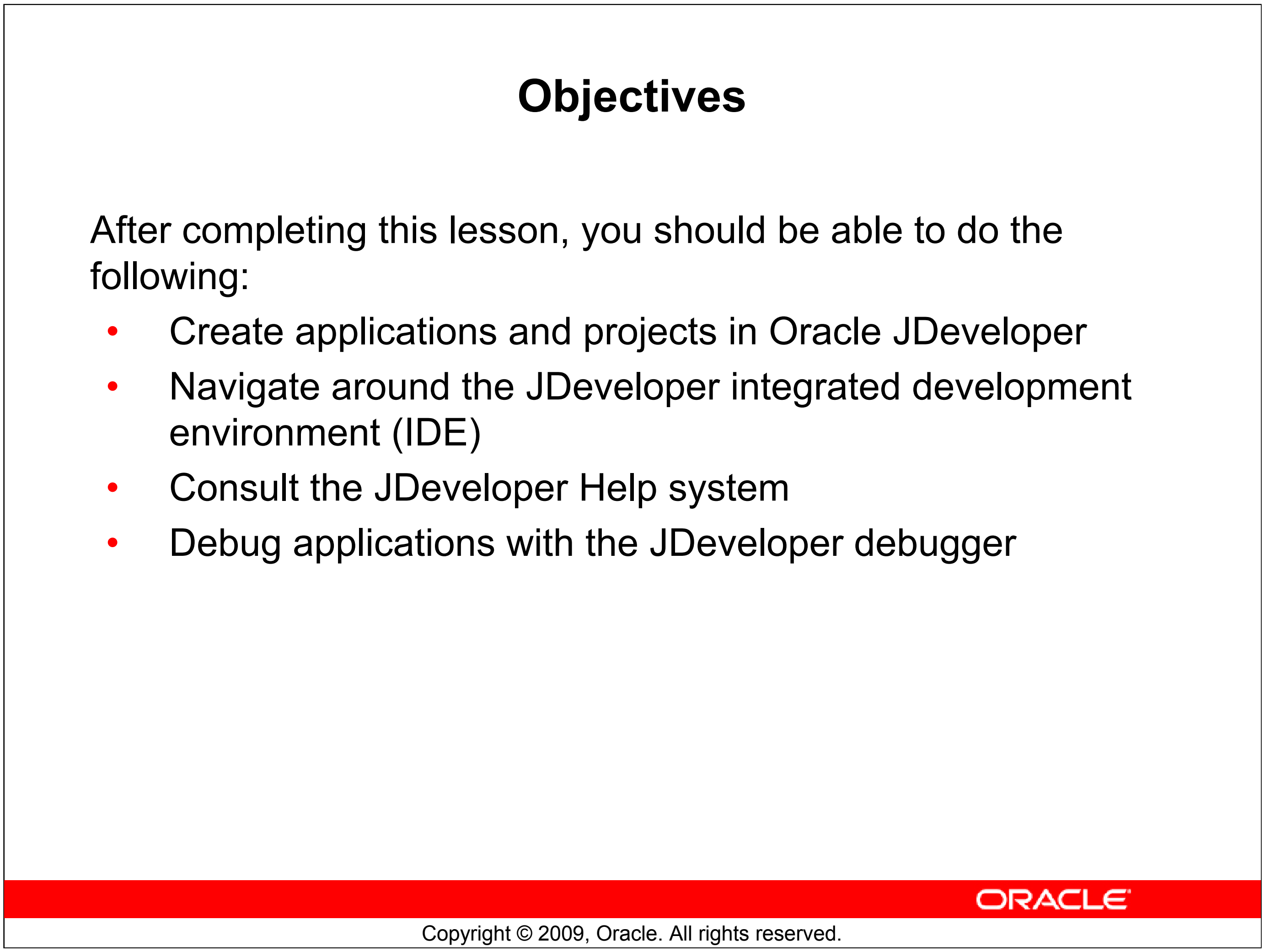
The goal of this practice is to make use of flow-control constructs that provide methods to determine the number of days in a month and to handle leap years.

Note: If you have successfully completed the previous practice, continue using the same directory and files. If the compilation from the previous practice was unsuccessful and you want to move on to this practice, change to the `les04` directory and continue with this practice.

Remember that if you close a DOS window or change the location of the `.class` files, you *must* set the `CLASSPATH` variable again.

5

Building Applications with Oracle JDeveloper (11*g*)



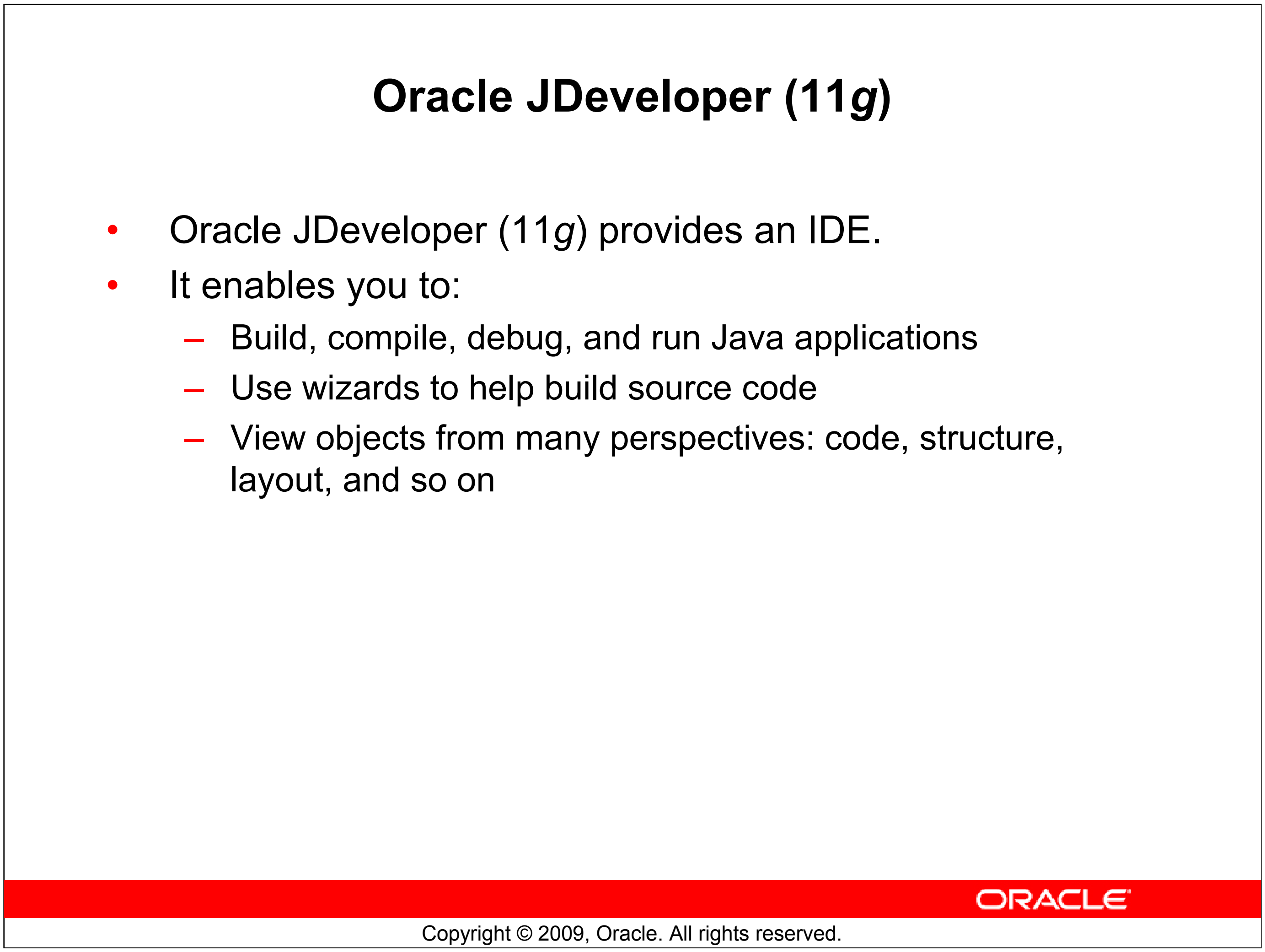
Objectives

After completing this lesson, you should be able to do the following:

- Create applications and projects in Oracle JDeveloper
- Navigate around the JDeveloper integrated development environment (IDE)
- Consult the JDeveloper Help system
- Debug applications with the JDeveloper debugger

Lesson Objectives

This lesson introduces Oracle JDeveloper, which provides you with an IDE that simplifies the writing of Java code and makes you much more productive. You learn how to create new projects and applications and how to use JDeveloper to write good, sound source code.



Oracle JDeveloper (11*g*)

- Oracle JDeveloper (11*g*) provides an IDE.
- It enables you to:
  - Build, compile, debug, and run Java applications
  - Use wizards to help build source code
  - View objects from many perspectives: code, structure, layout, and so on

Oracle JDeveloper

You can use Oracle JDeveloper to build a number of different types of Java components. This lesson focuses on using the JDeveloper IDE for building Java applications.

Wizard Driven

A wizard is a graphical tool that provides step-by-step guidance through the process of defining a new element in the IDE. Oracle JDeveloper provides numerous wizards, including:

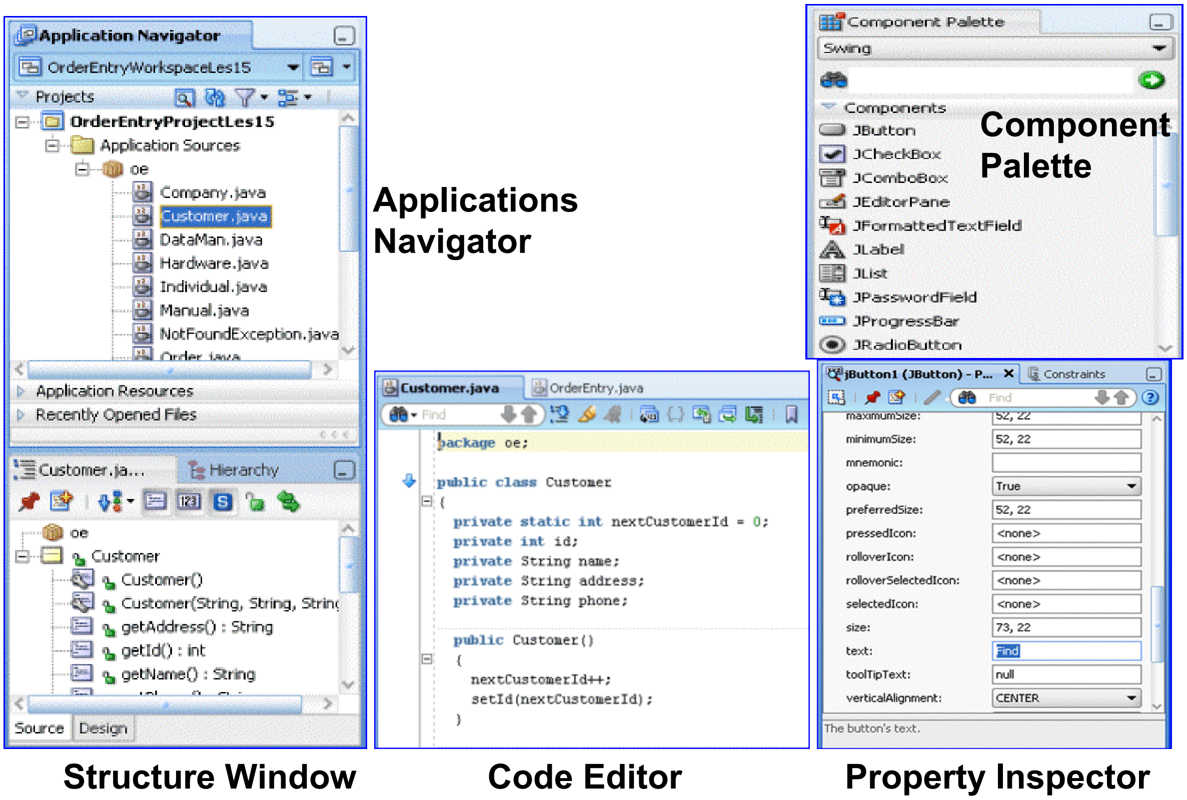
- **Application Wizard:** Defines a new application and associated projects
- **EJB Wizard:** Defines a new Enterprise JavaBean (EJB) and adds it to the specified project
- **JSP Wizard:** Defines a new JavaServer Page (JSP) and adds it to the specified project
- **HTTP Servlet Wizard:** Defines a new servlet and adds it to the specified project

Designing a User Interface

Oracle JDeveloper helps you specify the following features of your user interface:

- Size and position of controls
- Properties for each control, such as labels, enabled or disabled status, font, and so on
- Event-handler methods

Oracle JDeveloper (11*g*) Environment



```
package oe;

public class Customer
{
  private static int nextCustomerId = 0;
  private int id;
  private String name;
  private String address;
  private String phone;

  public Customer()
  {
    nextCustomerId++;
    setId(nextCustomerId);
  }
```

maximumSize: 52, 22
minimumSize: 52, 22
mnemonic:
opaque: True
preferredSize: 52, 22
pressedIcon: <none>
rolloverIcon: <none>
rolloverSelectedIcon: <none>
selectedIcon: <none>
size: 73, 22
text: Find
toolTipText: null
verticalAlignment: CENTER

The button's text.

ORACLE

Oracle JDeveloper (11*g*) Environment

JDeveloper contains five major user interface components. These components are what you use to edit code, design and manage the user interface, and navigate around your program.

Applications Navigator

The Applications Navigator comprises several components:

- The Navigator pane shows a list of files or classes in a project. The files may be Java source files, `.class` files, graphics files, HTML, XML documents, and so on.
- The Application Resources pane contains a list of the resources that your application may be using, for example any database connections that have been created.
- There is also a compartment that lists the files that you have been working with most recently.
- The Structure window lists all the methods, fields, and graphical user interface (GUI) components in a selected class. In the example on the slide, the `Customer.java` file is selected in the Navigator, and so the Structure window displays the structure of this `.java` file.

Code Editor

Editors are where most of the work takes place; this is where you write code and design user interfaces. You open the editor by double-clicking the file you want to edit or view.

Oracle JDeveloper (11*g*) Environment (continued)

Component Palette

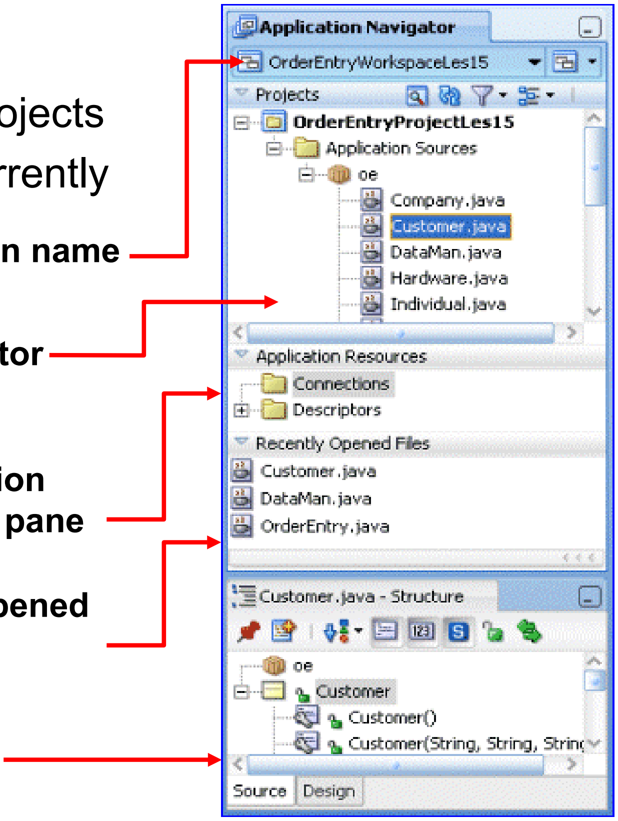
The Component Palette displays the visual elements of the JDeveloper component library. You drag and drop components from the palette when assembling a user interface. The Component Palette in the slide displays Swing components.

Property Inspector

The Property Inspector is the window that shows the properties and events that are associated with a selected component in the design section of an editor.

Application Navigator

- May contain multiple projects
- Enables you to view currently used objects



Application name

Application Navigator pane

Application Resources pane

Recently opened files

Structure window

Application Organization

Oracle JDeveloper (11*g*) uses a well-defined structure to manage Java programming applications. The structure is hierarchical and supports applications, projects, images, `.html` files, and so on.

Applications

Application is the highest level in the control structure. It is a view of all the objects you currently need while you are working. An application keeps track of the projects you use and the environment settings while you are developing your Java program. When you open JDeveloper, the last application used is opened by default so that you can resume your work.

Applications are stored in files with the extension `.jws`. You do not edit an application file directly. Whenever you save your application, you are prompted to save all the current open files. To save the open and modified files, select the Save option (or the Save All option) from the File menu.

Note: You can view the contents of an application file by using any text editor.

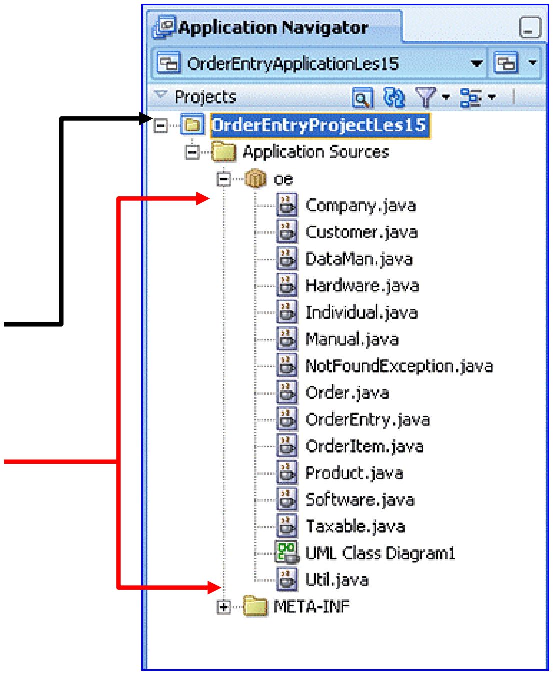
Application Organization (continued)

Determining Applications

You might consider an application to be a view of currently used objects. However, you can choose to create applications that group projects that were created in different applications. Application object groupings can be based around a business area (Accounts Payable, General Ledger, Accounts Receivable), a phase in a life cycle (analysis, design, deploy), or the structure of the application (UI, business logic, data structure).

Projects

- Contain related files
- Manage project and environment settings
- Manage compiler and debug options



Project node

Project files

Projects

JDeveloper projects organize the file elements that are used to create your program. A project file has the file extension `.jpr` and keeps track of the source files, packages, classes, images, and other elements that may be needed for your program. You can add multiple projects to an application to easily access, modify, and reuse your source code. You can view the contents of a project file using any text editor.

Projects manage environment variables, such as the source and output paths used for compiling and running your program. Projects also maintain compiler, run time, and debugging options so that you can customize the behavior of those tools for each project.

In the Navigator pane, projects are displayed as the second level in the hierarchy under the application.

When you select a `.java` or `.html` file in the Applications Navigator, the Structure pane (by default located below the Navigator pane) displays the elements of the file in a tree format. For example, when you select a `.java` source file, the classes, interfaces, methods, and variables are displayed.

To edit source code, double-click the file in the navigation list to display the contents in the appropriate editor. The Structure pane can be used to quickly locate specific areas of code in your Java source files and to browse the class hierarchy.

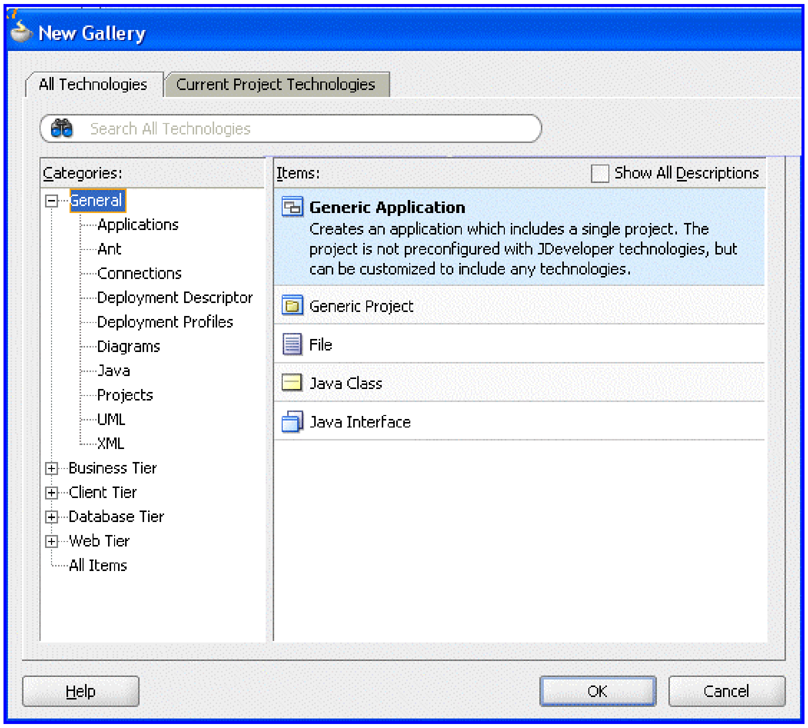
When you are working with the visual designer, the Structure pane displays the components of your user interface and their associated event-handling methods in a hierarchical tree format.

Note: Italic style is used to indicate file names that have not yet been saved.

Creating JDeveloper Items

JDeveloper items:

- Are invoked by selecting File > New
- Are categorized by type:
  - General
  - Business Tier
  - Client Tier
  - Database Tier
  - Web Tier



Creates an application which includes a single project. The project is not preconfigured with JDeveloper technologies, but can be customized to include any technologies.

Creating JDeveloper Items

You can create any JDeveloper item from this window. The context for creating the item must be correct. You must have the correct element selected in the Category column to be able to view and create the appropriate, related item.

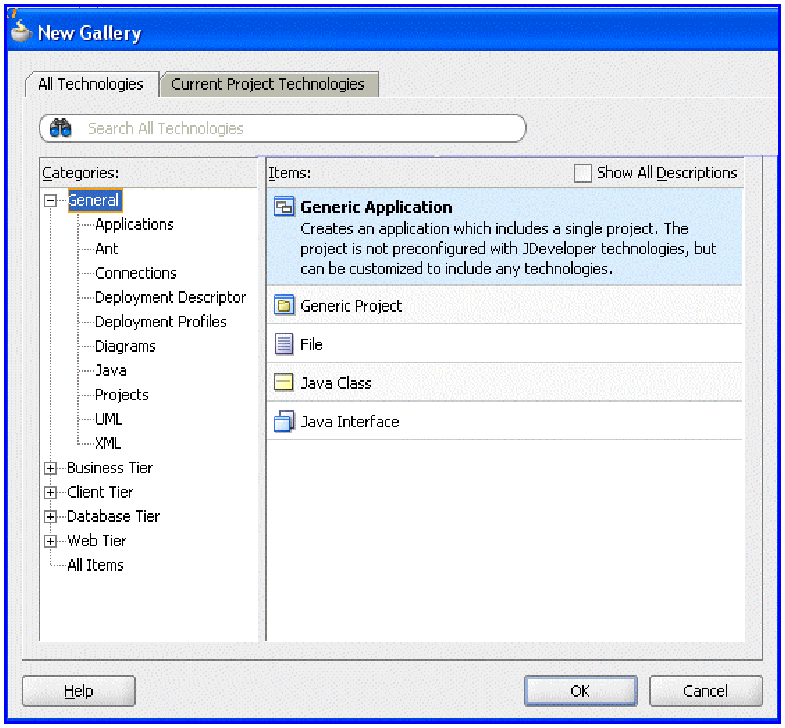
When creating the application, you can define the paths used for the files stored. Use the following convention for all stored files:

```
application\project\package
```

Use the Filter By list to view specific types of elements.

Creating an Application

In the General category, select Generic Application to invoke the Create Generic Application wizard.



New Gallery

All Technologies | Current Project Technologies

Search All Technologies

Categories:

- General
  - Applications
  - Ant
  - Connections
  - Deployment Descriptor
  - Deployment Profiles
  - Diagrams
  - Java
  - Projects
  - UML
  - XML
- Business Tier
- Client Tier
- Database Tier
- Web Tier
- All Items

Items: □ Show All Descriptions

Generic Application
Creates an application which includes a single project. The project is not preconfigured with JDeveloper technologies, but can be customized to include any technologies.

Generic Project

File

Java Class

Java Interface

Creating an Application

JDeveloper makes it easy for you to get started working with the tool quickly and easily. You need an application to work with, and so when you open JDeveloper for the first time you are presented with the option to open an existing application (by browsing the file system) or creating a new one. If you click the option to create a new application, the New Gallery is invoked. With the General category selected, as in the screenshot in the slide, you have the option to create a generic application. If you select Applications (the first subcategory in the General category), you can then choose from a number of specific application types.

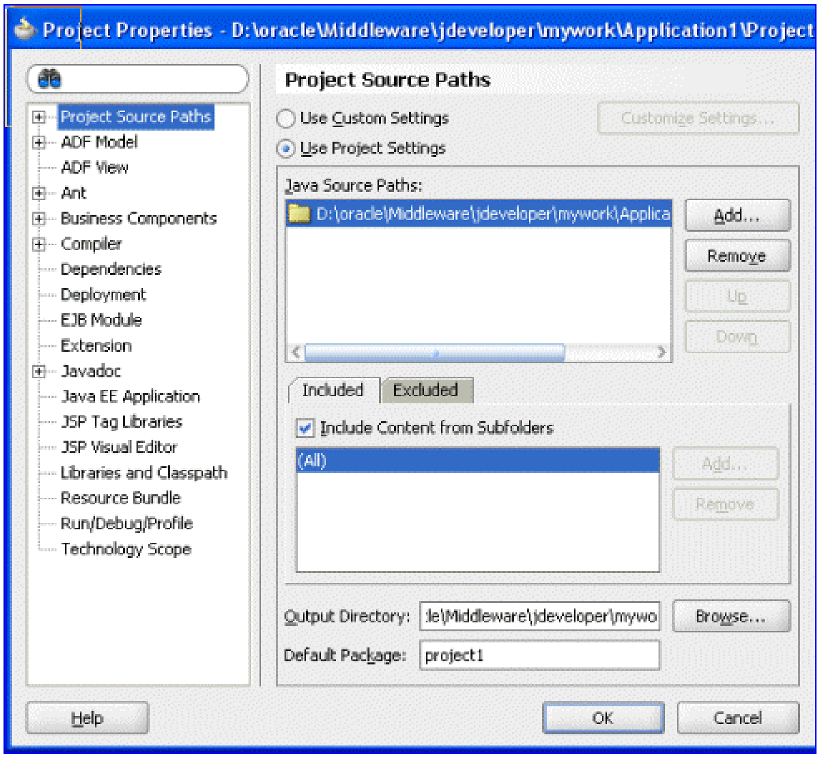
The Create Generic Application Wizard

When you click OK in the New Gallery, as in the screenshot in the slide, the Create Application wizard is invoked. The first page of the wizard asks you to name the application and specify a directory for it—or you can accept the defaults. You can also optionally specify a prefix for the application. If you are really impatient to get started with JDeveloper, you can just click Finish at the bottom of the page and a generic application and associated empty project are created for you.

Creating an Application (continued)

If you have the time, you can click Next instead, and view the second page of the wizard. This page allows you to specify a name for the project associated with your application and to select the technology that the project is to use. For a Java project, you select Java in the Available pane, and shuttle it across to the Selected pane. As before, you can click Finish at this point, or click Next to progress to the third page of the wizard. This page allows you to create a default package for your Java project (or accept the default), and to specify the path to your Java source code files and the output directory where your compiled `.class` files will be placed.

Project Properties: Specifying Project Details



Project Properties - D:\oracle\Middleware\jdeveloper\mywork\Application1\Project

Project Source Paths

○ Use Custom Settings

● Use Project Settings

Java Source Paths: D:\oracle\Middleware\jdeveloper\mywork\Applica

☑ Include Content from Subfolders

Included: (All)

Output Directory: :le\Middleware\jdeveloper\mywo

Default Package: project1

Project Properties: Specifying Project Details

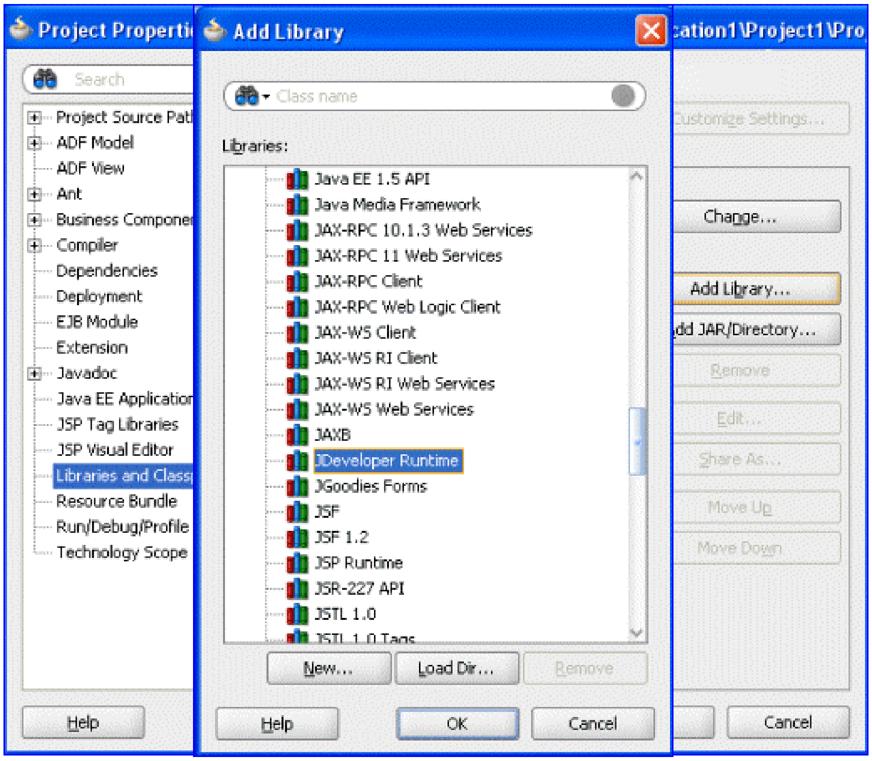
You can create and modify a variety of project properties. The properties are categorized and are accessible from the nodes on the left side of the pane.

The package name is the default for the project. Any Java source created in the project is therefore automatically put in the specific package because the package keyword is added to the source file with the name specified.

In particular, also note that the package name is implicitly appended to the project source and output directories as the target directories for the saved source (`.java`) files and for the compiled (`.class`) files, respectively.

Note: The output directory is added as the first path in the `–classpath` parameter for applications at compile and run time. In addition, together with this output directory, the libraries selected in the next screen of the wizard form the `CLASSPATH` for the code in this project.

Project Properties: Selecting Additional Libraries



ORACLE

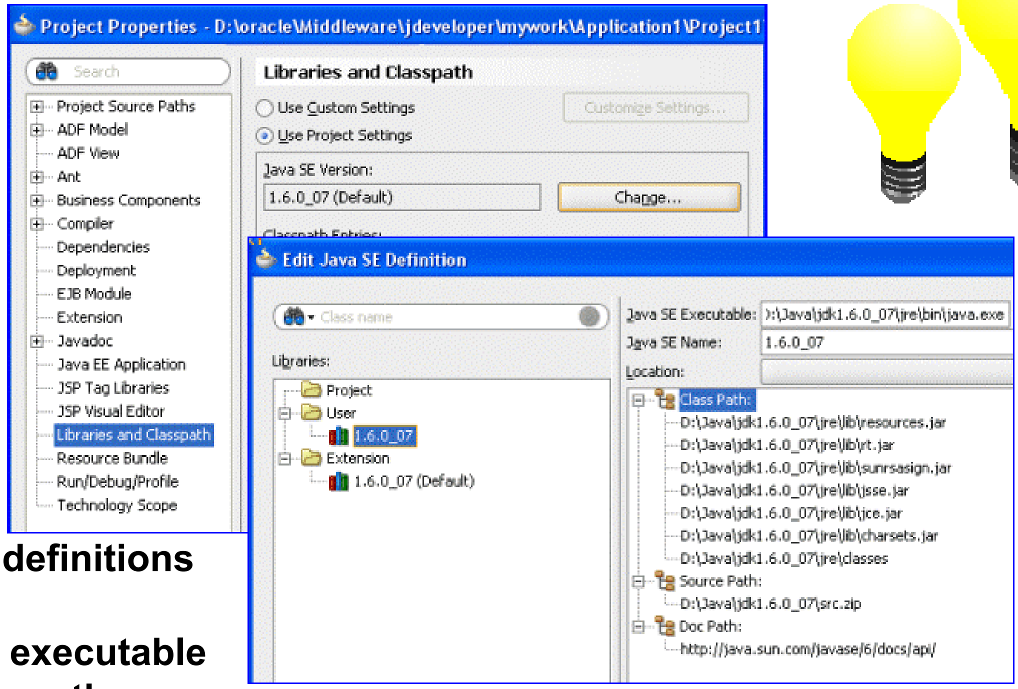
Project Properties: Selecting Additional Libraries

In the Libraries property, you can change the Java SE version that you want to use. You can also click the Add Library button to add existing libraries and create and add new ones.

The libraries listed here are delivered by Oracle, Sun Microsystems, and third-party packages and classes. Typically, they are delivered as classes in a `.jar` file. Use the existing ones or add your own.

Note: All selected libraries become part of the project application `CLASSPATH` in addition to the classes compiled into the project output directory. All the classes in the libraries and output directory must be deployed with the application.

Adding a New Java SE



○ Use Custom Settings
● Use Project Settings

Java SE Version: 1.6.0_07 (Default)

Java SE Executable:):\Java\jdk1.6.0_07\jre\bin\java.exe

Java SE Name: 1.6.0_07

Location:

```
Class Path:
    D:\Java\jdk1.6.0_07\jre\lib\resources.jar
    D:\Java\jdk1.6.0_07\jre\lib\rt.jar
    D:\Java\jdk1.6.0_07\jre\lib\sunrsasign.jar
    D:\Java\jdk1.6.0_07\jre\lib\jsse.jar
    D:\Java\jdk1.6.0_07\jre\lib\jce.jar
    D:\Java\jdk1.6.0_07\jre\lib\charsets.jar
    D:\Java\jdk1.6.0_07\jre\classes
Source Path:
    D:\Java\jdk1.6.0_07\src.zip
Doc Path:
    http://java.sun.com/javase/6/docs/api/
```

Java SE definitions include:

- **Java executable**
- **Class path**
- **Source path**
- **Doc path**

Defining a New Java SE in JDeveloper

Every JDeveloper project uses a Java SE definition to determine the version of the Java API to use for compiling and running. Each Java SE definition includes the following:

- Java executable used to launch programs
- A class path containing the classes available in the Java SE environment
- A source path housing the source files for the Java SE classes
- A doc path storing the javadoc files for the Java SE classes

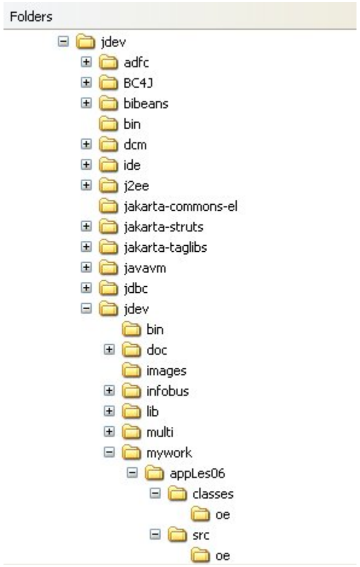
Additional definitions can be created from any available Java SE. These new definitions can be created in either the user libraries or the project libraries. In a multiuser environment, Java SE definitions created in the user libraries are user-specific, whereas those created in the project libraries are shared by all users.

Note: The default Java SE for JDeveloper 11*g* is Java JDK 1.6.0_07.

Directory Structure

JDeveloper creates and stores `.java` and `.class` files using the following conventions:

- `<ORACLE_HOME>\jdev\mywork`
- Followed by the application name
- Followed by the project name:
  - `\classes\<package name>\`
  - `\src\<package_name>\`
- Followed by `class` and `src` files



Directory Structure

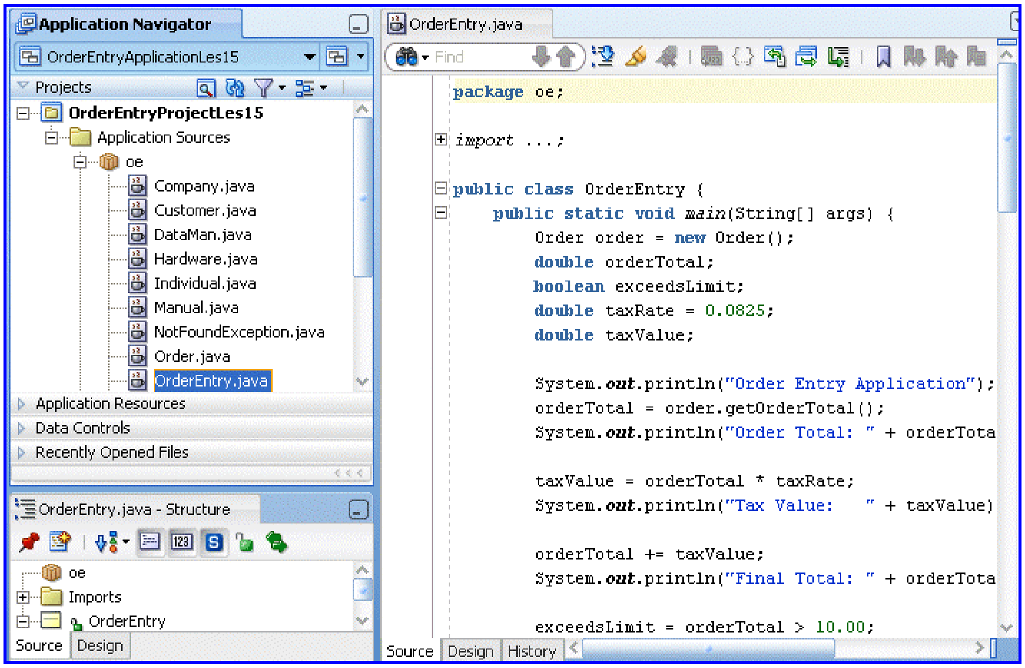
When you install JDeveloper, the installer creates a `JDEV_HOME` directory for the JDeveloper product files. A subdirectory for all the JDeveloper executables and support code is stored in the `\jdev\` directory. One of the directories under `\jdev\` is `\mywork\`, which is used to store the files you create in JDeveloper.

Each application has its own directory, and below it is a directory for each project. The name of the project directory is dependent on the type of application you create. For Web development, two projects are created—one for the `Model` and the other for the `View`. For a Java\Swing application, a single project is created named `Client`.

Each Java file you create is stored in a `\src\<package_name>\` directory under the project in which it was created. All the compiled files are stored in the `\classes\<package_name>\` directory. However, you can create your application and project directories anywhere in the file system.

Exploring the Skeleton Java Application

Java classes are listed in the Application Navigator; the Code Editor displays the source code:



```
package oe;

import ...;

public class OrderEntry {
    public static void main(String[] args) {
        Order order = new Order();
        double orderTotal;
        boolean exceedsLimit;
        double taxRate = 0.0825;
        double taxValue;

        System.out.println("Order Entry Application");
        orderTotal = order.getOrderTotal();
        System.out.println("Order Total: " + orderTota

        taxValue = orderTotal * taxRate;
        System.out.println("Tax Value:   " + taxValue)

        orderTotal += taxValue;
        System.out.println("Final Total: " + orderTota

        exceedsLimit = orderTotal > 10.00;
```

Exploring the Skeleton Java Application

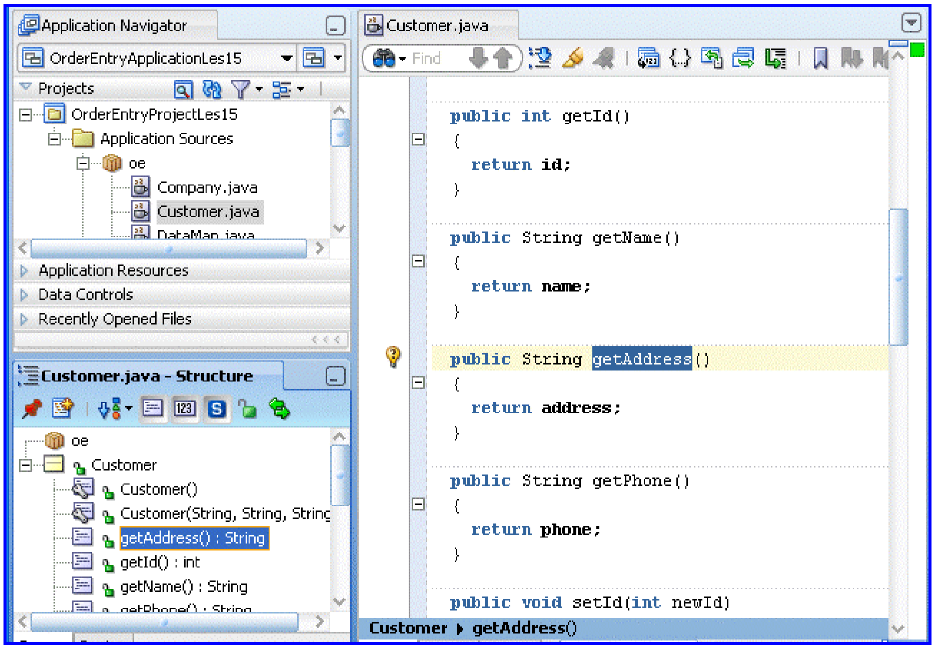
The slide shows the OrderEntry application that you work with in the practices for this course.

- The Application Navigator displays the hierarchy for the application with the Workspace/ Application file at the top level, with the project file below that, and then the `.java` files under the package node.
- The Code Editor displays the source code for a selected `.java` file, in this case, `OrderEntry.java`.
- The Structure window displays the structure of the selected file.

You can create a connection to a database from JDeveloper, and then query the database from a Java program.

Finding Methods and Fields

Find methods and fields using the Structure pane:



```
public int getId()
{
  return id;
}

public String getName()
{
  return name;
}

public String getAddress()
{
  return address;
}

public String getPhone()
{
  return phone;
}

public void setId(int newId)
```

Finding Methods and Fields

As projects evolve, classes can become quite large, containing numerous methods and fields. To help you find your way around complex classes, JDeveloper provides the Structure pane, which is the lower pane in the Applications Navigator.

The Structure pane lists all the methods and fields for the currently selected class. If you double-click an item in the Structure pane, JDeveloper takes you to the definition of that item in the source code, displaying and highlighting it in the Code Editor. For example, if you double-click the `getAddress()` method in the Structure pane, the declaration of the `getAddress()` method is highlighted in the Code Editor.

You can also search the Navigator and Structure pane components for strings by using a [*letter*]. The search is a hierarchical search based on the first letter of each component. As you type in subsequent letters, the structure list highlights the first component that begins with that set of letters. If there is more than one occurrence, use the up and down arrow keys to scroll through the result set.

Note: JDeveloper (11*g*) provides some features to facilitate navigation through your Java code. One of these is navigation between members, which allows you to quickly navigate between fields and methods in the Code Editor by using the Previous Member (Alt + Up) and Next Member (Alt + Down) accelerators.

Supporting Code Development with Profiler and Code Coach

- Improve code quality with Code Coach.
- Evaluate execution stack with the Execution Sample profiler.
- Examine heap memory usage with the Memory profiler.
- Analyze event occurrence and duration with the Event profiler for:
  - JVM events
  - Business components for Java events
  - Custom events

ORACLE

Code Coach

Code Coach creates more efficient Java programs by helping you write higher quality, better-performing code. You run Code Coach on a class to obtain advice on how to make your code better.

Profilers

Profilers gather statistics on your program, enabling you to more easily diagnose performance issues. With profilers, you can examine and analyze your data.

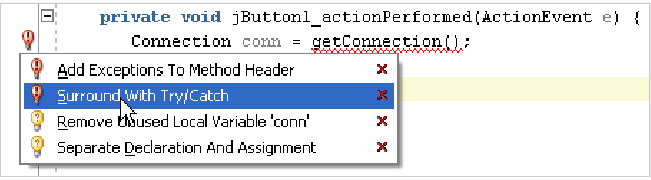
Code Editor

When you pause momentarily while typing in the Code Editor, JDeveloper automatically scans your code to look for syntax errors. If there are any, you see them represented in the Structure pane or in the Log window.

Code Insight

If you pause after typing a "." (period), JDeveloper invokes Code Insight. Code Insight displays valid methods and members from which you can select. JDeveloper also provides Code Insight for parameters when you pause after typing a left parenthesis.

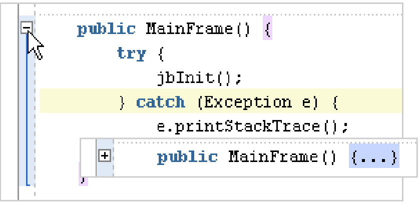
Code Editor Features



```
private void jButton1_actionPerformed(ActionEvent e) {
    Connection conn = getConnection();
```

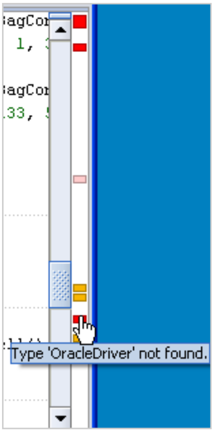
Add Exceptions To Method Header
Surround With Try/Catch
Remove Unused Local Variable 'conn'
Separate Declaration And Assignment

Code Assist

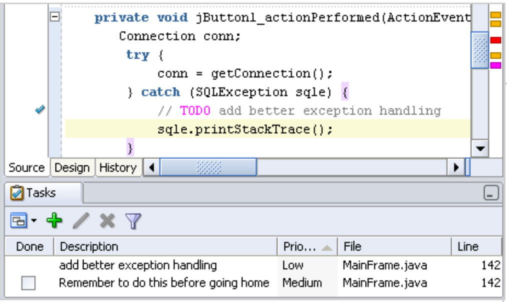


```
public MainFrame() {
    try {
        jbInit();
    } catch (Exception e) {
        e.printStackTrace();
        public MainFrame() {...}
```

Scope and code folding



Type 'OracleDriver' not found.

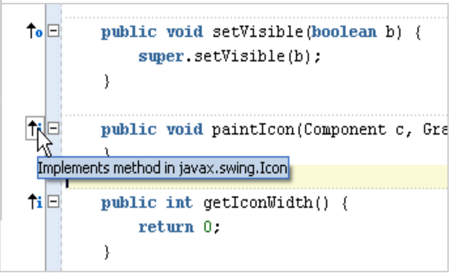


```
private void jButton1_actionPerformed(ActionEvent
    Connection conn;
    try {
        conn = getConnection();
    } catch (SQLException sqle) {
        // TODO add better exception handling
        sqle.printStackTrace();
    }
```

Done	Description	Prio...	File	Line
	add better exception handling	Low	MainFrame.java	142
□	Remember to do this before going home	Medium	MainFrame.java	142

Tasks list

Overview margin



```
public void setVisible(boolean b) {
    super.setVisible(b);
}

public void paintIcon(Component c, Gra
}

public int getIconWidth() {
    return 0;
}
```

Implements method in javax.swing.Icon

Overridden and Implemented Method Definitions

Code Editor Features

Code Assist

Code Assist is a feature in JDeveloper that deals with adherence to coding standards rather than syntactical correctness.

It examines your code in the Java Source Editor and offers suggestions to fix coding problems or breaches of coding standards. A light bulb icon appears in the margin beside a line where JDeveloper has a suggestion for a code change. You click the icon to display the suggestion. You can accept the suggestion by clicking it and amending the code; or you can reject the suggestion and suppress the light bulb by closing the suggestion. In many cases, if you accept the suggestion, JDeveloper makes the appropriate code modifications for you automatically.

You can choose which rules you want Code Assist to use, or you can disable the feature altogether by selecting `Tools > Preferences >Audit: Profiles`.

Code Folding

Code folding allows you to shrink sections of code, making large programs more readable and more manageable. You use the + and – buttons in the blue vertical bar on the left of the code to expand or contract classes.

Code Editor Features (continued)

Tasks List

By using the Tasks list, you can create and keep track of tasks. The tasks can be of any nature, but the feature has been provided primarily for tasks connected with the process of software development. For this reason, a direct link has been provided between the Code Editor and the Tasks list: When writing code, a task is created whenever you enter `//TODO`. A check mark is displayed in the margin to the left of the line, indicating the presence of a task.

Overview Margin

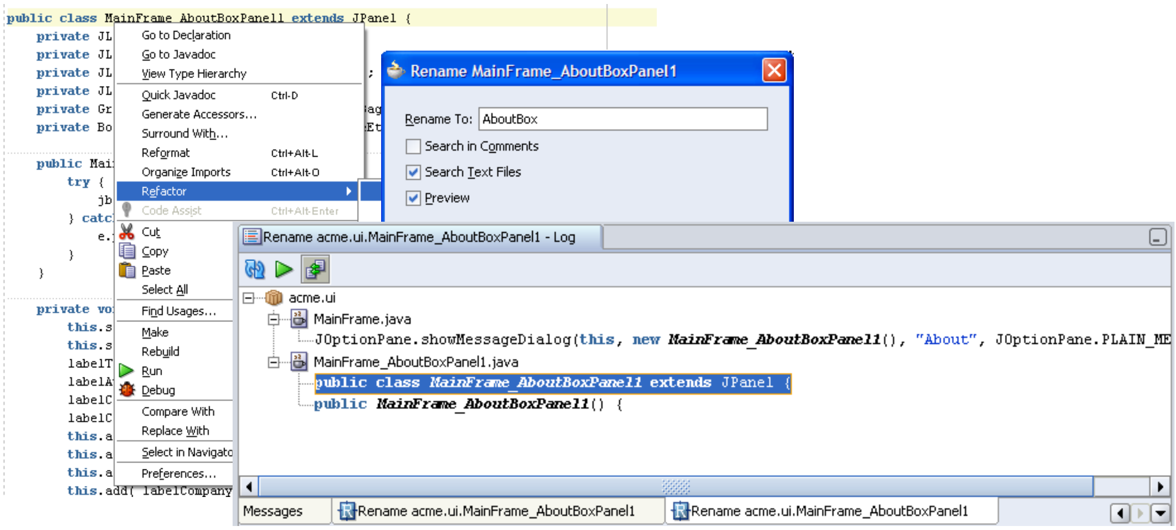
The Overview margin is displayed vertically on the right side of the Code Editor. If there are no problems with the code, a small green marker appears at the top of the margin. An orange marker indicates a warning, and a red marker denotes an error. When you position the cursor over an orange or red marker, you see a brief displayed description of the error or warning.

Overridden and Implemented Method Definitions

When working in the Java Source Editor, you can identify methods that override superclass definitions or implement interface declarations. Overriding definitions are marked with an upward-pointing arrow and the letter *o* in the left margin; clicking this letter takes you to the overridden definition. Similarly, an upward-pointing arrow and the letter *i* in the margin identifies a method that implements an interface; clicking the letter takes you to the implemented method declaration.

Refactoring

Modify the structure of code without changing its behavior (or breaking it).



```
public class MainFrame_AboutBoxPanel1 extends JPanel {
```

Rename MainFrame_AboutBoxPanel1

Rename To: AboutBox

☐ Search in Comments

☑ Search Text Files

☑ Preview

Rename acme.ui.MainFrame_AboutBoxPanel1 - Log

```
acme.ui
  MainFrame.java
    JOptionPane.showMessageDialog(this, new MainFrame_AboutBoxPanel1(), "About", JOptionPane.PLAIN_ME
  MainFrame_AboutBoxPanel1.java
    public class MainFrame_AboutBoxPanel1 extends JPanel {
    public MainFrame_AboutBoxPanel1() {
```

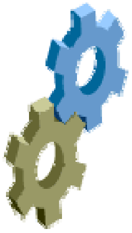
Refactoring

Refactoring is the process of improving an application by reorganizing its internal structure without changing its external behavior. In JDeveloper, you can make these changes without breaking any dependent files on which your project relies, because these files are automatically updated for you.

Oracle JDeveloper provides a powerful refactoring framework; as a result, performing refactoring actions is fast and much smoother. You can right-click a method or field in the Java Structure window (or in the Code Editor) to initiate a refactoring action. When refactoring, JDeveloper searches your entire project and any projects that are listed in Tools > Project Properties, Dependencies.

Before proceeding with a refactoring action, you have the option of previewing the occurrences that will be updated (as the slide shows). You can then choose to continue with the refactoring action or cancel it. You can even undo a refactoring if needed.

Refactoring



- Drag-and-drop refactoring
- Refactoring across entire application
- Refactoring across source control
- Over 35 new refactoring operations, including:

  - **Rename Class**
  - **Rename Field**
  - **Rename Method**
  - **Rename Package**
  - **Rename Parameter**
  - **Change Method Signature**
  - **Introduce Variable**
  - **Introduce Field**
  - **Extract Interface**
  - **Use Supertype Where Possible**
  - **Move Class**
  - **Duplicate Class**
  - **Pull Members Up**
  - **Safe Delete**

ORACLE

Refactoring (continued)

You can refactor across an entire application. In particular, you can refactor across multiple projects (via project dependencies).

Some of the refactorings that are available are:

Rename Class: Renames a class, its constructors, and its source file and updates all references to the class in the project

Rename Field: Renames a field and updates all references to the field in the project

Rename Method: Renames a method and updates all references to the method in the project

Rename Package: Renames a package and updates all references to the package in the project (including organization of subpackages)

Rename Local Variable: Renames a local variable and updates all references to the variable

Rename Parameter: Renames a parameter and updates all references to the parameter

Introduce Variable/Field/Constant/Parameter: Replaces the selected expression with a new expression of the same type

Refactoring (continued)

Extract Method: Creates a new method from the selected code, setting up parameters for any variables that need to be passed to the new method

Extract Interface: Creates a new interface from any of the public methods in the current type definition and implements that interface for the current type

Use Supertype Where Possible: Replaces all occurrences of a type with one of its supertypes if applicable

Pull Members Up: Promotes the declaration of methods or fields to the supertype and updates references accordingly

Push Members Down: Moves the declaration of methods or fields to all the subtypes of the current type and updates references accordingly

Delete Safely: Checks to make sure that the element you are attempting to delete is not actually being used in your code before allowing the delete operation to proceed. If references are found, you are warned and given the option to cancel the delete operation. You can use Delete Safely when deleting types, methods, and fields.

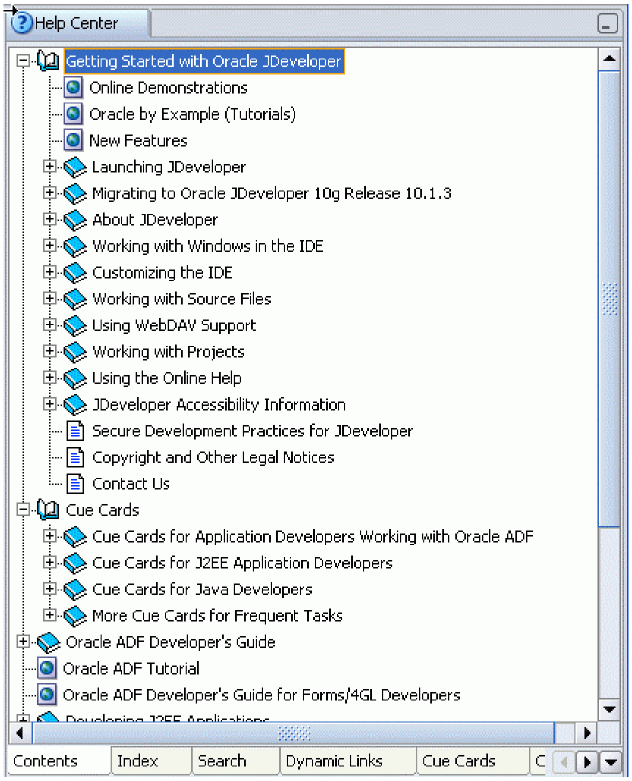
Using Javadoc

- Javadoc is the industry standard for documenting Java classes.
- When you work in Jdeveloper, you can browse the Javadoc-generated reference documentation for classes or interfaces.
- You can also add documentation comments to your source files when you develop your applications.

Using Javadoc

Javadoc is the industry standard for documenting Java classes. JDeveloper allows you to browse the Javadoc-generated reference documentation for classes or interfaces. JDeveloper provides two options from the context menu: Quick Javadoc and Go to Javadoc (full Javadoc).

JDeveloper Help System



Help Center

- Getting Started with Oracle JDeveloper
  - Online Demonstrations
  - Oracle by Example (Tutorials)
  - New Features
  - Launching JDeveloper
  - Migrating to Oracle JDeveloper 10g Release 10.1.3
  - About JDeveloper
  - Working with Windows in the IDE
  - Customizing the IDE
  - Working with Source Files
  - Using WebDAV Support
  - Working with Projects
  - Using the Online Help
  - JDeveloper Accessibility Information
  - Secure Development Practices for JDeveloper
  - Copyright and Other Legal Notices
  - Contact Us
- Cue Cards
  - Cue Cards for Application Developers Working with Oracle ADF
  - Cue Cards for J2EE Application Developers
  - Cue Cards for Java Developers
  - More Cue Cards for Frequent Tasks
- Oracle ADF Developer's Guide
- Oracle ADF Tutorial
- Oracle ADF Developer's Guide for Forms/4GL Developers
- Developing J2EE Applications

JDeveloper Help System

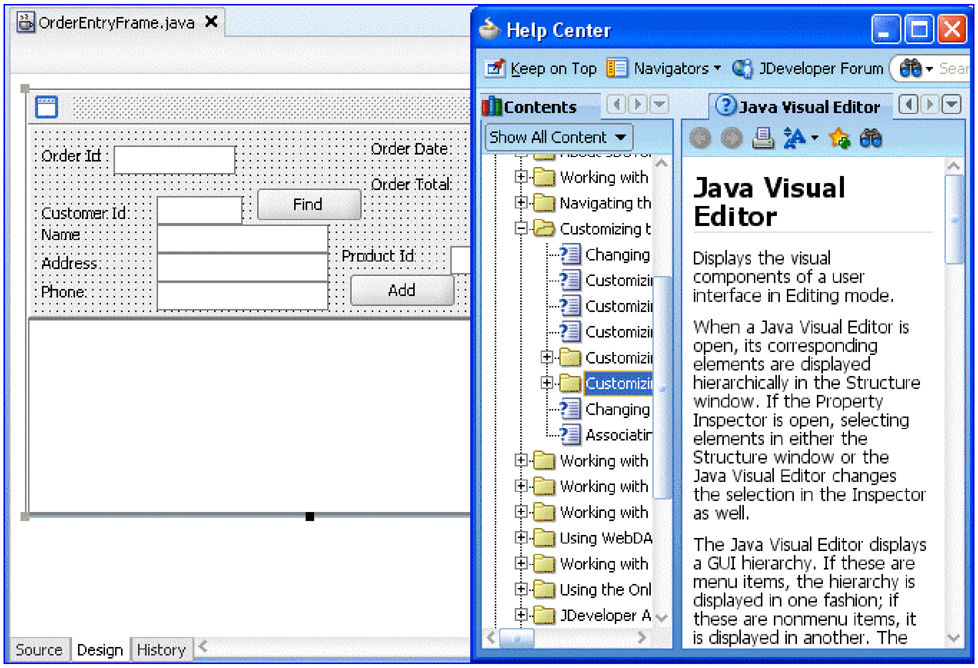
To make the best use of JDeveloper tools and libraries, and of Java itself, there is a comprehensive help system that covers all aspects of Java development.

To access the help system:

- Select Help > Help Topics in the main menu. The help system appears.
- Select one of the topics from the content navigator at the left of the window. After you select a topic, the topic expands to display subtopics.
- Right-click a topic to display the help text in the content window.
- Use hypertext links to navigate within a topic or to related topics.

Obtaining Help on a Topic

Press F1 to invoke context-specific help.



Java Visual Editor

Displays the visual components of a user interface in Editing mode.

When a Java Visual Editor is open, its corresponding elements are displayed hierarchically in the Structure window. If the Property Inspector is open, selecting elements in either the Structure window or the Java Visual Editor changes the selection in the Inspector as well.

The Java Visual Editor displays a GUI hierarchy. If these are menu items, the hierarchy is displayed in one fashion; if these are nonmenu items, it is displayed in another. The

Obtaining Help on a Topic

Pressing the F1 key invokes a context-sensitive Help topic window.

Oracle JDeveloper Debugger

- Helps find and fix program errors:
  - Run-time errors
  - Logic errors
- Allows control of execution
- Allows examination of variables

Oracle JDeveloper Debugger

Debugging is the process of looking for program errors that prevent your program from doing what you intended. There are two basic types of program errors: run-time errors and logic errors. Remember that the compiler catches any syntax problems.

If your program successfully compiles but gives run-time exceptions or hangs, you have a run-time error. That is, your program contains valid statements but is encountering errors when they are executed. For example, you may be trying to open a file that does not exist or you may be trying to divide by zero.

Logic errors are errors in the design and implementation of your program. That is, your program statements are valid and do something, but the results are not what you intended. This type of error is usually the most difficult to find.

The debugger enables you to control the execution of your program. It provides the ability to execute parts of the code step by step. You can also set breakpoints that pause the program execution when it reaches the line of code you want to examine.

When the program is paused, you can inspect and even modify program variables. This helps you examine loops and other control structures to make sure that what is happening is what you intended.

Oracle JDeveloper Debugger (continued)

Breakpoints

Breakpoints are a convenient way of tracing the cause of a problem in a program. When the debugger encounters a breakpoint, it pauses program execution. You can resume execution, stepping through the code line by line and examining variables and conditions. Or you can simply stop the program. You can set as many breakpoints as you want. Breakpoints are particularly useful when you know where your programming error starts. You can then set a breakpoint at that line and have the program execute until it reaches the breakpoint.

Watchpoints

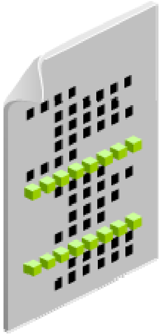
A watchpoint is a type of breakpoint. A watchpoint is a breakpoint that breaks on a value change.

Watchpoints enable you to pause the debugger when the value of a specified field is accessed or modified. You set a watchpoint by right-clicking a variable in the Code Editor and selecting Toggle Watchpoint from the context menu.

Breakpoints

Setting breakpoints:

- Manage multiple breakpoints
- Manage conditional breakpoints
- Define columns displayed in window:
  - Description
  - Type
  - Status
- Control scope of action
  - Global > Application > Project



Breakpoints

Setting Breakpoints

To set a breakpoint, you select a line of code in the source code window, right-click, and then select Toggle Breakpoint. You can click in the left margin to set a new breakpoint. After you start debugging, breakpoints that are known to be valid have a check mark in the breakpoint icon. A breakpoint without a check mark may mean that this line does not represent code where the debugger can stop. However, it might simply mean that the debugger does not yet know whether the breakpoint is valid or invalid.

Viewing Breakpoints

To view all the currently enabled breakpoints, select View > Breakpoints from the menu bar. A window appears showing all the breakpoints that are set in the program. To disable or remove a breakpoint, right-click the breakpoint and select an action from the menu.

Conditional Breakpoints

To set the conditions on which you want a breakpoint to be activated, right-click a breakpoint and select Edit Breakpoint. On the Breakpoint Conditions tab, you can specify information about how and when the breakpoint is activated (including valid Java conditional statements and thread-specific conditions).

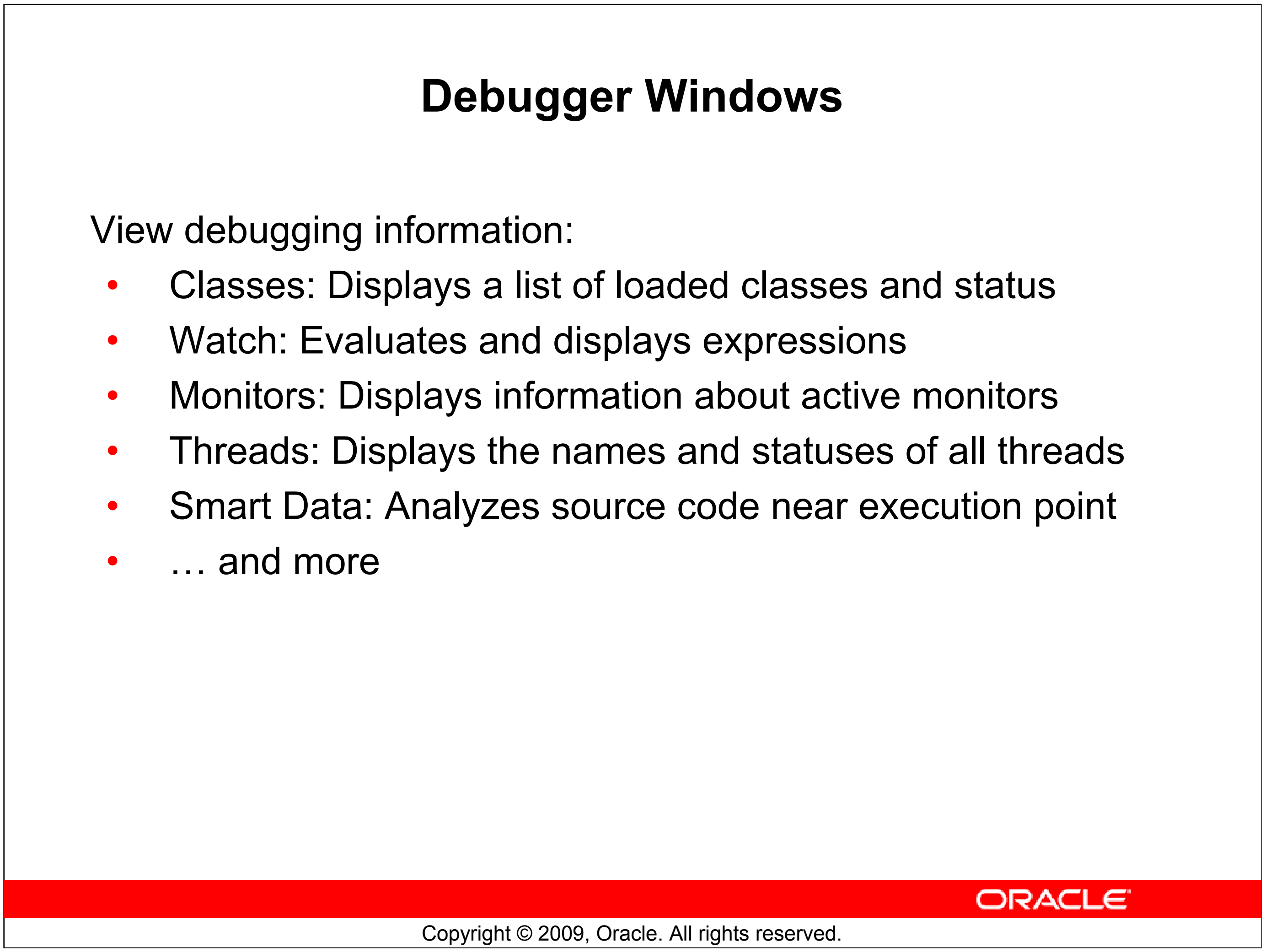
Breakpoints (continued)

Display Settings

To select the columns that are displayed in the breakpoints window, right-click in the breakpoints window and select Settings. In the dialog box, select Debugger > Breakpoints in the navigator tree on the left, and then select the columns to display.

Scope Settings

To select the scope for debugging, right-click in the breakpoints window, select Change Scope, and then select the appropriate value.



Debugger Windows

View debugging information:

- Classes: Displays a list of loaded classes and status
- Watch: Evaluates and displays expressions
- Monitors: Displays information about active monitors
- Threads: Displays the names and statuses of all threads
- Smart Data: Analyzes source code near execution point
- … and more

Debugger Windows

Make sure that the project is selected, and then click the Debug icon. Alternatively, in the Menu bar you can select Debug > Debug <Project Name>`.jpr`. This causes any required files to be compiled and then starts your program in debug mode.

Debugging Windows

As soon as you start the debugger, three tabs are added to a new window at the lower-right side of JDeveloper: the Smart Data tab, the Data tab, and the Watch tab. A new tab is added to the existing message window. This tab lets you monitor the code as it executes. Fields for each window can be modified in the Tools > Preferences menu in the Debugger node.

- **Smart Data tab:** Displays only the data that appears to be relevant to the source code you are stepping through
- **Data tab:** Displays all the arguments, local variables, and static fields for the current method
- **Watch tab:** Displays the current value of an expression that you have flagged to appear during execution of the program

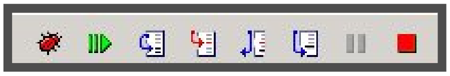
Remote Debugging

You can manually launch the program you want to debug and then start the debugger. In the Host list, select the name of the machine where the program has started. After the debugger is attached to the running program, remote debugging is similar to local debugging.

Stepping Through a Program

Use the buttons on the debugger toolbar:

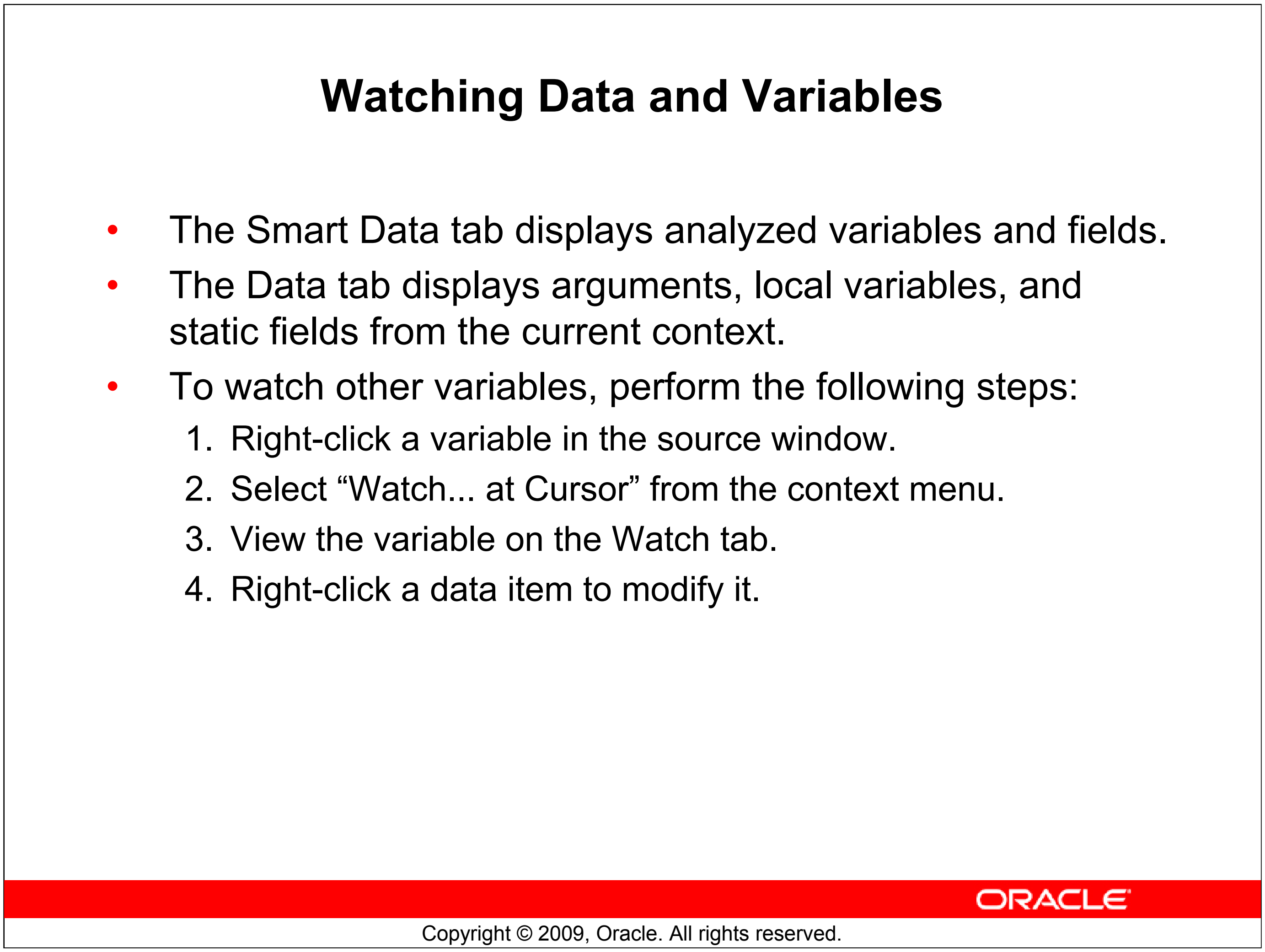
- Start the debugger.
- Resume the program.
- Step over a method call.
- Step into a method call.
- Step out of a method call.
- Step to the end of the method.
- Pause execution.
- Stop the debugger.



Stepping Through a Program

Click the following buttons in the debugger toolbar:

- **Start the debugger:** Executes the program in debug mode. The program is paused when it encounters a breakpoint. If no breakpoints are set, you can pause the program by using the “Pause execution” button.
- **Resume the program:** Resumes the program after stopping at a breakpoint
- **Step over a method call:** Executes the method at the current position in the program at full speed rather than tracing the method line by line
- **Step into a method call:** Traces a method line by line. This is useful when you suspect that the method may be the one that is causing the problem.
- **Step out of a method call:** Enables you to step out of the current method and return to the next instruction of the calling method
- **Step to the end of the method:** Jumps to the end of the method
- **Pause execution:** Pauses a running program at its current position
- **Stop the debugger:** Stops the execution of a running program. This is a useful way to kill the program.



Watching Data and Variables

- The Smart Data tab displays analyzed variables and fields.
- The Data tab displays arguments, local variables, and static fields from the current context.
- To watch other variables, perform the following steps:
  1. Right-click a variable in the source window.
  2. Select “Watch... at Cursor” from the context menu.
  3. View the variable on the Watch tab.
  4. Right-click a data item to modify it.

Watching Data and Variables

Viewing Analyzed Data on the Smart Data Tab

The debugger analyzes the source code near the execution point, looking for variables and field expressions that are used in the lines of code. By default, the debugger analyzes only one line of code for each location.

Viewing Local Variables on the Data Tab

The Data tab is the lower window that is displayed when you click the Debug tab at the bottom of the Application Navigator. The Data tab automatically displays a list of local variables, static fields, and arguments that are in scope. As you jump from one method to the next, the list of variables displayed on the Data tab changes.

Selecting Other Variables and Expressions to Watch

Other variables and expressions can be viewed by following the steps described in the slide. Select a variable or expression such as `age+10`. Right-click the variable or expression and select “Watch… at Cursor” from the context menu. A dialog box displays the selected variable or expression; click OK to accept it and add it to the Watch tab. View the variables that you have selected to monitor on the Watch tab. To modify a data value, right-click it and select Modify Value from the context menu.

Summary

In this lesson, you should have learned how to:

- Create applications and projects in Oracle JDeveloper
- Navigate around the JDeveloper IDE
- Use the JDeveloper Help system effectively
- Debug applications with the JDeveloper debugger

Additional Reading

You can learn more about the features of JDeveloper on the Oracle Technology Network (http://otn.oracle.com). The following are some examples of the topics covered.

Overview

Overview of JDeveloper, JDev IDE, Creating a BC4J Application, Java Concepts in JDeveloper, Naming Conventions, Debugging the Code, Deployment Alternatives

ADF Application Development Framework

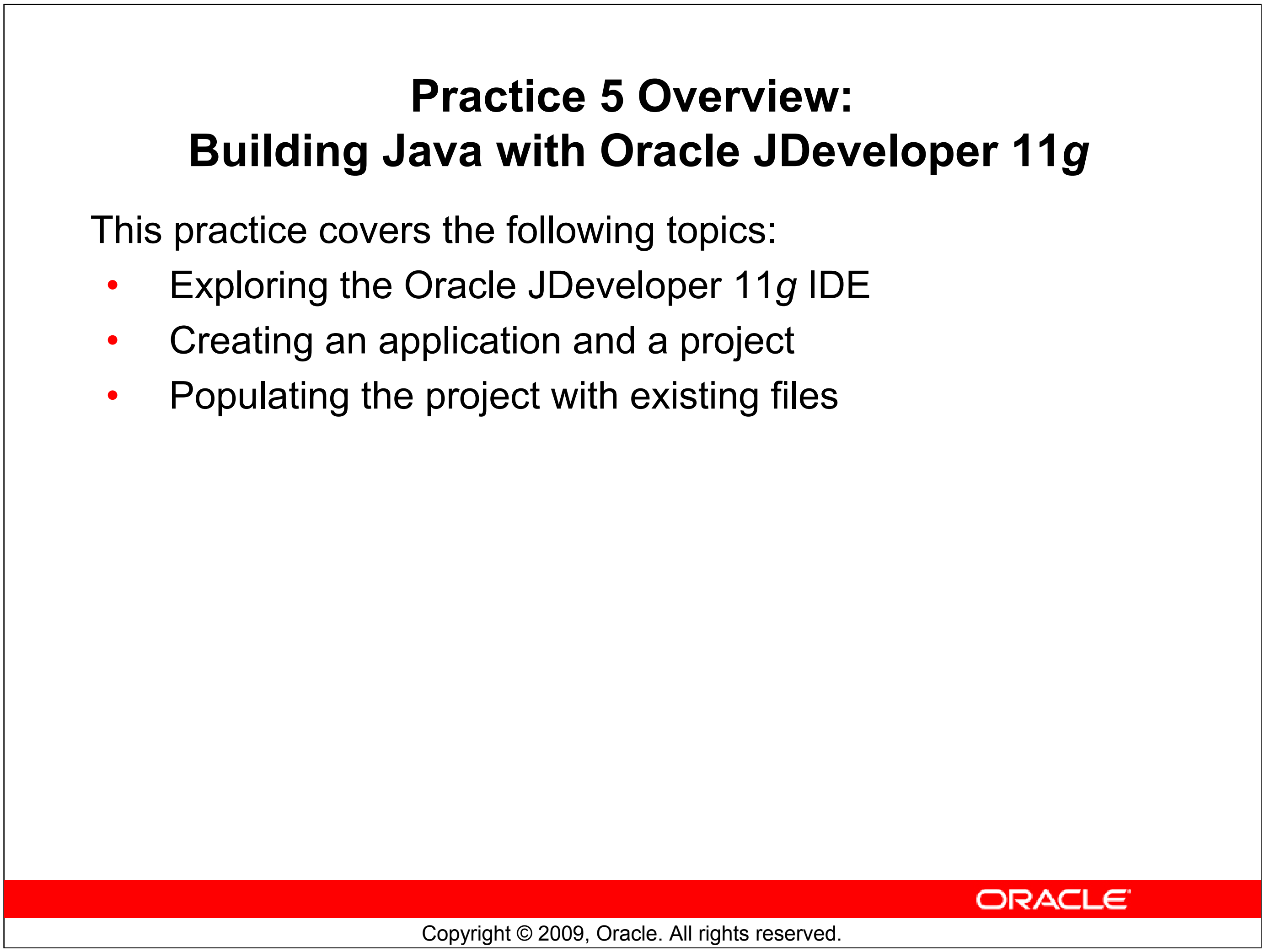
Reduces the complexity of Java EE development by providing visual and declarative development; increases development productivity; encourages Java EE best practices by implementing standard Java EE design patterns MVC; and provides a flexible and extensible environment by allowing multiple technology choices and development styles

JSPs

JSP Overview, Creating JSPs, JSP Components

Advanced Topics

Oracle ADF Tutorial, Oracle by Example (Tutorials)



Practice 5 Overview: Building Java with Oracle JDeveloper 11*g*

This practice covers the following topics:

- Exploring the Oracle JDeveloper 11*g* IDE
- Creating an application and a project
- Populating the project with existing files

Practice 5 Overview: Building Java with Oracle JDeveloper 11*g*

Starting in Practice 5, you use JDeveloper to build an application by using techniques you use during real-world development. The practice supports the technical material presented in the lesson and incorporates best practices to use when developing a Java application.

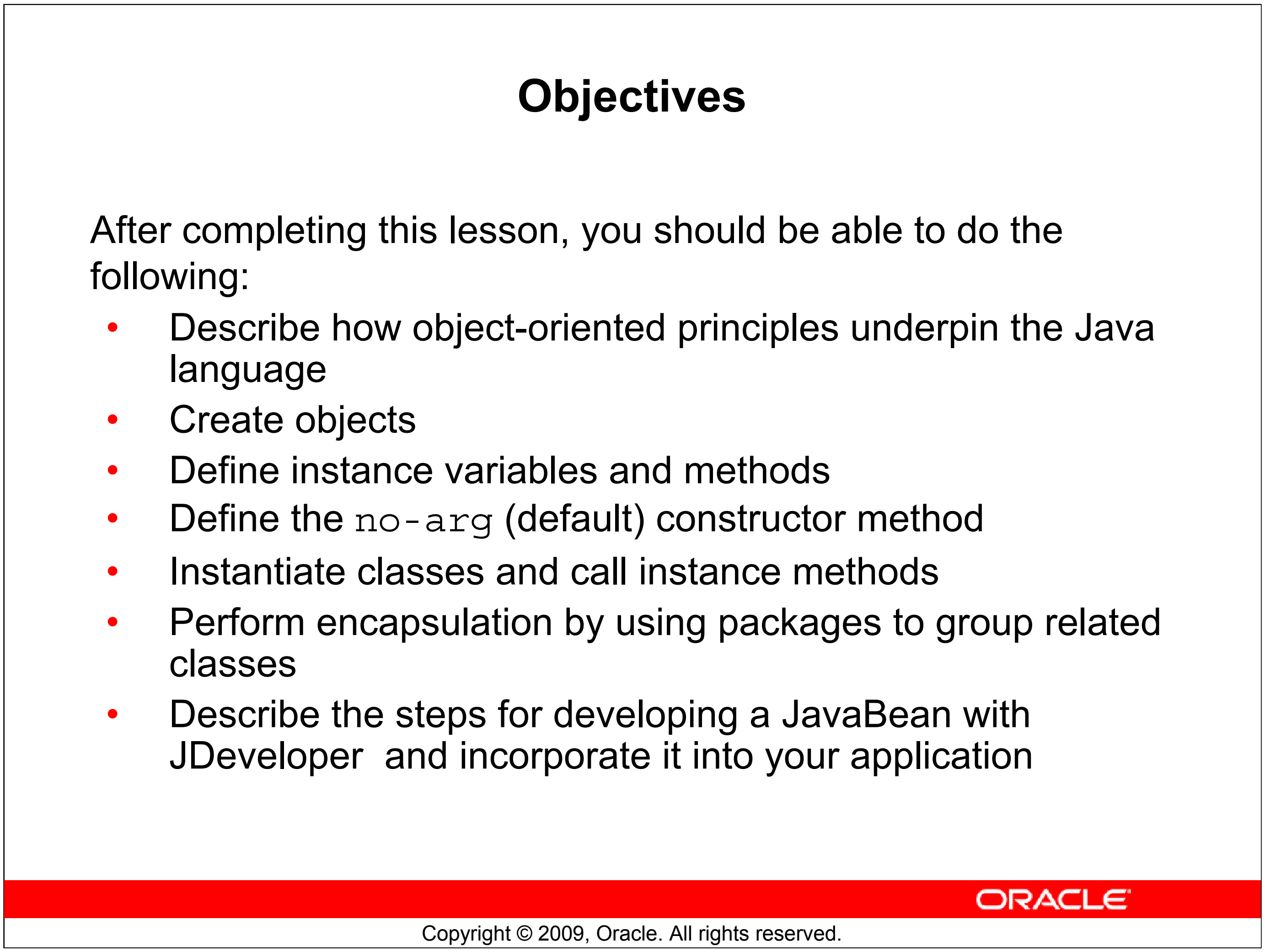
Goal

In this practice, you explore using the Oracle JDeveloper IDE to create an application and a project so that you can manage your Java files more easily during the development process. You learn how to create one or more Java applications and classes using the rapid code-generation features (such as the Code Editor and debugger).

More importantly, you now start using JDeveloper for most of the remaining lab work for this course (occasionally returning to the command line for various tasks). By the end of the course, you will have built and deployed the course GUI application while continuing to develop your Java and JDeveloper skills.

6

Creating Classes and Objects



Objectives

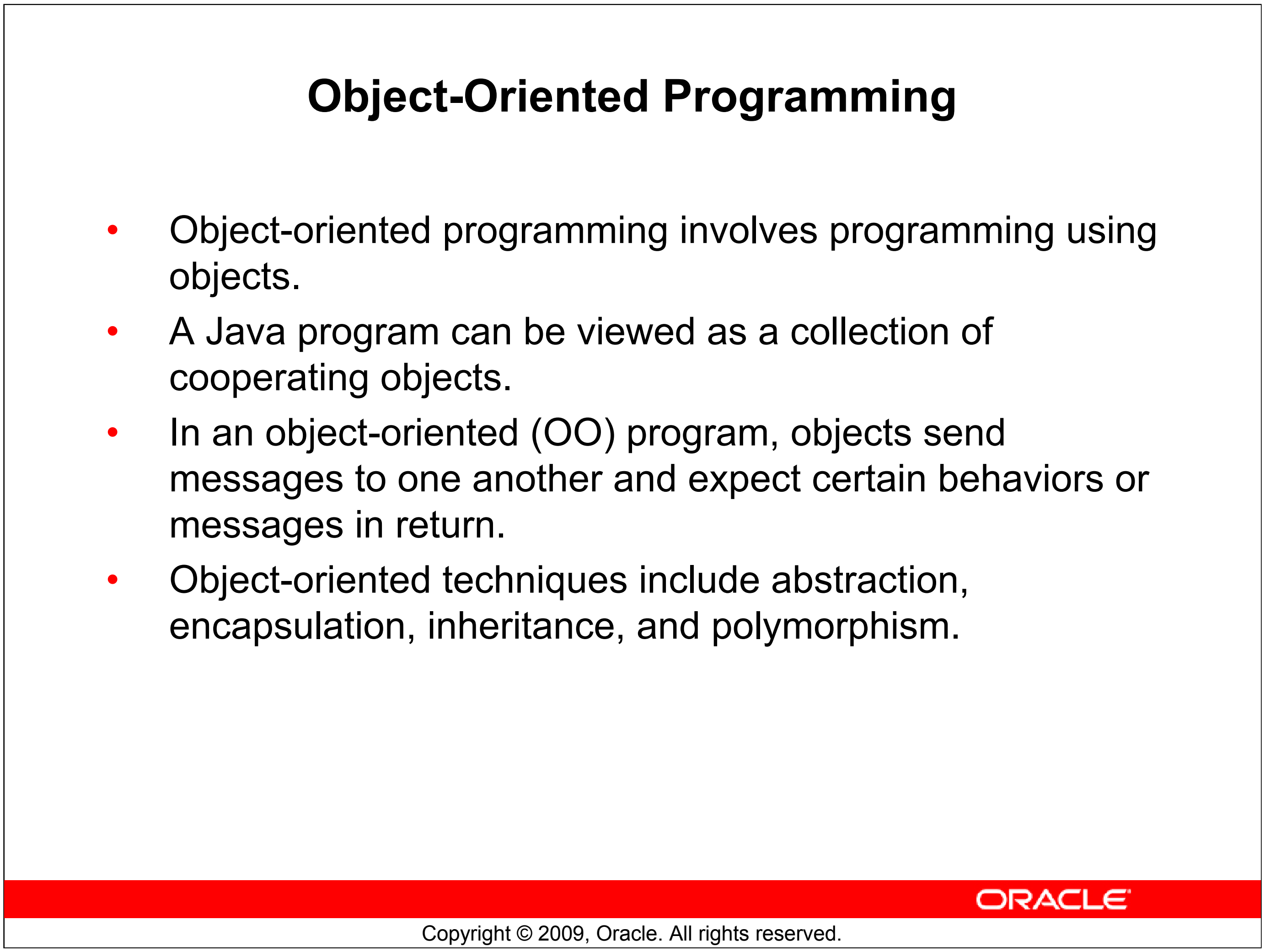
After completing this lesson, you should be able to do the following:

- Describe how object-oriented principles underpin the Java language
- Create objects
- Define instance variables and methods
- Define the `no-arg` (default) constructor method
- Instantiate classes and call instance methods
- Perform encapsulation by using packages to group related classes
- Describe the steps for developing a JavaBean with JDeveloper and incorporate it into your application

Lesson Objectives

This lesson introduces the principles of object-oriented programming and shows how they underpin the Java language. You learn how to create classes and you then define instance variables and methods for the classes. You will also see how to create objects using basic object initialization and how to manipulate the objects using instance methods.

JavaBeans are reusable software components written in Java for Java applications. Because they are platform independent, they can be reused not only within a particular application but also across different platforms and even in a distributed network environment such as the Internet. A bean is essentially a Java class (or group of classes) that conforms to a particular design pattern or rule and is designed for a specific purpose. In this lesson, you learn how to build a JavaBean and incorporate it into JDeveloper, ready for use in an application.



Object-Oriented Programming

- Object-oriented programming involves programming using objects.
- A Java program can be viewed as a collection of cooperating objects.
- In an object-oriented (OO) program, objects send messages to one another and expect certain behaviors or messages in return.
- Object-oriented techniques include abstraction, encapsulation, inheritance, and polymorphism.

Overview of Object-Oriented Programming

Object-oriented programming (OOP) is a paradigm for creating computer programs that are adaptable and reusable and can stand the test of time. The functionality comes from the design of discrete classes that contain information about objects and about expected behaviors. In an OO program, objects send messages to one another and expect certain behaviors or messages in return.

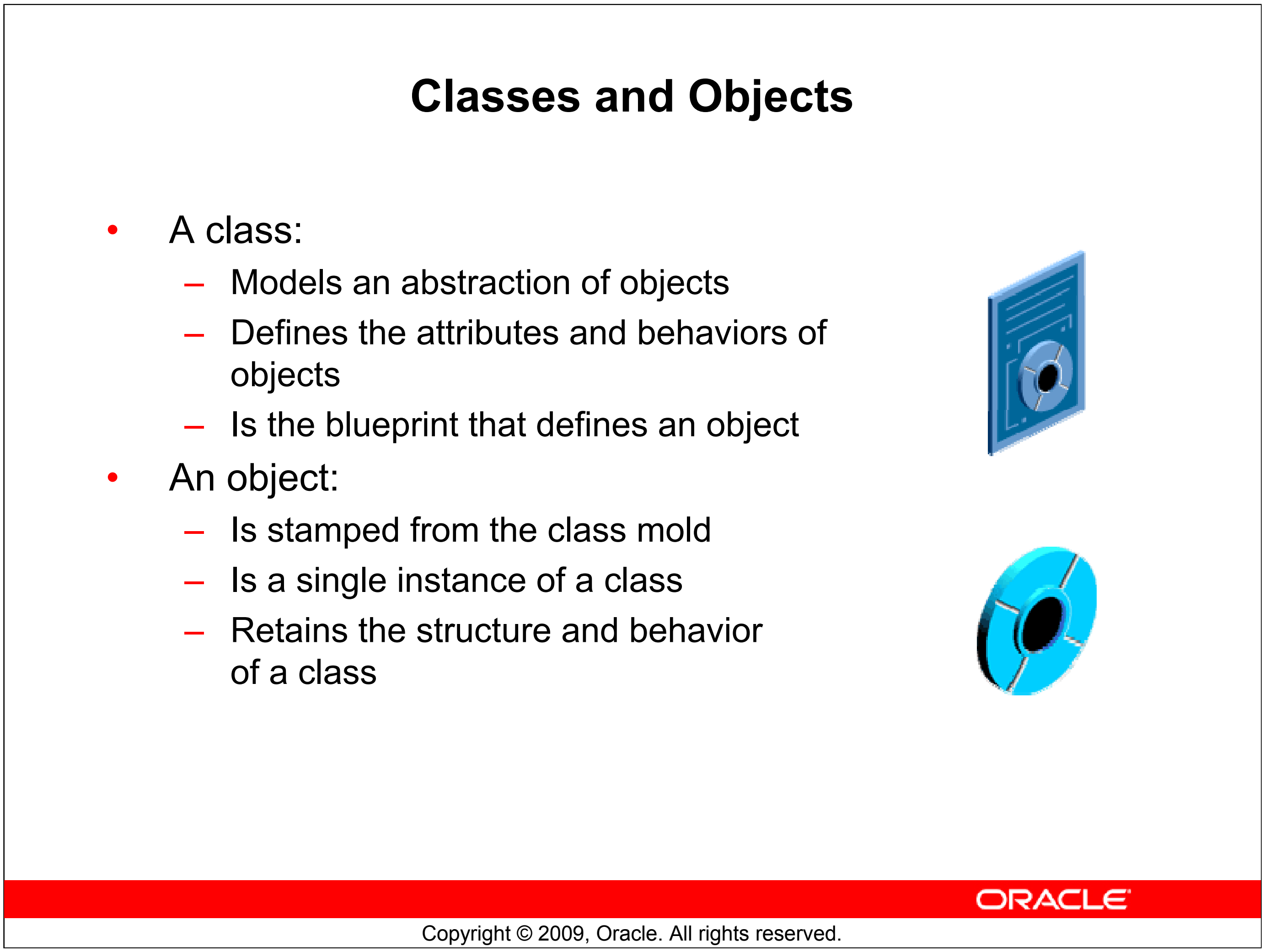
A good example of object orientation is the personal computer (PC). Although this is not a programming example, it serves as an example of what OOP can deliver in a programming environment.

Each PC is made up of components that are manufactured by several unrelated companies. Each component is built to a specification that includes information and behaviors. A CD drive, for example, is expected to return data from a CD when the operating system asks for it. The PC manufacturer does not need to be concerned with the internal workings of the CD drive, only that it responds to requests appropriately.

In the same way, an OO program may make calls to objects without knowing all the details of the objects. The program simply expects to get information or produce a specific behavior. Because each of these objects is defined separately, the internal workings of each object can change as long as the way objects are called and how they behave stay the same.

Overview of Object-Oriented Programming (continued)

OOP uses a number of techniques to achieve adaptability including abstraction, encapsulation, inheritance, and polymorphism. Each of these techniques is introduced in the next few slides.



Classes and Objects

- A class:
  - Models an abstraction of objects
  - Defines the attributes and behaviors of objects
  - Is the blueprint that defines an object
- An object:
  - Is stamped from the class mold
  - Is a single instance of a class
  - Retains the structure and behavior of a class

What Is an Object?

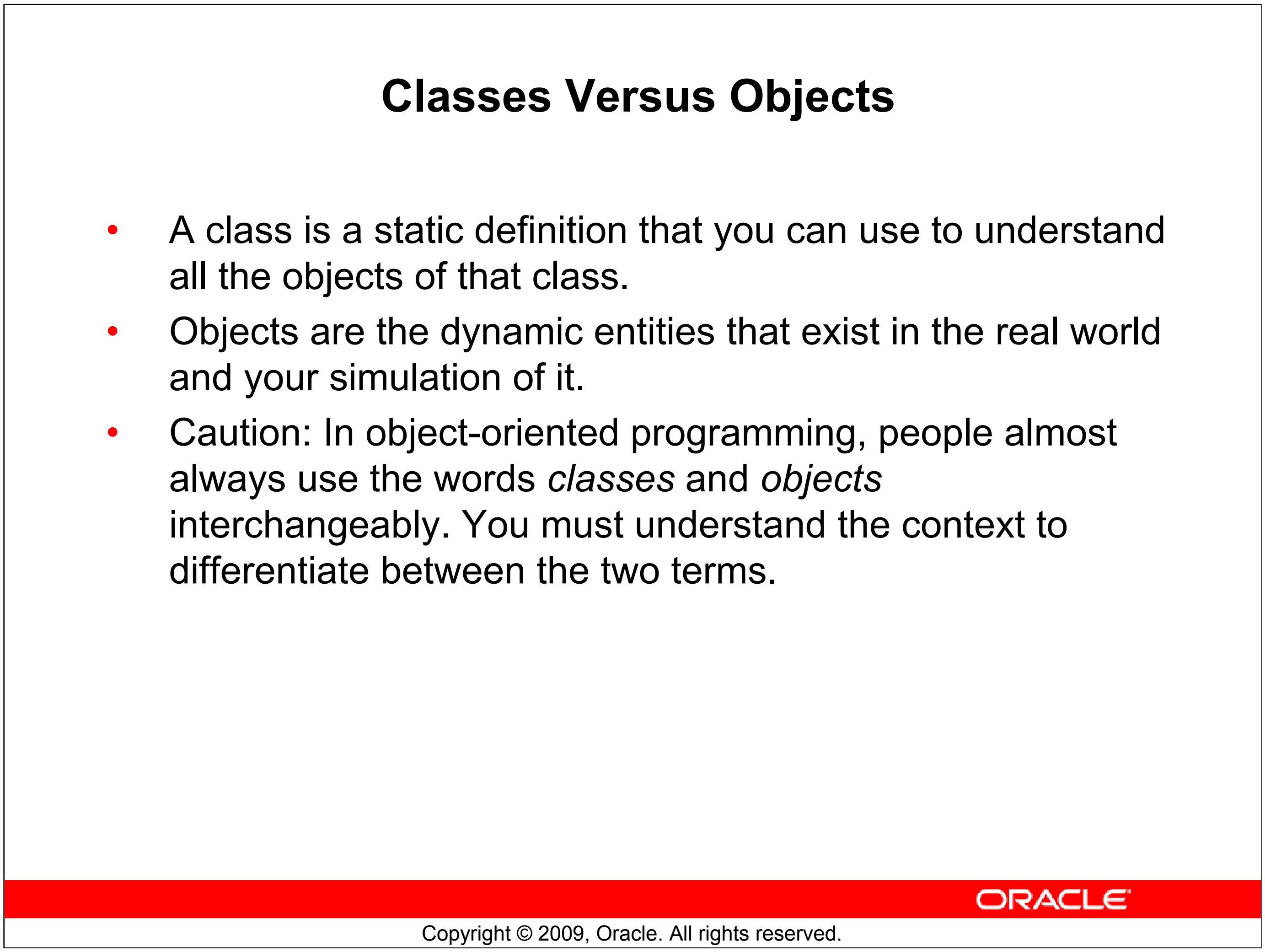
An object is something tangible (such as a chair or a house) or something that can be alluded to and thought about.

Object-oriented programs consist of several objects. These objects communicate with each other by sending messages from one object to another. In a true object-oriented program, that is all you have: a coherent group of communicating objects. New objects are created when needed, old ones are deleted, and existing objects carry out operations by sending messages.

Some examples of objects in an OO program might be Customer, Invoice, RentalAgreement, Video, and so on. Each of these objects holds information about itself (properties) and performs certain behaviors. A customer has a name, address, and phone number. A customer may also rent a video, return a video, pay a bill, and have other behaviors.

What Is a Class?

A class is a model of abstraction from real-world objects. A class defines the properties and behaviors of a set of objects. A class denotes a category of objects and acts as a blueprint for creating that type of object.



Classes Versus Objects

- A class is a static definition that you can use to understand all the objects of that class.
- Objects are the dynamic entities that exist in the real world and your simulation of it.
- Caution: In object-oriented programming, people almost always use the words *classes* and *objects* interchangeably. You must understand the context to differentiate between the two terms.

Classes Versus Objects

Objects exist only at run time. They hold attribute values, provide operations to be executed, and communicate by sending messages to each other. There can be many instances of a particular type of object.

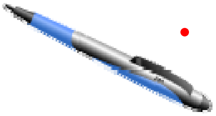
Classes are loaded into the run-time environment and used as a template to create the object instances. Although they are static definitions, they must be available at run time to be able to manufacture objects with all the qualities (for example, structure and function) defined in the class.

Here is an analogy from a purely structural perspective: A database table is a definition of a row. That is, the table structure can be loosely thought of as a class definition, and each row holds specific values as a separate instance with a structure that is defined by the table (the class). Both the table definition and its rows must exist in the database. There are many rows, but there is only one table definition.

Objects Are Modeled as Abstractions

- A Java object is modeled as an abstract representation of a real-world object.
- Model only those attributes and operations that are relevant to the context of the problem.

Problem domain: Product catalog
Real-world attributes and operations that you may want to model:



- **Attributes: Model, manufacturer, price**
- **Operations: Change price**

Real-world attributes and operations that you may *not* want to model:

- **Attributes: Ink color**
- **Operations: Refill, change color, point, write**

Objects Are Modeled as Abstractions

How do you decide what operations and attributes are relevant to the model of the `my blue pen` object? The answer is simple. You *must* understand how the object will be used by the other objects in the context of this particular system. You model the object as an abstraction of the real-world example in the context in which it exists.

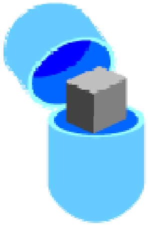
For example, in the context of a product catalog, the relevant attributes of a pen are reported, such as the model or name, price, and manufacturer (for example, Mont Blanc). The operations that are relevant to this catalog would be to change the price of this pen.

You may need to know whether the pen can be used to write text, be refilled, or have its ink color changed by replacing a cartridge. However, these latter operations are more relevant to the way the pen is used by the customer who is purchasing the pen, and therefore the refill, write, and change ink operations are not relevant to the catalog application context and must not be modeled. When deciding on the attributes and operations for an object, always ask whether they have relevance in the domain of the application. Always evaluate the attributes and operations in the application context by asking yourself the following question:
"Are they required to successfully implement the system to meet business requirements?"

Encapsulation

Encapsulation hides the internal structure and operations of an object behind an interface.

- A bank ATM is an object that gives its users cash.
  - The ATM hides (encapsulates) the actual operation of withdrawal from the user.
  - The interface (way to operate the ATM) is provided by the keyboard functions, screen, cash dispenser, and so on.
  - Bypassing encapsulation is bank robbery.
- Bypassing encapsulation in object-oriented programming is impossible.



ORACLE

Benefits of Encapsulation

With encapsulation, the developer can use an object and ignore the low-level details about how the object is structured and how it works internally. This frees the developer to think at a higher level of abstraction, resulting in the ability to comprehend more objects and to understand more complex systems.

In the real world, if you had to understand how things worked before being able to use them, you would not be able to deal with real-life objects such as ATMs, airplanes, microwave ovens, computers, or video recorders. To use these things, you deal with them by using their interfaces and ignoring their implementations.

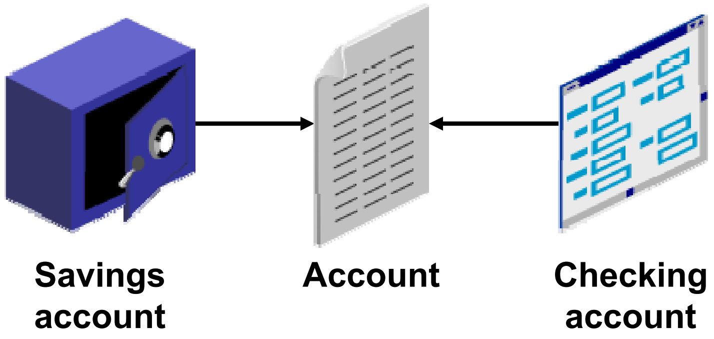
An added benefit is that the implementation of the operations may change, and you should still be able to use them in the usual manner.

Think again about the ATM. If the bank rewrites the software or changes the hardware of the ATM, it does not have to tell everyone how to use the new system. This is because the interface has not changed; you still use the ATM in the same manner. When you see minor changes to the interface of the ATM, it probably means that the software behind it has been completely rewritten.

In software terms, encapsulation is a mechanism to hide information and functionality, and it enforces security to control access to internal data and functionality. Programming rules prevent the encapsulation of an object from being violated.

Inheritance

- There may be a commonality between different classes.
- Define the common properties in a superclass.



- The subclasses use inheritance to include those properties.

Inheritance

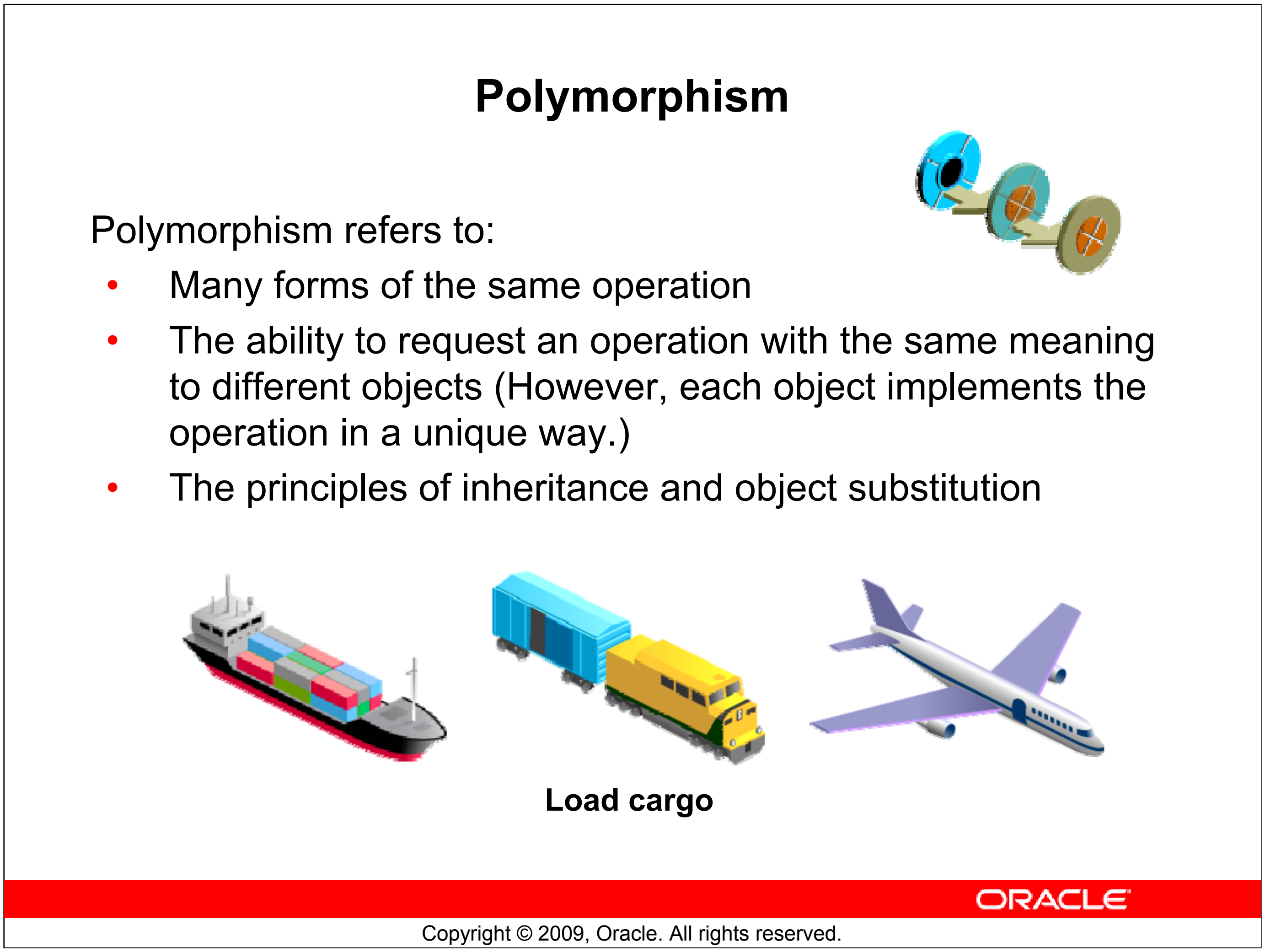
Inheritance is a relationship between classes in which one class declares another as its parent. When an object of the child class is created, it inherits all the properties of the parent class in addition to those defined in the child class itself. The child class can provide additional attributes and behavior that are relevant to it and can also redefine operations that are specified in the parent class if a different implementation is required.

Inheritance Clarifies Your Design

The use of classes to divide the world into a relatively small number of types is intuitive because humans have a natural tendency to classify. However, the model of the world is not flat. When you define two classes, you may notice that the classes share a lot of common attributes and operations. For example, imagine that you are implementing an air-traffic control system, and you define an airplane class and a helicopter class. These two classes have certain characteristics in common, and it makes sense to factor the common features into a parent class called aircraft. The airplane and helicopter classes can then inherit from aircraft; all you must specify in these classes is how they differ from the aircraft class.

Inheritance Also Improves Productivity

As well as improving the clarity of your design, inheritance can also improve productivity because new classes can be created quickly based on existing classes. Where a class exists but does not quite meet your needs, the original class need not be modified at all. Instead, a new subclass can be defined, and the differences can be specified in the new class.



Polymorphism

Polymorphism refers to:

- Many forms of the same operation
- The ability to request an operation with the same meaning to different objects (However, each object implements the operation in a unique way.)
- The principles of inheritance and object substitution

Load cargo

Polymorphism

The term *polymorphism* (Greek for “many forms”), as applied to object technology, means that the same operation can be defined for different classes, and each class can implement the operation in a different way. The implementation of an operation is called a *method*, and so polymorphism means that there may be many ways to implement methods for one operation. In the example in the slide, there is one operation to load passengers, but each class has a different method to implement the operation.

Why Is Polymorphism Important?

The importance of polymorphism is that the “load cargo” operation can be invoked on an object of any of the three classes, and the correct method is called automatically. This frees the caller from interrogating the object to determine its precise class. Polymorphism requires that an object send the same message (or a common message) to different objects, enabling the different objects to respond in their own way.

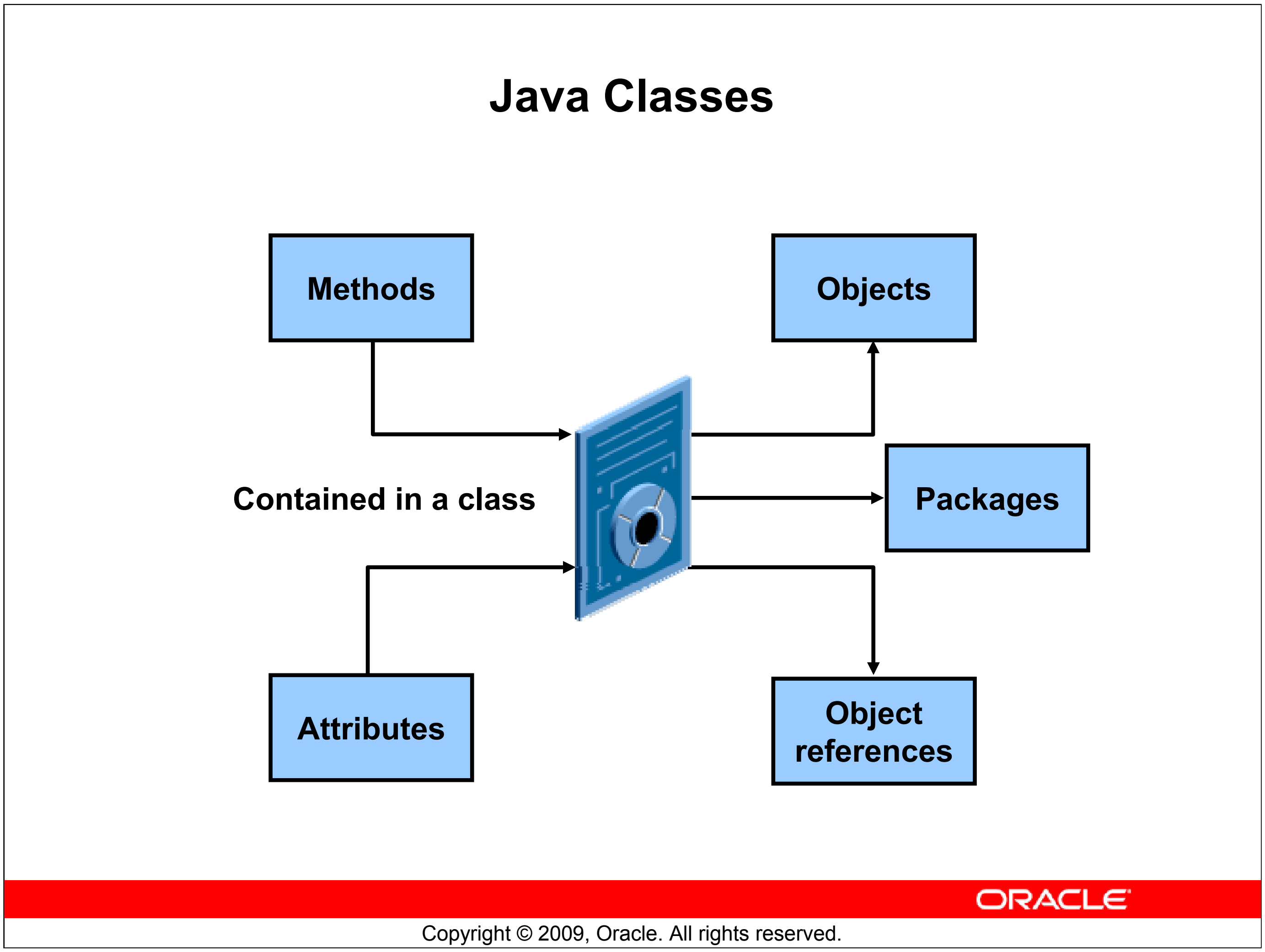
Inheritance and polymorphism are covered in detail in the “Inheritance and Polymorphism” lesson.

Guided Practice: Spot the Operations and Attributes

Guided Practice: Spot the Operations and Attributes

For each object in the slide, specify at least three attributes and two or three operations.

Objects	**Operations**	**Attributes**
My pencil	Write, erase	Lead color
My pen	Write, refill	Ink color, ink amount
Jaws	Eat, swim	Capacity, speed
Car		
Truck		
Satellite		



Java Classes

Methods

Objects

Contained in a class

Packages

Attributes

Object references

ORACLE

Java Classes

A class is a template or blueprint that is used in creating multiple objects. A class encapsulates all the data and behaviors that make up an object. When you ask Java to create or instantiate a class, Java uses the class definition as a template for building the new object.

A class contains attributes that are assigned to all new objects of that class. Attributes comprise the information or data that describes, classifies, categorizes, or quantifies an object. Each object of a given class has its own set of the class attributes. For example, an object of the Customer class may have a name, a billing address, and a telephone number attribute.

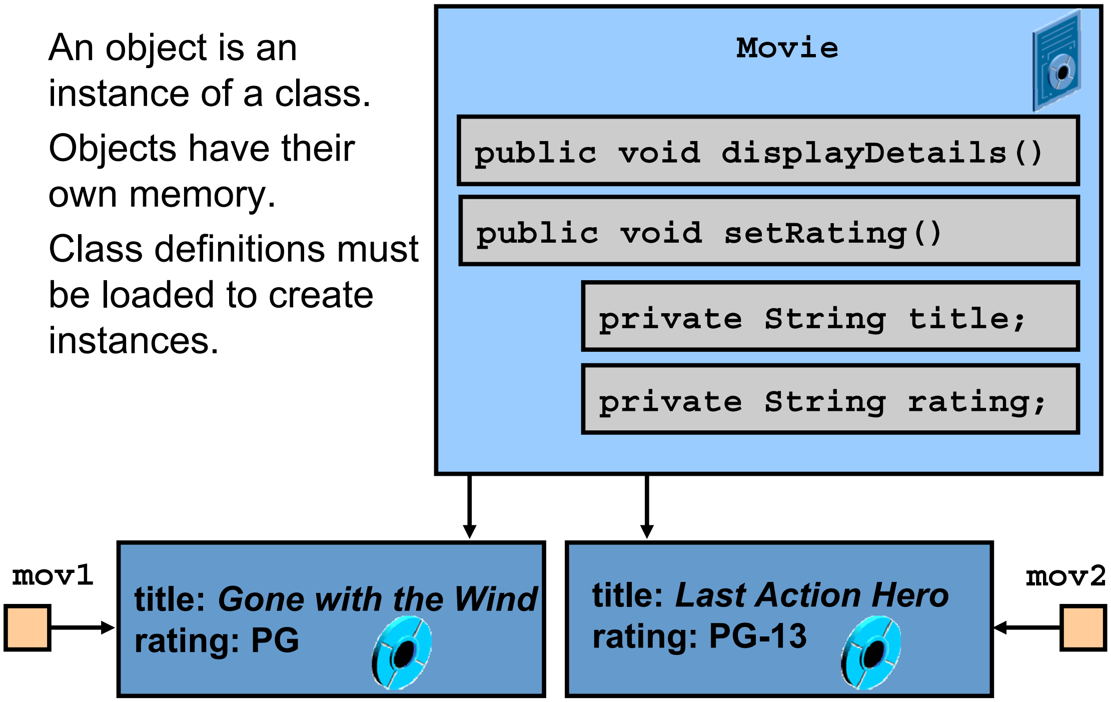
Data that is stored in an object can be primitive data types (such as integers or characters) or references to objects.

A class also contains methods—or functions—that specify the behavior or actions that an object can perform. For example, a customer can rent a video, make a payment, or change the billing address.

Java uses packages to group classes that are logically related. Packages consist of all the classes in a subdirectory. Packages are also used to control access from programs outside the package.

Comparing Classes and Objects

- An object is an instance of a class.
- Objects have their own memory.
- Class definitions must be loaded to create instances.



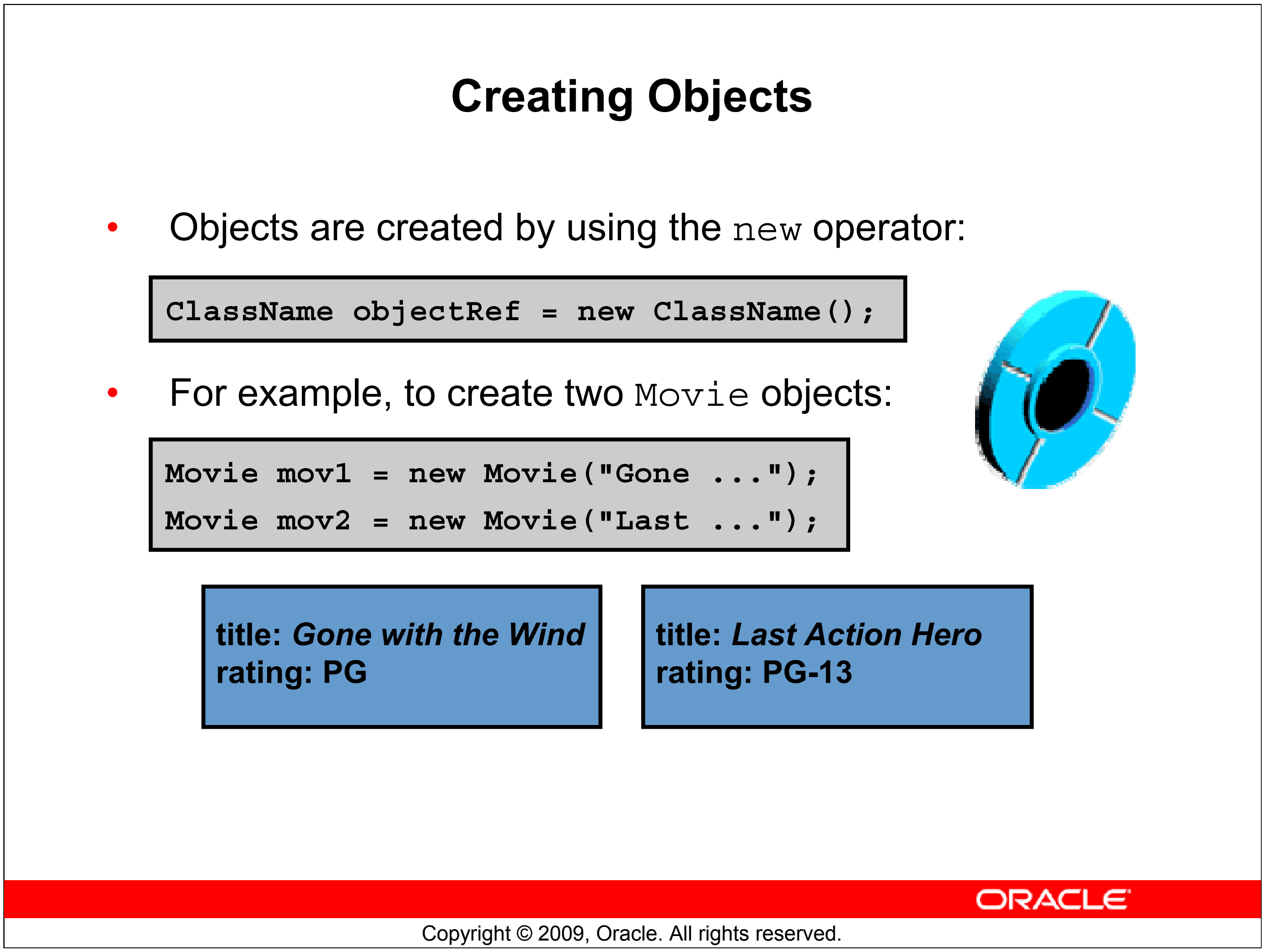
Comparing Classes and Objects

A class is a template for building objects of that class. In object-oriented terminology, an object is an instance of a class. Each object of that class has the same data structure and operations. However, the values that are held in this data structure are unique to each object; these values are therefore called instance variables. The operations of an object act on the instance variables in that object.

The class contains variables and methods called members. Those members owned by the class are "static members," and those owned by the objects from the class are called "instance members."

In the case of the `Movie` class, each individual movie is an instance of `Movie`. *Gone with the Wind* is one distinct instance of `Movie` and *Last Action Hero* is another. Each has its own set of variables that are separate and distinct from the variables of any other movie, or object, of the `Movie` class.

Each new object is identified in Java by a unique object reference. Java distinguishes among objects by using this reference. Objects are uniquely identifiable even if all their properties are the same.



Creating Objects

- Objects are created by using the `new` operator:

```
ClassName objectRef = new ClassName();
```

- For example, to create two `Movie` objects:

```
Movie mov1 = new Movie("Gone ...");
Movie mov2 = new Movie("Last ...");
```

title: *Gone with the Wind*
rating: PG

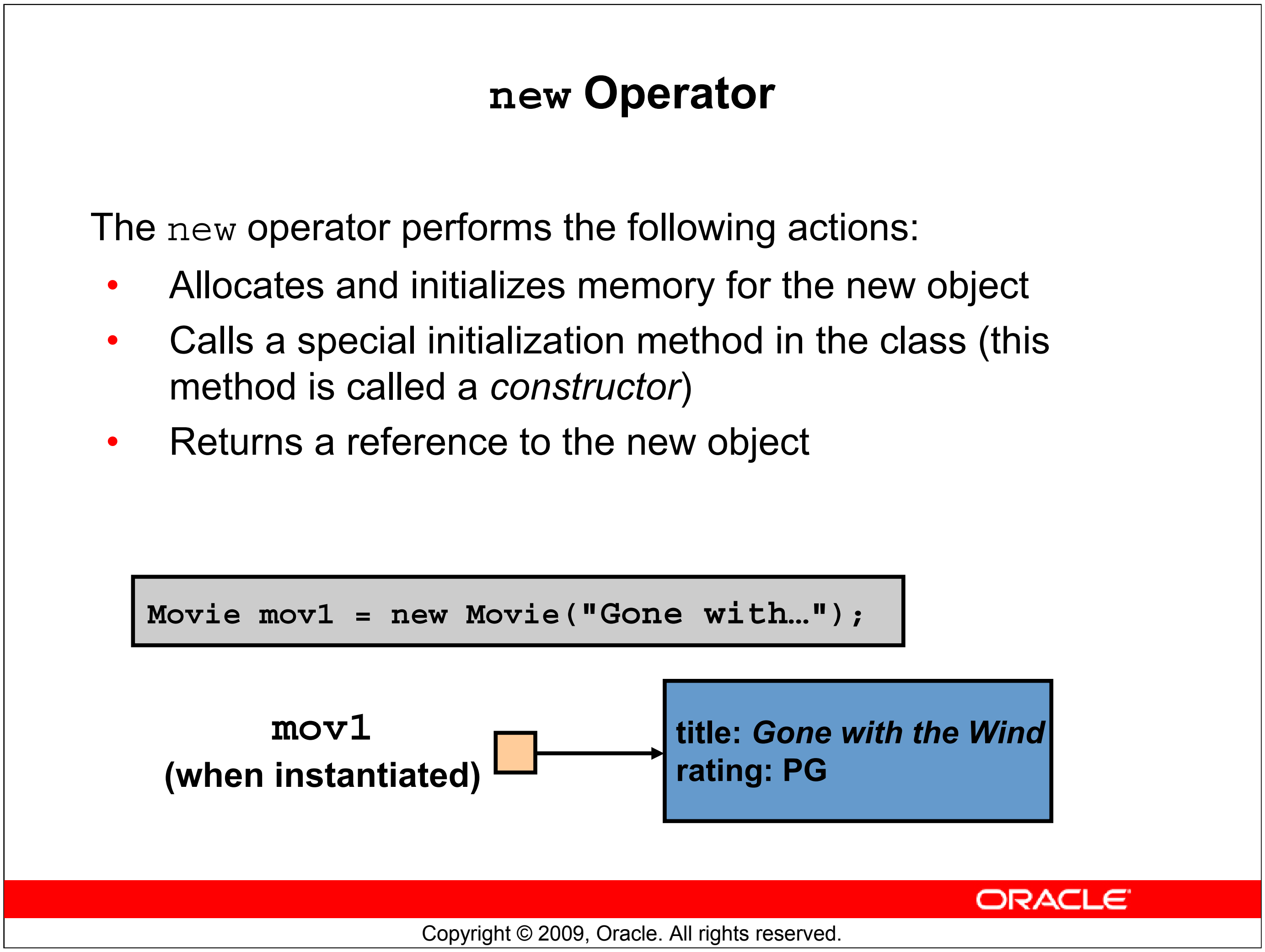
title: *Last Action Hero*
rating: PG-13

Creating Objects

In Java, you can create objects by using the `new` operator. The `new` operator creates an instance of a class and returns the reference of the new object.

```
        Movie mov1 = new Movie();
```

This statement creates an instance variable of the `Movie` type named `mov1`. It then creates a new instance of `Movie` by using the `new` operator, and then it assigns the object reference to the `mov1` instance variable. It is important to remember that the `new` operator returns a reference to the new object that points to the location of that object in memory.



`new` Operator

The `new` operator performs the following actions:

- Allocates and initializes memory for the new object
- Calls a special initialization method in the class (this method is called a *constructor*)
- Returns a reference to the new object

```
Movie mov1 = new Movie("Gone with…");
```

`mov1`
(when instantiated)

title: *Gone with the Wind*
rating: PG

`new` Operator

The `new` operator performs the following three tasks:

- Allocating memory for the new object. It knows how much memory is required by looking in the class definition to see what instance variables are defined in the class.
- Calling a constructor to initialize the instance variables in the new object. A constructor is a special method that is supplied by the class.
- Returning a reference to the newly created object. To refer to this object in the future, you must store this reference in a variable.

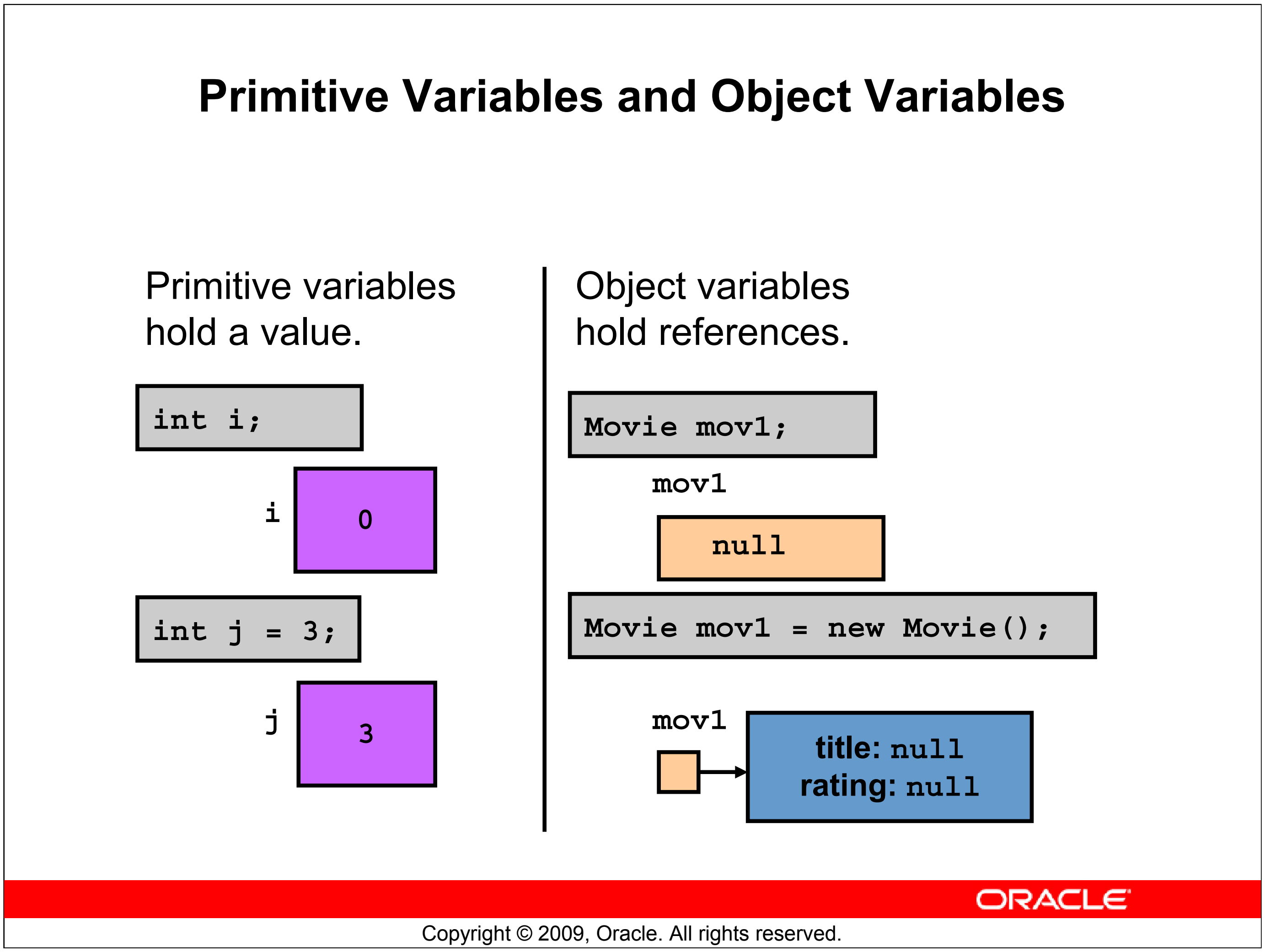
Separating Variable Declaration from Object Creation

The declaration of an object reference and the creation of an object are completely independent. In the previous examples, these two parts were combined in a single statement:

```
Movie mov1 = new Movie();
```

However, you can achieve the same effect with two separate statements:

```
Movie mov1;          // Declare an object reference,
                     // capable of referring to a Movie.
mov1 = new Movie();  // Create Movie object, and return the
                     // reference to the mov1 variable.
```



Primitive Variables and Object Variables

Primitive variables hold a value.

`int i;`

i 0

`int j = 3;`

j 3

Object variables hold references.

`Movie mov1;`

mov1 null

`Movie mov1 = new Movie();`

mov1

title: null
rating: null

Primitive Variables and Object Variables

Primitive variables are treated very differently from object variables in Java. It is important to understand the differences.

Primitive Variables

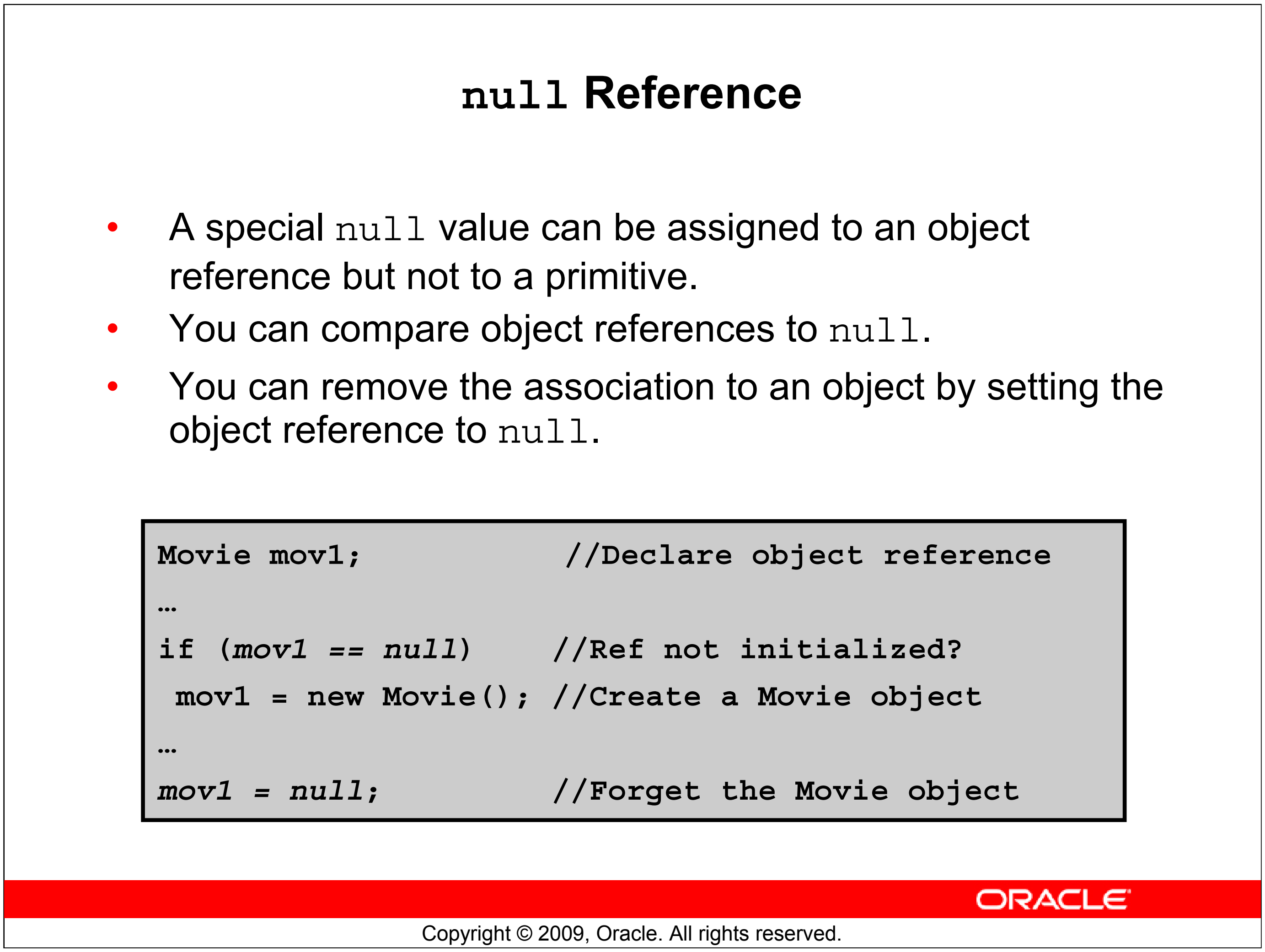
When you declare a primitive variable, Java allocates a chunk of memory to hold a variable of the specified primitive type. If you define a primitive variable as an instance or a class variable, then the variable is initialized to `0` if it is a number, `false` if it is a `Boolean`, or `\0` if it is a `char`.

Primitive variables hold their values directly. For example, if you declare an `int` variable and assign it the value `3`, the value is stored directly in the four bytes of the `int`.

Object Variables

When you declare an object variable, you also receive a chunk of memory, but this memory is only large enough to hold a reference to an object. You may find it useful to think of a reference as a "pointer" to an object.

As mentioned previously, declaring an object variable does not create an object of the specified class. Consequently, an object instance variable is initialized to `null` to indicate that it does not yet refer to an object. Use the assignment operator to make an object variable refer to an object. The assignment can be to an existing object or to a new object by using the `new` operator.



null Reference

- A special null value can be assigned to an object reference but not to a primitive.
- You can compare object references to null.
- You can remove the association to an object by setting the object reference to null.

```
Movie mov1;               //Declare object reference
…
if (mov1 == null)        //Ref not initialized?
 mov1 = new Movie(); //Create a Movie object
…
mov1 = null;             //Forget the Movie object
```

null Reference

Consider the following statement:

```
Movie mov1 = null;
```

This statement declares a variable called mov1 with a special reference called null, indicating that the reference does not yet refer to a real object.

Checking Object References Against null

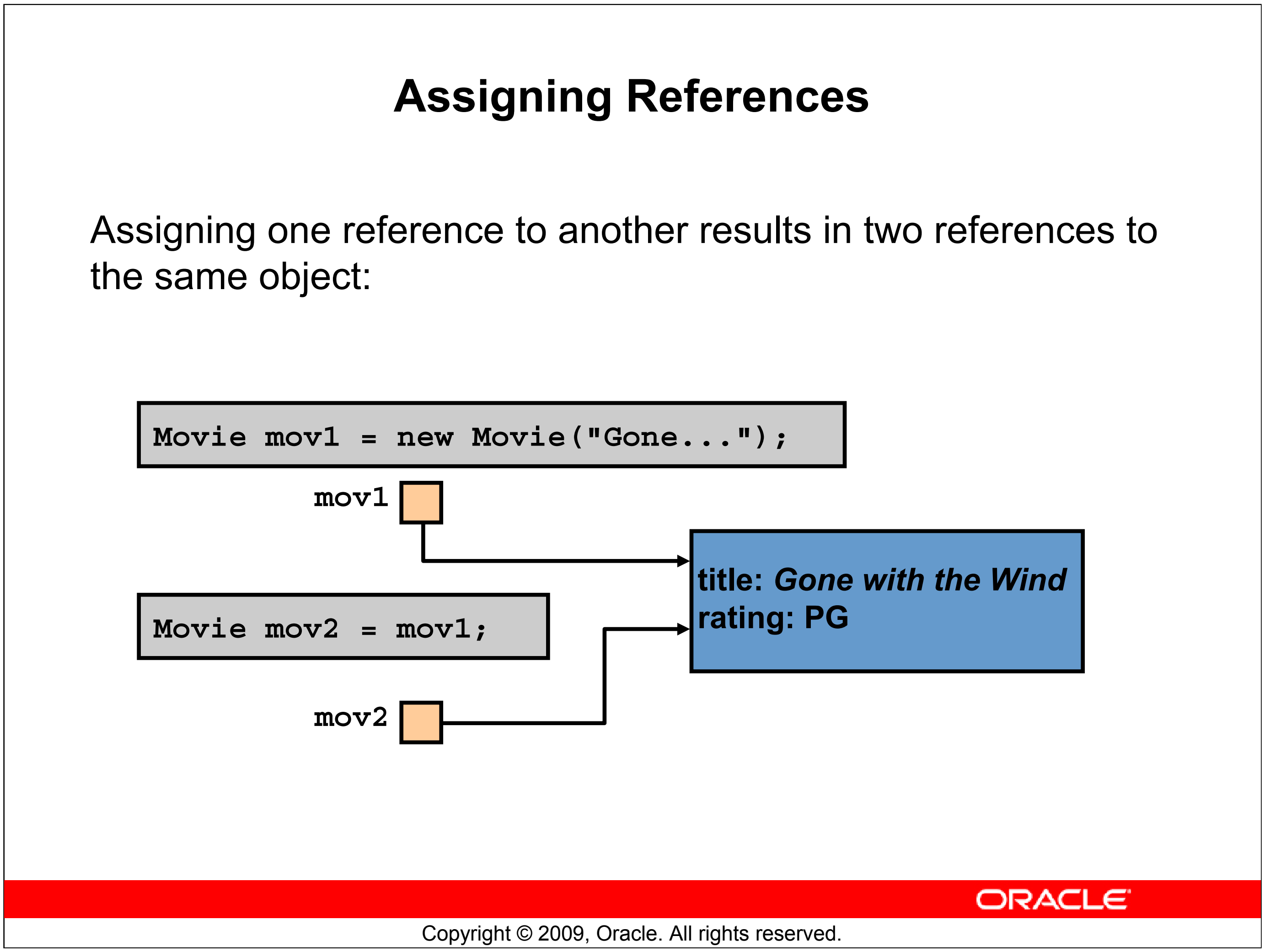
null is a keyword in Java. You can use it with the equality operator to determine whether an object reference has been initialized:

```
if (mov1 == null) {
  // The mov1 variable has not been initialized,
  // so do something to initialize it …
}
```

Discarding an Object

After you finish using an object, you can set its object reference to null. This indicates that the variable no longer refers to the object. When there are no more live references to an object, the object is marked for garbage collection.

The Java Virtual Machine (JVM) automatically decrements the number of active references to an object whenever an object is dereferenced or goes out of scope, or when the stored reference is replaced by another reference.



Assigning References

Assigning one reference to another results in two references to the same object:

```
Movie mov1 = new Movie("Gone...");
```

```
Movie mov2 = mov1;
```

Different Object References Can Refer to the Same Object

As previously mentioned, when you declare an object reference variable, it is initialized to `null`. Before you can use that variable, you must initialize it. Typically, you initialize it with a reference to a new object by using the following syntax:

```
Movie mov1 = new Movie();
```

However, it is also possible to initialize an object reference to an existing object as follows:

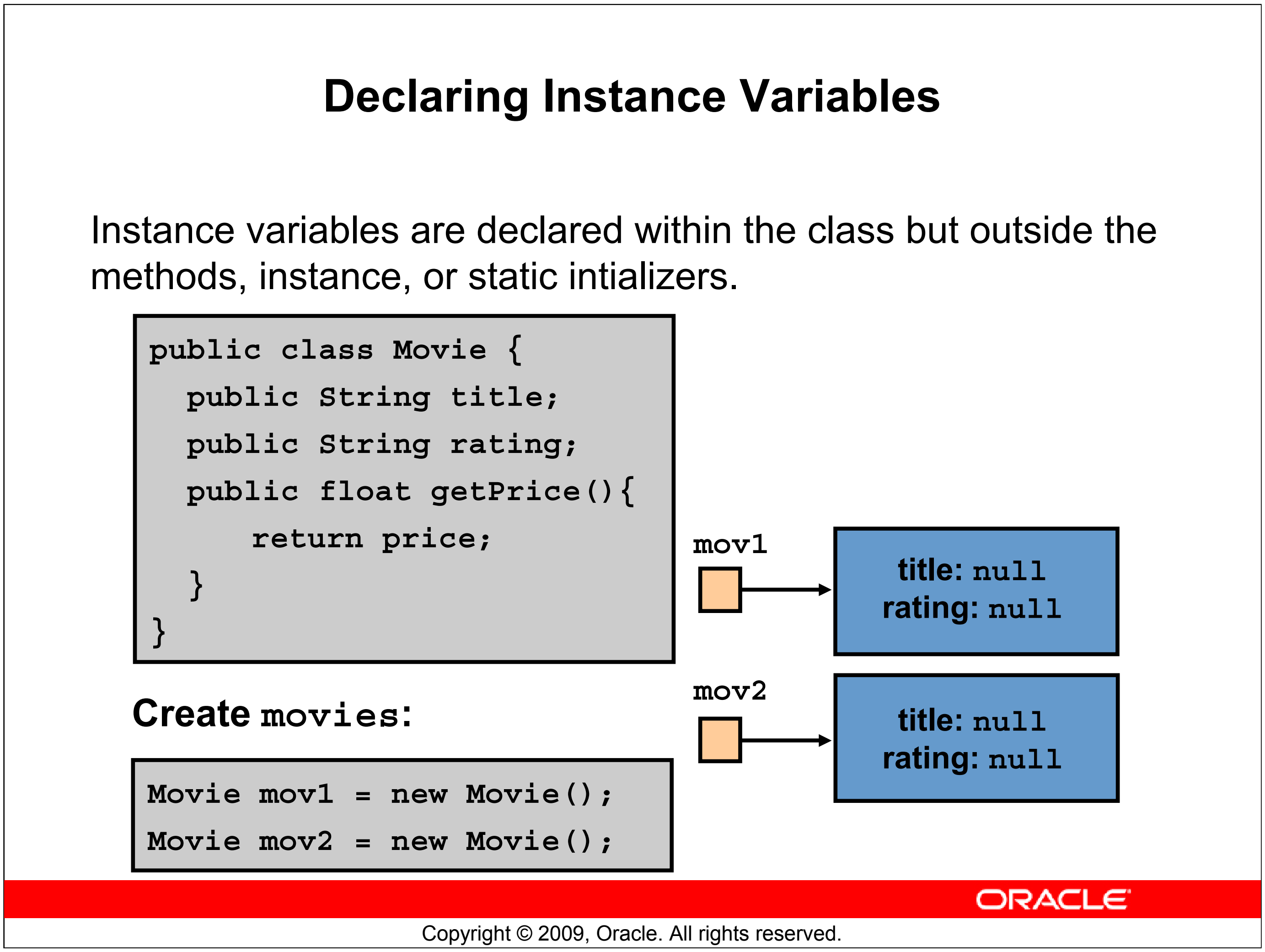
```
Movie mov2 = mov1;
```

This is perfectly legal Java syntax, but there is still only one `Movie` object. When you assign one object reference to another object reference, the result is two references to the same object, not a copy of the other object.

The Object Can Be Accessed Through Either Reference

You can access the same object through either reference; however, there is still only one object.

You can change either reference to refer to a different object without affecting the other reference. However, if you really need a separate object rather than multiple references to a single object, you must create a new object.



Declaring Instance Variables

Instance variables are declared within the class but outside the methods, instance, or static intializers.

```
public class Movie {
  public String title;
  public String rating;
  public float getPrice(){
     return price;
  }
}
```

Create movies:

```
Movie mov1 = new Movie();
Movie mov2 = new Movie();
```

mov1
title: null
rating: null
mov2
title: null
rating: null

Declaring Instance Variables

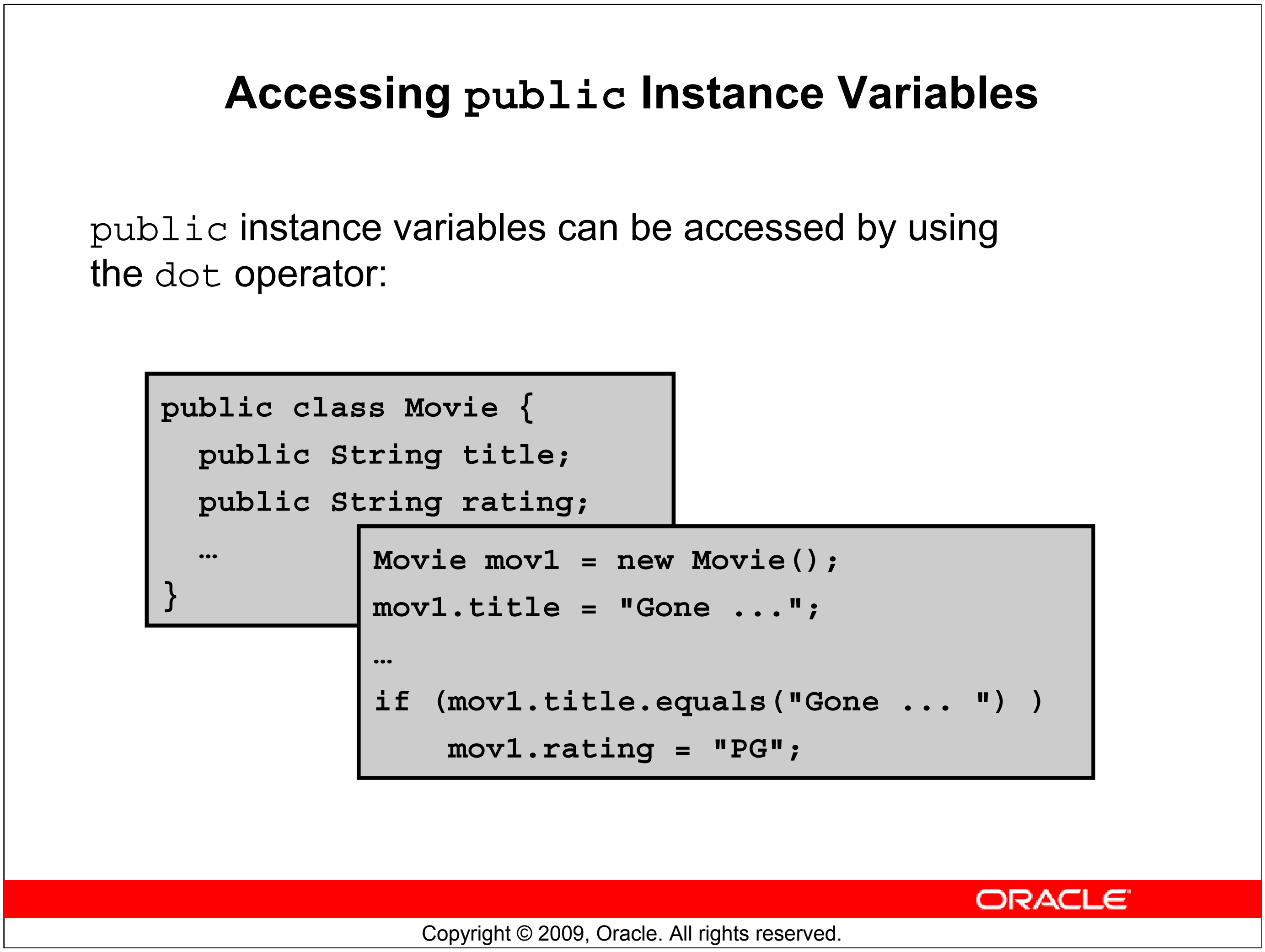
Instance variables must be declared within the class definition. In this example, these instance variables have been declared as public to allow users of the class to access them directly. Although this is not a good practice, it is done here to keep the example simple.

Declaring instance variables as public is a violation of encapsulation. After you have seen how to write instance methods, the instance variables will be made private to prevent them from being manipulated directly by users of the class.

What Does an Object Look Like?

The example creates two instances of the Movie class. That is, two Movie objects have been created. The first Movie object is referenced by mov1, and the second Movie object is referenced by mov2.

Although these two Movie objects are completely independent, they have an identical data structure. In other words, each object has its own separate and distinct copies of the instance variables that you declared in the Movie class.



Accessing `public` Instance Variables

`public` instance variables can be accessed by using the `dot` operator:

```
public class Movie {
  public String title;
  public String rating;
  …
}
```

```
Movie mov1 = new Movie();
mov1.title = "Gone ...";
…
if (mov1.title.equals("Gone ... ") )
    mov1.rating = "PG";
```

Accessing `public` Instance Variables

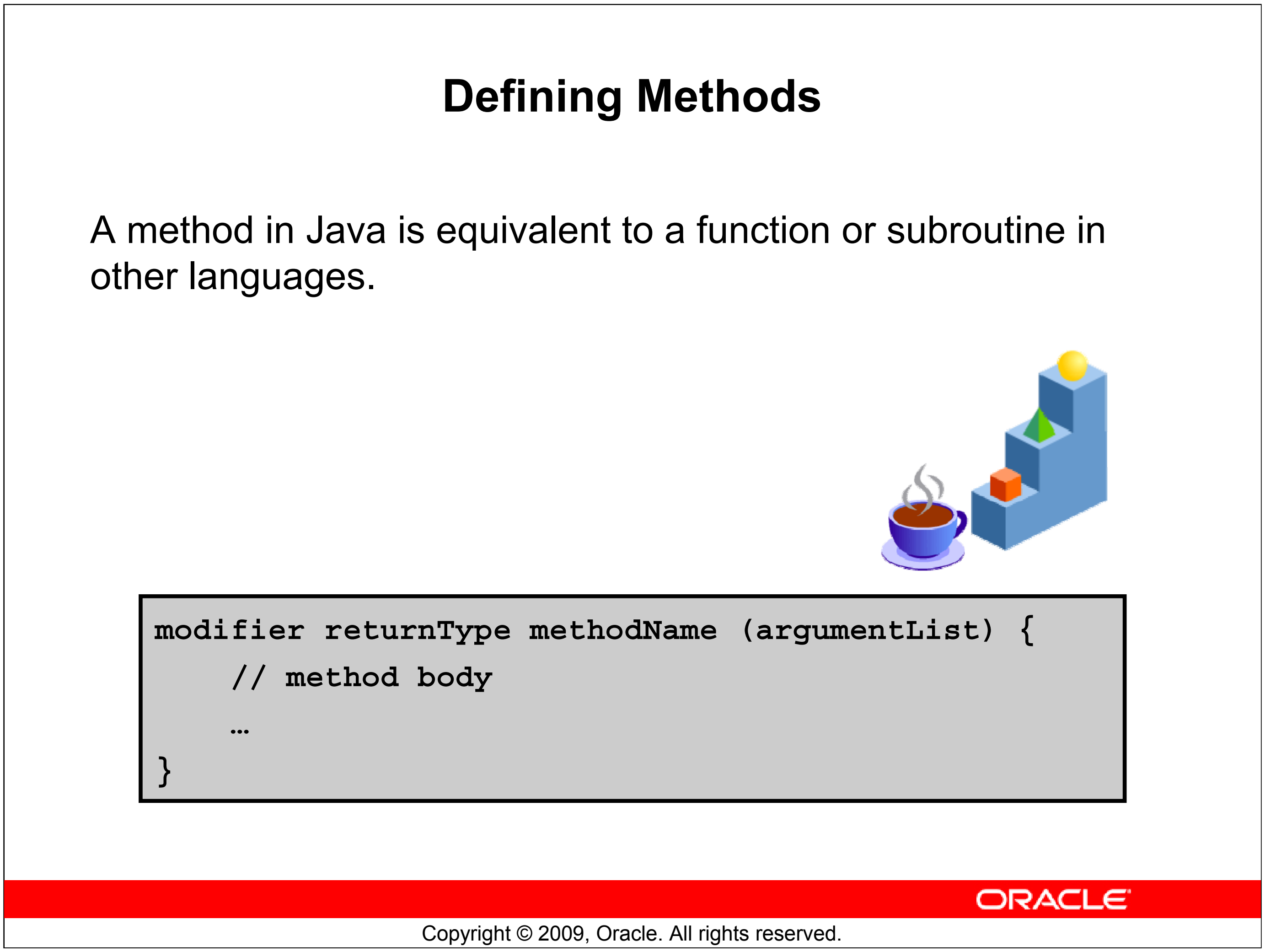
The example shows how to access `public` instance variables in an object by using the `dot` operator. This operator takes an object reference on the left and the name of an instance variable on the right:

```
objectRef.instanceVarName
```

Remember that this syntax is allowed only if the instance variable has been declared as `public`. In a fully encapsulated object, in which the instance variables are `private`, the only way to change the state of an object is by calling a suitable instance method on the object. Because the variables are exposed as `public`, anyone using your class can manipulate instance data without validation or control.

Users can set the `rating` to anything they want, regardless of business rules governing that data.

Note: `equals(...)` compares object types.



Defining Methods

A method in Java is equivalent to a function or subroutine in other languages.

```
modifier returnType methodName (argumentList) {
    // method body
    …
}
```

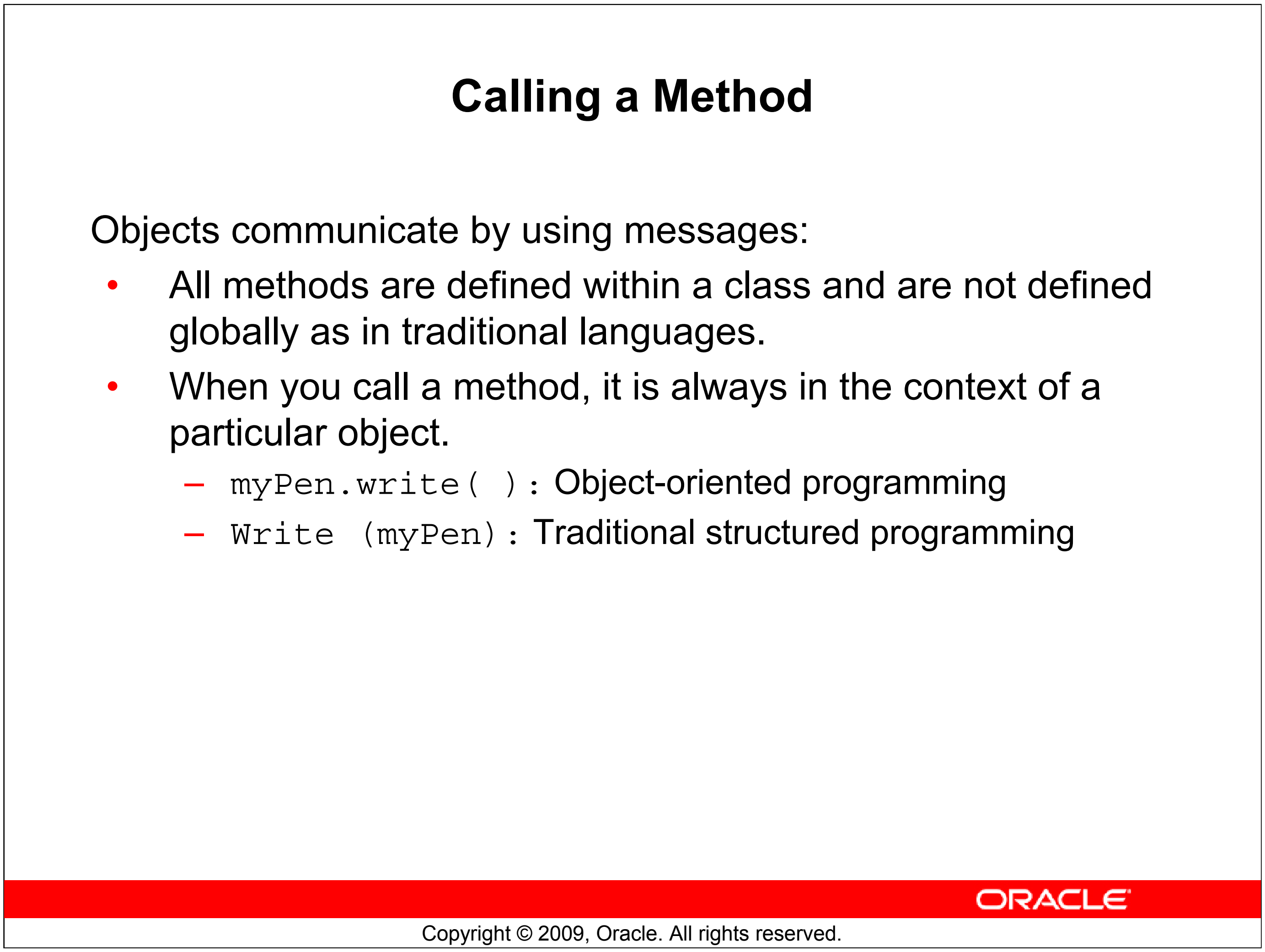
Defining Methods

When you define a class for an object-oriented program, you implement all of the behavior of that class in one or more methods. A Java method is equivalent to a function, procedure, or subroutine in other languages, except that it must be defined inside a class definition.
That is, there is no support for global methods in Java; every method must be defined within a class.

Anatomy of an Instance Method

The key components of an instance method are:

- A modifier, such as `public` or `private`. If you specify an instance method as `public`, it can be called from anywhere in the program. If you specify a method as `private`, it can be invoked only by other methods in the same class. If no modifier is specified, the method can be invoked by any method in any class in the same package. An instance method may also have a protected or a default modifier.
- A return type. This can be a primitive type (such as `int`), a class type (such as `String`), or the `void` keyword, if the method does not return a value.
- The name of the method. A Java naming convention expects method names to begin with a lowercase letter. Compound words in the method name must begin with an uppercase character.
- An optional argument list inside parentheses, separated by commas. If the method does not take any arguments, simply leave the parentheses empty.
- The method body enclosed in braces



Calling a Method

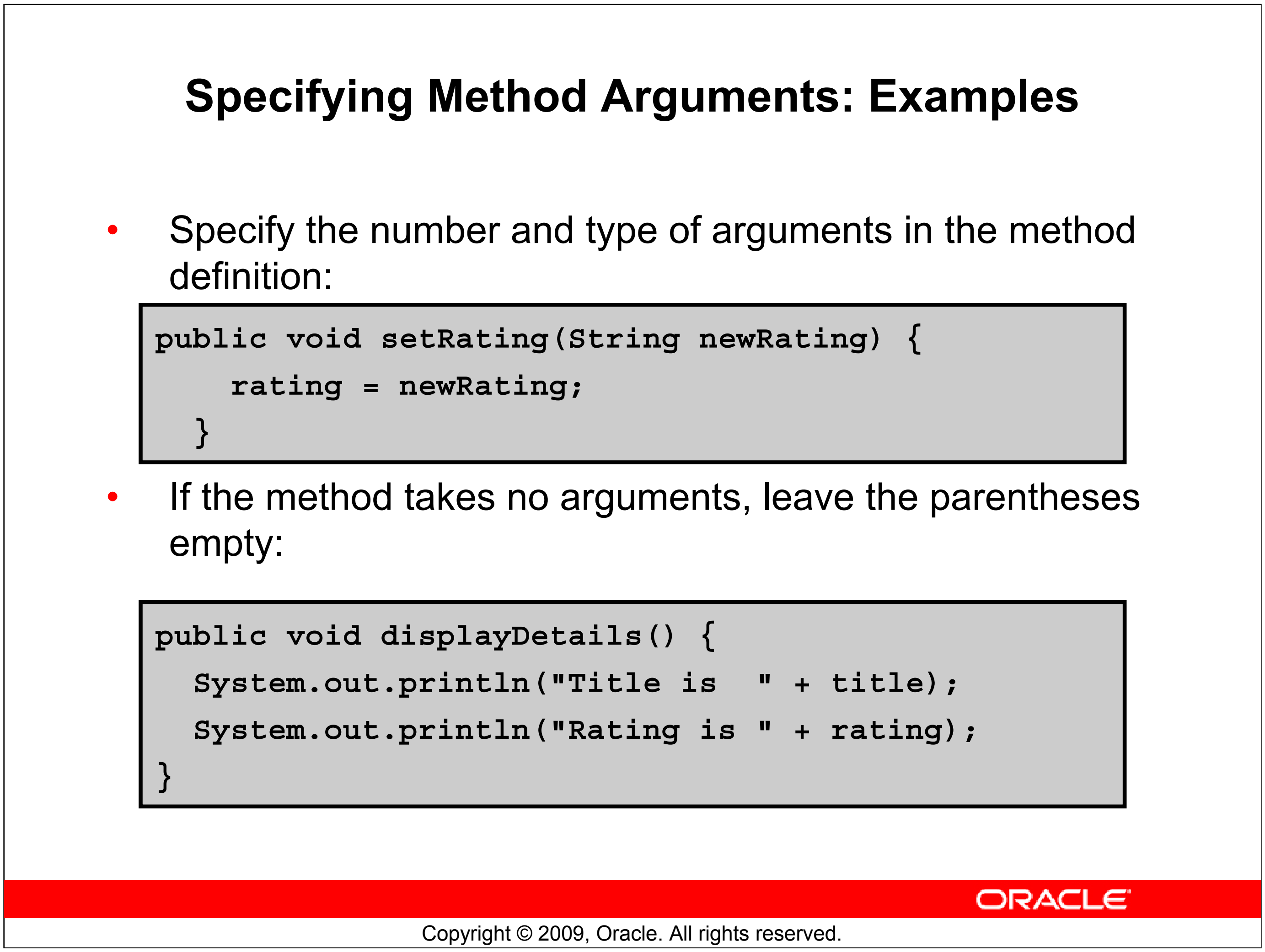
Objects communicate by using messages:

- All methods are defined within a class and are not defined globally as in traditional languages.
- When you call a method, it is always in the context of a particular object.
  - `myPen.write( )`: Object-oriented programming
  - `Write (myPen)`: Traditional structured programming

Objects Must Associate with Each Other

Objects must associate with other objects to enable messages to be sent. To send a particular message, an object must know the recipient of the message. This visibility may be achieved dynamically and transiently, or an object can “remember” another object through a *reference*. (Two classes may associate with each other, and two objects may have a link between them.)

A link is simply a relationship between objects. In programming terms, it may be implemented in various ways—for example, when a variable in one object contains a reference to the other object.



Specifying Method Arguments: Examples

- Specify the number and type of arguments in the method definition:

```
public void setRating(String newRating) {
    rating = newRating;
  }
```

- If the method takes no arguments, leave the parentheses empty:

```
public void displayDetails() {
  System.out.println("Title is  " + title);
  System.out.println("Rating is " + rating);
}
```

Specifying Method Arguments: Examples

A method can have zero or more arguments, also known as parameters; this definition is called the *method signature*. When you define a method in a class definition, you specify its formal arguments in an argument list. Each formal argument is a placeholder for a variable of some type, either a primitive or an object reference. Like any other variable, each argument must be declared in terms of its type, such as `String`, and a name, such as `title`.

Specifying Multiple Arguments

If a method has multiple arguments, each declaration must be separated by commas, as in the following example:

```
public void setMovieDetails(String pTitle, String pRating) {
  title = pTitle;
  rating = pRating;
  …
}
```

Specifying No Arguments

If a method has no arguments, simply leave the parentheses empty in the definition.

Specifying Method Arguments: Examples (continued)

varargs

varags (variable argument lists) were a language enhancement introduced in JDK 5.0.

Previously, a method that took an arbitrary number of values required you to create an array and put the values into the array before invoking the method. The following example uses the `MessageFormat` class to format a message:

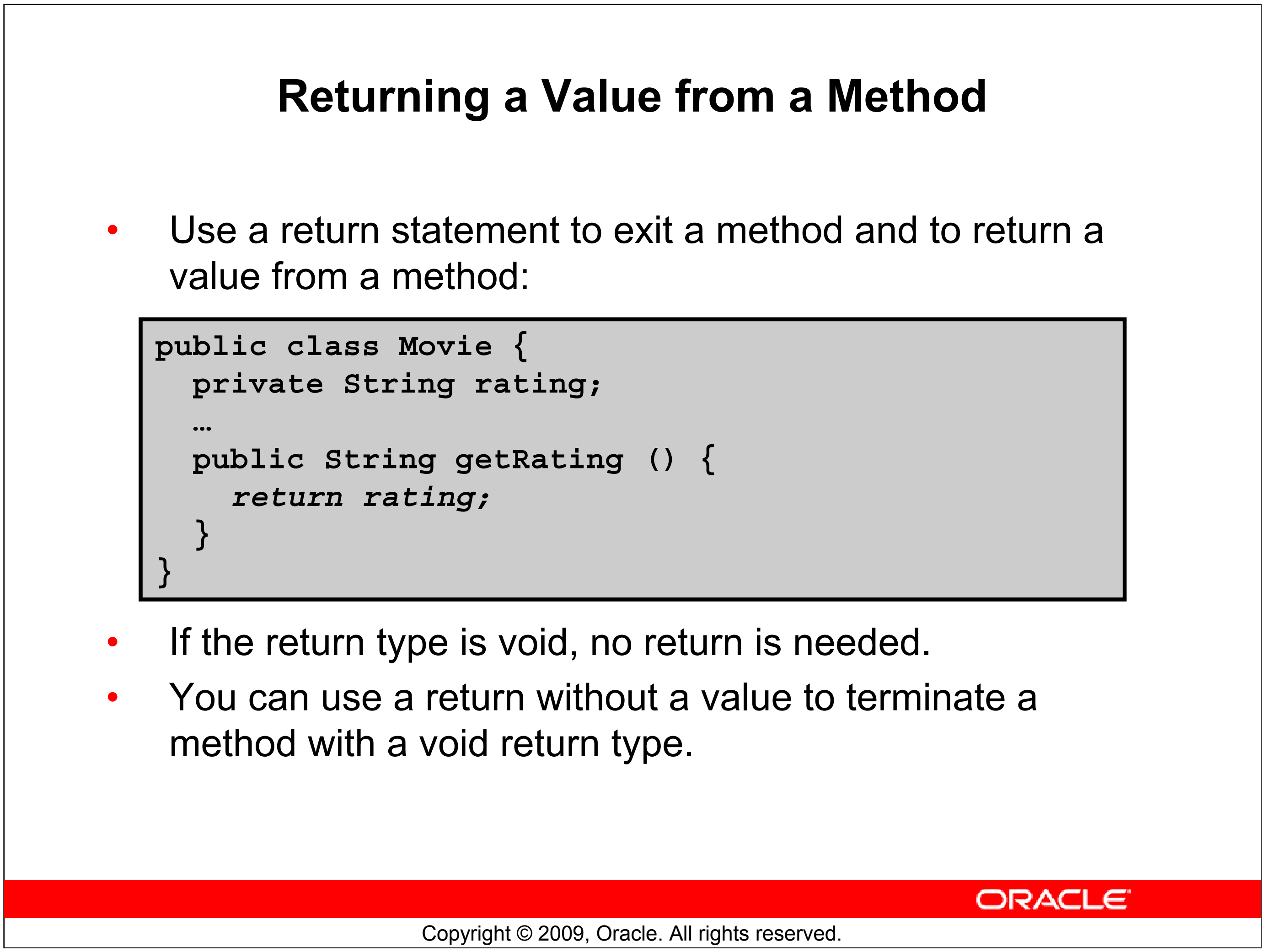
```
Object[] arguments = {
     new Integer(7),
     new Date(),
     "a disturbance in the Force‟
      };


String result = MessageFormat.format(
     "At {1,time} on {1,date}, there was {2} on planet "
     + "{0,number,integer}.", arguments);
```

It is still true that multiple arguments must be passed in an array, but the `varargs` feature automates and hides the process. The `MessageFormat.format` method now has this declaration:

```
public static String format(String pattern,
                             Object… arguments);
```

The three periods after the final parameter's type indicate that the final argument may be passed as an array *or* as a sequence of arguments. `varargs` can be used *only* in the final argument position. Given the new `varargs` declaration for `MessageFormat.format`, the preceding invocation can be replaced by the following shorter invocation:

```
String result = MessageFormat.format(
     "At {1,time} on {1,date}, there was {2} on planet ‟
     + "{0,number,integer}.",
     7, new Date(), "a disturbance in the Force");
```



Returning a Value from a Method

- Use a return statement to exit a method and to return a value from a method:

```
public class Movie {
  private String rating;
  …
  public String getRating () {
    return rating;
  }
}
```

- If the return type is void, no return is needed.
- You can use a return without a value to terminate a method with a void return type.

Returning a Value from a Method

A method can return a single value or expression to a caller of that method. If so, it must be defined with a return type such as `int` in front of the method name, as follows:

```
public int getLength() {
   …
}
```

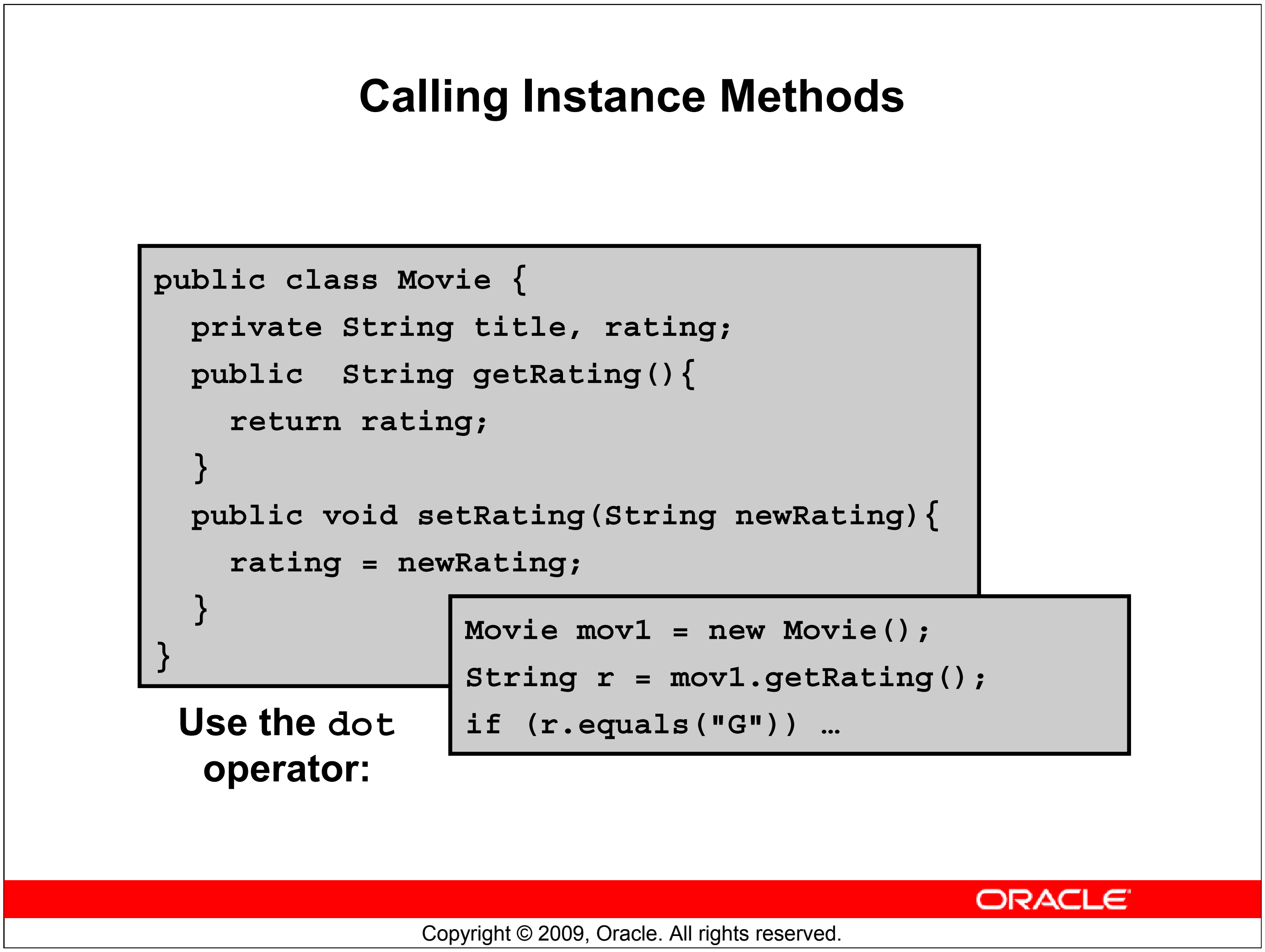
Methods are also allowed to return a reference to an object; this can be useful if you want to return more than one value. Here is a method that returns a reference to a `String` object:

```
public String getRating() {
   …
}
```

`return` Statement

A `return` statement is the mechanism by which methods pass the required value back to the calling method. A `return` statement takes a single value or expression, which must be compatible with the return type that is specified at the beginning of the method. When a `return` statement is encountered, the method exits immediately and ignores any statements that lie between the `return` statement and the closing brace of the method. This can be used for `void` methods that have no return value or it can be used to exit the function immediately:

```
return;    // Use this syntax to return void from a method
```



Calling Instance Methods

```
public class Movie {
  private String title, rating;
  public  String getRating(){
    return rating;
  }
  public void setRating(String newRating){
    rating = newRating;
  }
}
```

Use the dot operator:

```
Movie mov1 = new Movie();
String r = mov1.getRating();
if (r.equals("G")) …
```

Calling Instance Methods

As with instance variables, you can use the dot operator to call an instance method on an object. The general syntax is as follows:

```
objectRef.methodName(arguments … );
```

If the method takes no arguments, you still must include the parentheses when you call the method, but you leave them empty:

```
objectRef.methodName();   // Call method that takes no args
```

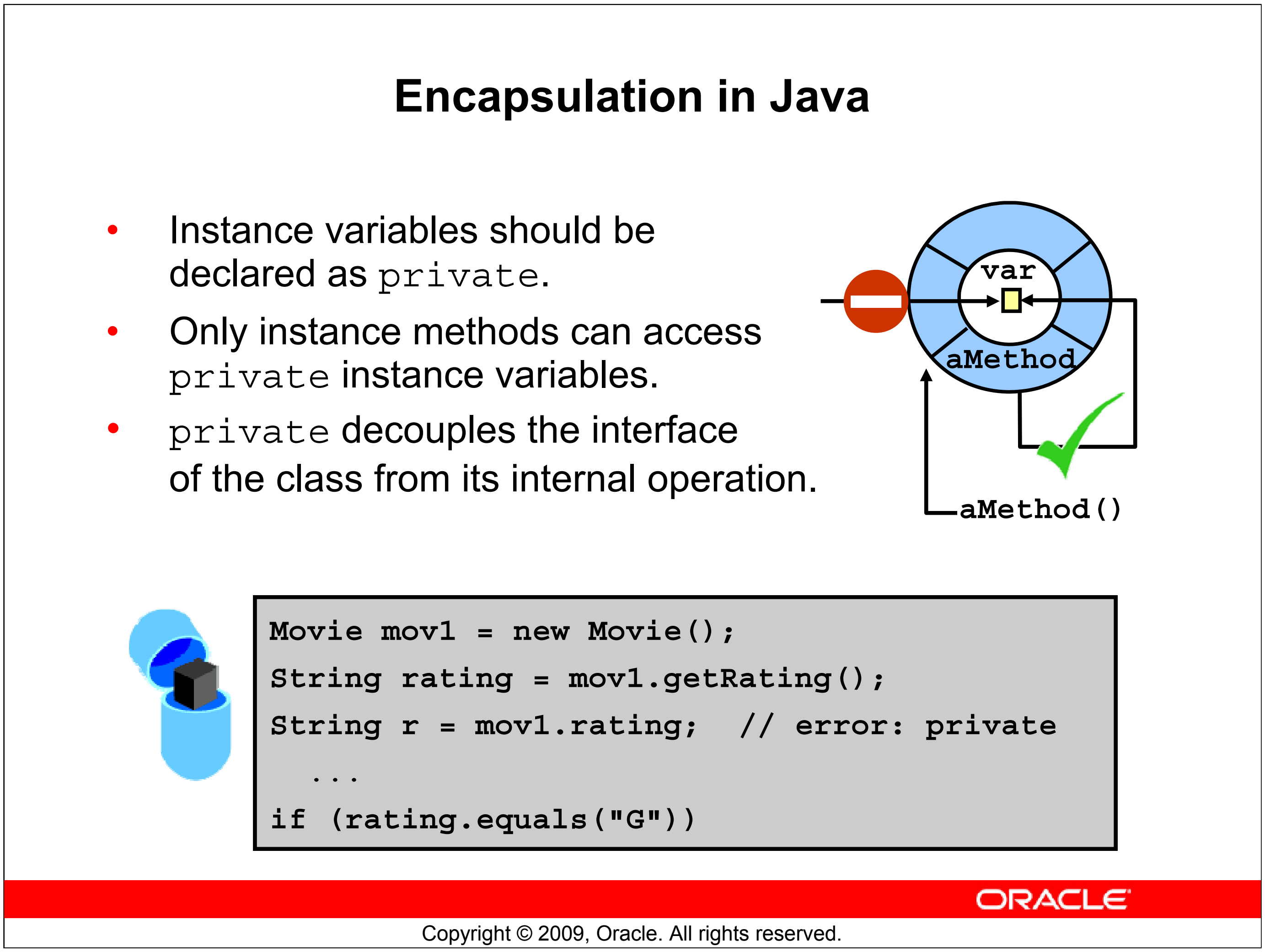
Example

In the example, two instance methods in the Movie class are defined: getRating() and setRating(). You then create a Movie object and check its rating by using getRating():

```
if ( mov1.getRating.equals("G") ) …
```

If necessary, the movie rating can be reset by using setRating():

```
mov1.setRating("PG");
```



Encapsulation in Java

- Instance variables should be declared as `private`.
- Only instance methods can access `private` instance variables.
- `private` decouples the interface of the class from its internal operation.

var
aMethod
aMethod()

```
Movie mov1 = new Movie();
String rating = mov1.getRating();
String r = mov1.rating;  // error: private
  ...
if (rating.equals("G"))
```

Encapsulation in Java

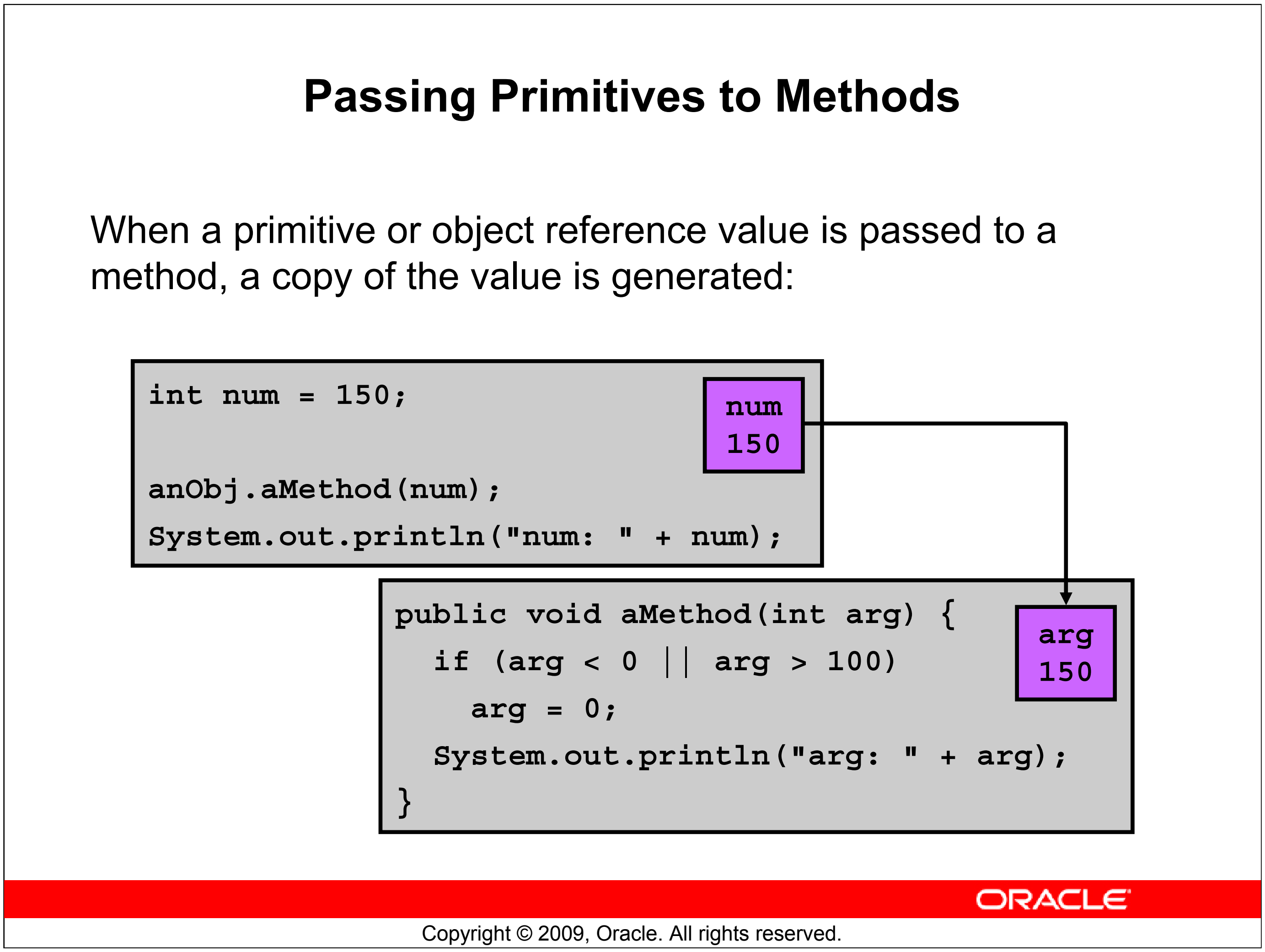
As you saw earlier in this lesson, encapsulation is a key concept in object-oriented programming. A well-defined class must decouple its public interface from its internal implementation. To achieve this, all instance variables of a class must be made `private` to hide them from users of the class. Only the instance methods of a class can access the `private` instance variables of that class.

Users of the class must invoke one of the `public` methods of the class to access (`get`) or change (`set`) the state of an object. For example, if the `Movie` class is properly encapsulated, it is impossible for a user of the `Movie` class to directly access instance variables such as `title` and `rating`.

Benefits of Encapsulation

As a class evolves, it is likely that you will need to modify the way in which the class is implemented internally. However, as long as you preserve the same interface to the class, the rest of the program will not need to be modified at all. You preserve the interface by retaining the same `public` instance methods with exactly the same signatures or parameter lists.

By maintaining this encapsulation, you will not break existing code that uses the `Movie` class.



Passing Primitives to Methods

When a primitive or object reference value is passed to a method, a copy of the value is generated:

```
int num = 150;

anObj.aMethod(num);
System.out.println("num: " + num);
```

```
public void aMethod(int arg) {
  if (arg < 0 || arg > 100)
    arg = 0;
  System.out.println("arg: " + arg);
}
```

Passing Primitives to a Method

When a primitive value is passed to a method, a copy of its value is passed to the method argument. If the method changes the value of the argument in any way, only the local argument is affected. When the method terminates, this local argument is discarded and the original variable in the calling method is left unchanged.

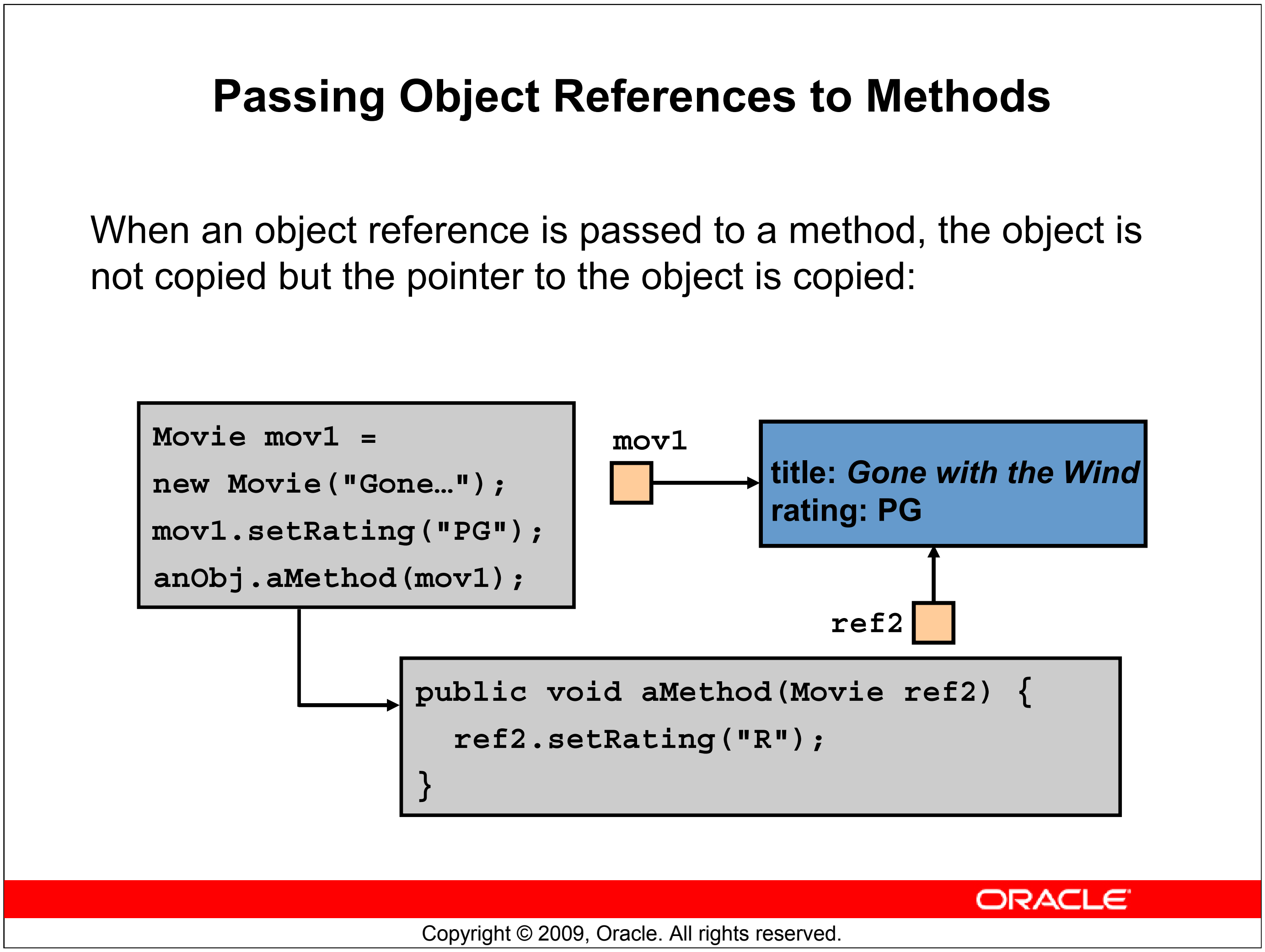
Example

The example in the slide illustrates the way in which primitive variables are “passed by value” to methods. The code on the left declares an `int` variable called `num` and assigns it the value `150`. When `num` is passed to `aMethod()`, a copy of its current value is passed to the method argument, which you have called `arg`. Initially, `arg` is `150`.

Inside `aMethod()`, `arg` is reset to `0`. However, when `aMethod()` terminates, `arg` is discarded and you return to the calling method, where `num` still has the value of `150`.

As a result, this example prints the following messages:

```
arg: 0
num: 150
```



Passing Object References to Methods

When an object reference is passed to a method, the object is not copied but the pointer to the object is copied:

```
Movie mov1 =
new Movie("Gone…");
mov1.setRating("PG");
anObj.aMethod(mov1);
```

```
public void aMethod(Movie ref2) {
  ref2.setRating("R");
}
```

Passing Object References to Methods

When an object reference is passed to a method, a copy of the passed reference is generated, which refers to the original object. The reference contains the address where the object is located in memory. Any changes that the method makes to the argument will change the original object. When the method terminates, any changes that you have made to the object during the method remain in force.

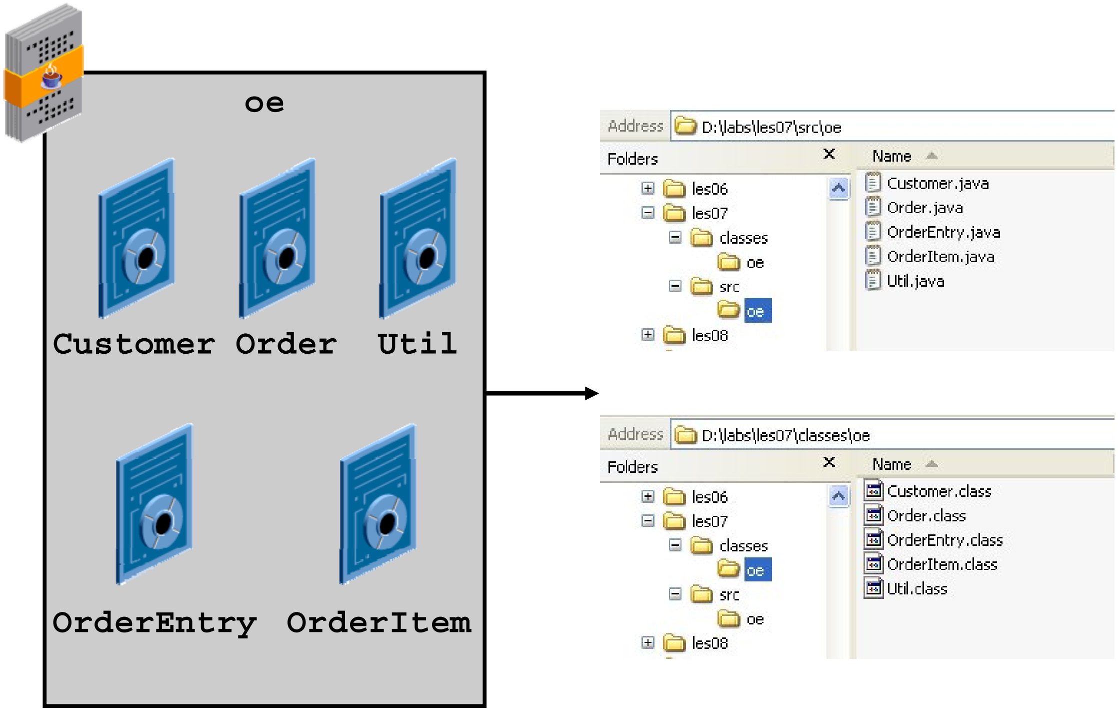
Example

The example in the slide illustrates the way in which objects are “passed by reference” to methods. The code on the left creates a new `Movie` object and stores a reference to it in `mov1`. The movie rating is then set to `"PG."`

When `mov1` is passed to `aMethod()`, the method receives a reference to the original `Movie` object. Inside `aMethod()`, the movie rating is changed to `"R."`

When `aMethod()` terminates, the original `Movie` object that is referred to by `mov1` has a rating of `"R"` rather than `"PG."`

Java Packages



Java Packages

A package is a container of classes that are logically related, either by application or function. A package consists of all the Java classes within a directory on the file system. Package names are used within a Java run-time environment to manage the uniqueness of identifiers as well as to control access from other classes. They also help by segmenting related parts of complex applications into manageable parts.

Namespace

The JVM uses a construct called *namespace* to manage identifier names in a Java program. A namespace is a chunk of memory that is allocated specifically to manage objects. Objects are placed in specific namespaces depending on the source of the code. For example, a class that is loaded from a local package is loaded into one namespace, whereas a class loaded from a network source goes into another separate namespace.

Identifier names must be unique in a namespace. Without the internal namespace construct, identifier names need to be unique across all Java classes. In other words, if the J2SE or any other class that you need uses an identifier named `count`, your program cannot define a variable named `count`.

Java uses namespaces to manage the identifier names so that you do not have to worry about names that are used by other classes. You manage uniqueness only within your program.

Grouping Classes in a Package

- Include the `package` keyword followed by the package name at the top of the Java source file. Use the dot notation to show the package path.
- If you omit the `package` keyword, the compiler places the class in a default “unnamed” package.
- Use the `-d` flag with the `javac` compiler to create the package tree structure relative to the specified directory.
- Running a `main()` method in a packaged class requires that:
  - The `CLASSPATH` contain the directory having the root name of the package tree
  - The class name be qualified by its package name

Grouping Classes in a Package

The package represents the organization of the Java bytecode of classes and interfaces. It is not the source-code organization, which is represented by the `.java` files. The Java compiler reads and uses any needed packaged classes that you specify.

Note: The `CLASSPATH` environment variable is critical when using packages. A missing directory in the `CLASSPATH` environment causes most run-time problems with code that uses packages.

When running the Java application, you must include the package name in the command:

```
c:\>java <package_name>.<class>
c:\>java practice06.MaintainCustomers
```

If a class is included in a package, the compiler can be requested to put the class file in a subdirectory reflecting the package name. To create the package directory names during compile time, use the `–d` option. For example, suppose that you compile a class called `RentItem` in a package called `rentals` as follows:

```
javac -d c:\acmevideo RentItem.java
```

Then the class file that is created is called:

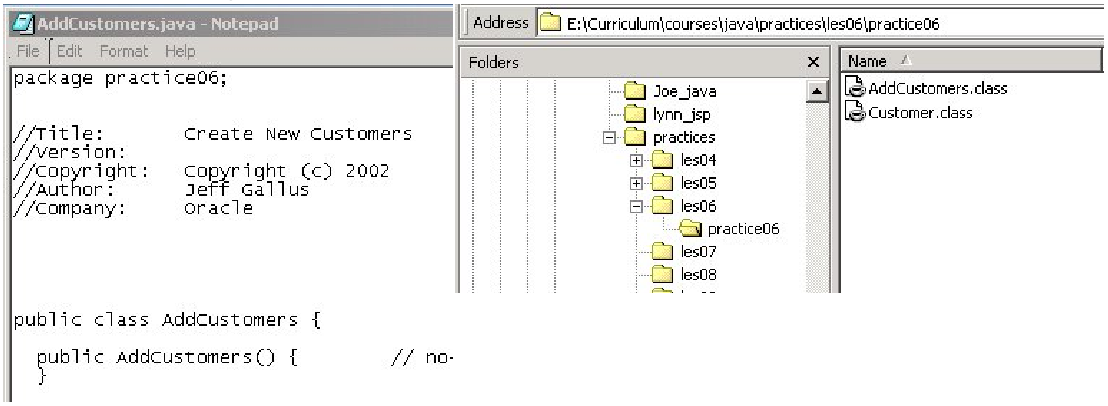
```
c:\acmevideo\rentals\RentItem.class
```

The default behavior for the `javac` command without the `-d` option is to put the class file in the same directory as the source file.

Setting the `CLASSPATH` with Packages

The `CLASSPATH` includes the directory containing the top level of the package tree:

Package name **`.class` location**



AddCustomers.java - Notepad

```
package practice06;


//Title:       Create New Customers
//Version:
//Copyright:   Copyright (c) 2002
//Author:      Jeff Gallus
//Company:     Oracle




public class AddCustomers {

  public AddCustomers() {        // no-
  }
```

Address E:\Curriculum\courses\java\practices\les06\practice06

CLASSPATH

```
C:\>set CLASSPATH=E:\Curriculum\courses\java\practices\les06
```

Setting the `CLASSPATH` with Packages

The `CLASSPATH` must point to the directory above the classes. For example, suppose that you want the Java interpreter to be able to find classes in the `practice06` package. Assume that the path to the `practice06` classes directory is the following:

```
E:\Curriculum\courses\java\practices\les06\practice06
```

Then you would set the `CLASSPATH` variable from an operating system prompt as follows:

```
set CLASSPATH=E:\Curriculum\courses\java\practices\les06
```

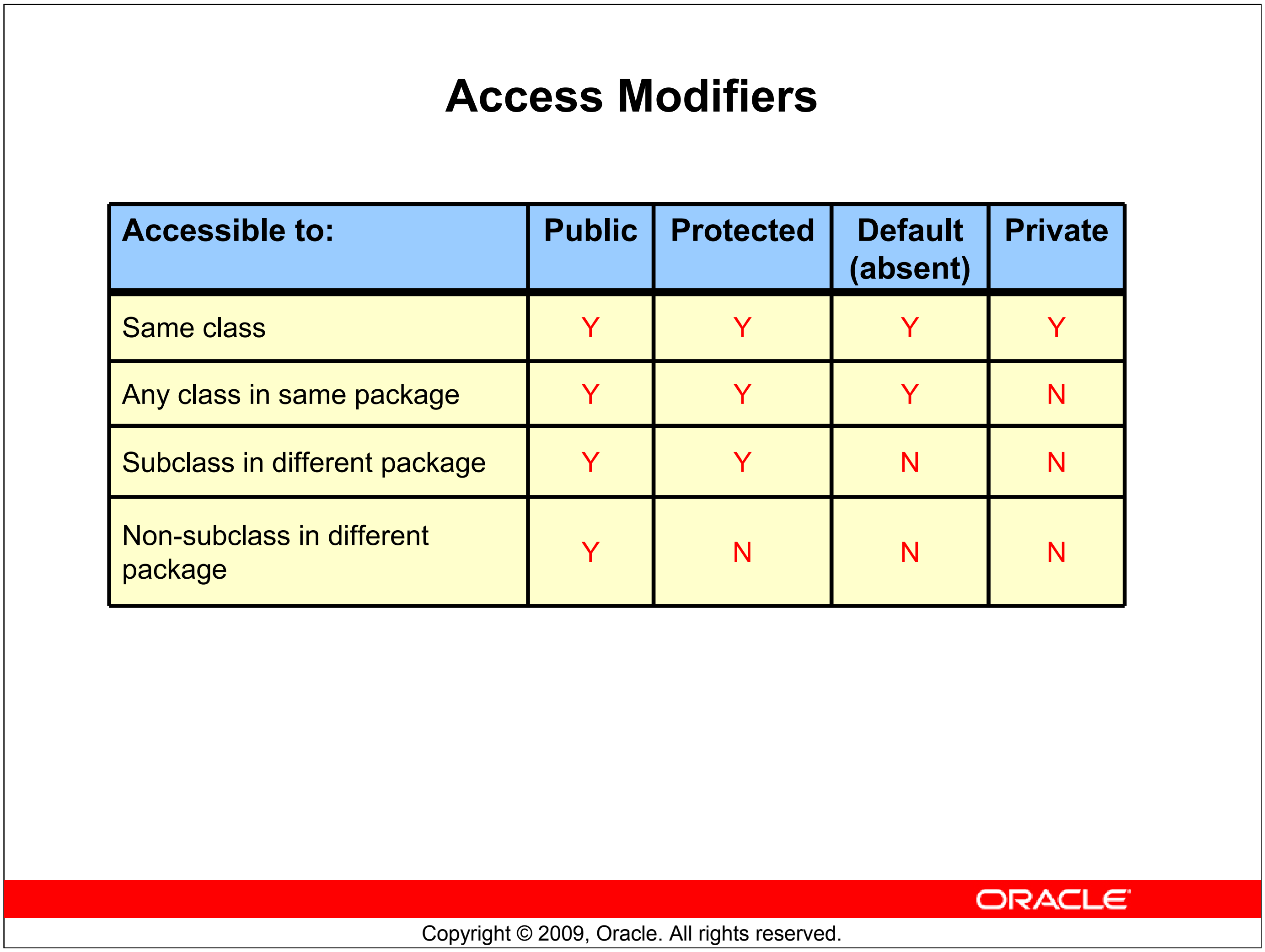
After you exit a DOS prompt, the `CLASSPATH` reverts to the permanent settings. The `CLASSPATH` is used by *both* the compiler and the class loader in the interpreter (JVM) to locate classes (in order to resolve references to class names) and to load the classes into memory at run time. The `CLASSPATH` can include:

- A list of directory names (separated by semicolons in Windows and colons in UNIX)
  - The classes are in a package tree relative to any of the directories in the list.
- A `.zip` or `.jar` file name that is fully qualified with its path name
  - The classes in these files must be zipped with the path names that are derived from the directories formed by their package names.

Setting the CLASSPATH with Packages (continued)

Note: The directory containing the root name of a package tree must be added to the CLASSPATH. Consider putting the CLASSPATH information in the command window or even in the Java command, rather than hard-coding it in the environment. Here is an example:

```
java -classpath E:\Curriculum\courses\java\les06
   practice06.AddCustomers
```



Access Modifiers

Accessible to:	Public	Protected	Default (absent)	Private
Same class	Y	Y	Y	Y
Any class in same package	Y	Y	Y	N
Subclass in different package	Y	Y	N	N
Non-subclass in different package	Y	N	N	N

Access Modifiers

Java controls access to variables and methods through the use of access modifiers. The access modifiers are `private`, `public`, `protected`, and `default (no modifier)`.

The least restrictive access modifier is `public`. Variables and methods that are declared as `public` can be seen and used by any other class.

If an access modifier is not specified (called *package access* or *default access*), the variables and methods can be used by any other class within the same package.

The next level of access modifier is `protected`. Variables and methods that are declared as `protected` can be seen from any subclass of that class. They can also be seen from any class within the package in which they exist.

The most restrictive access modifier is `private`. A `private` method or variable cannot be accessed by any other class. The restriction applies to all other classes and subclasses regardless of their package.

The final access modifier is `default (no modifier)`. In this modifier, only other members of the same package can access variables and methods.

Access Modifiers (continued)

Variable or Method Visibility

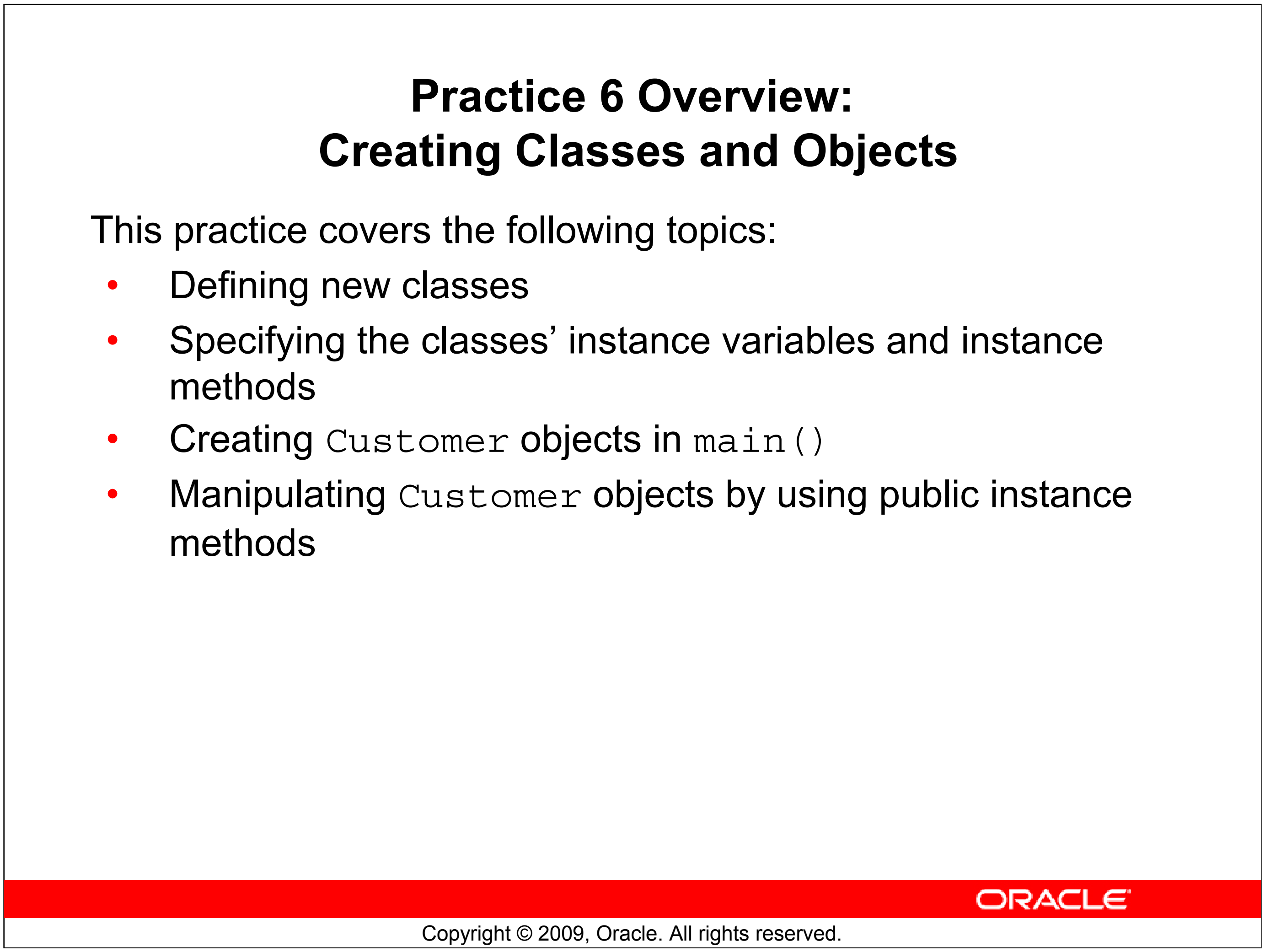
Access in order from least restrictive to most restrictive:

- **Public:** All
- **Protected:** Only other members of the same package or from a different package if inherited (using extends keyword)
- **Default (no modifier):** Only other members of the same package
- **Private:** Only other members of the same class

Examples:

- A `protected double getWage()` method is visible in all the classes that are in the same package as the class in which this method is defined and all the subclasses of that class.
- The `int getCount()` method is visible in all the classes that are in the same package as the class in which this method is defined.

Other modifiers that are used to control access to variables and methods are `final`, `static`, and `abstract`. These keywords are discussed in later lessons.



Practice 6 Overview:
Creating Classes and Objects

This practice covers the following topics:

- Defining new classes
- Specifying the classes' instance variables and instance methods
- Creating `Customer` objects in `main()`
- Manipulating `Customer` objects by using public instance methods

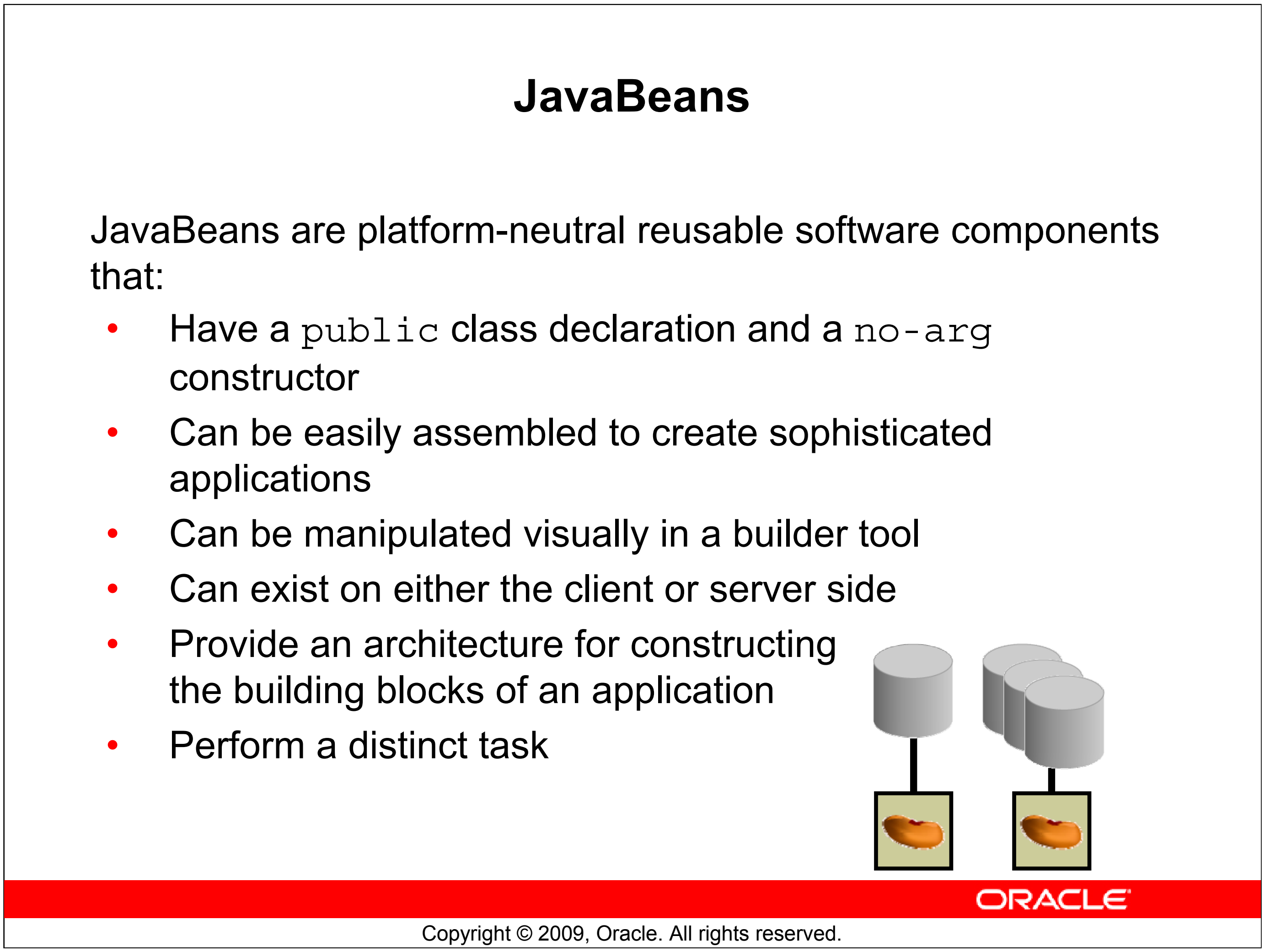
Practice 6 Overview: Creating Classes and Objects

Goal

The goal of this practice is to complete the basic functionality for existing method bodies of the `Customer` classes. You then create customer objects and manipulate them by using their public instance methods. You display the `Customer` information back to the JDeveloper message window.

Note: If you have successfully completed the previous practice, continue using the same directory and files. If the compilation from the previous practice was unsuccessful and you want to move on to this practice, change to the `les06` directory, load the `OrderEntryApplicationLes06` application, and continue with this practice.

Viewing the model: Remember that you can view the course application model up to this practice in the `UML Class Diagram1` file. To locate this file, load the `OrderEntryApplicationLes06` application, if you have not already done so. In the Applications Navigator node, select `OrderEntryApplicationLes06`, and then select File > Open. In the Open dialog box, double-click `model`, and then double-click `oe`. Select `UML Class Diagram1.oxd_ana` and click Open. This diagram displays all the classes created up to this point in the course.



JavaBeans

JavaBeans are platform-neutral reusable software components that:

- Have a `public` class declaration and a `no-arg` constructor
- Can be easily assembled to create sophisticated applications
- Can be manipulated visually in a builder tool
- Can exist on either the client or server side
- Provide an architecture for constructing the building blocks of an application
- Perform a distinct task

JavaBeans

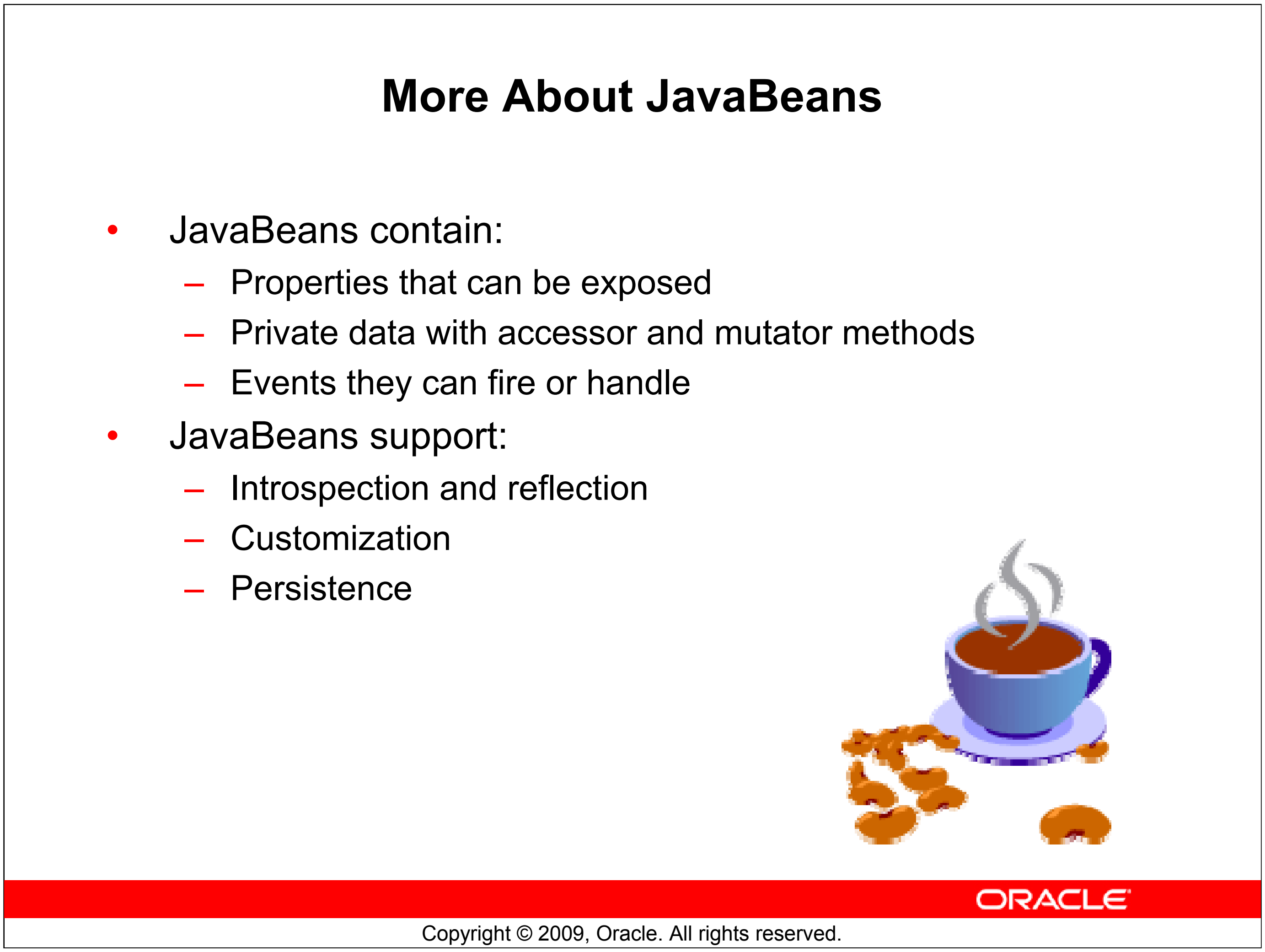
A JavaBean is a reusable platform-independent software component that can be manipulated visually in a builder tool like JDeveloper.

A JavaBean consists of a single class or a group of classes. At a minimum, a bean must have a `public` class declaration and a `no-parameter` constructor. Any classes can be part of a bean. There is no specific superclass that a bean must extend—unlike, for example, an applet, which must extend the `Applet` class. However, a JavaBean must conform to certain basic architecture rules.

Beans are not a special type of component. A bit of Java code does not have a special “bean” type or label that makes it a bean. A JavaBean has a set of criteria that make it flexible and reusable to others. It is not about, for example, different types of coffee, such as French Roast or Columbian. Rather, it is about the preparation and packaging of the beans themselves so that they can be used in a variety of environments (for example, used in a grinder for a beverage or covered in chocolate as an after-dinner treat).

Examples of JavaBeans

Common examples of JavaBeans include Calculator and Calendar beans.



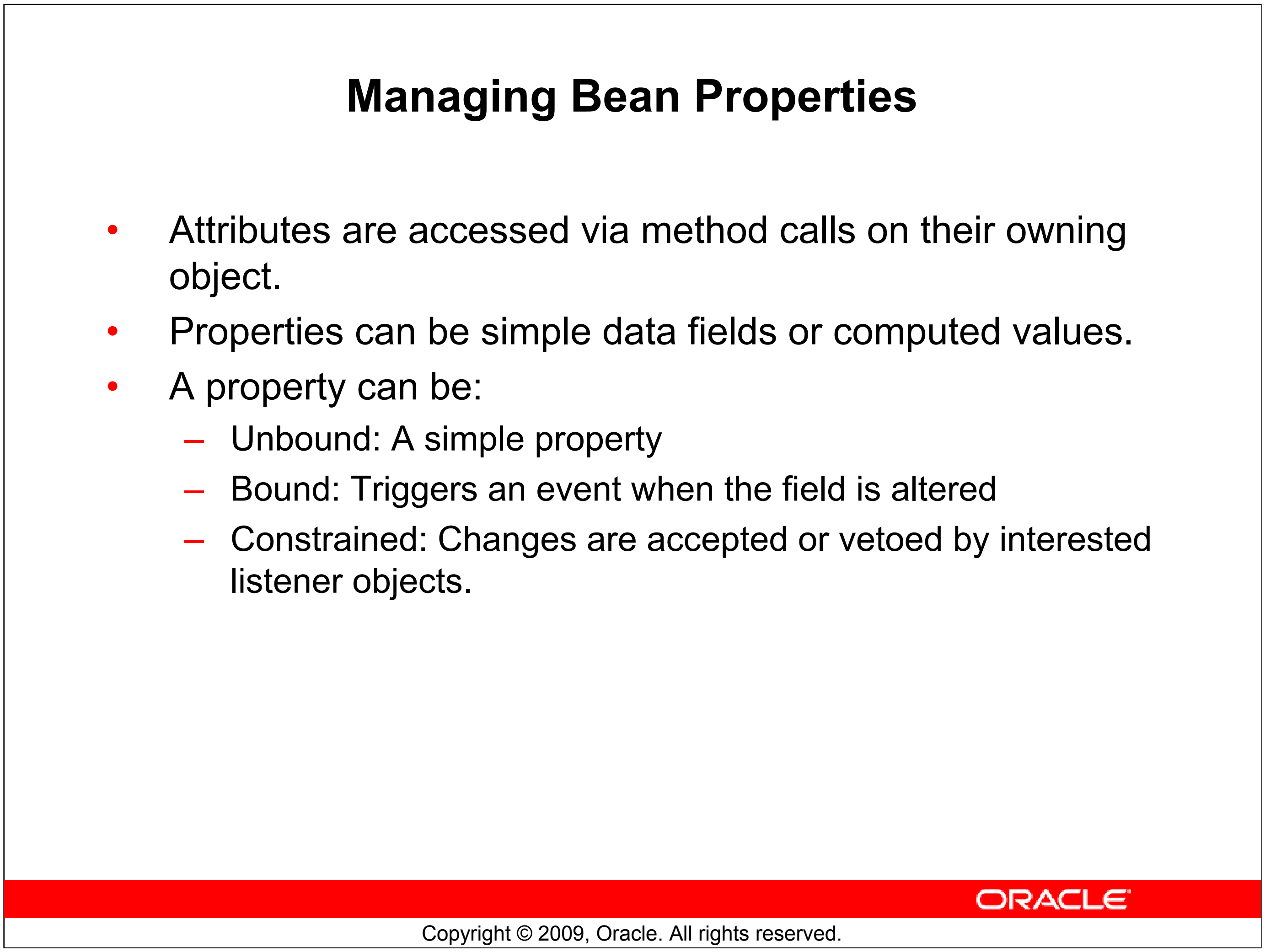
More About JavaBeans

- JavaBeans contain:
  - Properties that can be exposed
  - Private data with accessor and mutator methods
  - Events they can fire or handle
- JavaBeans support:
  - Introspection and reflection
  - Customization
  - Persistence

Using a Standard Protocol

A JavaBean is simply a Java class that obeys a strict protocol and usually consists of a group of support classes and resource files, which are packaged into an archive file. Consider a JavaBean as a black box, where all you need to know is the functionality of the box, and not the functionality of its contents. Individual JavaBeans will vary in the functionality that they support, but the typical features of the unifying architecture that distinguish a JavaBean are support for:

- Properties, both for customization and for programmatic use
- Accessor and mutator methods that perform all the necessary business logic to arrive at an appropriate result
- Introspection and reflection, so that at design time, a builder tool can analyze how a bean works. Introspection and reflection enable a Java class to dynamically determine the basic structure of the bean through its public interface definitions. This depends on the bean having been written according to a structured coding pattern.
- Customization, so that when using an application builder, a user can customize the appearance and behavior of a bean
- Persistence, so that a bean can be customized in an application builder and then have the state of its customized objects saved and reloaded later using serialization



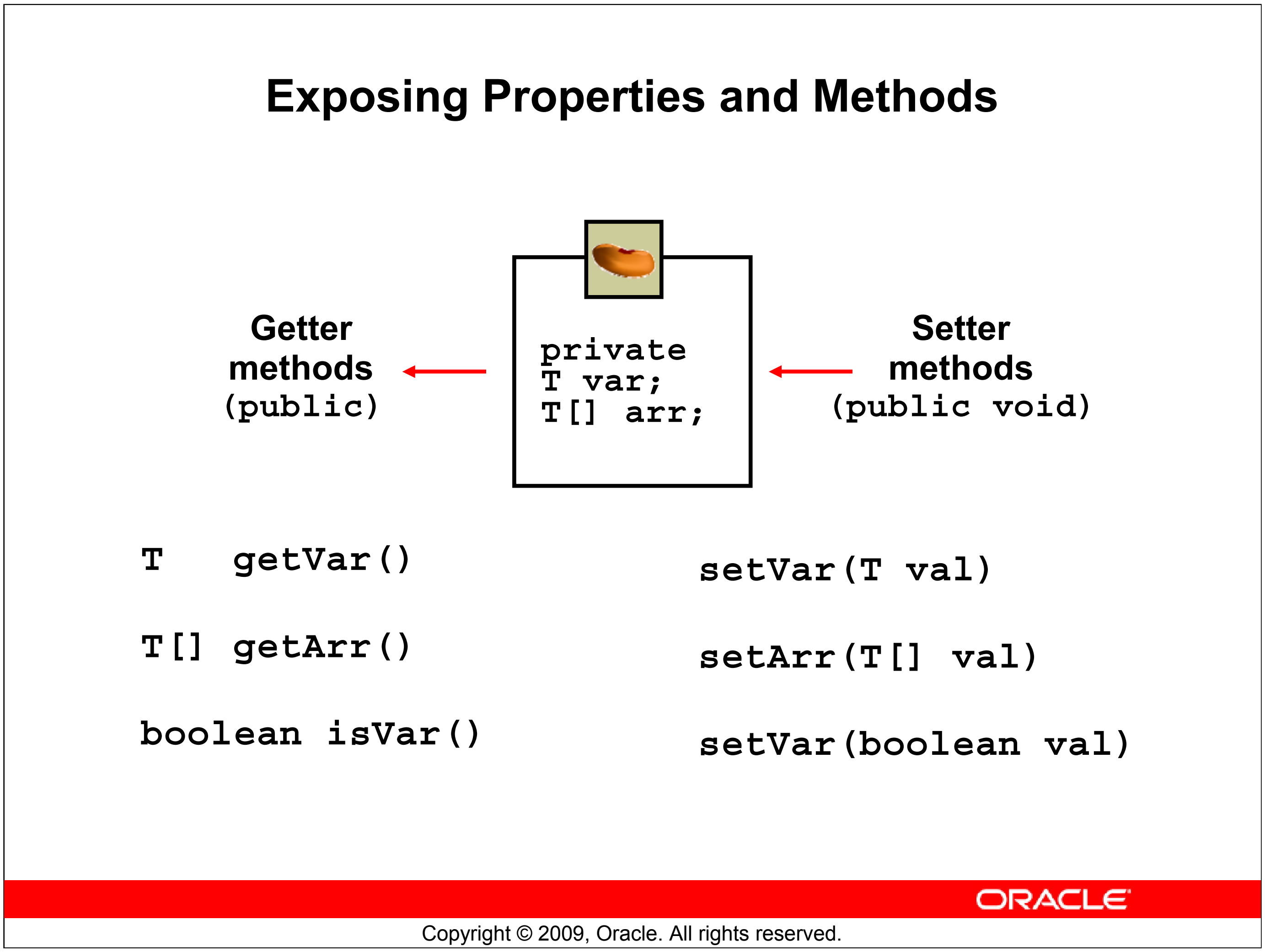
Managing Bean Properties

- Attributes are accessed via method calls on their owning object.
- Properties can be simple data fields or computed values.
- A property can be:
  - Unbound: A simple property
  - Bound: Triggers an event when the field is altered
  - Constrained: Changes are accepted or vetoed by interested listener objects.

Properties

Properties are the bean variables. They can be of any Java data type—that is, primitives or objects. In the context of a bean, variables or properties can have a binding that is stated as any of the following:

- **Unbound:** Unbound properties are simple properties that are accessed or modified by using the associated `get` and `set` methods.
- **Bound:** Bound properties are like simple properties, but when modified by using the set method they trigger the `PropertyChangeEvent` event. The `PropertyChangeEvent` object contains the old value and new value of the property, so that listeners can react appropriately to the change. For example, changing the connection property of a JClient `SessionInfo` object fires an event that redraws JDeveloper's structure pane.
- **Constrained:** Constrained properties are an extension to bound properties, in that they generate a `PropertyChangeEvent` when the property is modified, but the listeners handling the event can veto (prevent) the change by throwing `PropertyVetoException` if the change violates some rule or constraint. For example, resizing a visual component can be vetoed by other components based on layout management rules. The distinguishing characteristic between bound and constrained is what occurs when the property value is altered.



Exposing Properties and Methods

Getter methods `(public)`

```
private
T var;
T[] arr;
```

Setter methods `(public void)`

```
T    getVar()                    setVar(T val)

T[]  getArr()                    setArr(T[] val)

boolean isVar()                  setVar(boolean val)
```

Property and Method Naming Conventions

The naming conventions apply to the methods that an application calls to access and alter the properties of a bean. The name of the `set` and `get` methods are formed from the prefix `get` or `set`, followed by the capitalized name of the property or variable that is accessed.

The properties are commonly declared `private`, forcing users to invoke the access methods to interact with the bean. The following is an example of a `String` property and its corresponding setter and getter methods:

```
private String text;
public String getText() { return text };
public void setText(String newtext) { text = newtext; }
```

Note: The property name starts with a lowercase letter, but the method names are `setText` and `getText` (using the capitalized form of the property name). The return type of the getter method and the single argument of the setter method must be the same type as the property. For indexed properties, like arrays, it is common to provide two additional methods to get and set a value at a specified index. Here are some examples:

```
private String[] name;
public String[] getName() {...} ;
public void setName(String[] values) {...};
public String getName(int index) {...};
public void setName(int index, String value) {...};
```

Building and Using a JavaBean in JDeveloper

1. Develop a JavaBean.
   – Modified via code, class, or UI editors
2. Store the bean in an archive file.
3. Create a JDeveloper library identifying the archive.
4. Install the bean in the JDeveloper Component Palette via its library name.
5. Develop an application that uses the JavaBean component.

Building and Using a JavaBean in JDeveloper

Step 1: Develop a JavaBean.

Use JDeveloper to develop a bean. A bean is essentially a Java class, or a group of classes, that conforms to a particular design pattern or rules.

Step 2: Store the bean in an archive file.

To deploy the bean for reuse in another application, you must store all the classes in an archive file, known as a Java Archive (`.jar`) file. A deployment wizard is provided to create the `.jar` file.

Step 3: Create a JDeveloper library.

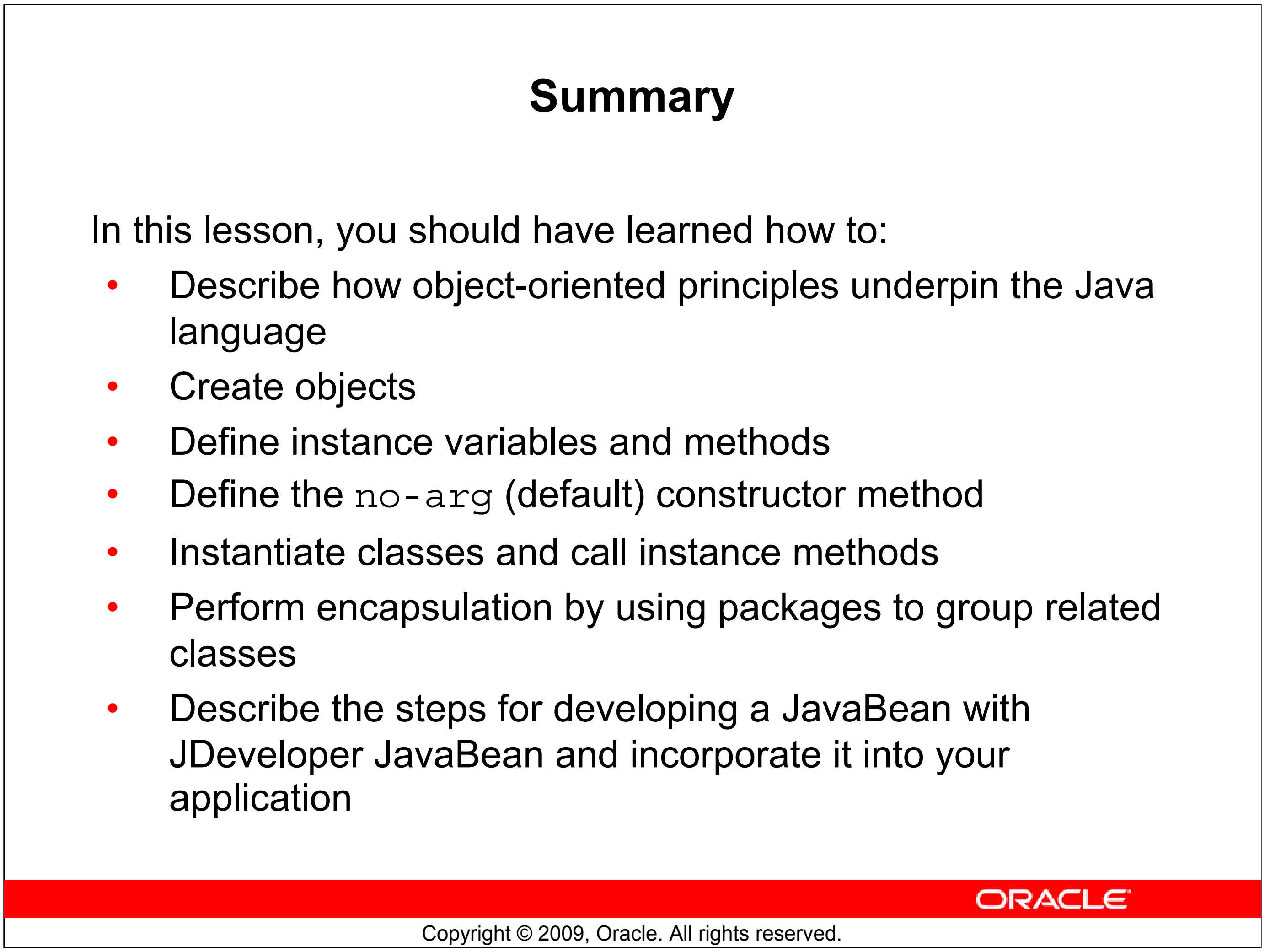
The Component Palette requires that a user-named JDeveloper library be created to identify the path to the JavaBean archive.

Step 4: Install the bean in a Component Palette in JDeveloper.

After the bean has been created and archived, you must install it in a Component Palette in JDeveloper. A series of menu options is used to achieve this. However, you must first create a named JDeveloper library to identify the path to the archive that was created in step 2.

Step 5: Develop an application.

After the bean is attached to a Component Palette, any application can use it.



Summary

In this lesson, you should have learned how to:

- Describe how object-oriented principles underpin the Java language
- Create objects
- Define instance variables and methods
- Define the `no-arg` (default) constructor method
- Instantiate classes and call instance methods
- Perform encapsulation by using packages to group related classes
- Describe the steps for developing a JavaBean with JDeveloper JavaBean and incorporate it into your application

Summary

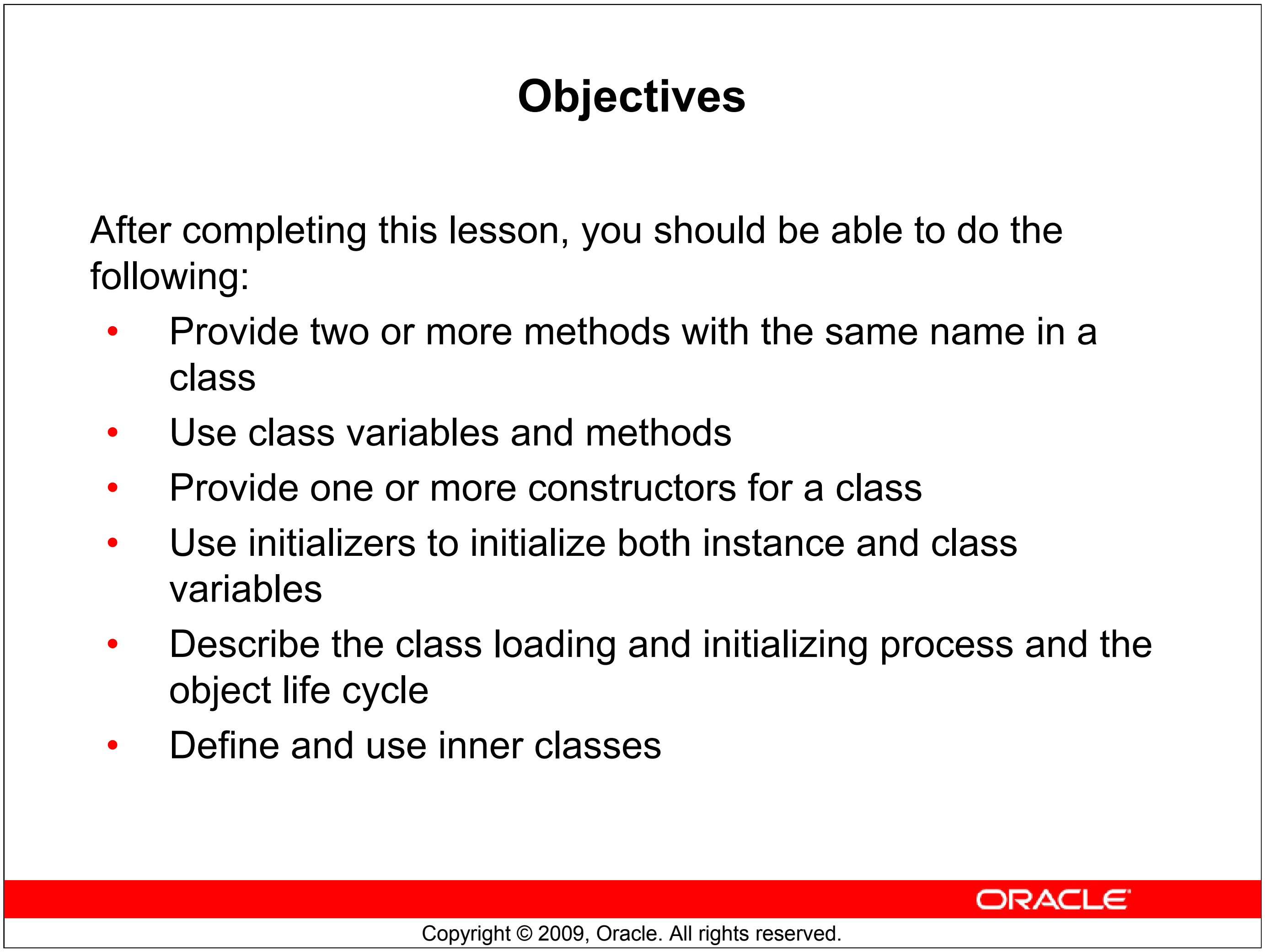
Object-oriented programming involves programming with objects. A Java program can be viewed as a collection of co-operating objects. Classes are static definitions from which object instances are created. You implement the behaviors of a class using methods. A Java method is equivalent to a function, procedure or subroutine in other languages.

A Java package is a container of classes that are logically related. The Java compiler reads and uses any needed packaged classes that you specify. The `CLASSPATH` environment variable is critical when using packages. The directory containing the root name of a package tree must be added to the `CLASSPATH`.

Access modifiers control access to variables and methods in Java.

7

Object Life Cycle and Inner Classes



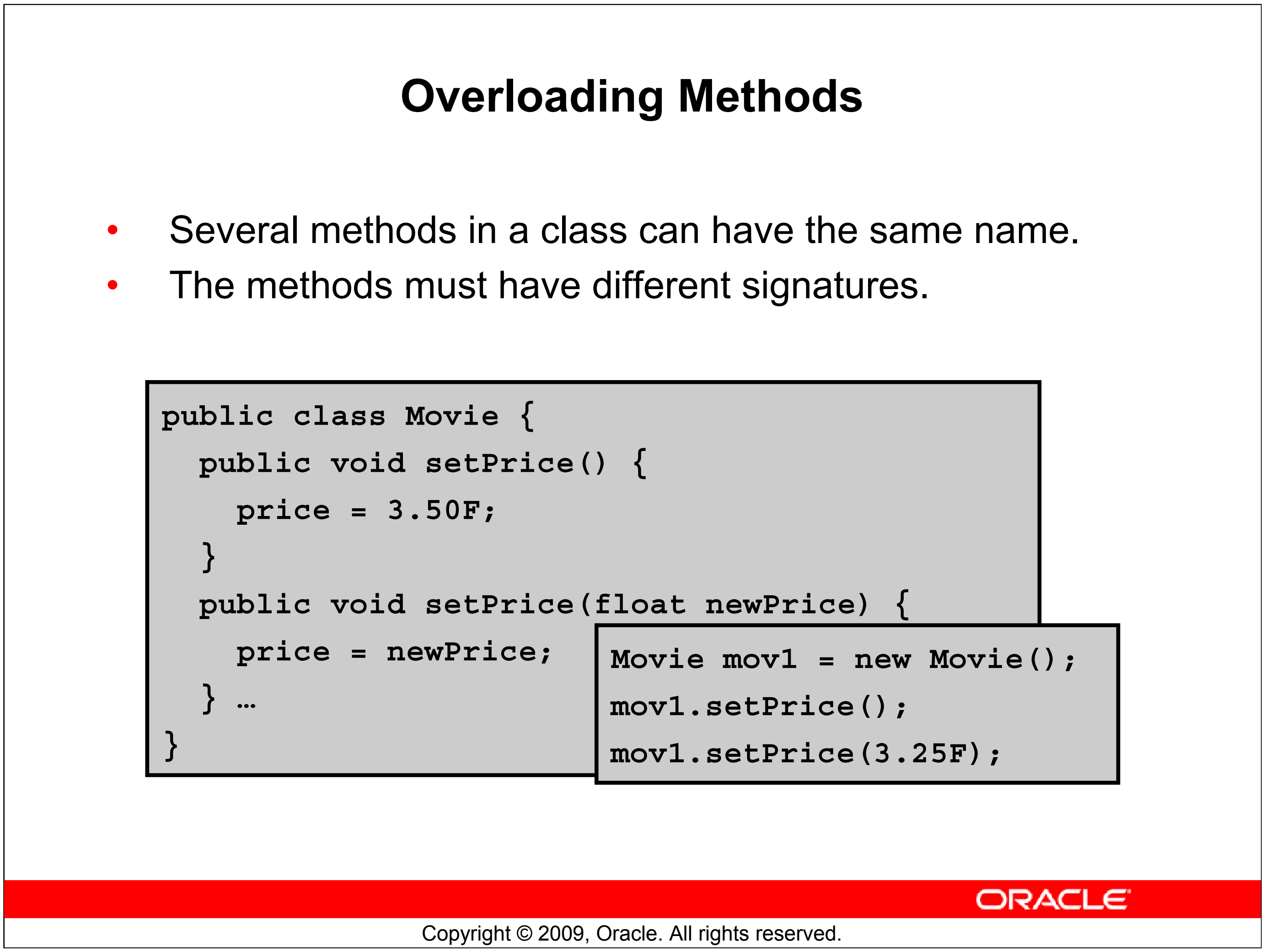
Objectives

After completing this lesson, you should be able to do the following:

- Provide two or more methods with the same name in a class
- Use class variables and methods
- Provide one or more constructors for a class
- Use initializers to initialize both instance and class variables
- Describe the class loading and initializing process and the object life cycle
- Define and use inner classes

Lesson Objectives

This lesson explores additional issues relating to class definitions. You learn how to provide and use overloaded methods in a class to provide a uniform appearance of the class to the outside world. You are also introduced to the use of constructors, enabling you to ensure that the instance variables of a class are properly initialized. The lesson discusses replacing and supplementing the default `no-arg` constructor with alternative constructors.



Overloading Methods

- Several methods in a class can have the same name.
- The methods must have different signatures.

```
public class Movie {
  public void setPrice() {
    price = 3.50F;
  }
  public void setPrice(float newPrice) {
    price = newPrice;
  } …
}
```

```
Movie mov1 = new Movie();
mov1.setPrice();
mov1.setPrice(3.25F);
```

Overloading Methods

Two or more methods in a class can have the same name, provided that they have different signatures. A method signature is formed from its name, together with the number and types of its arguments. The method return type is not considered part of the method signature.

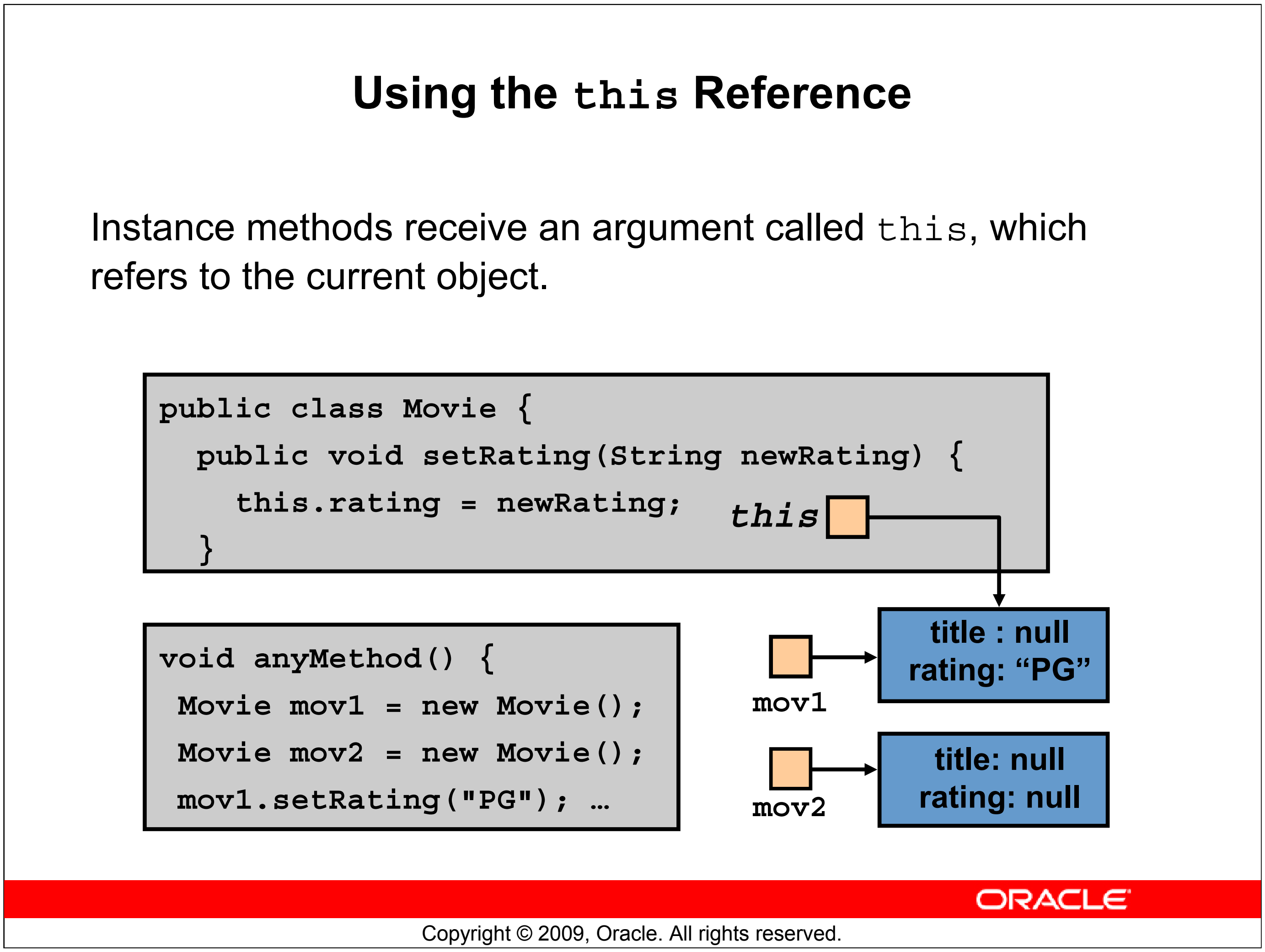
Defining two or more methods with the same name but different signatures is called *method overloading*. This technique is useful in presenting a unified interface to the outside world. The exact method that is called is determined by the parameters that are included in the call. Without method overloading, each method would require a unique name. For example, if you want to retrieve customer information by customer ID or by name, you need two methods: `getCustomerByID(id)` and `getCustomerByName(name)`. With overloading, you have two methods named `getCustomer()`, one with the ID parameter and the other with the name parameter.

How Does the Compiler Decide Which Overloaded Method to Call?

When a user of a class calls an overloaded method, the compiler chooses the correct method to call by comparing the argument types that are passed in the call with all the methods of that name in the class definition. If the compiler cannot find a compatible match, even after performing possible type conversions, it flags an error. Likewise, if more than one match is possible, the compiler flags an error because the method call is ambiguous.

Can Overloaded Methods Differ in Return Type Only?

No. Overloaded methods must be distinguishable by comparing their argument lists alone. If overloaded methods differ in their return type only, the compiler flags an error.



Using the `this` Reference

Instance methods receive an argument called `this`, which refers to the current object.

```
public class Movie {
  public void setRating(String newRating) {
    this.rating = newRating;
  }
```

```
void anyMethod() {
 Movie mov1 = new Movie();
 Movie mov2 = new Movie();
 mov1.setRating("PG"); …
```

Using the `this` Reference

All instance methods receive an implicit argument called `this`, which can be used inside any method to refer to the *current object*. The current object is the object on which the method was called. The `this` argument is an implicit reference to the calling object and as such is not required in most cases.

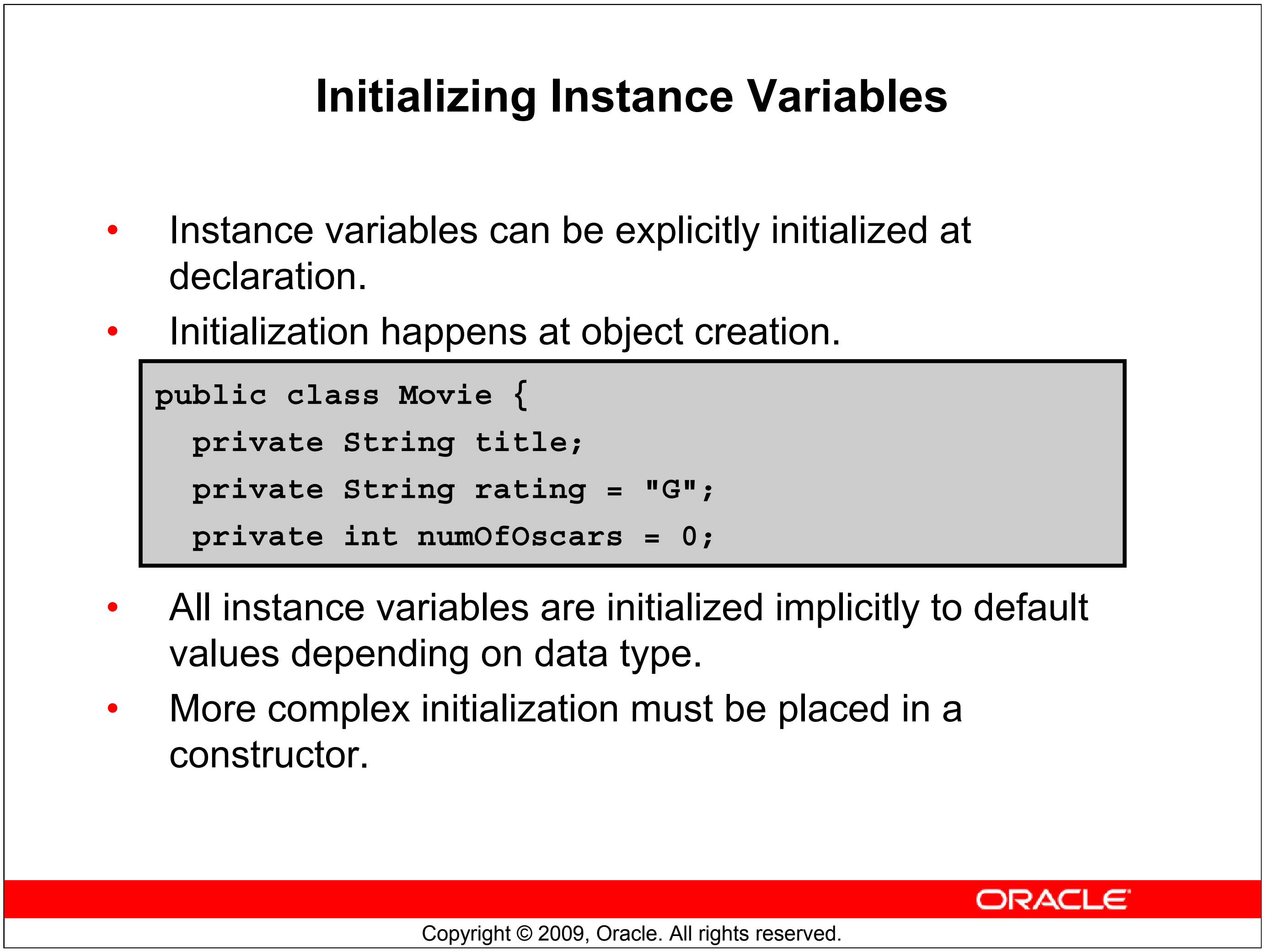
How Is `this` Used Inside an Instance Method?

Inside an instance method, any unqualified reference to an instance variable or instance method is implicitly associated with the `this` reference. For example, the two following statements are equivalent:

```
public void setRating(String inRating) {
  rating      = inRating;
  this.rating = inRating;
}
```

There are two circumstances where you must use an explicit `this` reference:

- When the name of an instance variable is hidden by a formal argument of an instance method. For example, if there is an instance variable called `name`, and an argument that is also called `name`, the argument hides the instance variable. Any reference to `name` accesses the argument rather than the variable. To access the instance variable, you must use `this.name`.
- When you need to pass a reference to the current object as an argument to another method



Initializing Instance Variables

- Instance variables can be explicitly initialized at declaration.
- Initialization happens at object creation.

```
public class Movie {
  private String title;
  private String rating = "G";
  private int numOfOscars = 0;
```

- All instance variables are initialized implicitly to default values depending on data type.
- More complex initialization must be placed in a constructor.

Initializing Instance Variables

When an object is created, Java performs default initialization for all the instance variables in the object:

- `char` variables are set to `\u0000`.
- `byte`, `short`, `int`, and `long` variables are set to `0`.
- `boolean` variables are set to `false`.
- `float` and `double` variables are set to `0.0`.
- Object references are set to `null`.

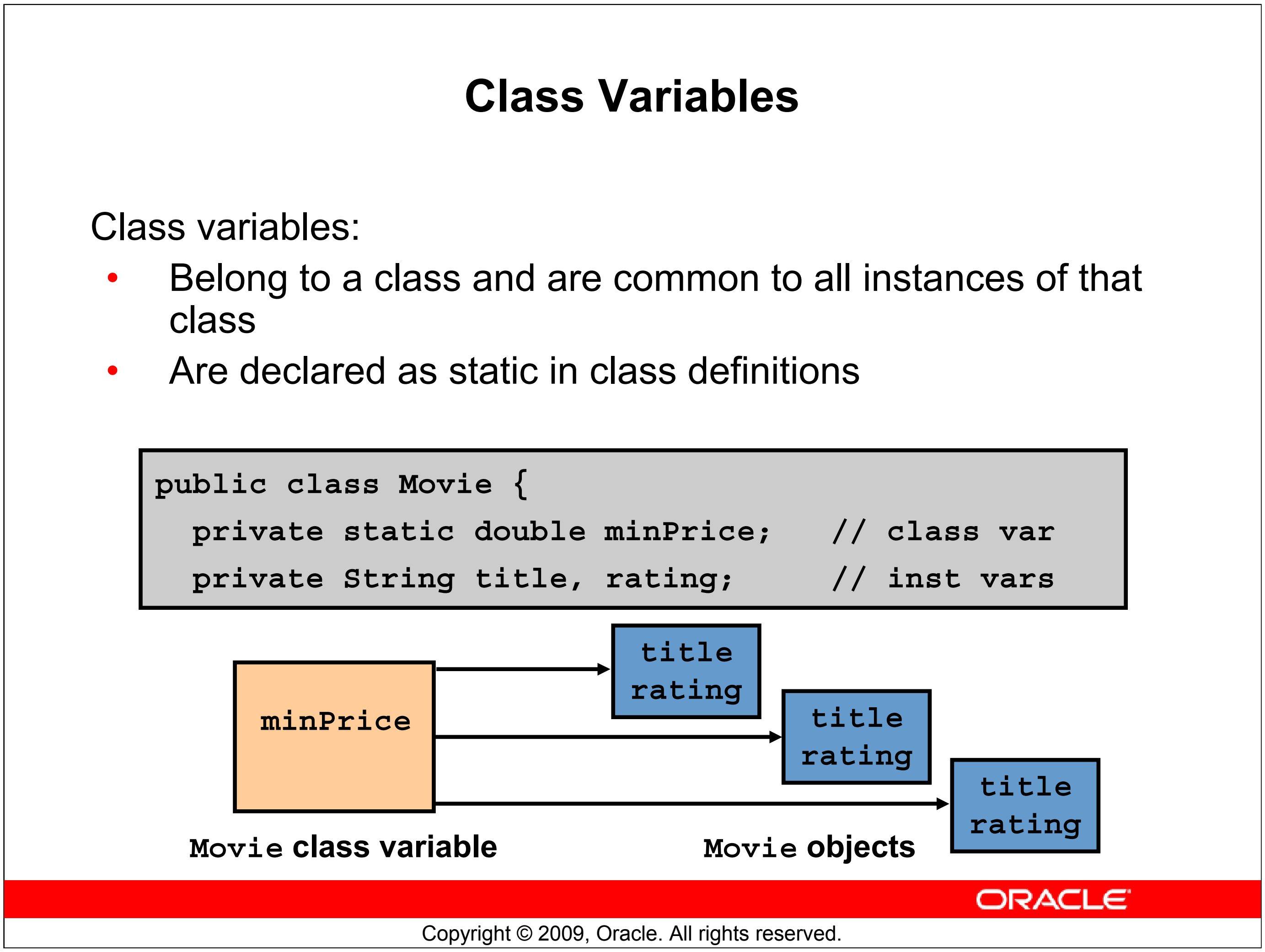
Explicit Initialization of Instance Variables

To initialize an instance variable to a nondefault value, you can specify initializers. This allows instance variables to be assigned an explicit value, as in the following example:

```
public class Movie {
  private String rating = "G";
  …
}
```

Complex Initialization of Instance Variables

Initializers are fine if you want to assign a simple value to an instance variable. However, if you want to carry out more sophisticated initialization, you must use a constructor. A constructor is a special instance method that is used to initialize new instances of a class.



Class Variables

Class variables:

- Belong to a class and are common to all instances of that class
- Are declared as static in class definitions

```
public class Movie {
  private static double minPrice;   // class var
  private String title, rating;     // inst vars
```

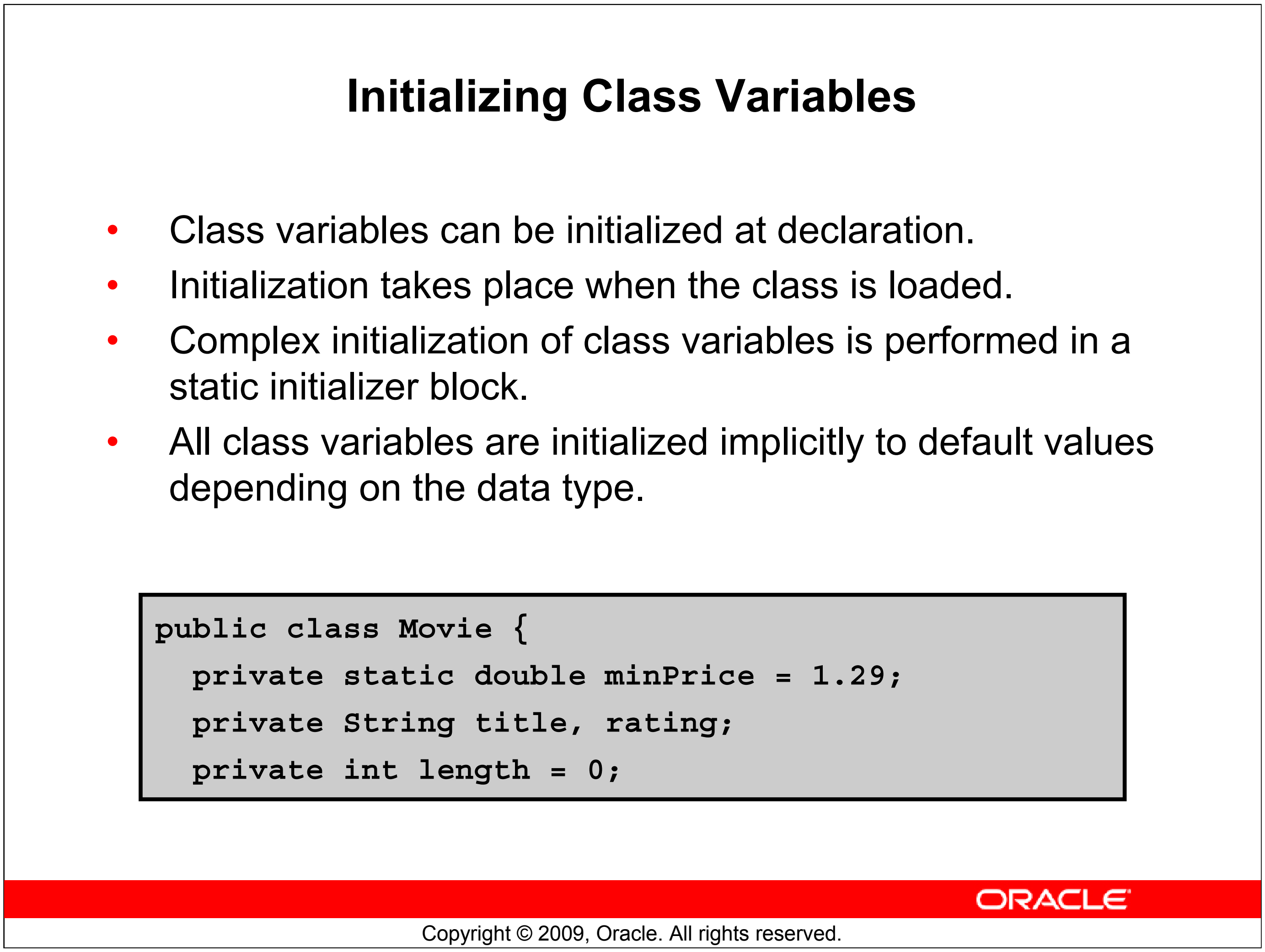
Class Variables

A class variable, which is also called a `static` variable, is a variable that belongs to a class and is common to all instances of that class. In other words, there is only one instance of a class variable, no matter how many instances of that class exist.

Defining Class Variables in Java

In Java, you declare class variables by using the `static` keyword. In the example, `minPrice` has been declared as a class variable because the minimum price is the same for all `Movie` objects. Notice that `minPrice` has been declared `private` because it must be accessed only by methods of the `Movie` class.

In this example, the `minPrice` is the same for all movies, regardless of title or rating.



Initializing Class Variables

- Class variables can be initialized at declaration.
- Initialization takes place when the class is loaded.
- Complex initialization of class variables is performed in a static initializer block.
- All class variables are initialized implicitly to default values depending on the data type.

```
public class Movie {
  private static double minPrice = 1.29;
  private String title, rating;
  private int length = 0;
```

Initializing Class Variables

Class variables are initialized when the class is loaded. Do not initialize class variables in a constructor; constructors are for initializing instance variables rather than class variables.

Default Initialization of Class Variables

Class variables have the same default values as instance variables: numbers are set to `0`, `boolean` variables are set to `false`, characters are set to `\u0000`, and references are set to `null`.

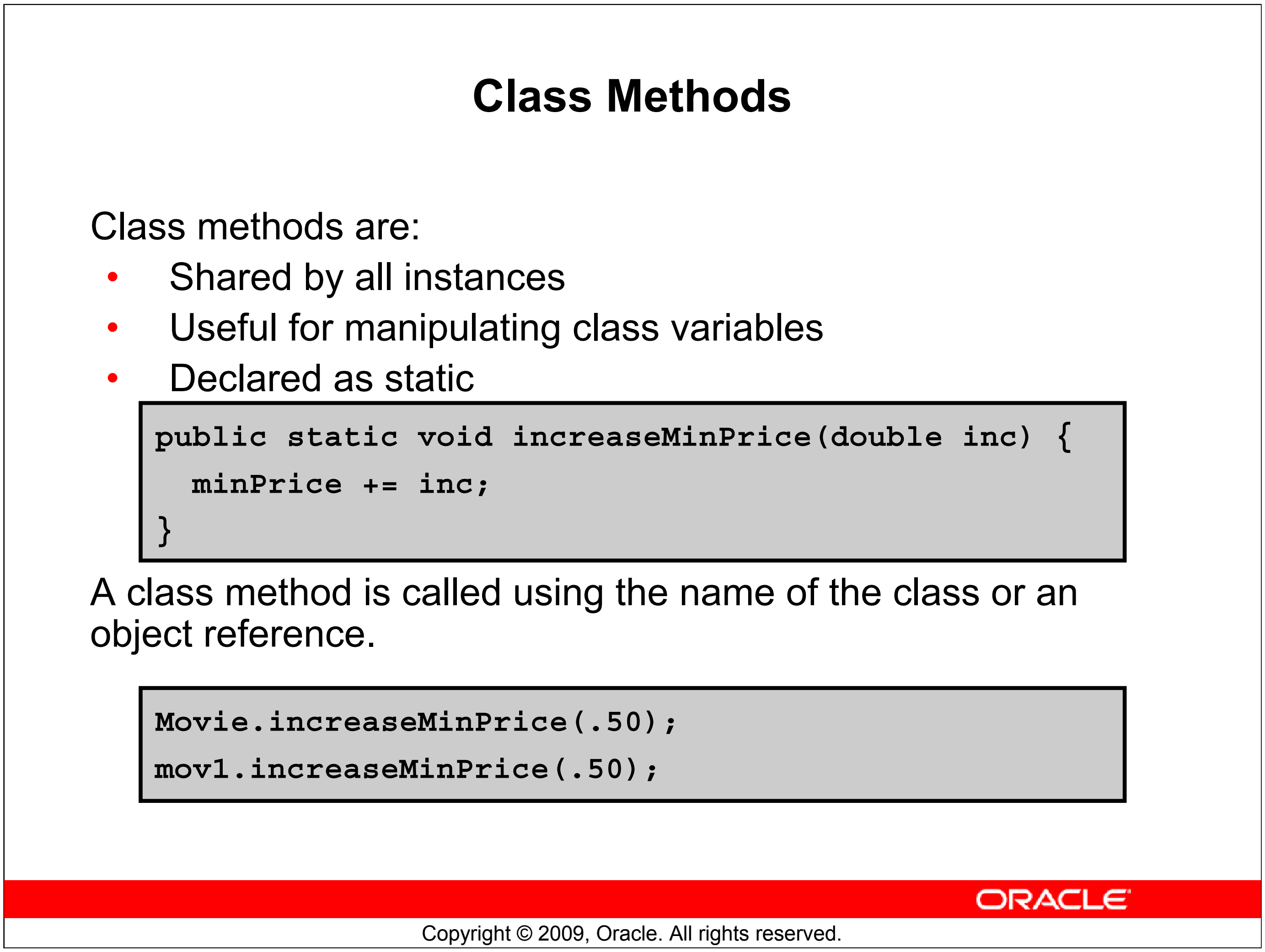
Explicit Initialization of Class Variables

Class variables can be initialized with nondefault values, just like instance variables. For example, the `minPrice` variable in the slide has been set to `1.29`.

Complex Initialization of Class Variables

Complex initialization of class variables is performed in a `static` initialization block, or *static initializer*. A static initializer is not named, has no return value, and begins with the `static` keyword, followed by a block of code inside braces. It is similar to a constructor except that it executes only once and does not depend on any instance of the class.

```
public class Movie {
  private static double minPrice;
  static {
    Date todaysDate = new Date();
    minPrice = getMinPrice(todaysDate);
  }
```



Class Methods

Class methods are:

- Shared by all instances
- Useful for manipulating class variables
- Declared as static

```
public static void increaseMinPrice(double inc) {
  minPrice += inc;
}
```

A class method is called using the name of the class or an object reference.

```
Movie.increaseMinPrice(.50);
mov1.increaseMinPrice(.50);
```

Class Methods

A class method, which is also called a `static` method, is a method that belongs to a class and is shared by all instances of that class. Unlike an instance method, a class method does not operate on a single object, and so it does not have a `this` reference. A class method can access only the class variables and class methods of its class.

Why Use Class Methods?

Class methods are an ideal way to access class variables. In fact, they are the only way if no instances of the class currently exist. For example, the `increaseMinPrice()` method in the slide changes the minimum price of all movies, whether they are currently instantiated or not.

How Do You Call a Class Method?

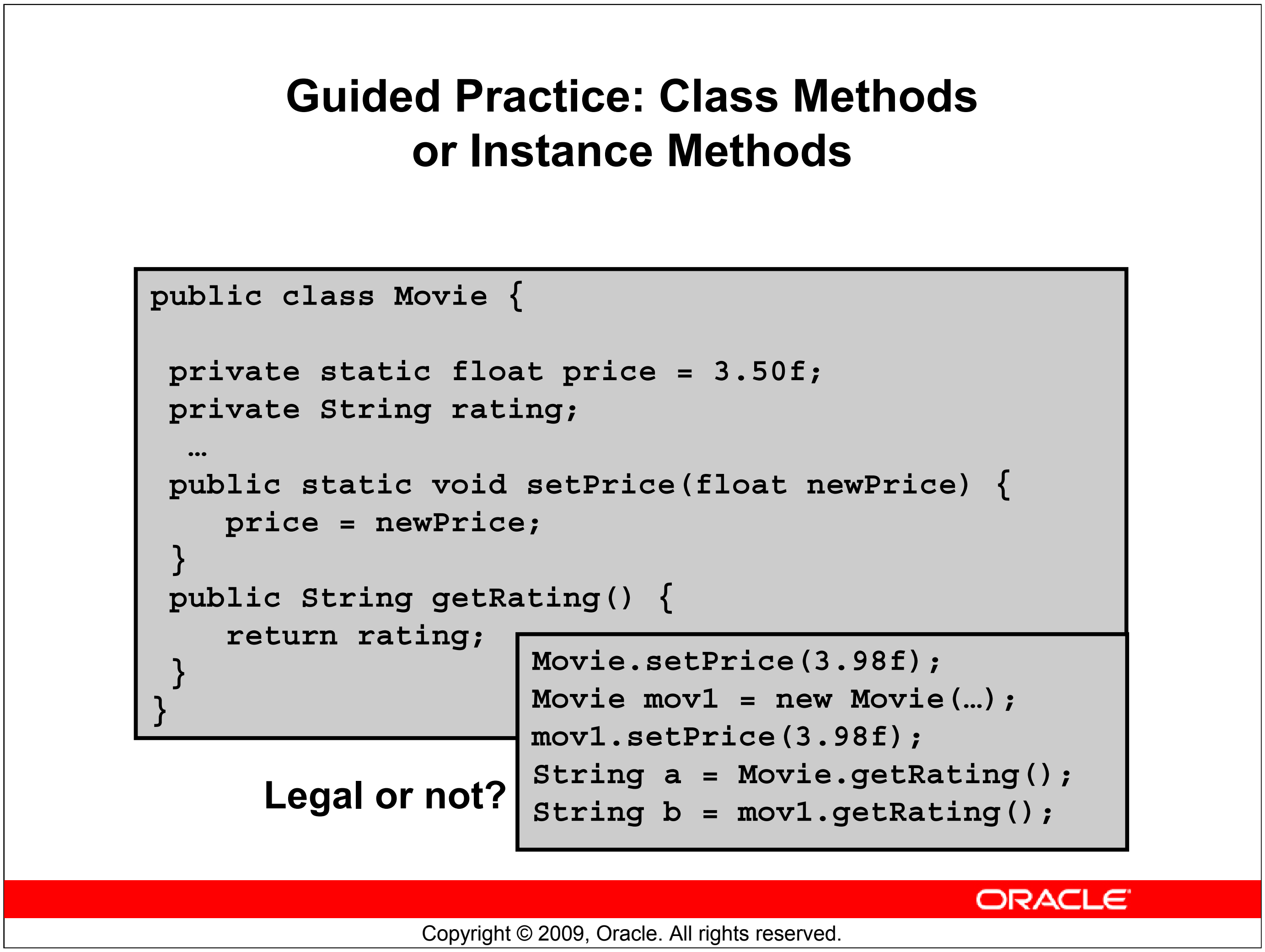
Class methods are called by using the following general syntax:

```
ClassName.classMethodName( … argumentList … );
```

You can also call class methods by using an object reference before the dot, rather than the name of the class, but the method can still access only class variables and class methods.

Static Methods

You may want to create a method that is used outside of any instance context. Declare a method to be static, which may only call other static methods directly. Static methods may not refer to their superclass or its methods.



Guided Practice: Class Methods or Instance Methods

```
public class Movie {

 private static float price = 3.50f;
 private String rating;
   …
 public static void setPrice(float newPrice) {
    price = newPrice;
 }
 public String getRating() {
    return rating;
 }
}
```

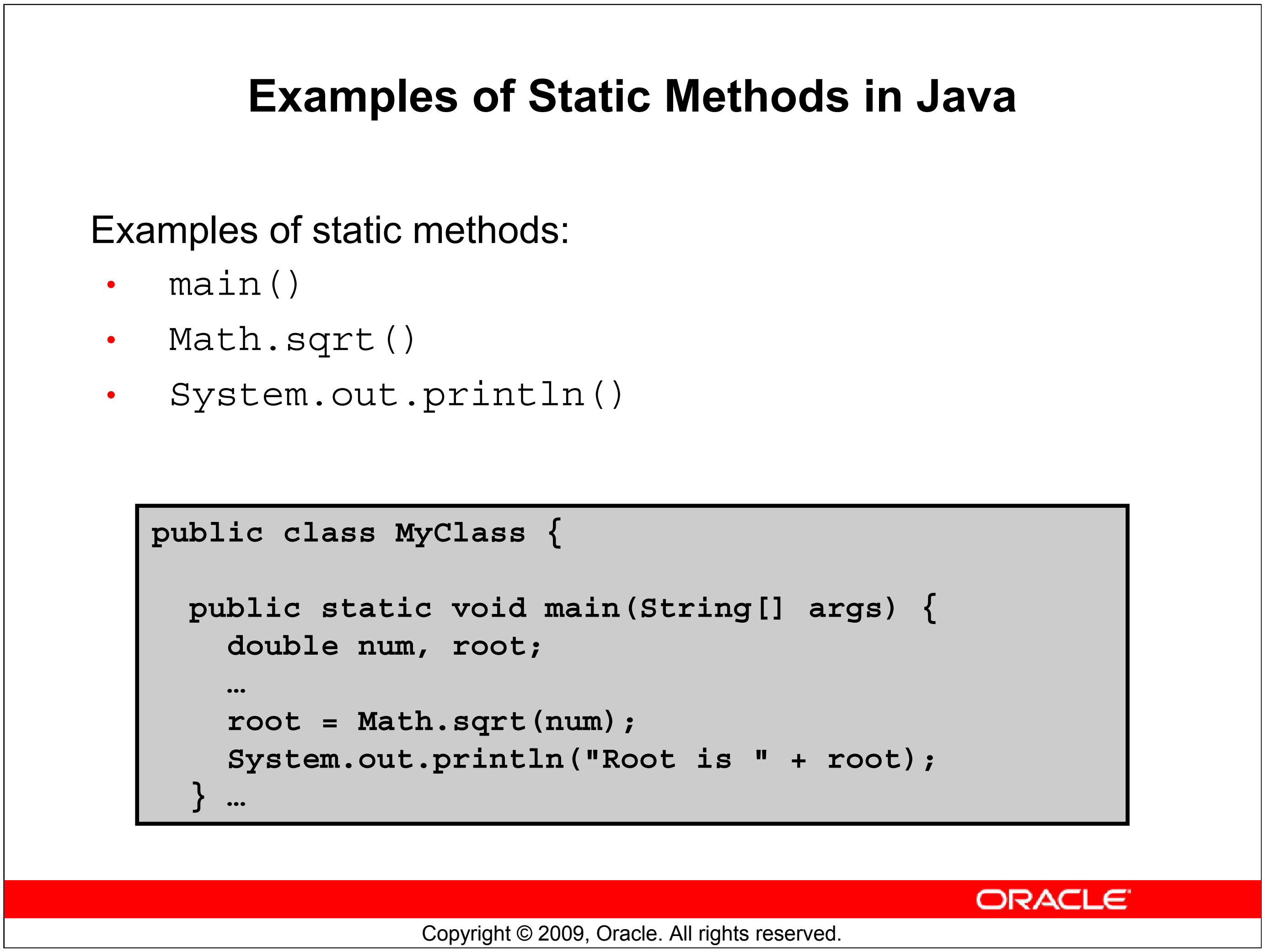
Legal or not?

```
Movie.setPrice(3.98f);
Movie mov1 = new Movie(…);
mov1.setPrice(3.98f);
String a = Movie.getRating();
String b = mov1.getRating();
```

Guided Practice: Class Methods or Instance Methods

Describe the definition of the `Movie` class in the slide. Then take a look at the test code, where a `Movie` object is created and various methods are called.

Explain the `Movie` class and the code that uses it. Which statements are legal and which are illegal? Why?



Examples of Static Methods in Java

Examples of static methods:

- `main()`
- `Math.sqrt()`
- `System.out.println()`

```
public class MyClass {

  public static void main(String[] args) {
    double num, root;
    …
    root = Math.sqrt(num);
    System.out.println("Root is " + root);
  } …
```

Examples of Static Methods in Java

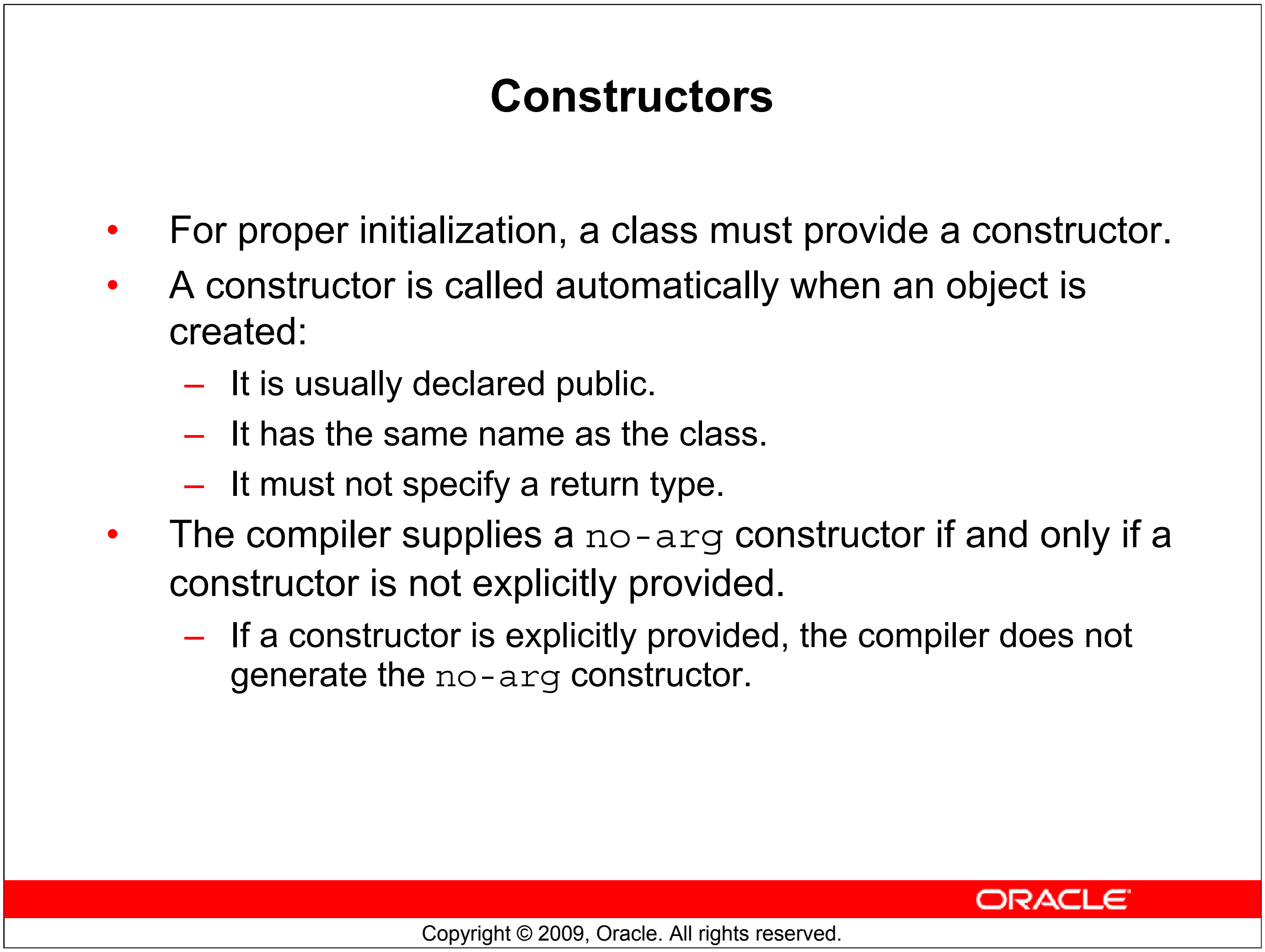
`main()` Is a Static Method

When you run a Java application, the virtual machine locates and calls the `main()` method of that class. Even though everything in a Java program must be contained within a class, you need not create an instance of the class if `main()` just calls class methods. If `main()` accesses instance methods or variables of its own class, it must first instantiate itself:

```
public class Movie {
  public void increaseMinPrice(double increase) { … }

  public static void main(String[] args) {
    Movie myMovie = new Movie();
    myMovie.increaseMinPrice(.20);
  }
}
```

`Math.sqrt()` Is a Static Method

The `Math` class provides class methods to compute many mathematical functions, such as trigonometric functions and logarithms. It also provides several class constants, such as `E` (`2.71828`…) and `PI` (`3.1415926`…).



Constructors

- For proper initialization, a class must provide a constructor.
- A constructor is called automatically when an object is created:
  - It is usually declared public.
  - It has the same name as the class.
  - It must not specify a return type.
- The compiler supplies a `no-arg` constructor if and only if a constructor is not explicitly provided.
  - If a constructor is explicitly provided, the compiler does not generate the `no-arg` constructor.

Constructors

When an object is created, its instance variables are initialized to their default values. However, you need to provide one or more constructors in a class to initialize its instance variables properly and to enable users of the class to specify the initial state of an object.

For example, with the `Movie` class, you have already seen examples such as:

```
Movie mov1 = new Movie ("Gone with the Wind");
```

For a user to create an object in this way, the `Movie` class must provide a constructor that initializes the state of a `Movie` object with the specified title of the movie.

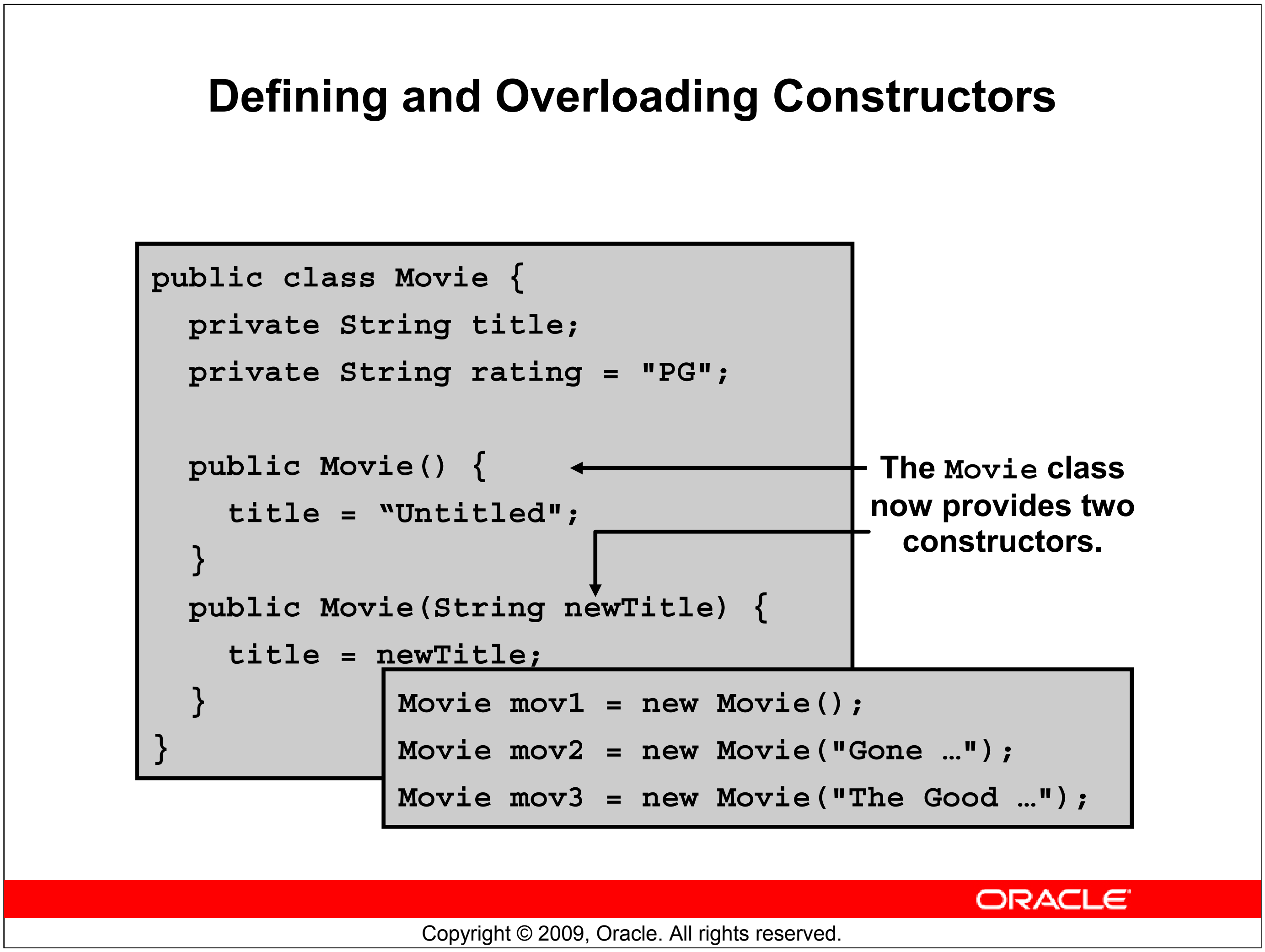
How and When Are Constructors Called?

A constructor is a special method that is called automatically by the run-time system when an object is created. A constructor has the same name as the class; it can have arguments but must not specify a return type. Constructors must be declared as `public` unless you want to restrict who can create instances of the class.

What Happens if You Do Not Provide a Constructor?

If you do not provide constructors, a default no-argument (`no-arg`) constructor is provided for you. This constructor takes no arguments and does nothing, but it does at least allow objects to be created. The `no-arg` constructor invokes the `no-arg` constructor of its parent class.

Note: If you want a specific `no-arg` constructor as well as constructors that take arguments, you must explicitly provide your own `no-arg` constructor.



Defining and Overloading Constructors

```
public class Movie {
  private String title;
  private String rating = "PG";

  public Movie() {
    title = "Untitled";
  }
  public Movie(String newTitle) {
    title = newTitle;
  }
}
```

The Movie class now provides two constructors.

```
Movie mov1 = new Movie();
Movie mov2 = new Movie("Gone …");
Movie mov3 = new Movie("The Good …");
```

Anatomy of a Constructor

A constructor is a special method that is called automatically when an object is created.
A constructor must have the same name as the class. It can have arguments, but it must not specify a return type.

Overloaded Constructors

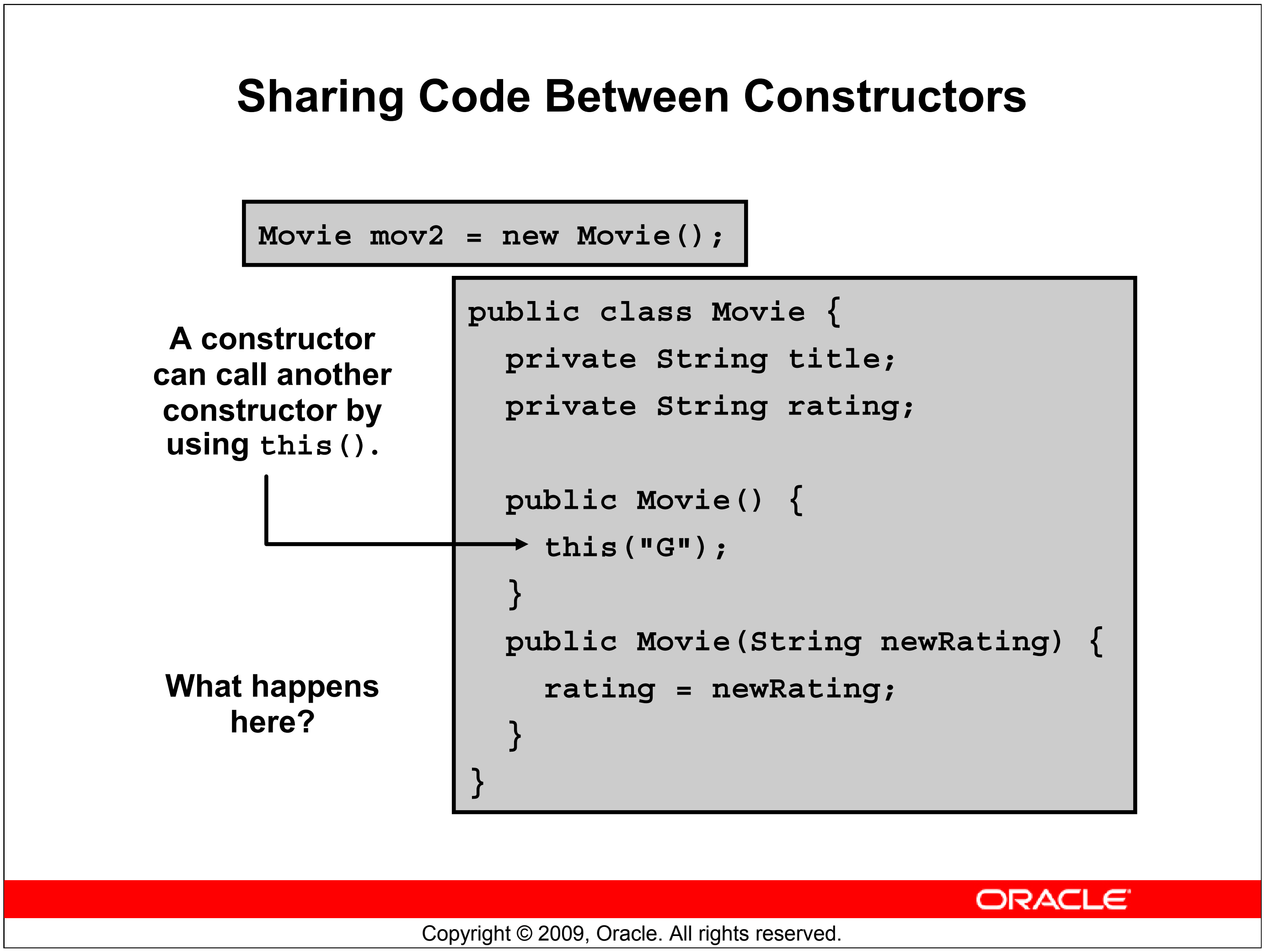
As with overloaded methods, if you want to provide more than one constructor in a class, each one must have a different signature. Because each constructor must always have the same name, this simply means that each constructor must have different numbers or types of arguments.

Example

In the example, the Movie class has two simple constructors: one with no arguments and the other that takes a String argument for the title of the movie.

Users of the Movie class can now create movies with different titles. When you create a new Movie object, the compiler decides which constructor to call based on the arguments that are specified in parentheses in the new statement, as in the following example:

```
Movie mov1 = new Movie();      // calls the no-arg constructor
                              // creates movies entitled "Untitled"
Movie mov2 = new Movie("Last ...");// calls the constructor with the
                                     String argument
```



Sharing Code Between Constructors

```
Movie mov2 = new Movie();
```

```
public class Movie {
  private String title;
  private String rating;

  public Movie() {
    this("G");
  }
  public Movie(String newRating) {
    rating = newRating;
  }
}
```

A constructor can call another constructor by using `this()`.

What happens here?

Sharing Code Between Constructors

A constructor can call another constructor of the same class by using the `this()` syntax.

```
public Movie() {             // First constructor
  this("G");
}
public Movie(String r) {    // Second constructor
  rating = r;
}
```

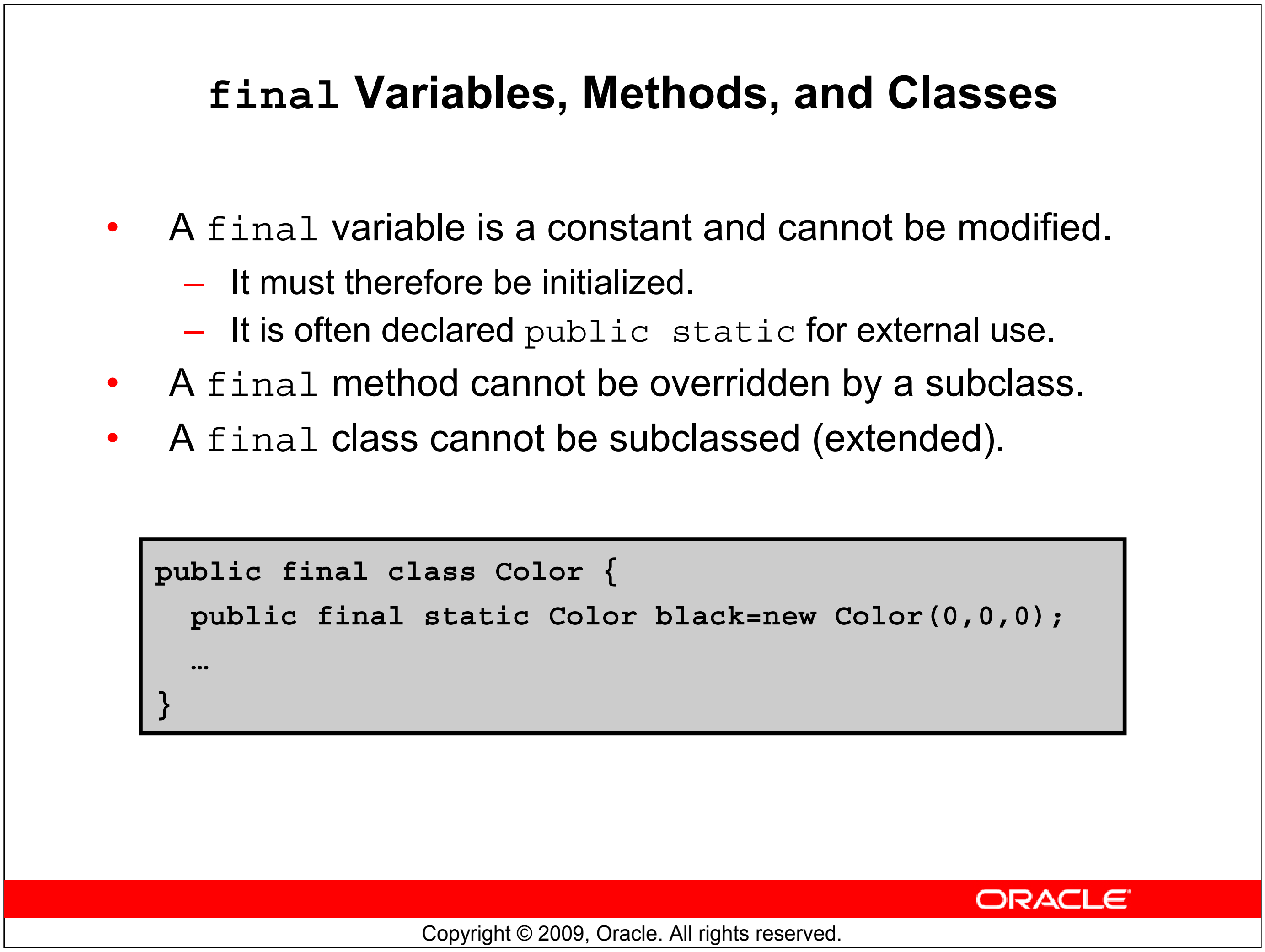
The first constructor calls the second constructor, passing `"G"` as an argument. The second constructor then copies the string into the `rating` instance variable. Using this technique ensures that the default rating for all `Movies` is `"G"` without duplicating the code in multiple constructors.

By using `this()`, you avoid duplicate code in multiple constructors. This technique is especially useful if the initialization routine is complex. All the complex code goes into one constructor that is called from all the others.

Syntax Rules

When one constructor calls another by using the `this()` syntax, there are a few rules of syntax that you need to be aware of:

- The call to `this()` must be the first statement in the constructor.
- The arguments to `this()` must match those of the target constructor.



final Variables, Methods, and Classes

- A final variable is a constant and cannot be modified.
  - It must therefore be initialized.
  - It is often declared public static for external use.
- A final method cannot be overridden by a subclass.
- A final class cannot be subclassed (extended).

```
public final class Color {
  public final static Color black=new Color(0,0,0);
  …
}
```

final Variables

By default, all variables can be modified and methods can be overridden. Specifying a variable as final prevents modification of its value, thereby making a constant value.

This is useful for guaranteeing that a value is consistent across all users of the class. These variables are usually declared public static final as class-wide constants.

final Methods

A final method is one that cannot be overridden in a subclass. In other words, if programmers inherit from the class, they are not allowed to provide an alternative version of this method. This is a useful technique to prevent programmers from inadvertently (or maliciously) redefining core methods that must work a certain way.

final Classes

You can also declare a class to be final. A final class is one that cannot be inherited from. In fact, the Color class that is shown in the slide is a final class. By declaring a class as final, you are making a strong design decision that the class is complete enough to meet all its current and future requirements and will never need to be extended to provide additional functionality.

Reclaiming Memory

- When all references to an object are lost, the object is marked for garbage collection.
- Garbage collection reclaims memory that is used by the object.
- Garbage collection is automatic.
- There is no need for the programmer to do anything, but the programmer can give a hint to `System.gc();`.

Reclaiming Memory

Memory management in Java is automatic. When an object is created, memory is allocated for the object from a heap. When there are no more references to that object, it is marked for garbage collection. When the garbage collector runs, it searches for marked memory and returns it to the heap.

There are no *free()* or *delete()* functions in Java as there are in C++. To force an object to be marked for garbage collection, simply remove all references to that object by setting the references to `null`.

When Does Garbage Collection Occur?

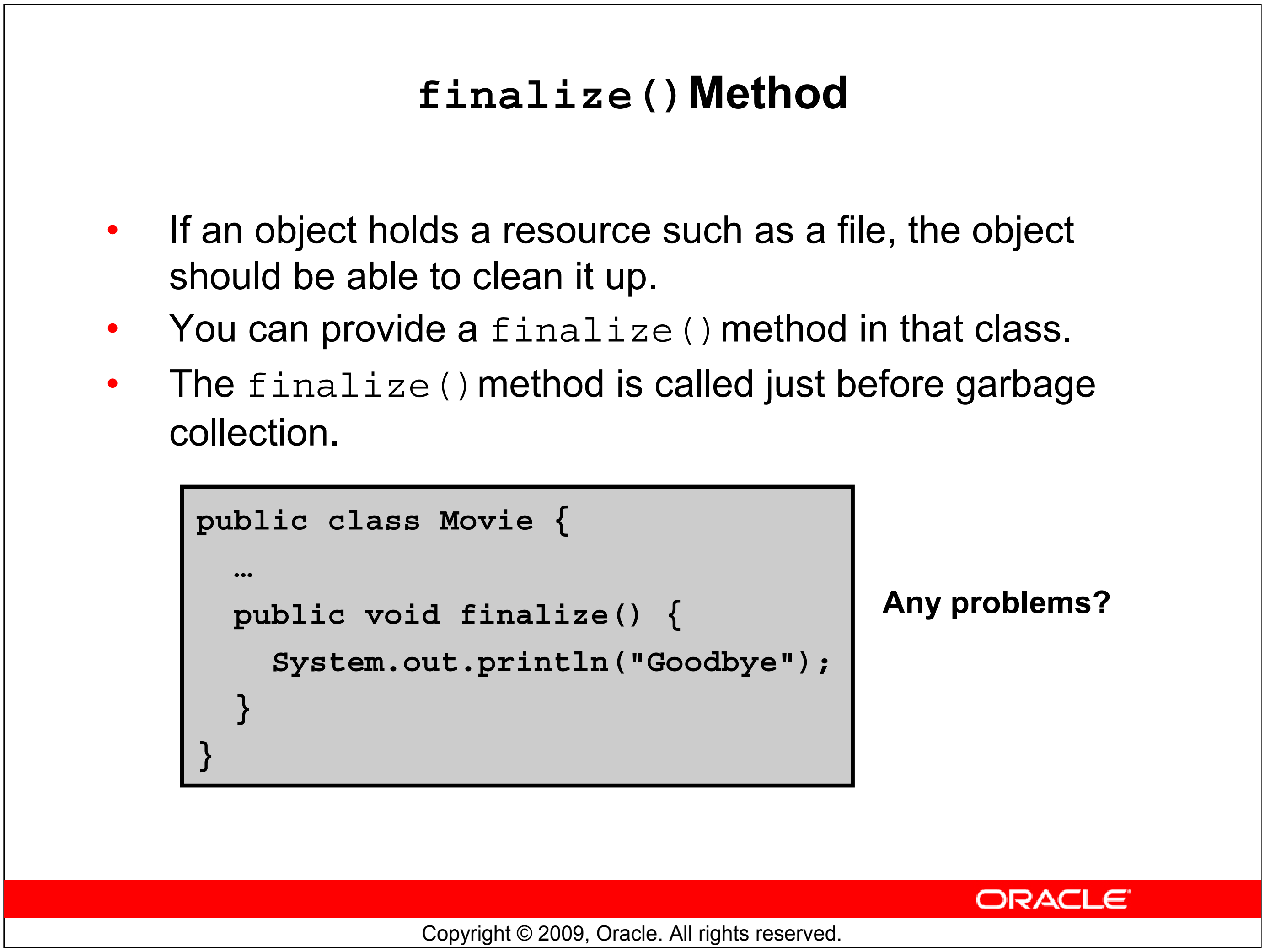
Garbage collection is implementation-specific. Some environments run garbage collection when the amount of free memory that is available to the Java Virtual Machine (JVM) falls below some arbitrary threshold. The JVM performs garbage collection in a low-priority thread. When the JVM has nothing else to do, the garbage collector thread receives some CPU time to see whether any memory can be reclaimed.

You can explicitly request the garbage collector to run by calling the `gc()` method:

```
System.gc();
```

This is only a request for garbage collection. It is not a guarantee that the JVM will comply.

Because of this internal garbage collection process, you do not automatically know when an object is deleted and whether the necessary amount of memory that is needed for this object is still in use.



`finalize()` Method

- If an object holds a resource such as a file, the object should be able to clean it up.
- You can provide a `finalize()` method in that class.
- The `finalize()` method is called just before garbage collection.

```
public class Movie {
  …
  public void finalize() {
    System.out.println("Goodbye");
  }
}
```

Any problems?

`finalize()` Method

In some languages (such as C++), a class can provide a *destructor*. A destructor is similar to a constructor except that it is called just before an object is destroyed. A destructor is normally used to free up resources that are held by the object, such as any secondary memory that is allocated by the object, open files, and so on.

Java Does Not Support Destructors

Java manages memory automatically, and thus an object need not explicitly free any secondary memory that it may have allocated. Consequently, Java does not support destructors. Instead, to enable an object to clean up resources other than memory, such as open files, Java permits a class to provide a `finalize()` method.

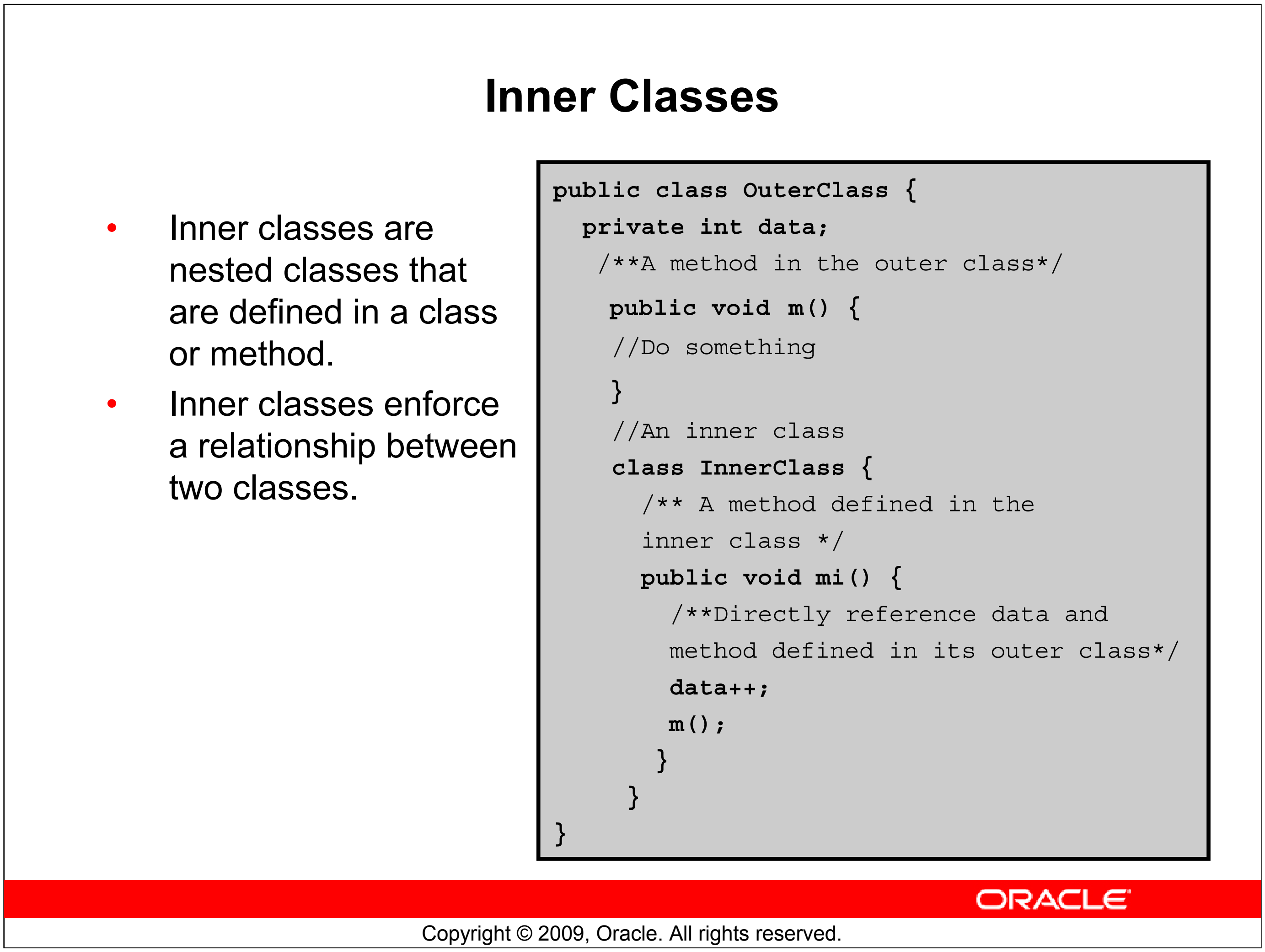
`finalize()` Method (continued)

Anatomy of the `finalize()` Method

The `finalize()` method is called when an object undergoes garbage collection. Unfortunately, as you have already seen, there is no guarantee regarding when this will happen or that it will happen before the program exits. The JVM reserves the right not to immediately collect the memory that is associated with an object after calling the finalizer. Such objects are known as *phantoms*.

Alternatives to `finalize()`

The unpredictability of when `finalize()` is called is unacceptable if resources are scarce. The only solution is to manage such resources manually. To take control of the process, you can define a public `dispose()` method in your class; users of your class must call this method when they have finished using an object of your class. You can still keep your `finalize()` method if you want to, as a final effort to clean up resources.



Inner Classes

- Inner classes are nested classes that are defined in a class or method.
- Inner classes enforce a relationship between two classes.

```
public class OuterClass {
  private int data;
   /**A method in the outer class*/
    public void m() {
    //Do something
    }
    //An inner class
    class InnerClass {
      /** A method defined in the
      inner class */
      public void mi() {
        /**Directly reference data and
        method defined in its outer class*/
        data++;
        m();
       }
     }
}
```

Inner Classes

An inner class is simply a class that is defined within the scope of another class.

You define inner classes because they functionally support the outer class or because they make sense in the context of the outer class. Inner classes have different privileges when accessing outer class members according to the type of inner class they are.

An inner class may be used just like a regular class. Normally you declare a class an inner class if it is only used by its outer class.

An inner class has the following features:

- An inner class is compiled into a class named *`OuterClassName$InnerClassName`*`.class`.
- An inner class can reference the data and methods defined in the outer class in which it nests, so you do not have to pass the reference of an object of the outer class in the constructor of the inner class. For this reason, inner classes can make programs simple and concise.
- An inner class can be declared with a visibility modifier subject to the same visibility rules applied to a member of the class.
- An inner class can be declared `static`. A `static` inner class can be accessed using the outer class name. A `static` inner class cannot access non-static members of the outer class.

Inner Classes (continued)

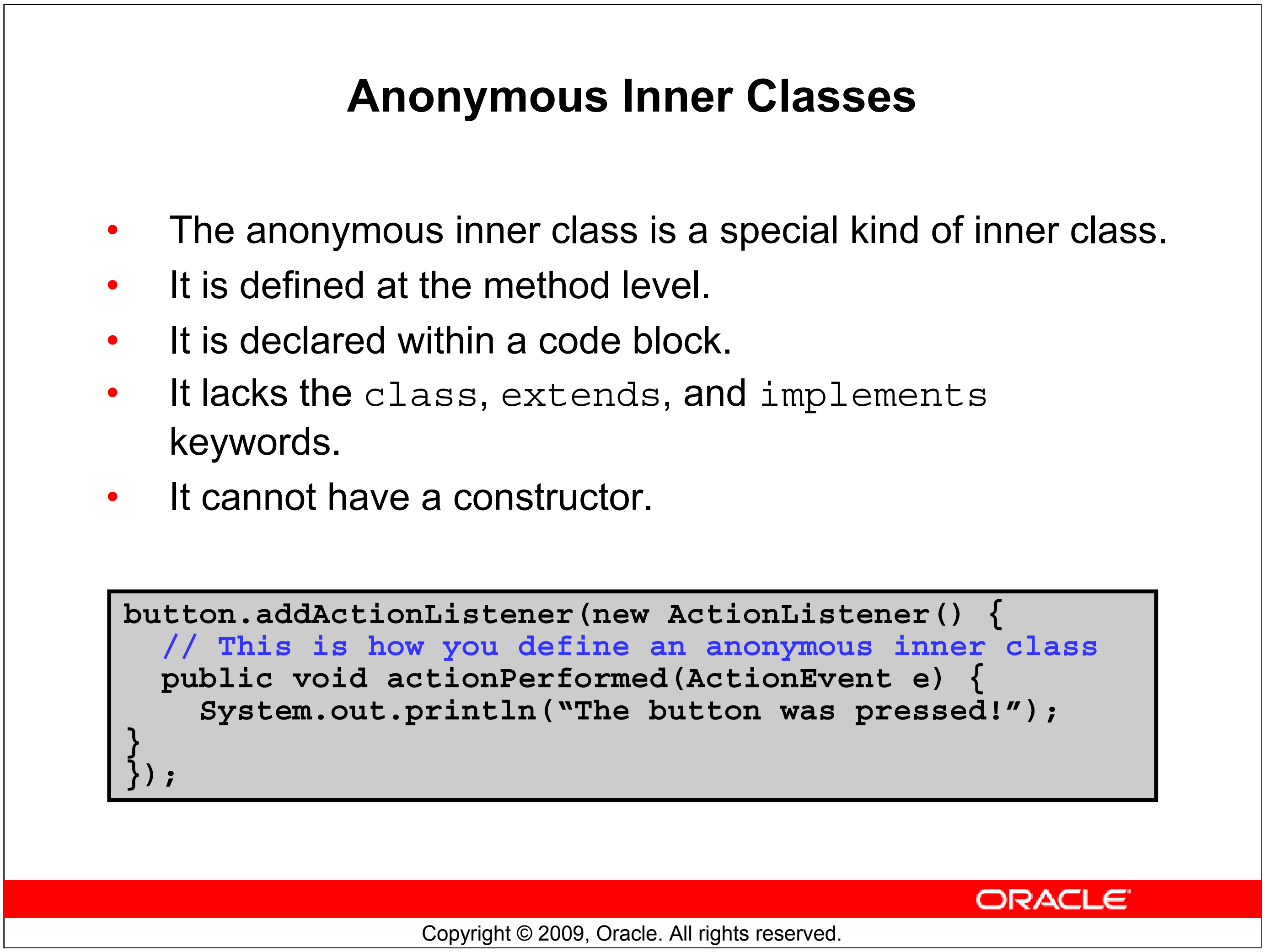
- Objects of an inner class are often created in the outer class. However you can also create an object of an inner class from another class. If the inner class is nonstatic, you must first create an instance of the outer class, and then use the following syntax to create an object for the inner class:

```
      OuterClass.InnerClass innerObject = outerObject.new
                                          InnerClass();
```

- If the inner class is `static`, use the following syntax to create an object for it:

```
      OuterClass.InnerClass innerObject = new
                                 OuterClass.InnerClass();
```

Note: A simple use of inner classes is to combine dependent classes into a primary class. This reduces the number of source files. It also makes class files easy to organize because all the class files are named with the primary class as the prefix.



Anonymous Inner Classes

- The anonymous inner class is a special kind of inner class.
- It is defined at the method level.
- It is declared within a code block.
- It lacks the `class`, `extends`, and `implements` keywords.
- It cannot have a constructor.

```
button.addActionListener(new ActionListener() {
  // This is how you define an anonymous inner class
  public void actionPerformed(ActionEvent e) {
    System.out.println(“The button was pressed!”);
}
});
```

Anonymous Inner Classes

An anonymous inner class is an inner class without a name. It combines declaring an inner class and creating an instance of the class in one step. The `class` keyword is omitted; also omitted are `public`, `protected`, `extends`, and `implements`. Anonymous inner classes are commonly used to implement user interface adapters to perform event handling when using AWT or Swing events.

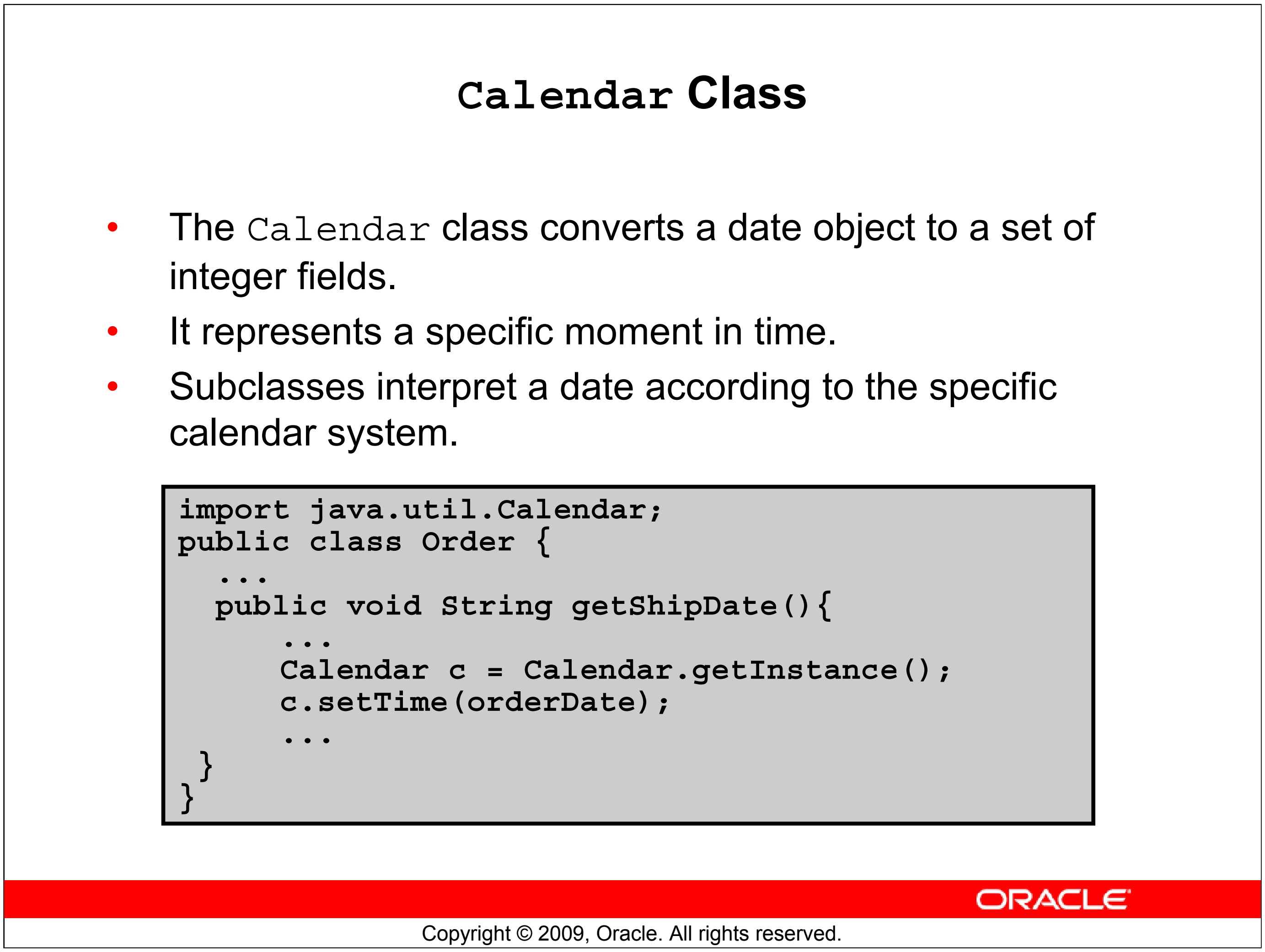
Because an anonymous inner class is a special kind of inner class, it is treated like an inner class in many ways. In addition it has the following features:

- An anonymous inner class must always extend a superclass or implement an interface, but it cannot have an explicit `extends` or `implements` clause.
- An anonymous inner class must implement all the abstract methods in the superclass or in the interface.
- An anonymous inner class always uses the `no-arg` constructor from its superclass to create an instance. If an anonymous inner class implements an interface, the constructor is `Object()`.
- An anonymous inner class can override the methods of the superclass.
- An anonymous inner class is compiled into a class named `OuterClassName$`*`n`*`.class`. For example, if an outer class `Test` has two anonymous inner classes, they are compiled into `Test$1.class` and `Test$2.class`.

Anonymous Inner Classes (continued)

Inner classes are often used as listener classes for GUI components like buttons. However the use of *anonymous* inner classes can make the program even more simple and concise, as in the following example:

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class SimpleEventDemoAnonymousInnerClass extends JFrame
{
  public SimpleEventDemoAnonymousInnerClass() {
     JButton jbtOK = new JButton(“OK”);
     setLayout(new FlowLayout());
     add(jbtOK);

     // Create and register anonymous inner class listener
     jbtOK.addActionListener(new ActionListener() {
       public void actionPerformed(ActionEvent e) {
           System.out.println(“It is OK”);
       }
      });
   }

   /** Main method */
   public static void main(String[] args) {
     JFrame frame = new SimpleEventDemoAnonymousInnerClass();
     frame.setTitle(“SimpleEventDemoAnonymousInnerClass”);
     frame.setLocationRelativeTo(null); //Center the frame
     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
     frame.setSize(220, 80);
     frame.setVisible(true);
   }
}
```



Calendar Class

- The Calendar class converts a date object to a set of integer fields.
- It represents a specific moment in time.
- Subclasses interpret a date according to the specific calendar system.

```
import java.util.Calendar;
public class Order {
   ...
  public void String getShipDate(){
       ...
       Calendar c = Calendar.getInstance();
       c.setTime(orderDate);
       ...
 }
}
```

Calendar Class

Calendar is an abstract base class for converting a Date object to a set of integer fields such as YEAR, MONTH, DAY, HOUR, and so on. (A Date object represents a specific instant in time with millisecond precision.)

Specific subclasses of Calendar interpret a Date according to the rules of a specific calendar system. One concrete example of such a subclass is GregorianCalendar.

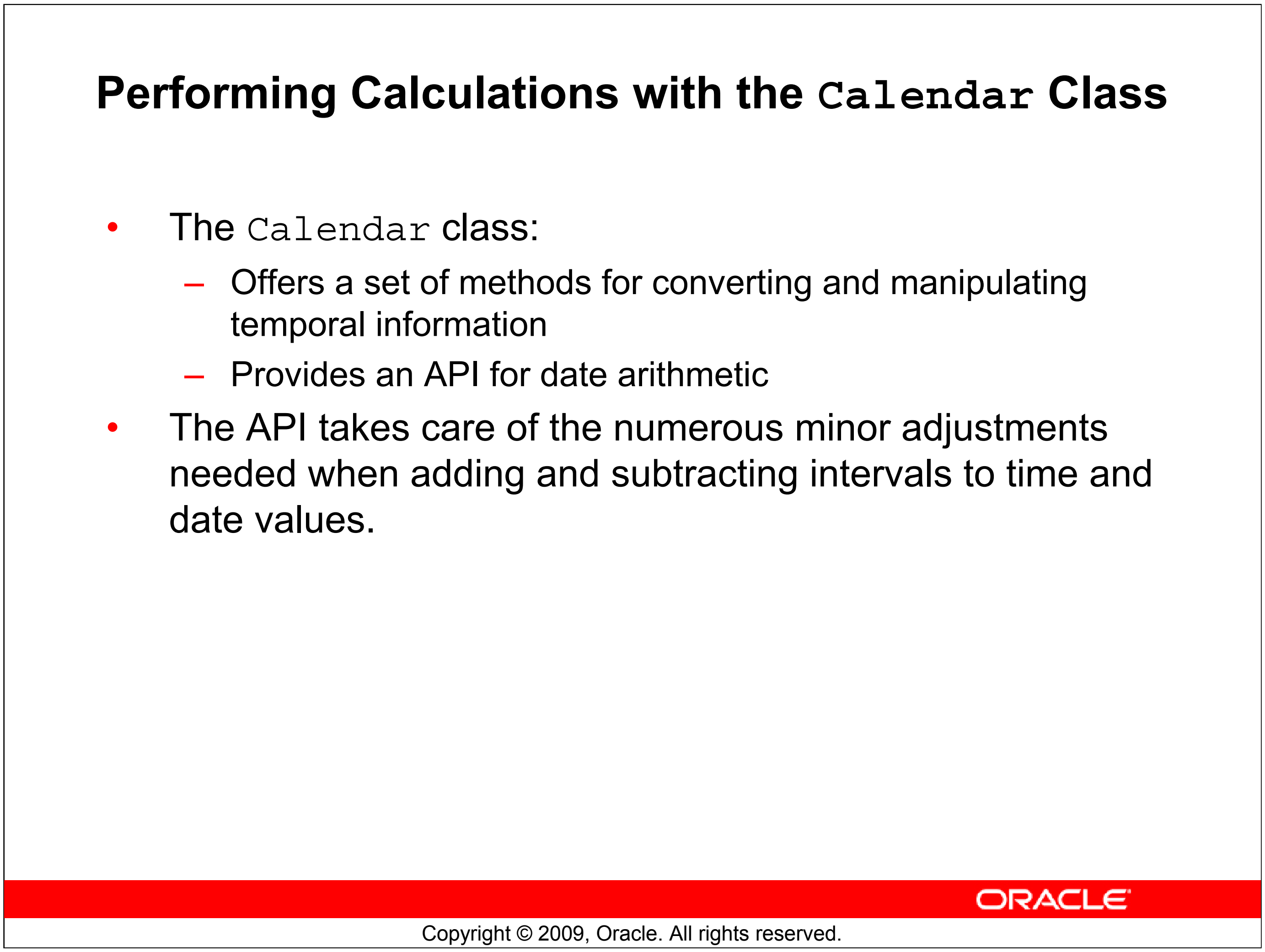
Using the Calendar Class

The Calendar object can produce all the time field values needed to implement the date-time formatting for a particular language and calendar style (for example, Japanese-Gregorian, Japanese-Traditional, and so on). Calendar defines the range of values returned by certain fields, as well as their meanings. For example, the first month of the year has the value MONTH == JANUARY for all calendars.

Like other locale-sensitive classes, Calendar provides a class method, getInstance, for getting a generally useful object of this type. A getInstance method returns a Calendar object whose time fields have been initialized with the current date and time:

```
Calendar rightNow = Calendar.getInstance();
```

The setTime method sets this particular calendar’s current time with the given date.



Performing Calculations with the `Calendar` Class

- The `Calendar` class:
  - Offers a set of methods for converting and manipulating temporal information
  - Provides an API for date arithmetic
- The API takes care of the numerous minor adjustments needed when adding and subtracting intervals to time and date values.

Performing Calculations with the `Calendar` Class

`Calendar`'s built-in date/time arithmetic API is extremely useful. For example, consider the number of lines of code that go into calculating what the date will be five months from today. You need to think about the number of days in the current month and in the intervening months, as well as to make end-of-year and leap year modifications to arrive at an accurate final result. These kinds of calculations are quite complex, and are easy for a novice developer to get wrong.

`Calendar` Class Example

The following example initializes a Calendar to 01 Jan 2008 and then adds two months and one day to it to obtain a new value.

```
package datetime;
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.GregorianCalendar;
```

Performing Calculations with the `Calendar` Class (continued)

`Calendar` Class Example (continued)

```
public class DateCalc {
    public static void main(String[] args) {
        DateCalc td = new DateCalc();
        td.doMath();
    }
     //  method to create a calendar object, add 2 m 1 d and
    //print result
    private void doMath() {
        // set calendar to 01 Jan 2008
        Calendar calendar = new GregorianCalendar(2008,
                             Calendar.JANUARY, 1);
        System.out.println("Starting date is: ");
        printCalendar(calendar);

        // add 2m 1d
        System.out.println("Adding 2m 1d...");
        calendar.add(Calendar.MONTH, 2);
        calendar.add(Calendar.DAY_OF_MONTH, 1);

        // print ending date value
        System.out.println("Ending date is: ");
        printCalendar(calendar);
    }

     //utility method to print a Calendar object using
    //SimpleDateFormat

    private void printCalendar(Calendar calendar) {
        // define output format and print
        SimpleDateFormat sdf = new SimpleDateFormat("d MMM yyyy
                                         hh:mm aaa");
        String date = sdf.format(calendar.getTime());
```

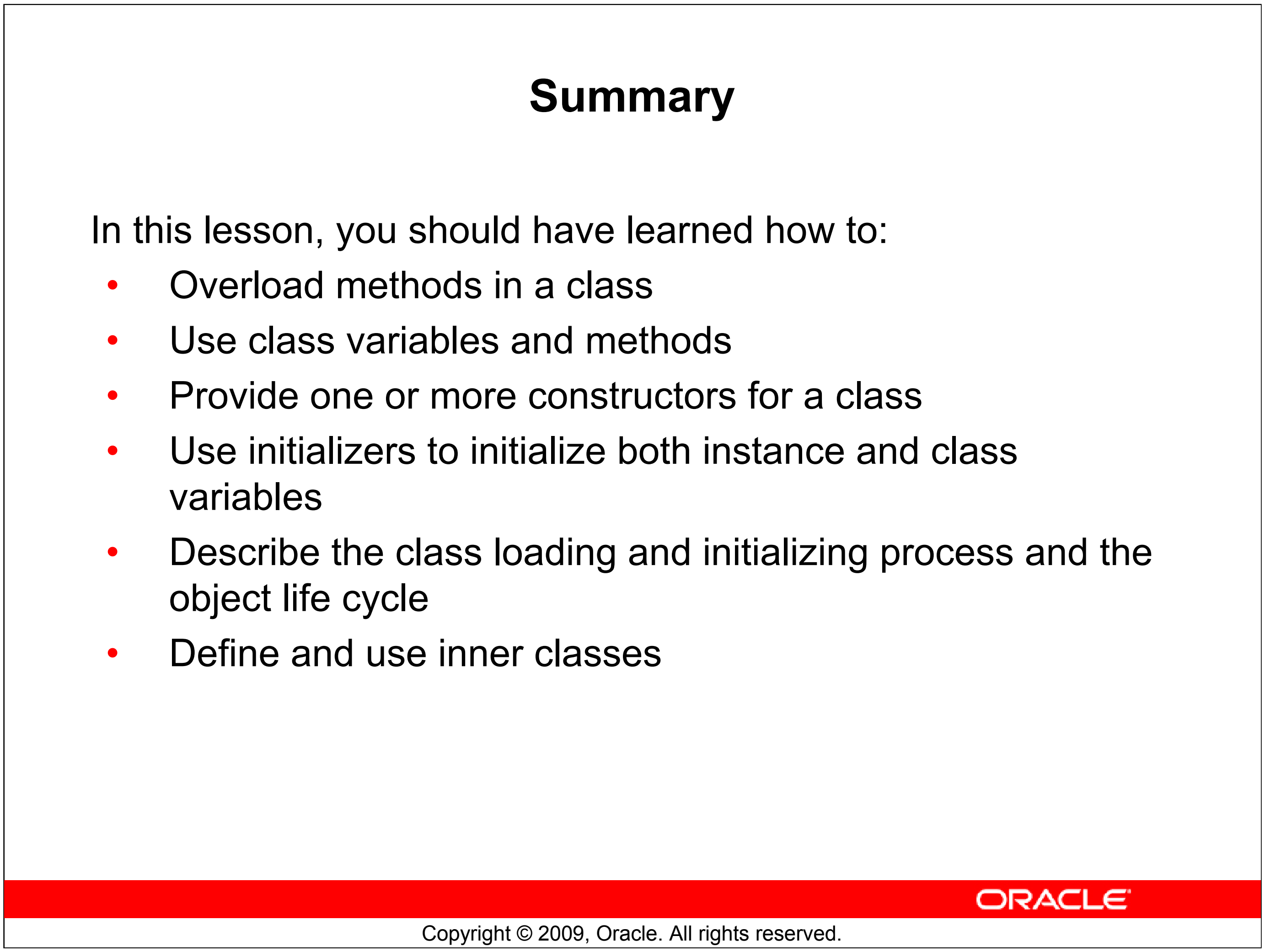
`Calendar` Class Example (continued)

```
    System.out.println(date);
          }
}
```

The main workhorse of this class is the `doMath()` method, which begins by initializing a new `GregorianCalendar` object to 1 Jan 2008. Next, the object's `add()` method is invoked; this method accepts two arguments: the name of the field to add the value to and the amount of time to be added. In this example, the `add()` method is called twice—first to add two months to the starting date and then to add a further one day to the result. After the addition is performed, the `printCalendar()` utility method is used to print the final result. Notice the use of the `SimpleDateFormat` object to turn the output of `getTime()` into a human-readable string.

When you run the class, this is the output you will see:

```
Starting date is:
1 Jan 2008 12:00 AM
Adding 2m 1d...
Ending date is:
2 Mar 2008 12:00 AM
```



Summary

In this lesson, you should have learned how to:

- Overload methods in a class
- Use class variables and methods
- Provide one or more constructors for a class
- Use initializers to initialize both instance and class variables
- Describe the class loading and initializing process and the object life cycle
- Define and use inner classes

Summary

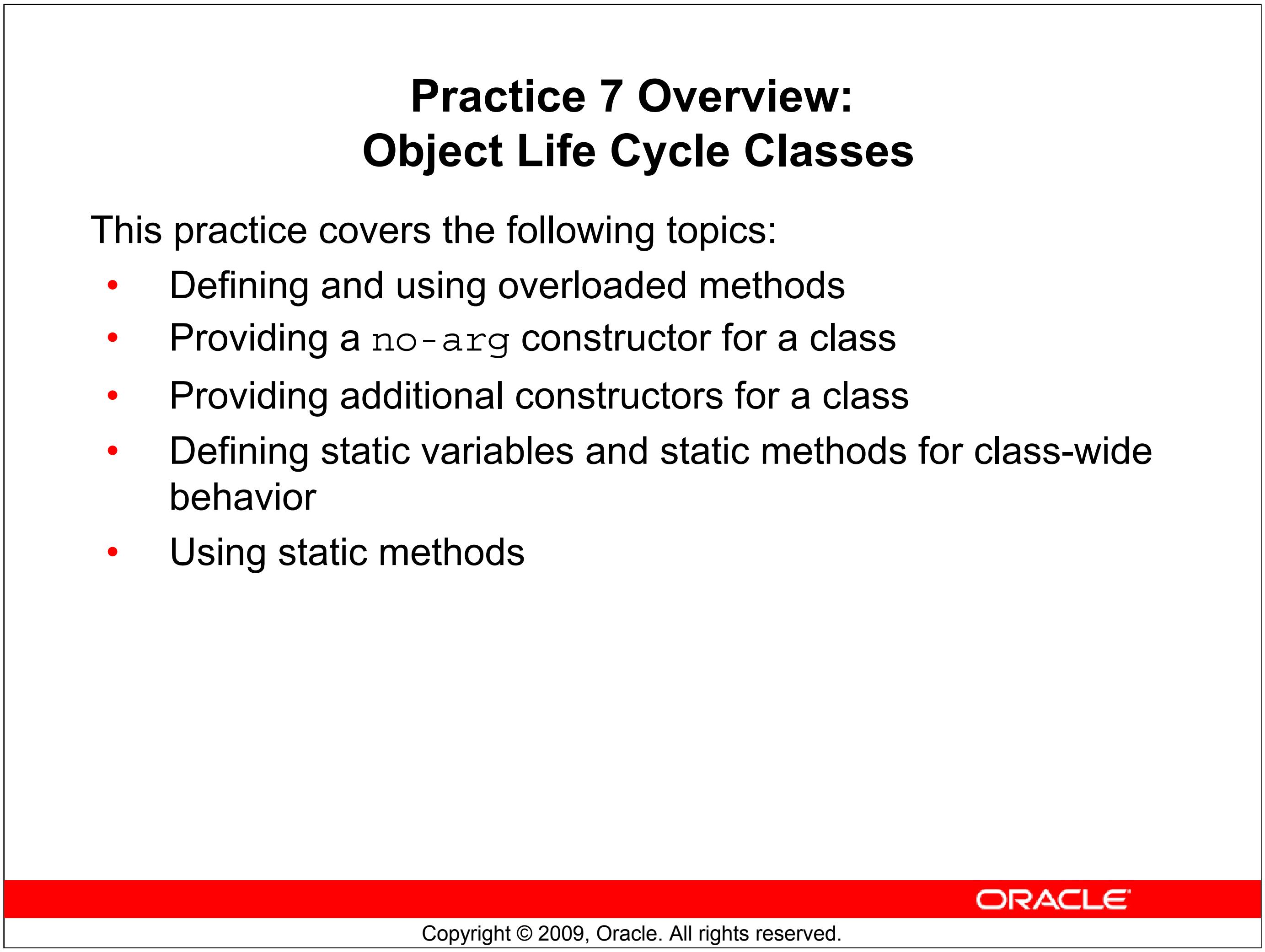
Two or more methods in a class can have the same name, provided that they have different signatures. This is known as *overloading*.

A variable can be defined at the class level as well as at the instance level. A class variable is common to all instances of the class. Similarly a class method belongs to a class and is shared by all instances of the class.

A constructor is a special method that is called by the run-time system when an object is created. A constructor is used to initialize a class' instance variables and to enable users of the class to specify the initial state of the object.

An inner class is a class that is defined within the scope of another class.You define inner classes because they functionally support the outer class or because they make sense in the context of the outer class. An anonymous inner class is an inner class without a name. It combines declaring an inner class and creating an instance of the class in one step. Inner classes are often used as listener classes for GUI components such as buttons.

`Calendar` is an abstract base class for manipulating `Date` objects.



Practice 7 Overview: Object Life Cycle Classes

This practice covers the following topics:

- Defining and using overloaded methods
- Providing a `no-arg` constructor for a class
- Providing additional constructors for a class
- Defining static variables and static methods for class-wide behavior
- Using static methods

Practice 7 Overview: Object Life Cycle Classes

Goal

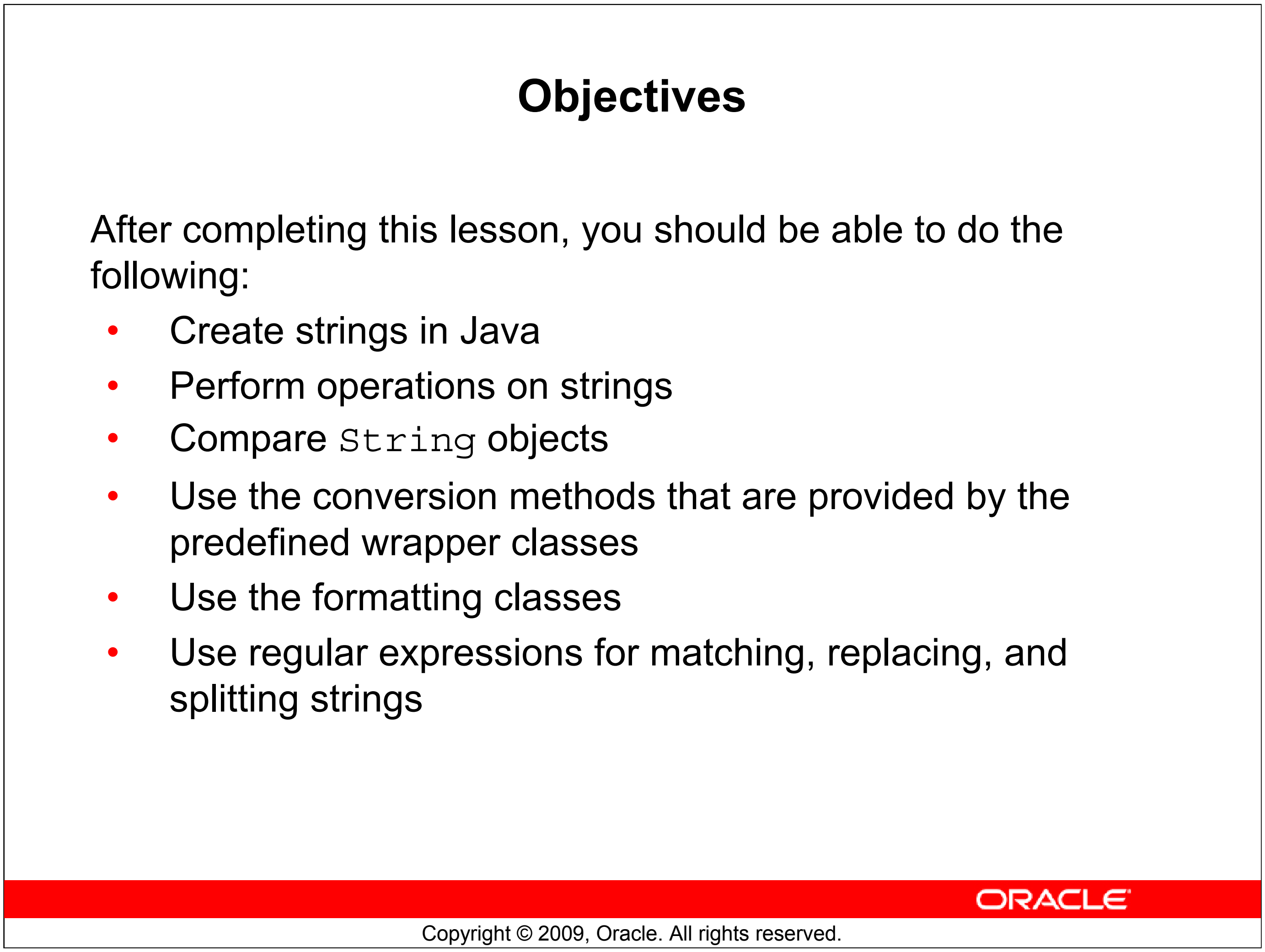
The goal of this practice is to provide experience with creating and using constructors, class-wide methods, and attributes. You also use an existing `DataMan` class to provide a data-access layer for finding customers and products in the `OrderEntry` application. Part of the practice is designed to help you understand method overloading by creating more than one constructor and/or method with the same name in the same class.

Note: If you have successfully completed the previous practice, continue using the same directory and files. If the compilation from the previous practice was unsuccessful and you want to move on to this practice, change to the `les07` directory, load the `OrderEntryApplicationLes07` application, and continue with this practice.

Viewing the model: To view the course application model up to this practice, expand `OrderEntryApplicationLes07 application – OrderEntryProjectLes07 - Application Sources – oe`, and double-click the `UML Class Diagram1` entry. This diagram displays all the classes created up to this point in the course.

8

Using Strings



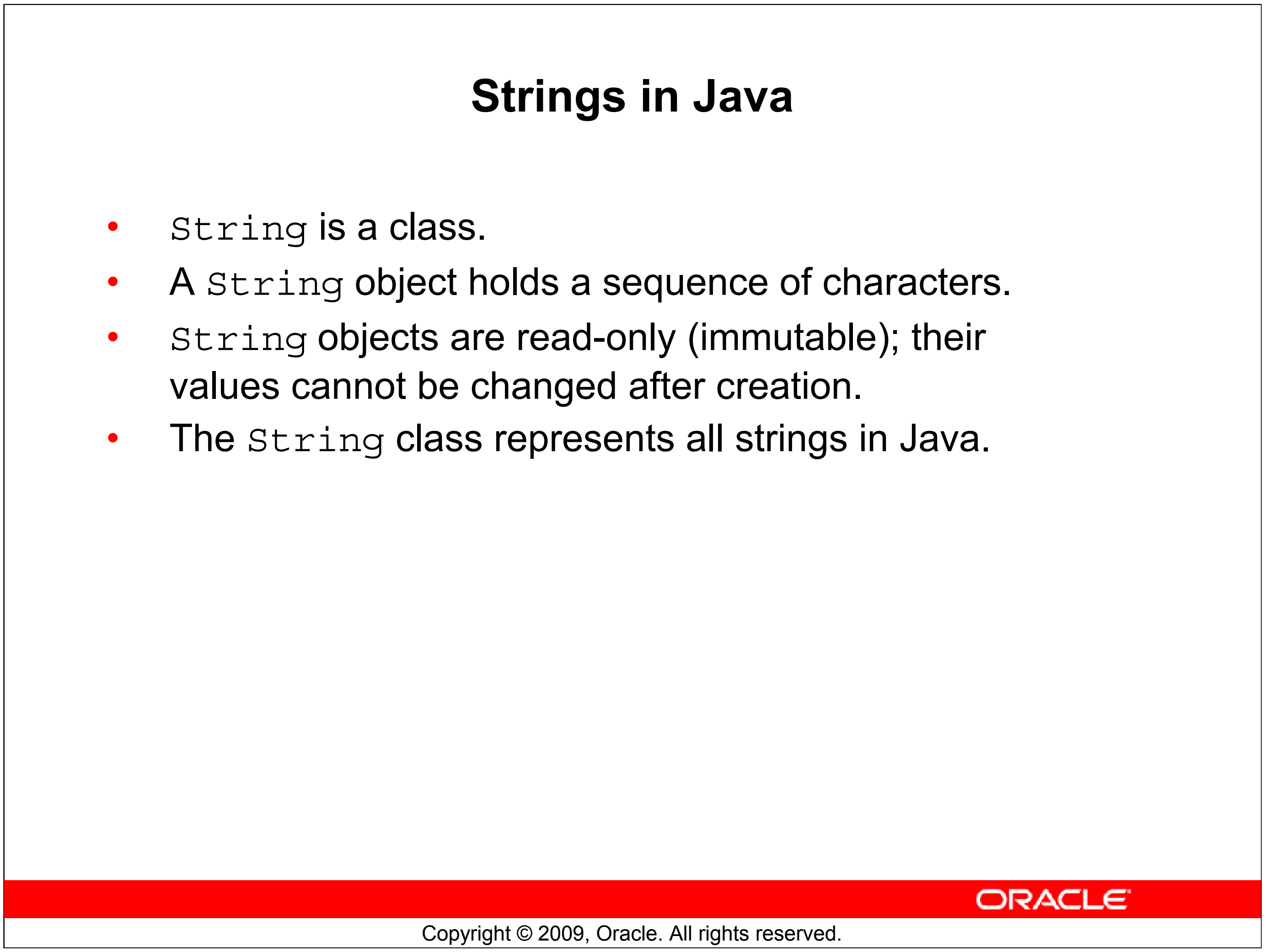
Objectives

After completing this lesson, you should be able to do the following:

- Create strings in Java
- Perform operations on strings
- Compare `String` objects
- Use the conversion methods that are provided by the predefined wrapper classes
- Use the formatting classes
- Use regular expressions for matching, replacing, and splitting strings

Lesson Objectives

This lesson shows you how to use the Java `String` class and `StringBuffer` class to manipulate character strings. You also use the Java wrapper classes to convert primitive types into other formats. The lesson also provides an introduction to Java I/O. It introduces the class hierarchies for the byte and character I/O streams, thus illustrating how to choose the appropriate class for a particular I/O circumstance.



Strings in Java

- String is a class.
- A String object holds a sequence of characters.
- String objects are read-only (immutable); their values cannot be changed after creation.
- The String class represents all strings in Java.

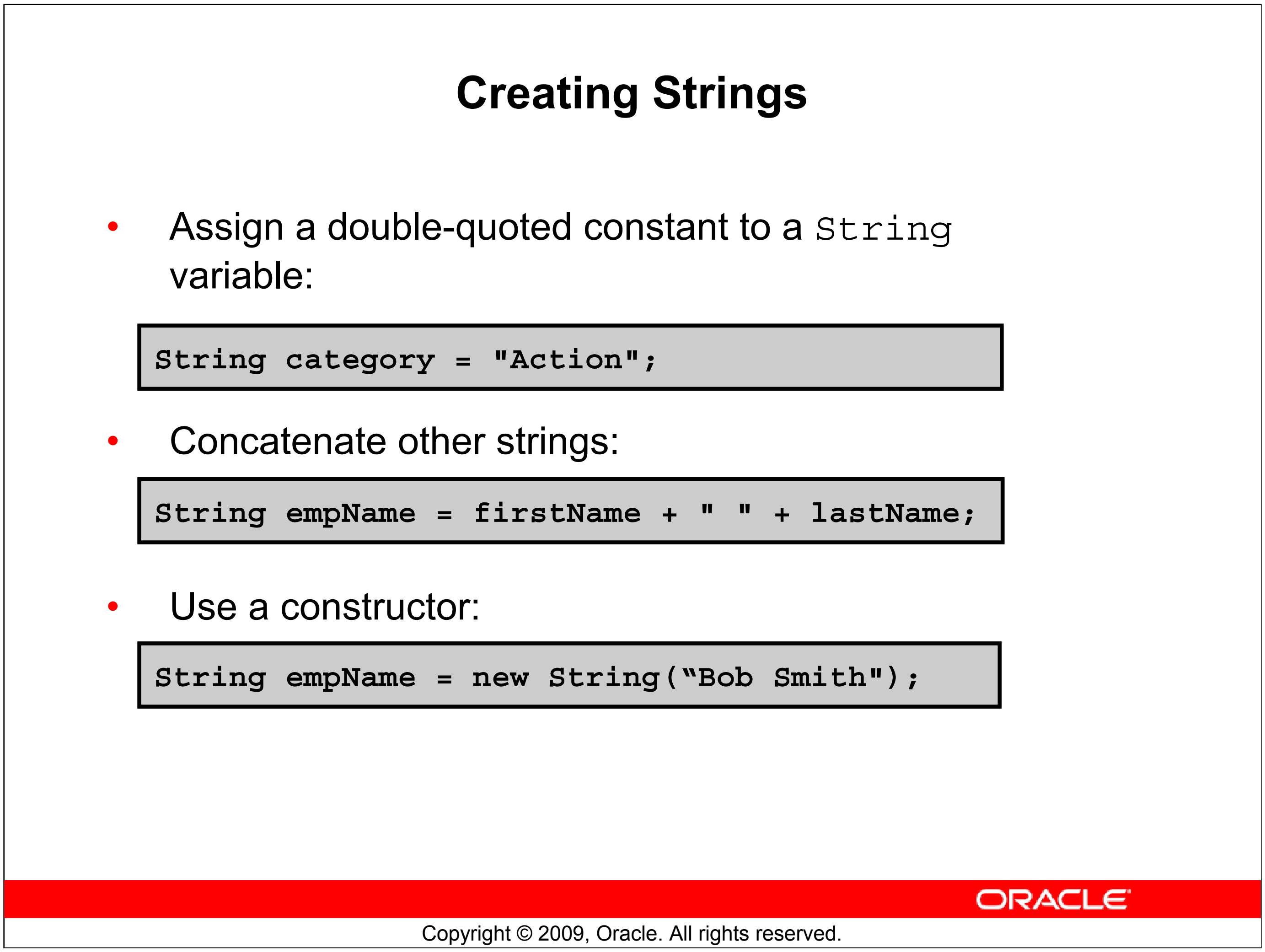
Strings in Java

As with most programming languages, strings are used extensively throughout Java. For this reason, the Java application programming interface (API) provides a String class to help you work with strings of character data. Literal string values in Java source code are turned into String objects by the compiler. The String objects can be used directly, passed as arguments to methods, or assigned to String object reference variables:

```
System.out.println( "Hello World..." );
String str = "Action";
```

Strings Cannot Be Modified

The String class represents an immutable string. This means that after creating a String object, you cannot change it. If you want to modify the contents of the string, you must use the StringBuffer class.



Creating Strings

- Assign a double-quoted constant to a `String` variable:

```
String category = "Action";
```

- Concatenate other strings:

```
String empName = firstName + " " + lastName;
```

- Use a constructor:

```
String empName = new String(“Bob Smith");
```

Creating Strings

The easiest way to create a string is from a double-quoted constant, as shown in the first example in the slide. You can use the + operator to concatenate two `String` objects. This is explained in more detail in the following slides.

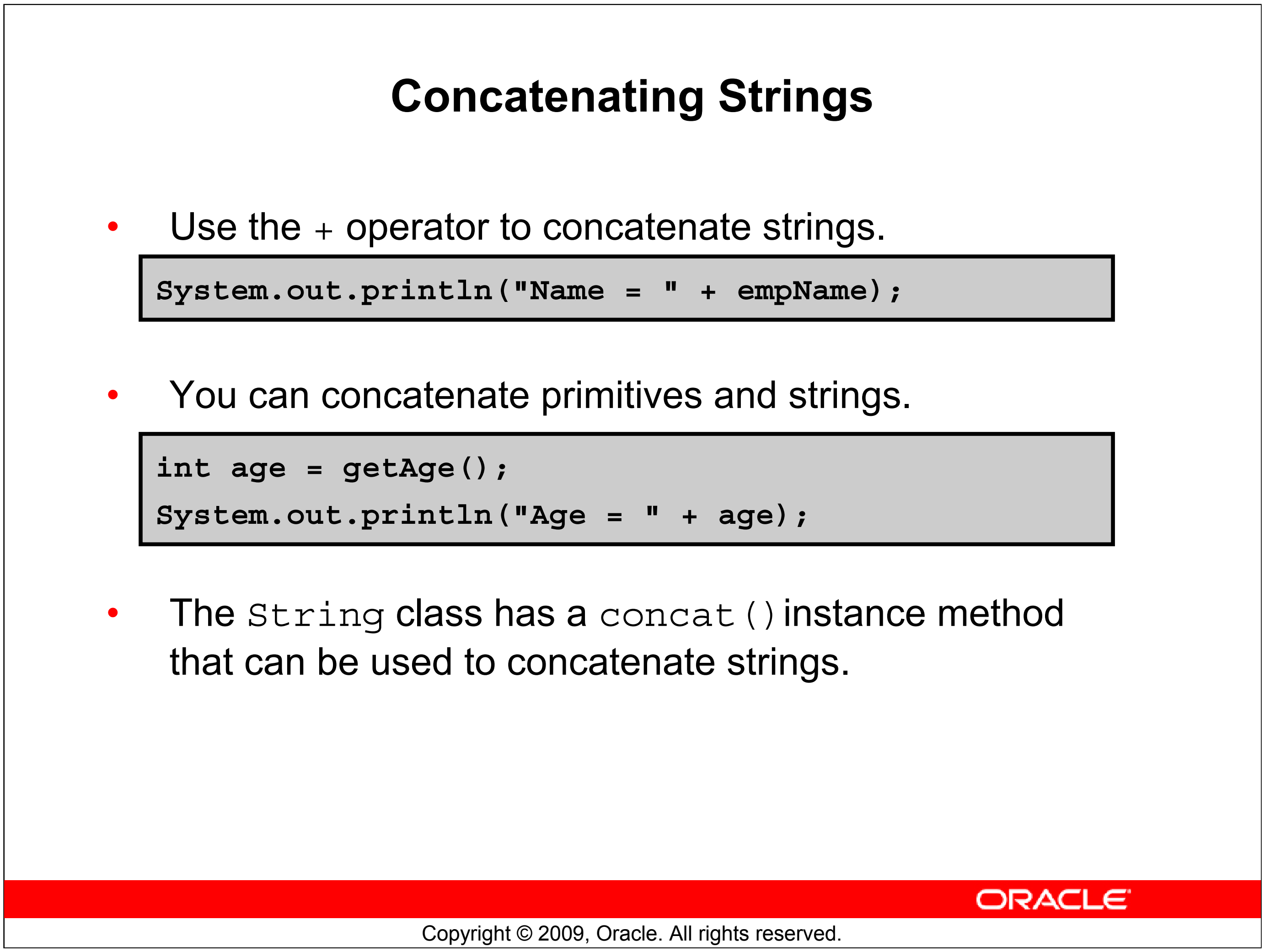
The `String` class provides several constructors. The following are some of the more useful ones:

- `String()` creates an empty string with the value `""`.
- `String(String str)` creates a copy of the specified `String` object, `str`.
- `String(char[] arr)` creates a string from the characters in a character array.

You can find a list of constructors in the online Java SE documentation for the `String` class, which is available in the Oracle JDeveloper online Help.

Using Strings in Your Code

The `String` class is part of the `java.lang` package. `java.lang` is automatically imported into all Java classes, so you do not need to add any import statements to your code to use `String` objects.



Concatenating Strings

- Use the + operator to concatenate strings.

```
System.out.println("Name = " + empName);
```

- You can concatenate primitives and strings.

```
int age = getAge();
System.out.println("Age = " + age);
```

- The `String` class has a `concat()` instance method that can be used to concatenate strings.

Concatenating Strings

Java uses + (the only overloaded operator in Java) for string concatenation. The `concat()` method in the `String` class is another way to concatenate strings.

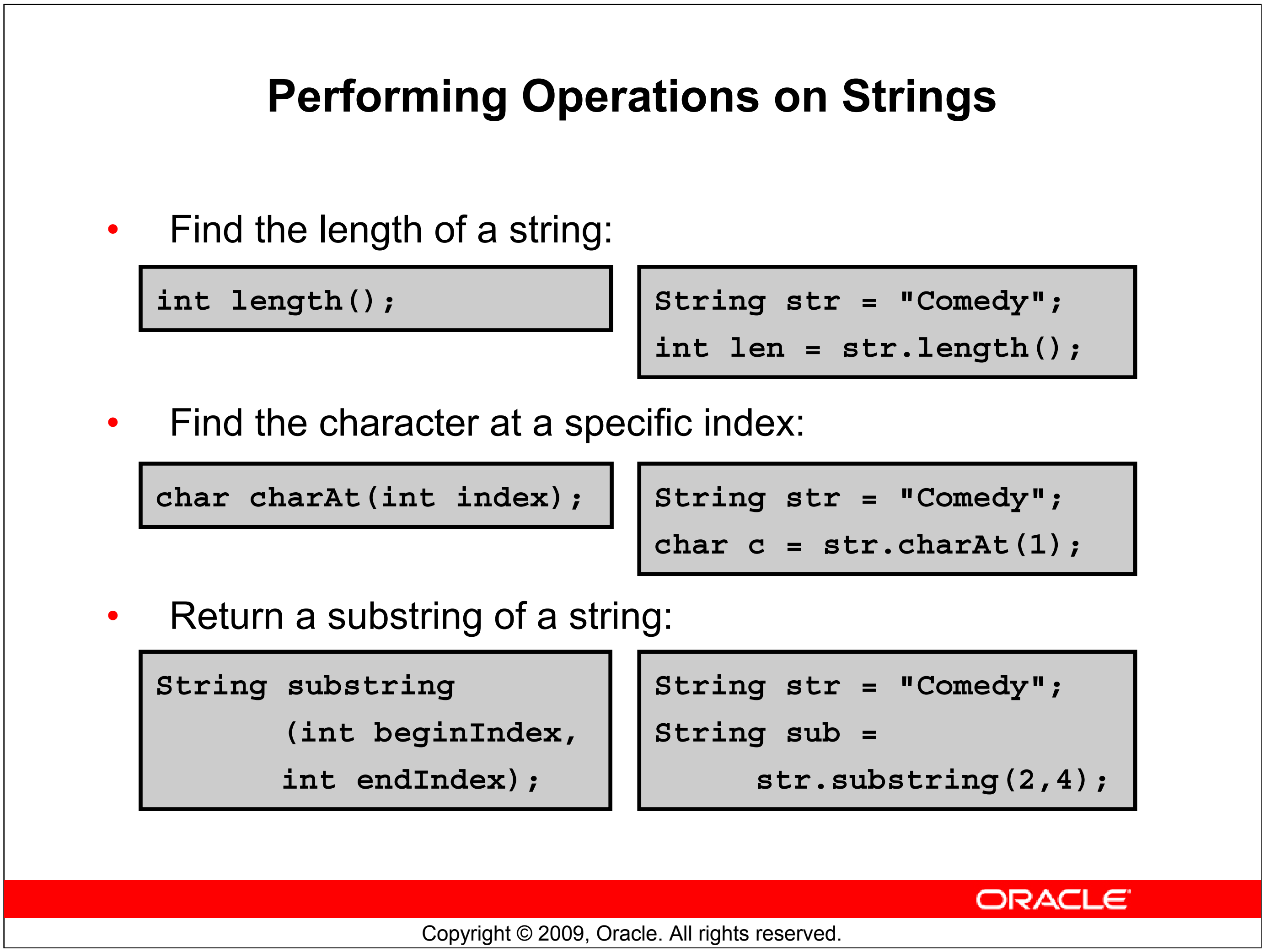
The following code produces equivalent strings:

```
String name = "Sam Wong";
String name = "Sam " + "Wong";
String name = "Sam ".concat("Wong");
```

The second example in the slide shows a primitive (in this case, an `int`) being concatenated with a `String`; the primitive is implicitly converted to a `String`.

Literal strings cannot span lines in Java source files, but you can concatenate lines to produce the same effect:

```
String song =
  "What good is sitting alone in your room" + "\n" +
  "Come hear the music play" + "\n" +
  "Life is a cabaret old chum" + "\n" +
  "Come to the cabaret" + "\n";
```



Performing Operations on Strings

- Find the length of a string:

```
int length();
```

```
String str = "Comedy";
int len = str.length();
```

- Find the character at a specific index:

```
char charAt(int index);
```

```
String str = "Comedy";
char c = str.charAt(1);
```

- Return a substring of a string:

```
String substring
       (int beginIndex,
       int endIndex);
```

```
String str = "Comedy";
String sub =
     str.substring(2,4);
```

Performing Operations on Strings

Find the Length of a String

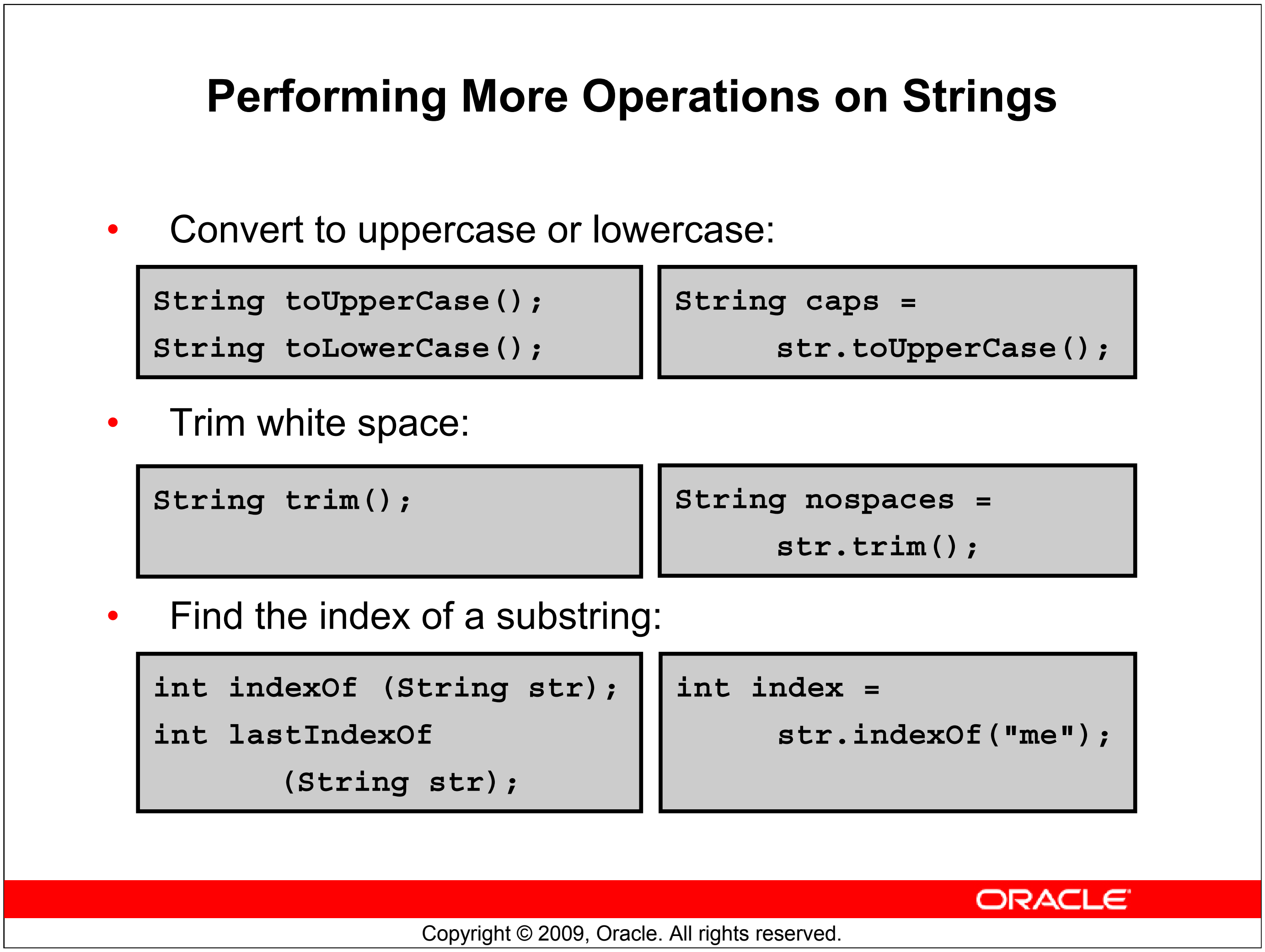
The `length()` method returns the number of characters in a string. In the example in the slide, `length` is set to `6`.

Return the Character at a Specified Index

The `charAt()` method returns the character at the specified index (indexing starts at `0`). In the slide example, `c` is set to the letter `o`.

Return a Substring of a String

The `substring()` method returns a specified substring of the string (starting at `beginIndex`, ending at `endIndex - 1`). In the slide example, `sub` is set to “`me`.”



Performing More Operations on Strings

- Convert to uppercase or lowercase:

```
String toUpperCase();
String toLowerCase();
```

```
String caps =
       str.toUpperCase();
```

- Trim white space:

```
String trim();
```

```
String nospaces =
       str.trim();
```

- Find the index of a substring:

```
int indexOf (String str);
int lastIndexOf
        (String str);
```

```
int index =
       str.indexOf("me");
```

Performing More Operations on Strings

Convert to Uppercase or Lowercase

The `toUpperCase()` method returns a new string containing an uppercase version of the original string. Similarly, the `toLowerCase()` method returns a new string containing a lowercase version of the original string.

Trim Whitespace

The `trim()` method returns a new string containing a copy of the old string with white space removed from both ends.

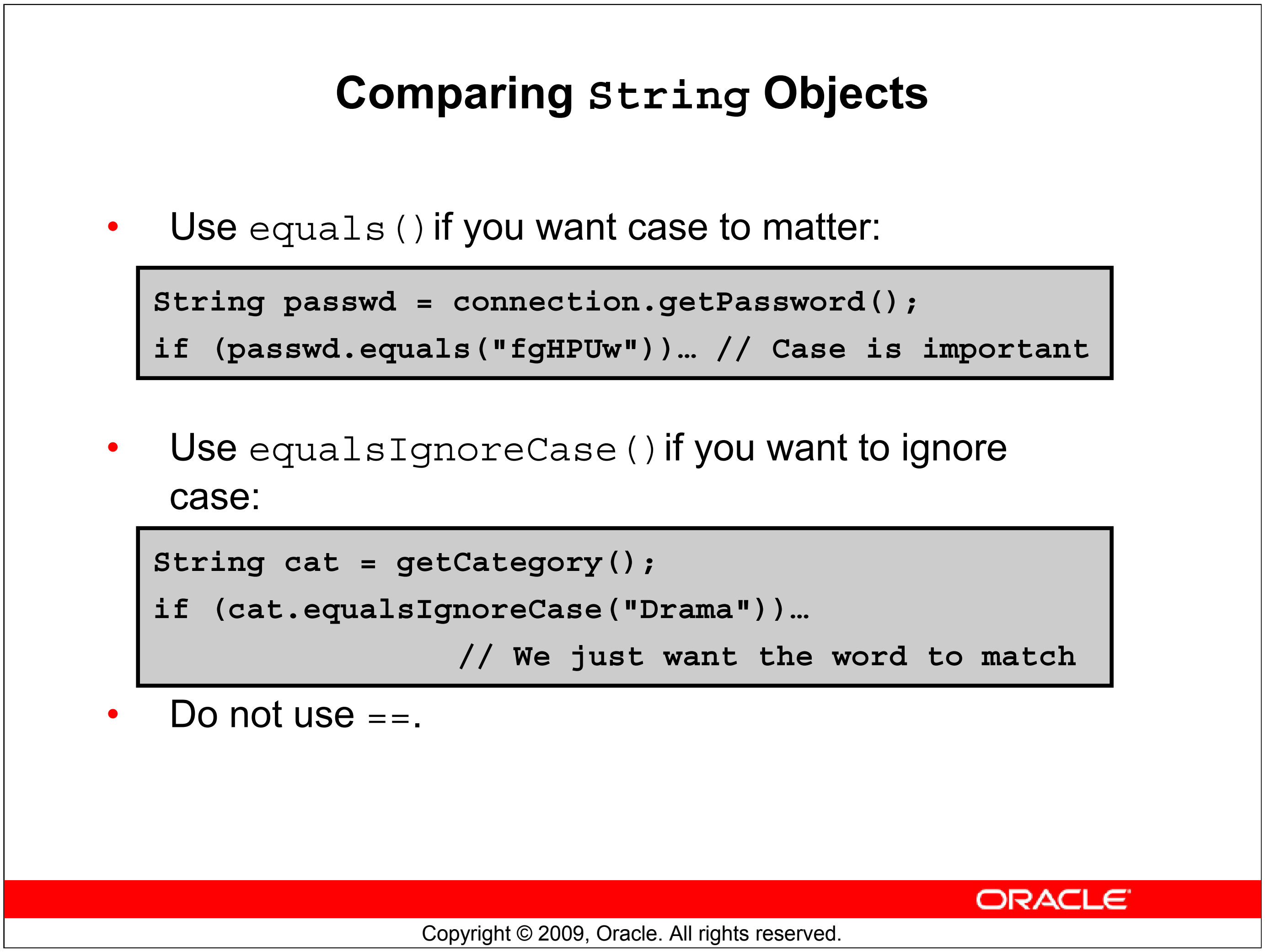
Find the Index of a Substring

There are two accessors that return the position of a specific character or string:

`indexOf` and `lastIndexOf`

The `indexOf()` method returns the index of a specified substring. It returns a zero-based position of a substring within `str` or `–1` if not found. The `lastIndexOf()` method returns the index of the last occurrence of a specified substring.

Note: There are multiple versions of both these methods. Check the online Java SE documentation for the `String` class for details of each one.



Comparing String Objects

- Use equals() if you want case to matter:

```
String passwd = connection.getPassword();
if (passwd.equals("fgHPUw"))… // Case is important
```

- Use equalsIgnoreCase() if you want to ignore case:

```
String cat = getCategory();
if (cat.equalsIgnoreCase("Drama"))…
                  // We just want the word to match
```

- Do not use ==.

Comparing String Objects

boolean equals(Object anObj) returns true if the specified strings contain the same text. It does not return true if anObj is null or is not a String object.

boolean equalsIgnoreCase(String s2) is similar to equals(), except that it ignores case.

Do not use the == operator to compare String objects or to compare any types of objects. With objects, == returns true only if the two references point to the same object.

Interning Strings

The String class has an intern method, which is used to set up pools of strings. Suppose that you have a string s such that:

```
 s = s.intern();
```

Then the contents of s are compared against an internal pool of unique strings and added if this particular string's contents are not already in the pool. A reference to the unique pool String is returned.

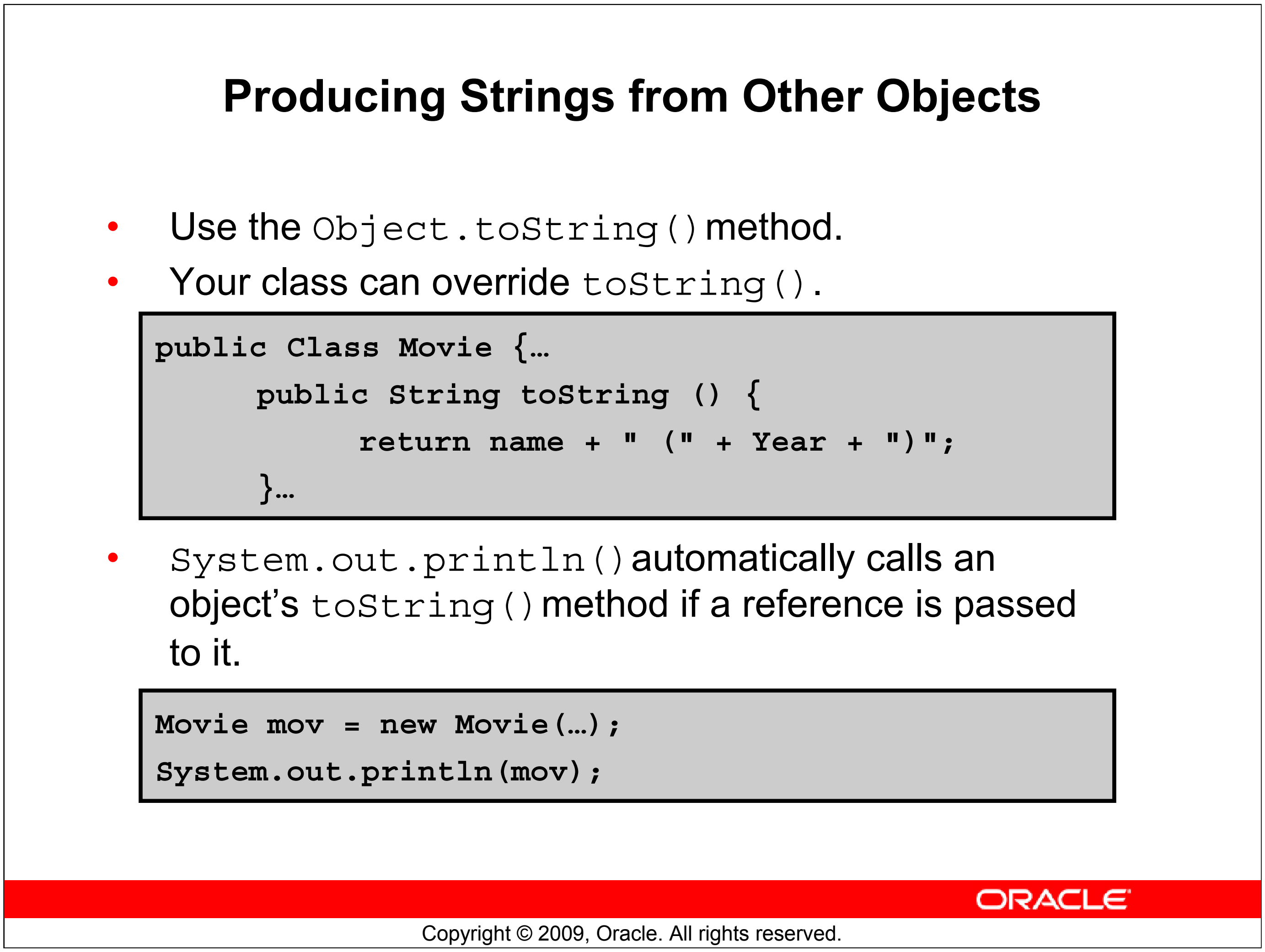
Strings that have been interned can be compared to each other using == because there is a unique string object for any given sequence of characters representing one of the strings.

Comparing `String` Objects (continued)

String literals and string-valued constant expressions are always interned. Thus, the following is always true:

```
"abc" == "abc"
```

Interning has some initial setup costs. But after it is implemented, it supports very efficient equality checking between strings because it compares pooled objects instead of character sequences. Interning offers a performance advantage when the same strings are used repeatedly.



Producing Strings from Other Objects

- Use the `Object.toString()` method.
- Your class can override `toString()`.

```
public Class Movie {…
     public String toString () {
          return name + " (" + Year + ")";
     }…
```

- `System.out.println()` automatically calls an object’s `toString()` method if a reference is passed to it.

```
Movie mov = new Movie(…);
System.out.println(mov);
```

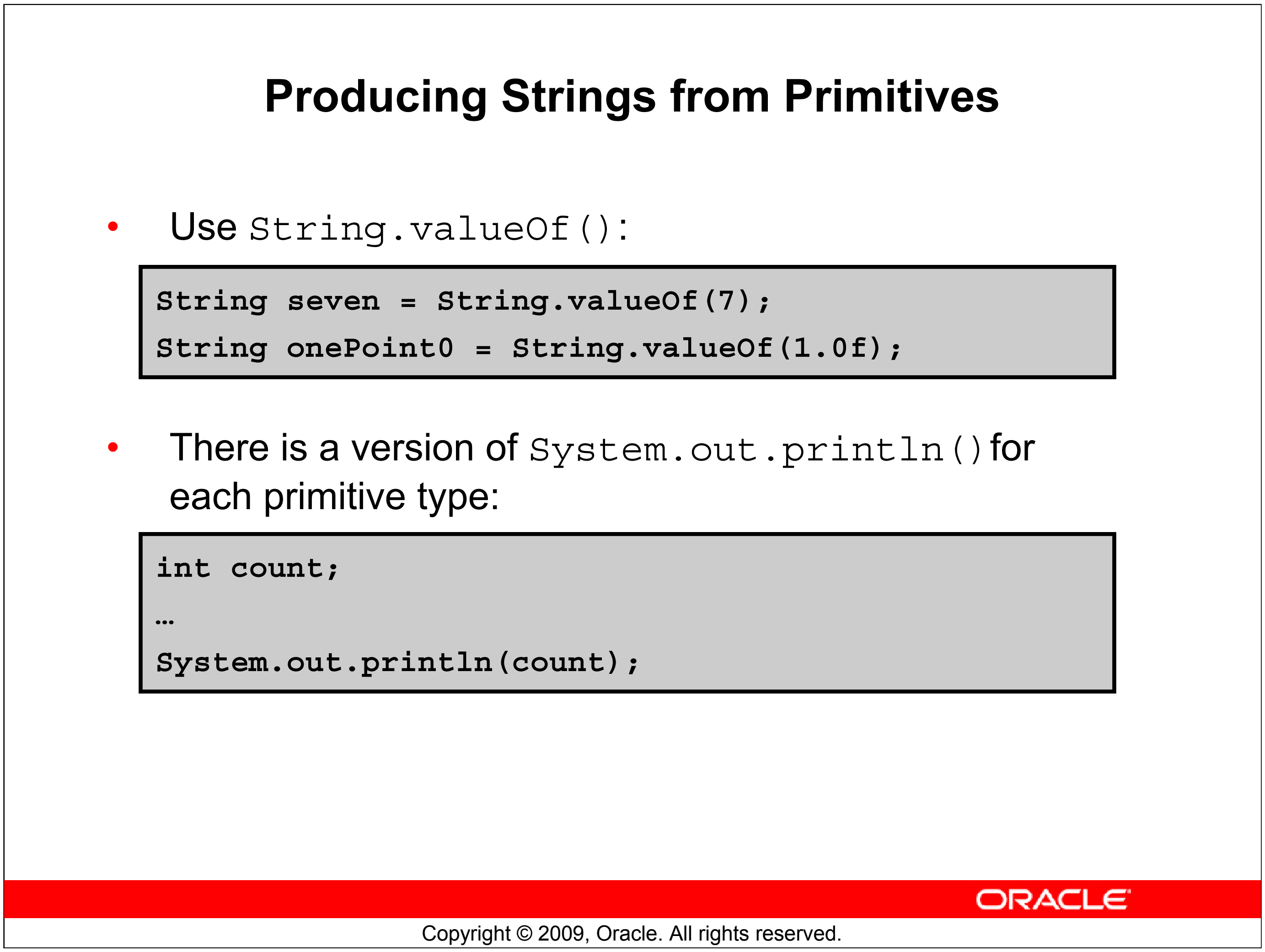
Producing Strings from Other Objects

The `toString` method for a class object returns a string consisting of the name of the class of which the object is an instance, the “@” character, and the unsigned hexadecimal representation of the hash code of the object. The `toString()` method is invoked whenever you use an object reference in a string concatenation expression, or whenever you pass it to `System.out.println`. In the example in the slide, the `Movie` class provides a `toString()` method that prints the name and year of the movie.

What Happens if a Class Does Not Provide a `toString()` Method?

If a class does not provide its own `toString()` method, it inherits one from the `Object` class. The string that is produced by `Object.toString()` is not very user-friendly; it consists of both the name of the class of which the object is an instance and a hexadecimal number representing a hash code. The equivalent code that is inherited from the `Object.toString()` method is:

```
public String toString() {
 return  getClass().getName() +
             '@' + Integer.toHexString(hashCode());
}
```



Producing Strings from Primitives

- Use `String.valueOf()`:

```
String seven = String.valueOf(7);
String onePoint0 = String.valueOf(1.0f);
```

- There is a version of `System.out.println()` for each primitive type:

```
int count;
…
System.out.println(count);
```

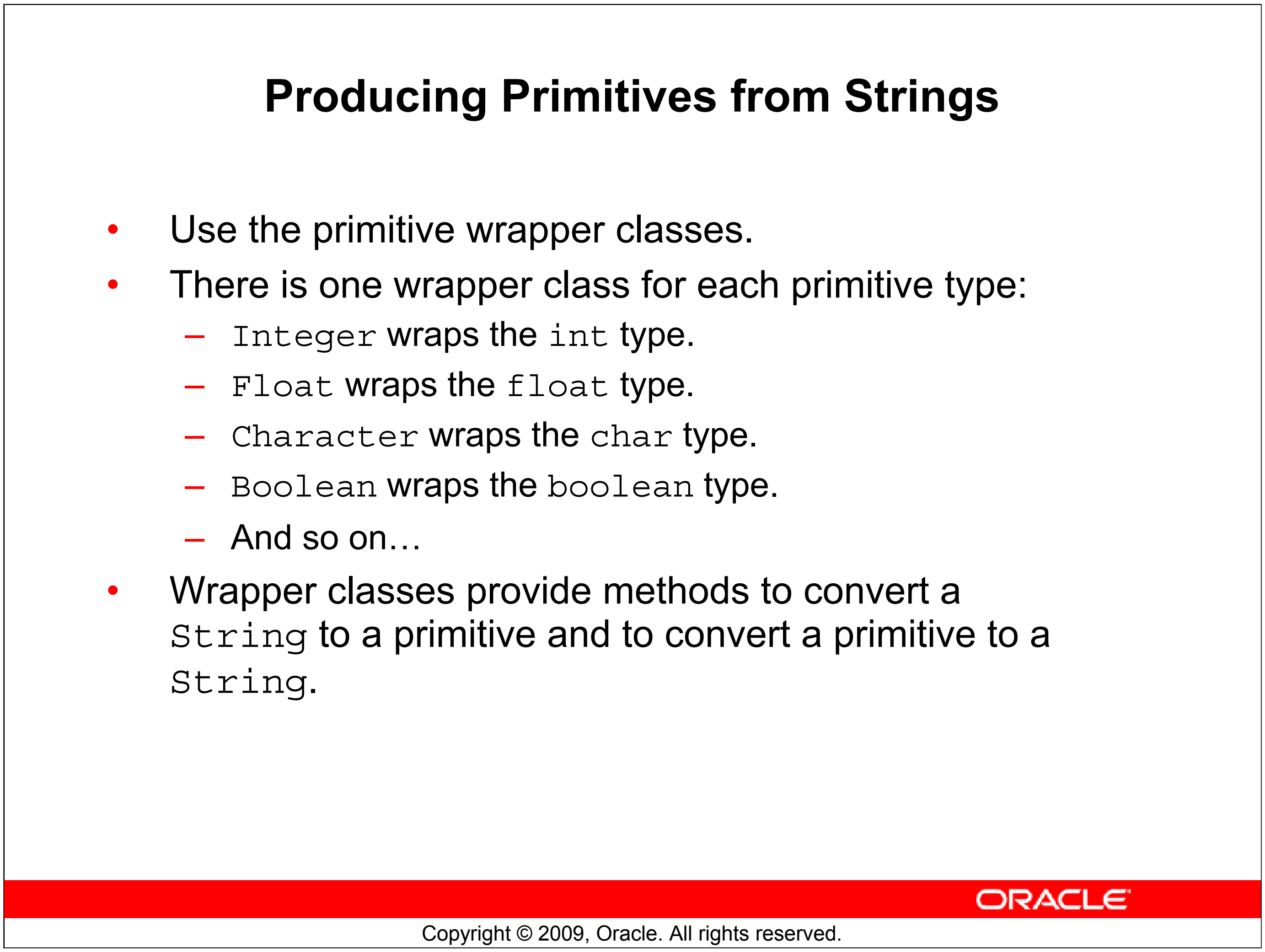
Producing Strings from Primitives

The `String` class provides a static method `valueOf()` that returns a string representation of a primitive type. However, a `String` does not have a `valueOf()` method for the `byte` or `short` types. There is a version of `valueOf()` for each remaining primitive type. The example in the slide uses two versions:

```
String valueOf(int i)
String valueOf(float f)
```

Printing Primitives

When a primitive is concatenated with a string, it is automatically converted to a string by calling `String.valueOf()`. When a primitive is passed to `System.out.println()` on its own, the appropriate version of `println()` is called; there is a version for each primitive type.



Producing Primitives from Strings

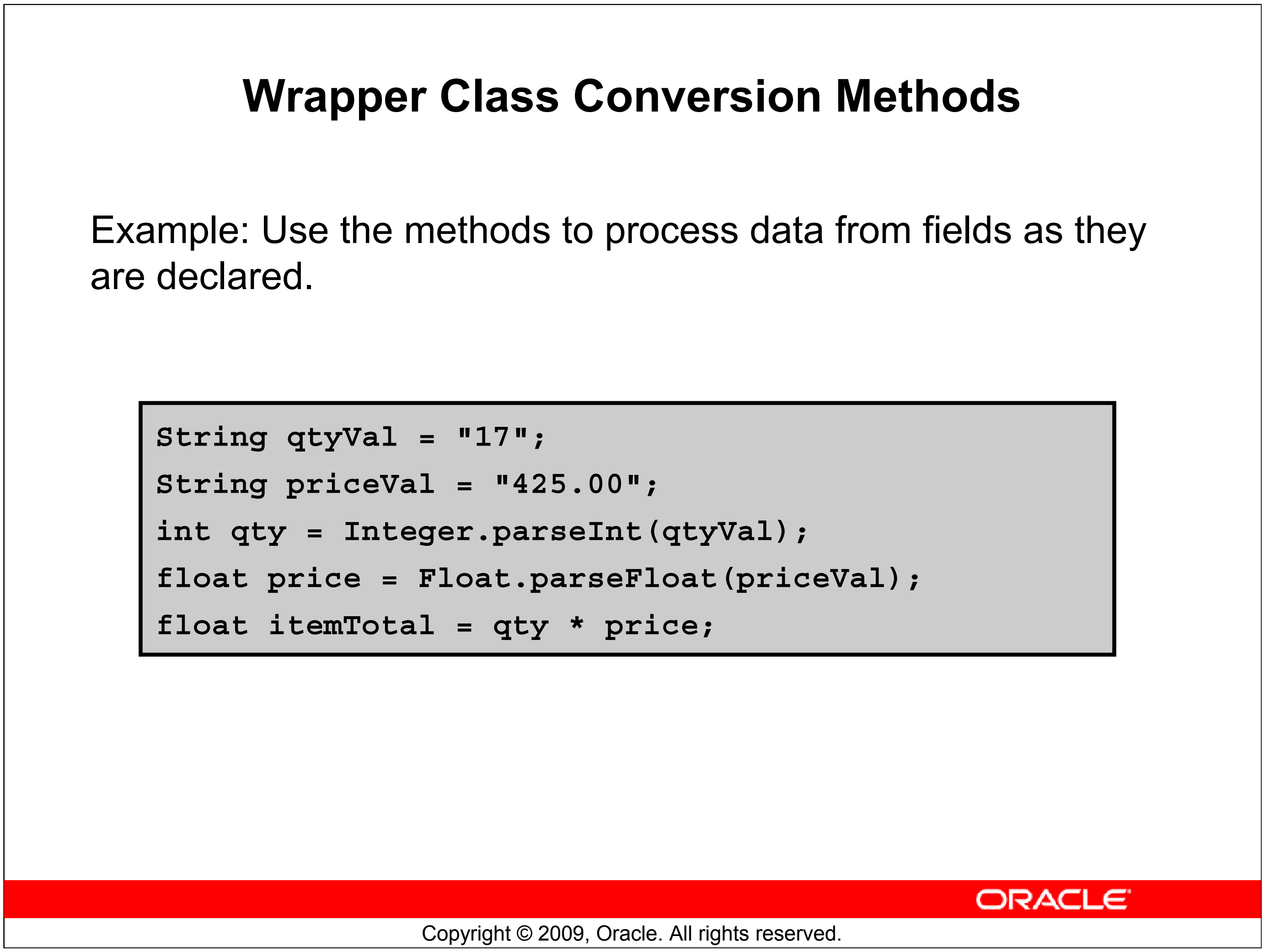
- Use the primitive wrapper classes.
- There is one wrapper class for each primitive type:
  - `Integer` wraps the `int` type.
  - `Float` wraps the `float` type.
  - `Character` wraps the `char` type.
  - `Boolean` wraps the `boolean` type.
  - And so on…
- Wrapper classes provide methods to convert a `String` to a primitive and to convert a primitive to a `String`.

Primitive Wrapper Classes

For each primitive type, Java provides a corresponding wrapper class that enables a primitive to be handled like an object. Each wrapper class provides a static method to convert a `String` to the corresponding primitive type. The following slides show how to use these conversion methods.

Other Uses of Wrapper Classes

- Wrapper classes are useful if you need to treat a primitive type as an object. For example, you cannot store primitive types in a `Vector`, which holds objects. Thus, you convert an `int` to an `Integer` object by using the `Integer(int)` constructor.
- Wrapper classes provide conversion methods that are related to primitive data types. For example, the `Integer` class has a number of methods, including `Integer.parseInt()`, that convert a `String` containing digits to an `int`.
- Wrapper classes also provide variables that are related to the type.



Wrapper Class Conversion Methods

Example: Use the methods to process data from fields as they are declared.

```
String qtyVal = "17";
String priceVal = "425.00";
int qty = Integer.parseInt(qtyVal);
float price = Float.parseFloat(priceVal);
float itemTotal = qty * price;
```

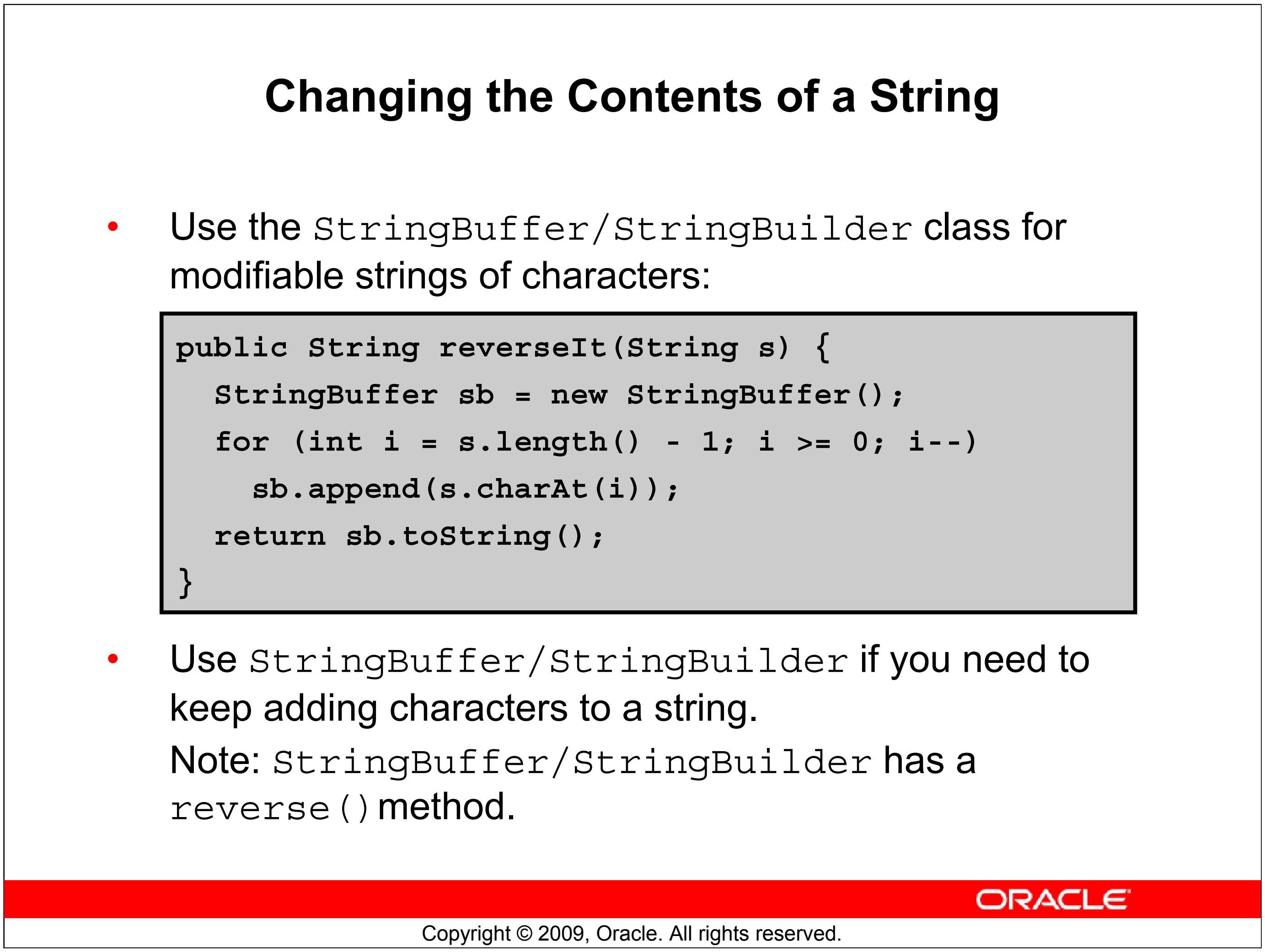
Wrapper Class Conversion Methods

The example in the slide shows hard-coded `String` variables containing numeric values. These values are used to perform a calculation. The `qtyVal` string is being converted into a `int` by using the `Integer.parseInt()` method, and the `priceVal` is being converted into a `float` by using the `Float.ParseFloat()` method.

Note that the conversion methods were called without instantiating either an `Integer` object or a `Float` object. The wrapper classes serve as a home for their conversion methods and need not be instantiated when the methods are called.

What Happens if Strings Contain a Value That Cannot Be Converted?

If the string contains a noninteger value in the first field, `Integer.parseInt()` fails and throws an exception. To handle such a case, you must add code to catch and handle this exception and to catch and handle a similar exception that is thrown by `Float.parseFloat()`.



Changing the Contents of a String

- Use the `StringBuffer/StringBuilder` class for modifiable strings of characters:

```
public String reverseIt(String s) {
  StringBuffer sb = new StringBuffer();
  for (int i = s.length() - 1; i >= 0; i--)
    sb.append(s.charAt(i));
  return sb.toString();
}
```

- Use `StringBuffer/StringBuilder` if you need to keep adding characters to a string.

  Note: `StringBuffer/StringBuilder` has a `reverse()` method.

Introducing the `StringBuffer/StringBuilder` Class

The `StringBuffer/StringBuilder` class is an alternative to the `String` class. In general, a `StringBuffer/StringBuilder` can be used whenever a string is used. `StringBuffer/StringBuilder` is more flexible than `String`. You can add, insert, or append new contents into a string buffer, whereas the value of a `String` object is fixed once the string is created.

The `StringBuilder` class, introduced in JDK 5.0, is similar to `StringBuffer` except that the update methods in `StringBuffer` are synchronized. Use `StringBuffer` if it may be accessed by multiple tasks concurrently. Using `StringBuilder` is more efficient if it is accessed by a single task. The constructors and methods in `StringBuffer` and `StringBuilder` are almost the same. The examples here use `StringBuffer`. You may replace `StringBuffer` with `StringBuilder`. The program can compile and run without any other changes.

The following example creates three new `String` objects and copies all the characters each time a new `String` is created:

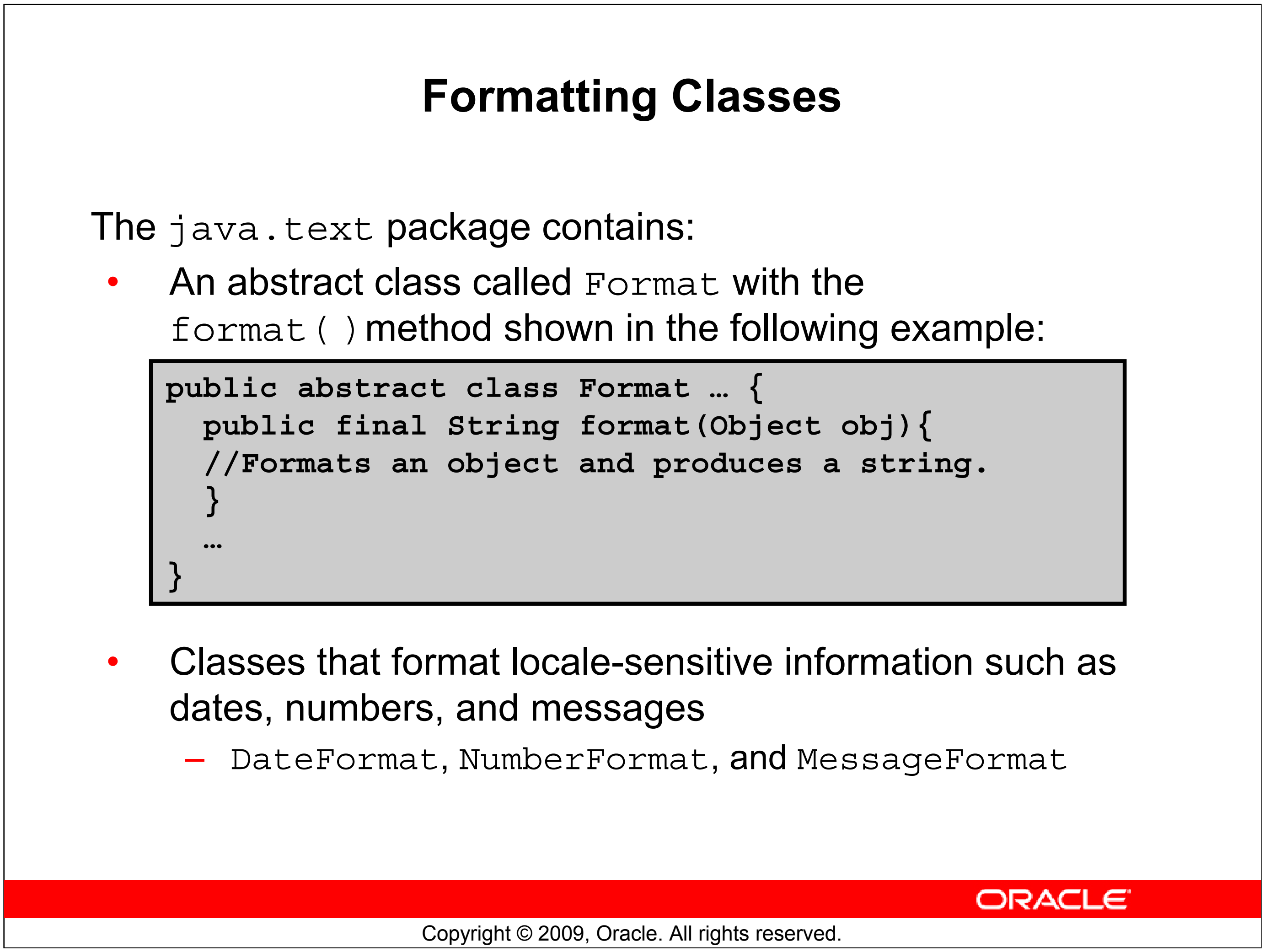
```
String quote = "Fasten your seatbelts, ";
quote = quote + "it’s going to be a bumpy ride.";
```

Introducing the StringBuffer/StringBuilder Class (continued)

It is more efficient to preallocate the amount of space that is required by using the StringBuffer constructor and its append() method as follows:

```
StringBuffer quote = new StringBuffer(60); // alloc
60 chars
 quote.append("Fasten your seatbelts, ");
 quote.append(" it's going to be a bumpy ride. ");
```

StringBuffer also provides several overloaded insert() methods for inserting various types of data at a particular location in StringBuffer.



Formatting Classes

The `java.text` package contains:

- An abstract class called `Format` with the `format( )` method shown in the following example:

```
public abstract class Format … {
  public final String format(Object obj){
  //Formats an object and produces a string.
  }
  …
}
```

- Classes that format locale-sensitive information such as dates, numbers, and messages
  - `DateFormat`, `NumberFormat`, and `MessageFormat`

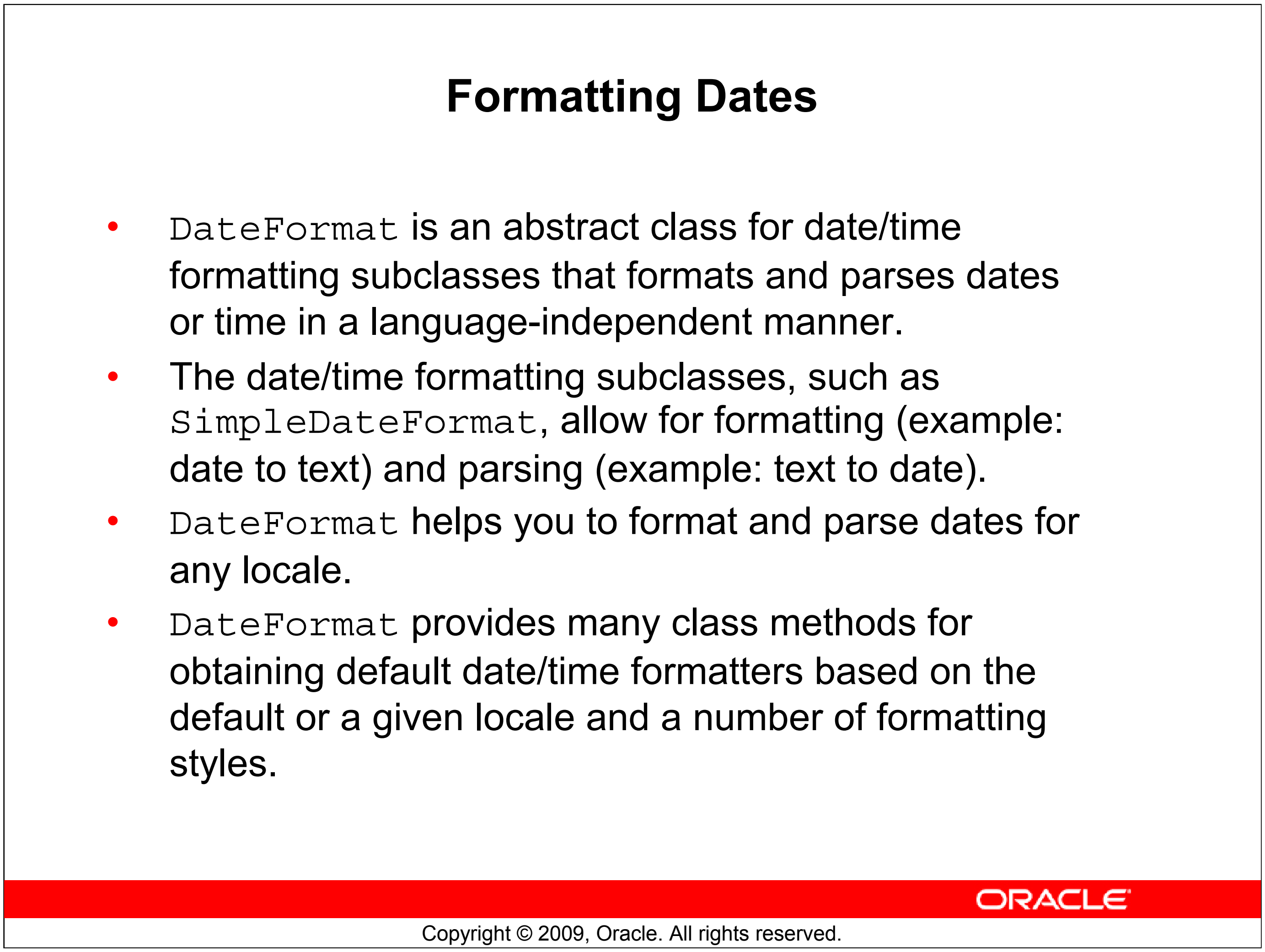
Formatting Classes

These formatting methods define the programming interface for formatting locale-sensitive objects into strings and for parsing strings back into objects (the `parseObject(…)` method). Any string that is formatted by the `format(…)` method is guaranteed to be parseable by `parseObject`.

If there is no match when parsing, `parseObject(String)` throws a `ParseException`, and `parseObject(String, ParsePosition)` leaves the `ParsePosition` index member unchanged and returns `null`.

Here is a hierarchy of the `Format` class:

`Format`
`DateFormat`
`SimpleDateFormat`
	`Format`
	`MessageFormat`
		`Format`
		`NumberFormat`
		`ChoiceFormat`
		`DecimalFormat`



Formatting Dates

- `DateFormat` is an abstract class for date/time formatting subclasses that formats and parses dates or time in a language-independent manner.
- The date/time formatting subclasses, such as `SimpleDateFormat`, allow for formatting (example: date to text) and parsing (example: text to date).
- `DateFormat` helps you to format and parse dates for any locale.
- `DateFormat` provides many class methods for obtaining default date/time formatters based on the default or a given locale and a number of formatting styles.

Formatting Dates

You saw an example of using `SimpleDateFormat` in the previous lesson, when calculating a date with the `Calendar` class.

Example of formatting a date for a specified locale and pattern

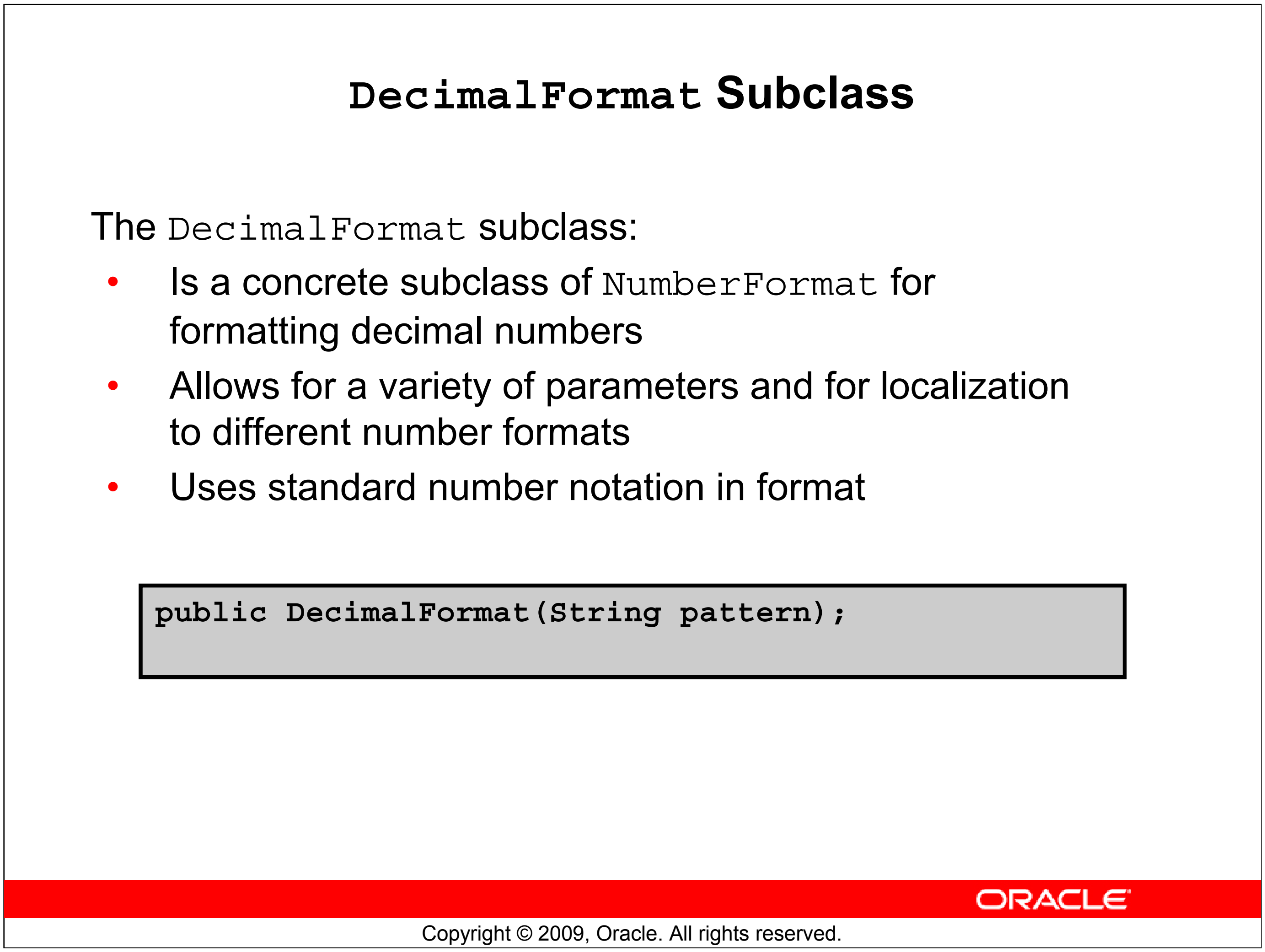
```
import java.text.*;
import java.util.Date;
import java.util.Locale;

public class DateFormLocale {
    public static void main(String[] args)
   {
     //Create a SimpleDateFormat object
```

Formatting Dates (continued)

```
        DateFormat df =
              DateFormat.getDateInstance(DateFormat.LONG,
                                              Locale.FRANCE);

          //Print out the date using supplied pattern
          System.out.println("A sample date looks like: " +
                               df.format(new Date("15-Aug-2007")));
      }
  }
```

The output is:

A sample date looks like: 15 août 2007



DecimalFormat Subclass

The DecimalFormat subclass:

- Is a concrete subclass of NumberFormat for formatting decimal numbers
- Allows for a variety of parameters and for localization to different number formats
- Uses standard number notation in format

```
public DecimalFormat(String pattern);
```

DecimalFormat Subclass

DecimalFormat is a concrete subclass of NumberFormat that formats decimal numbers. It has a variety of features designed to make it possible to parse and format numbers in any locale, including support for Western, Arabic, and Indic digits. It also supports different kinds of numbers, including integers (123), fixed-point numbers (123.4), scientific notation (1.23E4), percentages (12%), and currency amounts ($123). All of these can be localized.

A DecimalFormat comprises a *pattern* and a set of *symbols*. The pattern may be set directly using applyPattern(), or indirectly using the API methods.

Special characters are used in the parts of the pattern. For example, if you are using the decimal format constructor to create a format pattern for displaying decimal numbers, the structure of the pattern can be represented by the symbols in the following chart:

Symbol	Meaning
0	A digit
#	A digit; zero shows as absent
.	Placeholder for decimal separator
,	Placeholder for grouping separator
;	Separates formats
-	Default negative prefix
%	Multiply by 100 and show as percentage

DecimalFormat Subclass (continued)

unicode \u2030 Multiply by 1,000 and show as per mille

unicode \\u00A4 Currency sign; replaced by currency symbol; if doubled, replaced by international currency symbol

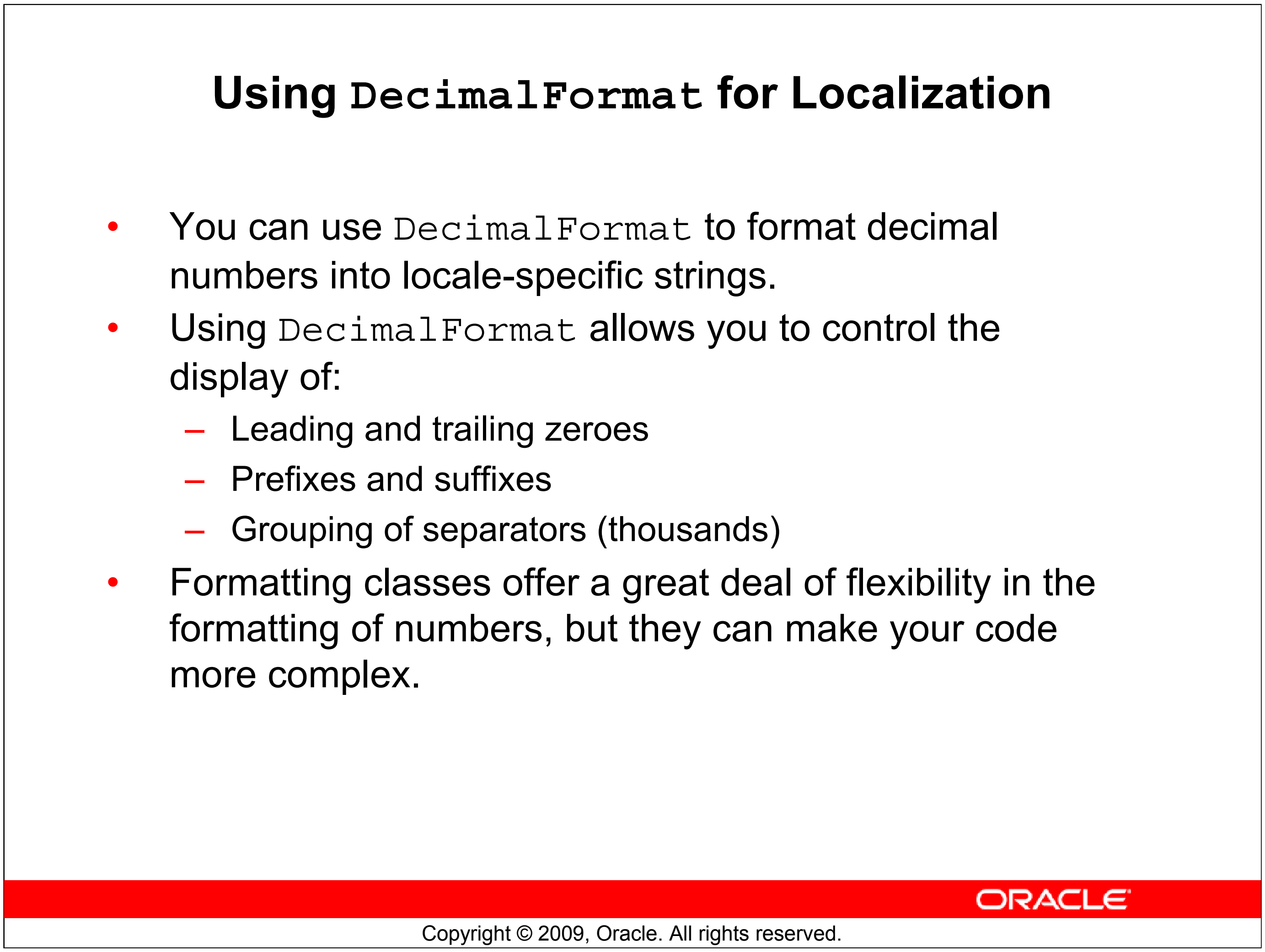
Simple Example of Using DecimalFormat

When carrying out a simple floating point (FP) operation such as a divide, the result is often a long fraction when you use the default conversion of a double to a String. The following example uses DecimalFormat to format the floating point value.

```
import java.text.DecimalFormat;

public class DecFormat {
    public static void main(String[] args) {

         //Create a string pattern
         String fmt = "0.00#";

         //Create an instance of the DecimalFormat class
   //with this pattern
         DecimalFormat df = new DecimalFormat(fmt);

        //Invoke the format method to create the formatted
  //string
        double q = 10.0 / 4.0;
        String str = df.format(q);
             System.out.println("q =  " + str);
}
   }
```

The output is

```
q =  2.50
```



Using `DecimalFormat` for Localization

- You can use `DecimalFormat` to format decimal numbers into locale-specific strings.
- Using `DecimalFormat` allows you to control the display of:
  - Leading and trailing zeroes
  - Prefixes and suffixes
  - Grouping of separators (thousands)
- Formatting classes offer a great deal of flexibility in the formatting of numbers, but they can make your code more complex.

Using `DecimalFormat` for Localization

Example of Locale-Sensitive Formatting

The example below uses the pattern `###,###.###` where the pound sign (#) denotes a digit, the comma is a placeholder for the grouping separator, and the period is the placeholder for the decimal separator. To create a `DecimalFormat` object for a specified locale, you instantiate a `NumberFormat` and then cast it to a `DecimalFormat`.

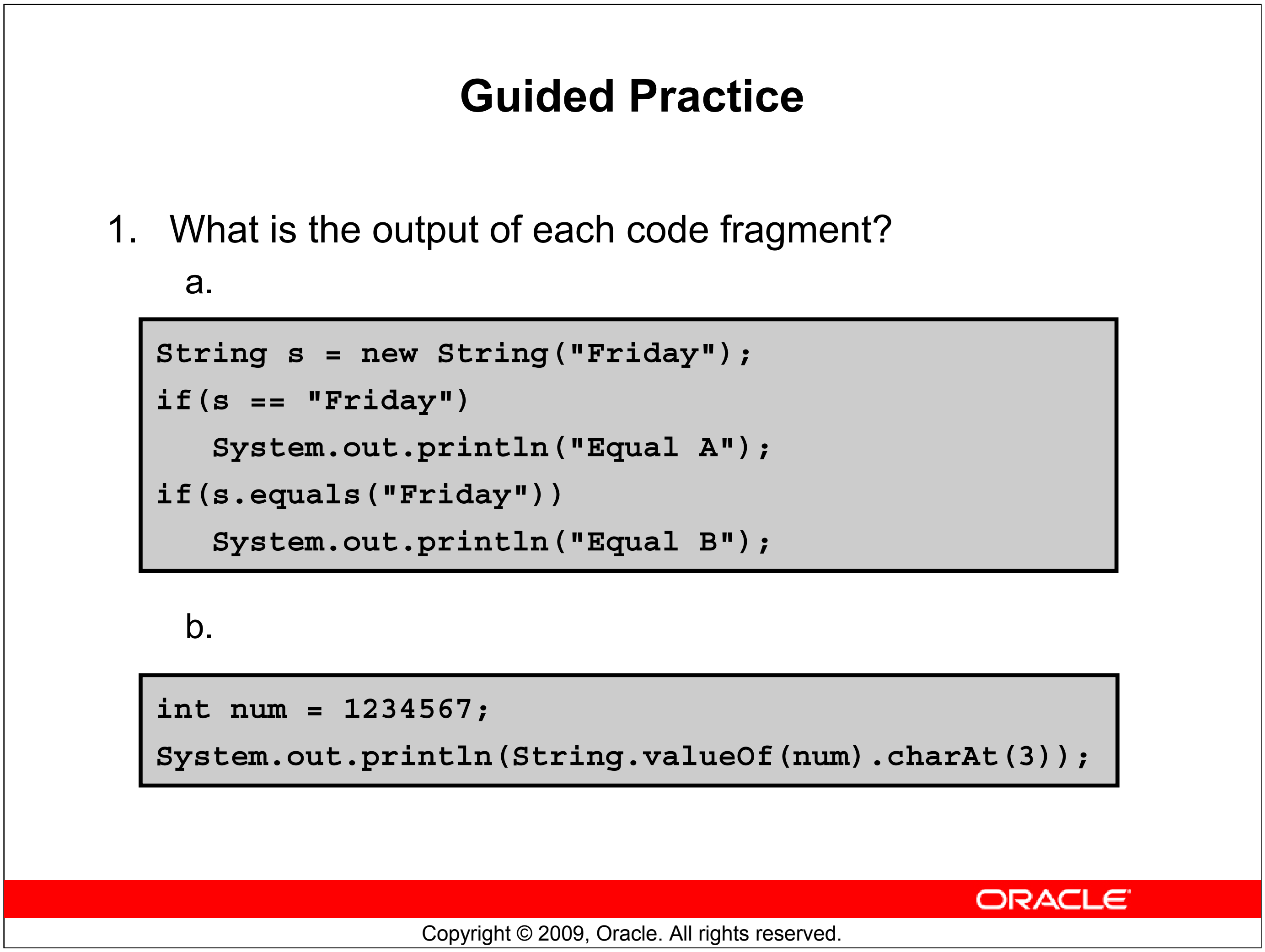
```
import java.text.DecimalFormat;
import java.text.NumberFormat;
import java.util.Locale;


public class DecFmtLoc {
    public static void localizedFormat(String pattern, double
                               value, Locale loc) {
      NumberFormat nf = NumberFormat.getNumberInstance(loc);
```

Using `DecimalFormat` for Localization (continued)

```
      DecimalFormat df = (DecimalFormat)nf;
        df.applyPattern(pattern);
        String output = df.format(value);
        System.out.println(pattern + "  " + output + "  " +
                                 loc.toString());
    }

    static public void main(String[] args) {
        localizedFormat("###,###.###", 123456.789, new
                                      Locale("en", "US"));
        localizedFormat("###,###.###", 123456.789, new
                                      Locale("de", "DE"));
        localizedFormat("###,###.###", 123456.789, new
                                      Locale("fr", "FR"));
    }
}
```

The output is

```
###,###.###  123,456.789  en_US
###,###.###  123.456,789  de_DE
###,###.###  123 456,789  fr_FR
```



Guided Practice

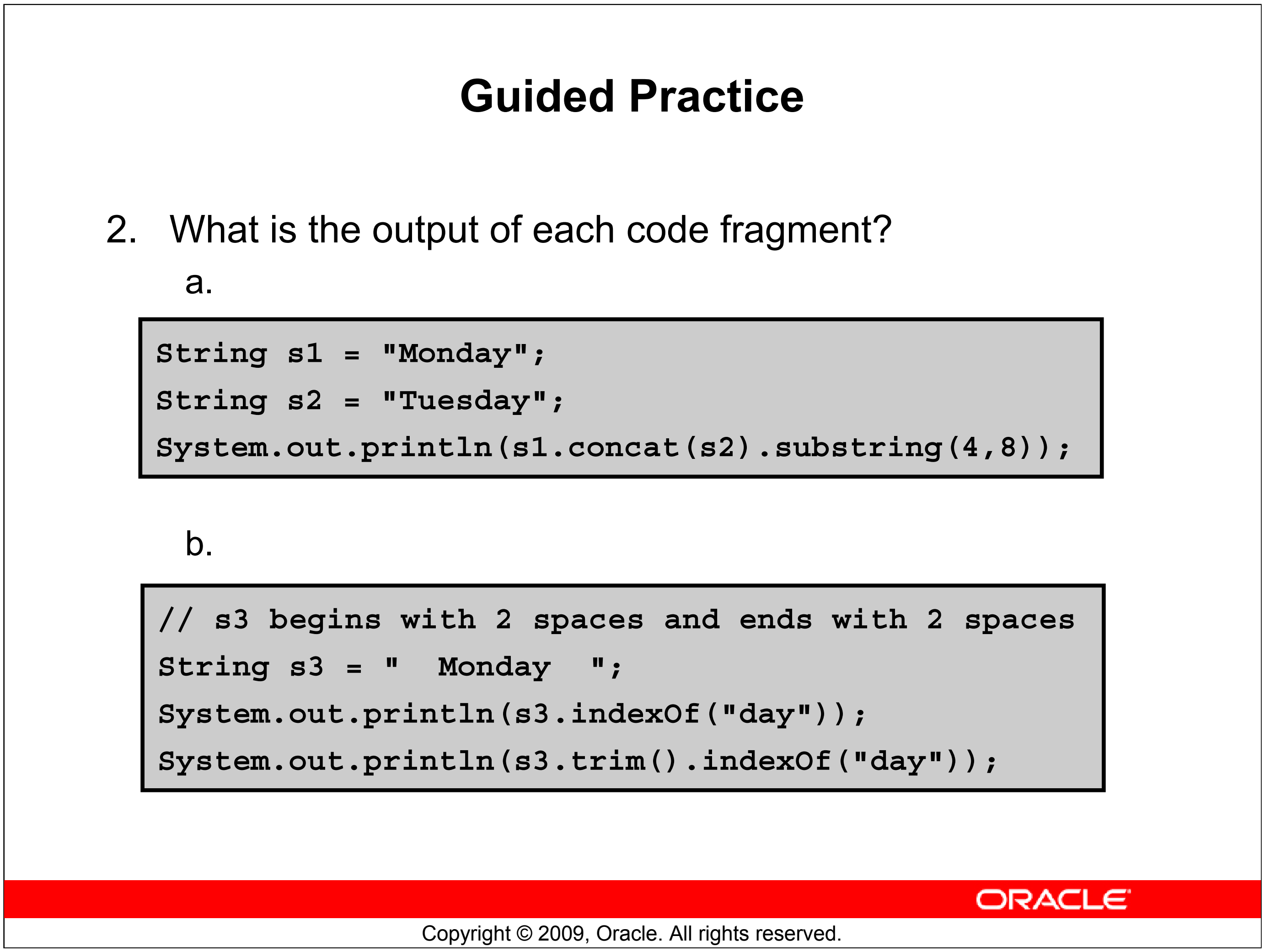
1. What is the output of each code fragment?

   a.

```
String s = new String("Friday");
if(s == "Friday")
   System.out.println("Equal A");
if(s.equals("Friday"))
   System.out.println("Equal B");
```

   b.

```
int num = 1234567;
System.out.println(String.valueOf(num).charAt(3));
```



Guided Practice

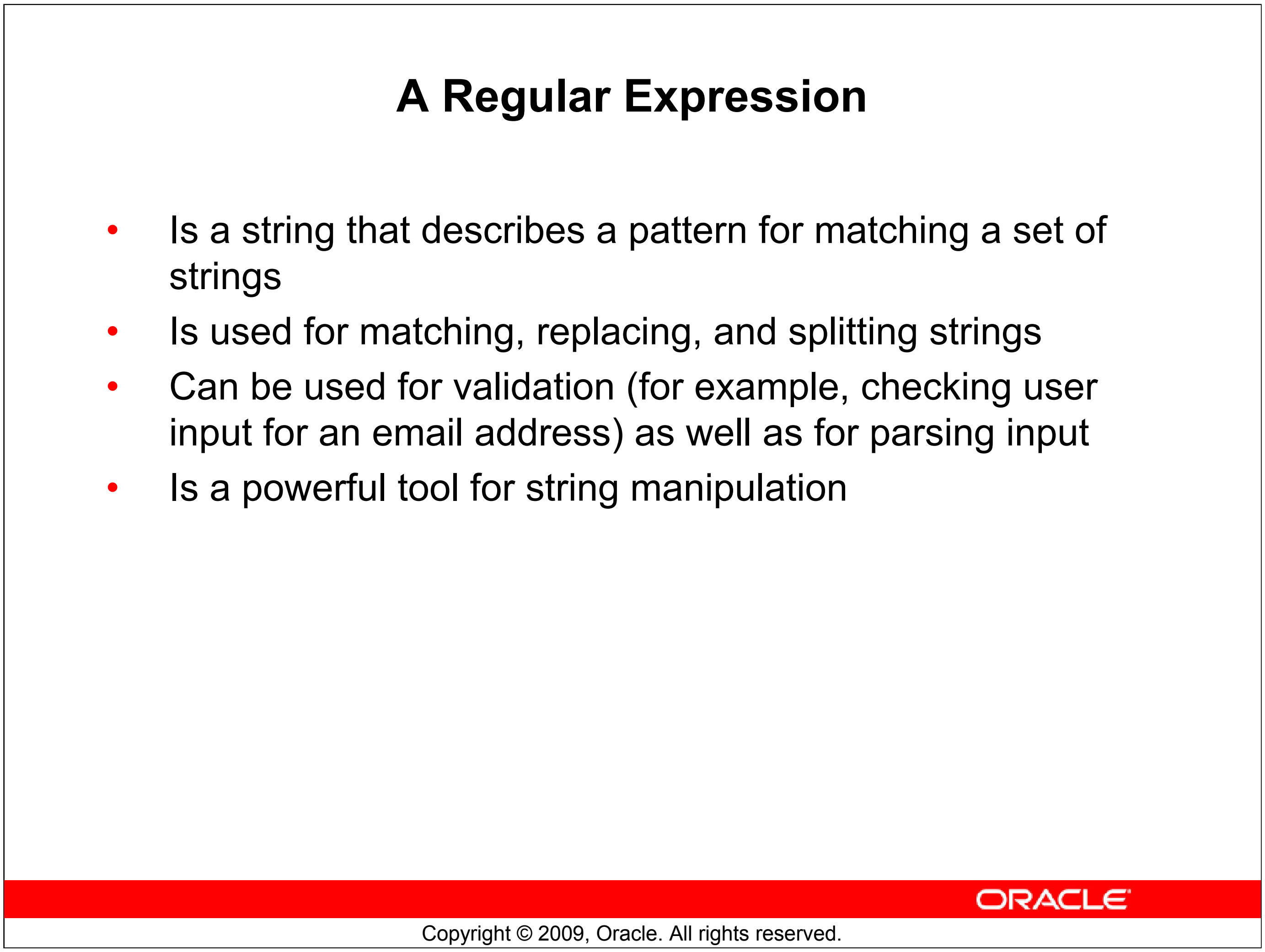
2. What is the output of each code fragment?

   a.

```
String s1 = "Monday";
String s2 = "Tuesday";
System.out.println(s1.concat(s2).substring(4,8));
```

   b.

```
// s3 begins with 2 spaces and ends with 2 spaces
String s3 = "  Monday  ";
System.out.println(s3.indexOf("day"));
System.out.println(s3.trim().indexOf("day"));
```

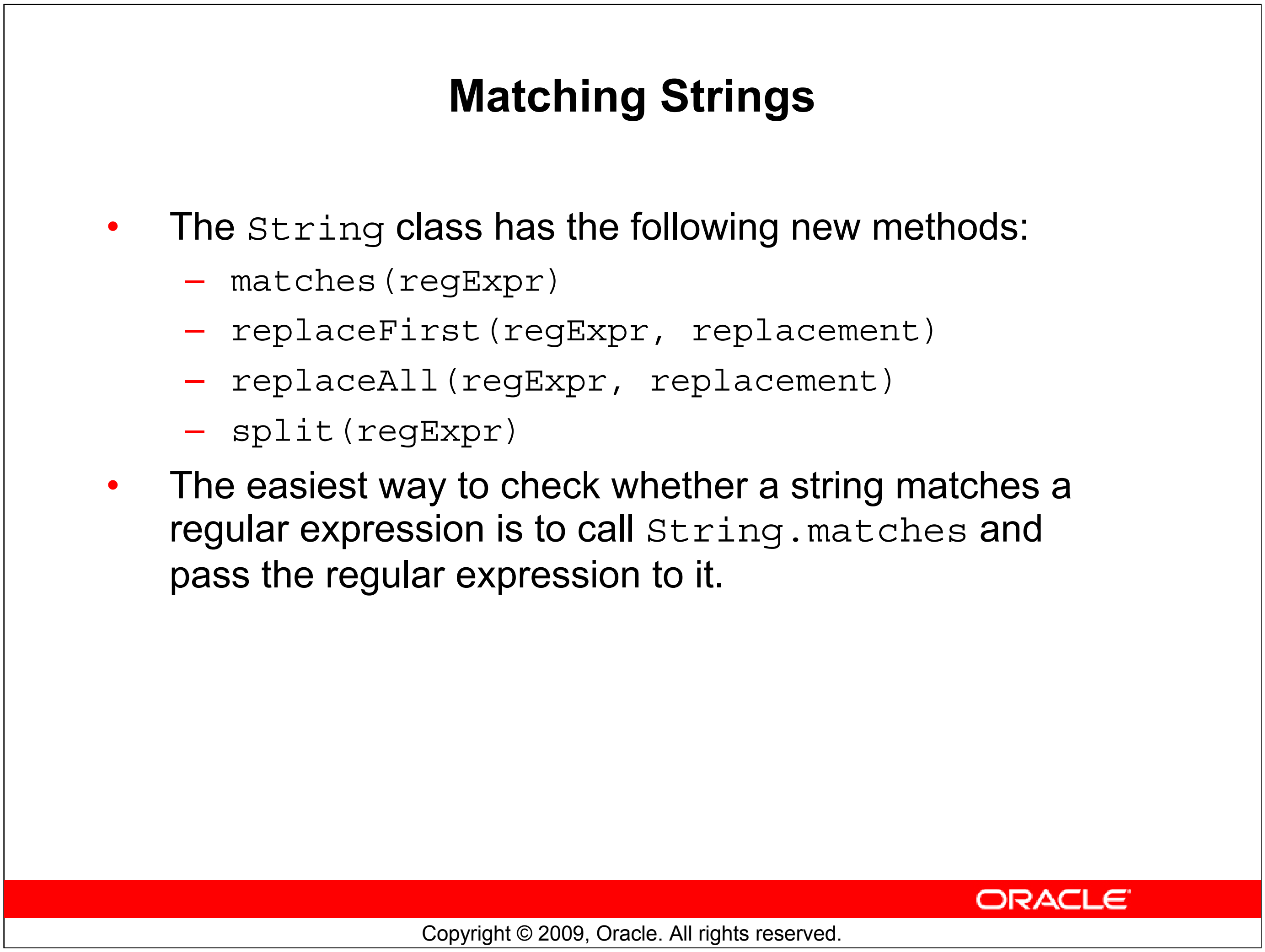


A Regular Expression

- Is a string that describes a pattern for matching a set of strings
- Is used for matching, replacing, and splitting strings
- Can be used for validation (for example, checking user input for an email address) as well as for parsing input
- Is a powerful tool for string manipulation

A Regular Expression

A regular expression, also known as a *regex* or a *regexp*, is a string whose pattern (template) describes a set of strings. The pattern determines what strings belong to the set and consists of literal characters and *metacharacters*, characters that have special meaning instead of a literal meaning. The process of searching text to identify matches—strings that match the regex's pattern—is known as *pattern matching*.



Matching Strings

- The `String` class has the following new methods:
  - `matches(regExpr)`
  - `replaceFirst(regExpr, replacement)`
  - `replaceAll(regExpr, replacement)`
  - `split(regExpr)`
- The easiest way to check whether a string matches a regular expression is to call `String.matches` and pass the regular expression to it.

Matching Strings

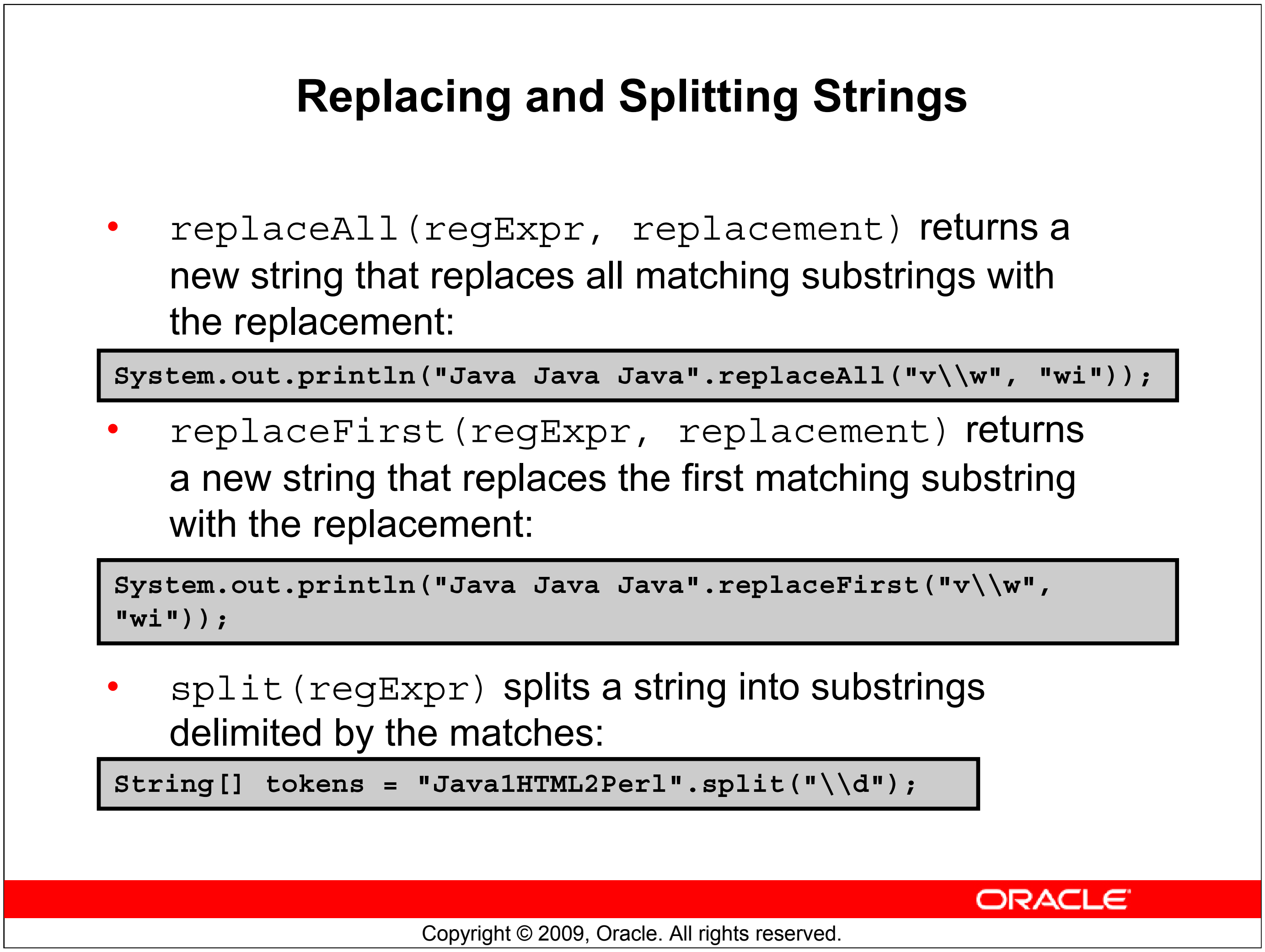
The `matches` method in the `String` class is very similar to the `equals` method. For example, the following two statements both evaluate to true:

```
"Java".matches("Java");
"Java".equals("Java");
```

However, the `matches` method is more powerful. It can match not only a fixed string, but also a set of strings that follow a pattern. For example, the following statements all evaluate to `true:`

```
"Java is fun".matches("Java.*")
"Java is cool".matches ("Java.*")
"Java is powerful".matches ("Java.*")
```

`"Java.*"` here is a regular expression. It describes a string pattern that begins with Java followed by zero or more characters. Here the substring `.*` matches zero or more characters.



Replacing and Splitting Strings

- `replaceAll(regExpr, replacement)` returns a new string that replaces all matching substrings with the replacement:

```
System.out.println("Java Java Java".replaceAll("v\\w", "wi"));
```

- `replaceFirst(regExpr, replacement)` returns a new string that replaces the first matching substring with the replacement:

```
System.out.println("Java Java Java".replaceFirst("v\\w",
"wi"));
```

- `split(regExpr)` splits a string into substrings delimited by the matches:

```
String[] tokens = "Java1HTML2Perl".split("\\d");
```

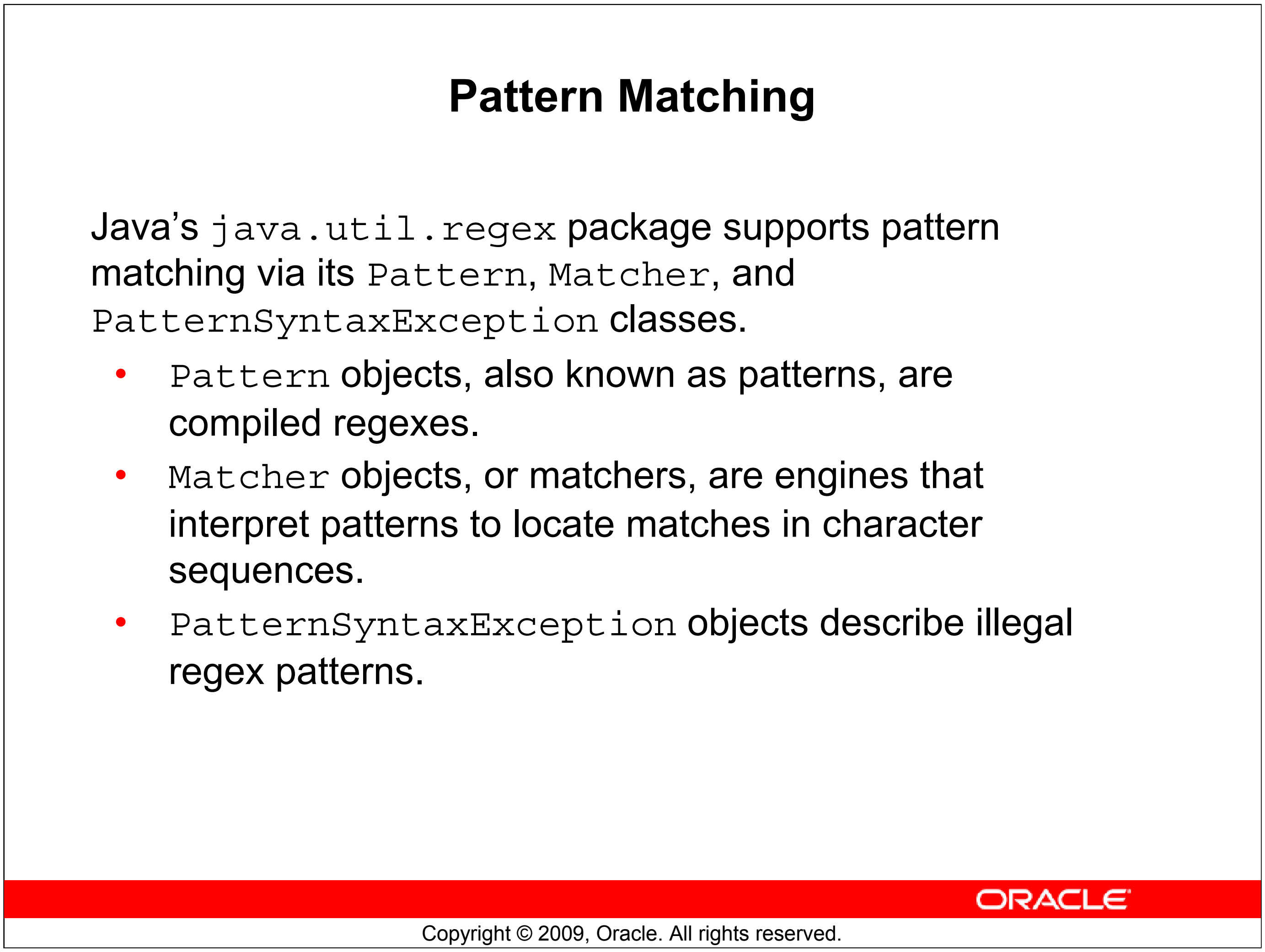
Replacing and Splitting Strings

The `String` class also contains the `replaceAll`, `replaceFirst`, and `split` methods for replacing and splitting strings.

The **`replaceAll`** example in the slide replaces the “va” substring in the word “Java” with “wi.” (The substring “v\\w” identifies the “v” character and “\w” matches any word characters following it). Hence the code displays “Jawi Jawi Jawi.”

In the second example, the `replaceFirst` method replaces “va” in the first “Java” string with “wi.” (As above, the substring “v\\w” identifies the “v” character and “\w” matches any word characters following it). Hence the code displays “Jawi Java Java.”

In the third example, the `split` method splits the “Java1HTML2Perl” string into “Java,” “HTML,” and “Perl,” returned in elements 0,1,2 of the String array *tokens*. (“\d” matches a digit).



Pattern Matching

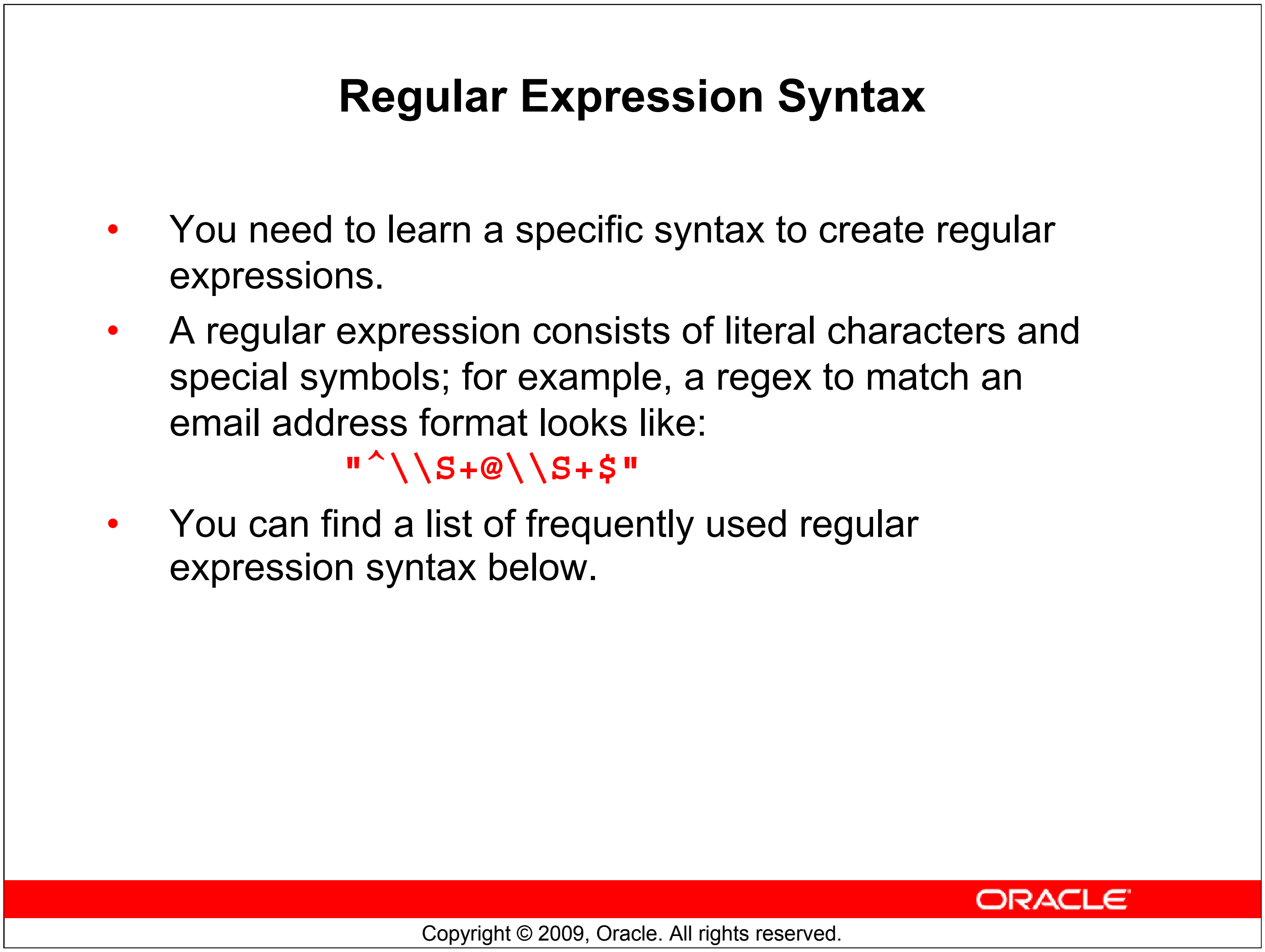
Java’s `java.util.regex` package supports pattern matching via its `Pattern`, `Matcher`, and `PatternSyntaxException` classes.

- `Pattern` objects, also known as patterns, are compiled regexes.
- `Matcher` objects, or matchers, are engines that interpret patterns to locate matches in character sequences.
- `PatternSyntaxException` objects describe illegal regex patterns.

Pattern Matching

The `java.util.regex` package primarily consists of three classes:

- A `Pattern` object is a compiled representation of a regular expression. The `Pattern` class provides no public constructors. To create a pattern you must first invoke one of its `public static compile` methods, which will then return a `Pattern` object. These methods accept a regular expression as the first argument.
- A `Matcher` object is the engine that interprets the pattern and performs match operations against an input string. Like the `Pattern` class, `Matcher` defines no public constructors. You obtain a `Matcher` object by invoking the `matcher` method on a `Pattern` object.
- A `PatternSyntaxException` object is an unchecked exception that indicates a syntax error in a regular expression pattern.



Regular Expression Syntax

- You need to learn a specific syntax to create regular expressions.
- A regular expression consists of literal characters and special symbols; for example, a regex to match an email address format looks like:

  `"^\\S+@\\S+$"`
- You can find a list of frequently used regular expression syntax below.

Regular Expression Syntax

The regex in the slide describes the format for an email address. Now you see how it is constructed. First examine what a standard email address looks like.

It STARTS WITH

A NONE SPACE CHARACTER

(and there's ONE OR MORE OF THOSE).

That's followed by

LITERALLY AN @ CHARACTER

Then A NONE SPACE CHARACTER

(and there's ONE OR MORE OF THOSE).

And that's then

THE END OF THE STRING

Translate this into Regular Expression syntax:

It STARTS WITH: ^

A NONE SPACE CHARACTER: \S

(and there's ONE OR MORE OF THOSE): +

That's followed by

LITERALLY AN @ CHARACTER: @

Regular Expression Syntax (continued)

Then A NONE SPACE CHARACTER: \S

(and there's ONE OR MORE OF THOSE): +

And that's then

THE END OF THE STRING: $

Now combine the individual elements into a single string and add extra \ characters to ensure that the \ of \S gets past the Java language and into the regex class.

Thus:

```
"^\\S+@\\S+$"
```

Special Metacharacters (Short List):

\escape character. \\, \n, \r, \t,\f matches \, newline, carriage return, tab, and formfeed, respectively.

^ matches the beginning of a string (or line if `MULTILINE` is set).

$ matches the end of a string (or line if `MULTILINE` is set).

\b matches on a word boundary.

xy matches x followed by y.

x | y matches either x or y.

\S matches a nonspace character.

\w matches a word character.

\d matches a digit (1-9).

(…) grouping operator groups into a unit that can be repeated with *,+, or ?.

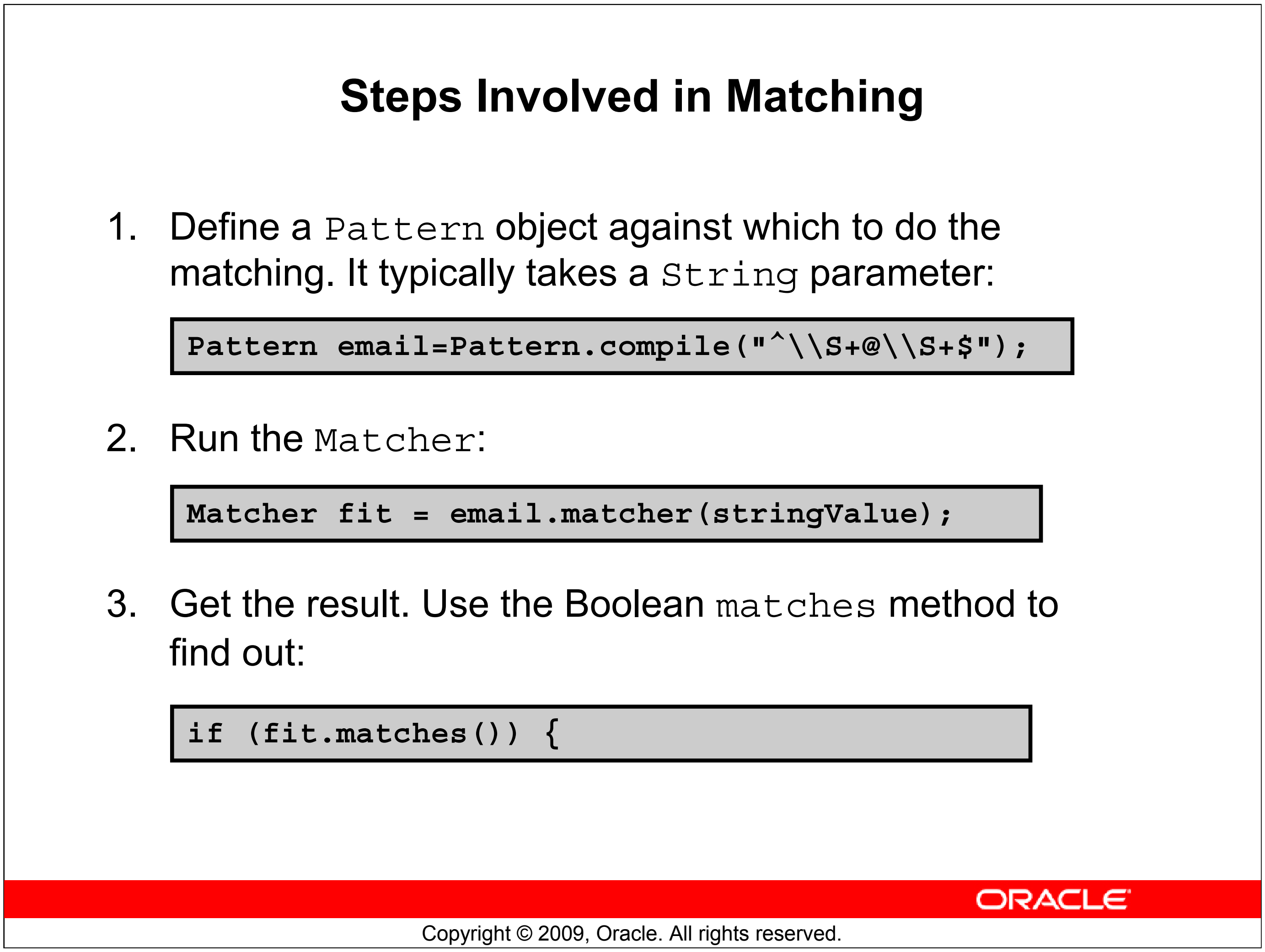
Character classes: [a-d] matches a single character “a” through “d.”

[^a-d] matches any character except characters “a” through “d.”

Repetition: x? matches zero or one occurrence of x.

x* matches 0 or more occurrences of x.

x+ matches 1 or more occurrences of x.



Steps Involved in Matching

1. Define a `Pattern` object against which to do the matching. It typically takes a `String` parameter:

```
Pattern email=Pattern.compile("^\\S+@\\S+$");
```

2. Run the `Matcher`:

```
Matcher fit = email.matcher(stringValue);
```

3. Get the result. Use the Boolean `matches` method to find out:

```
if (fit.matches()) {
```

Steps Involved in Matching

The code below takes the regular expression created in the previous slide and runs the matcher to see if it matches a valid email address format in a command-line parameter:

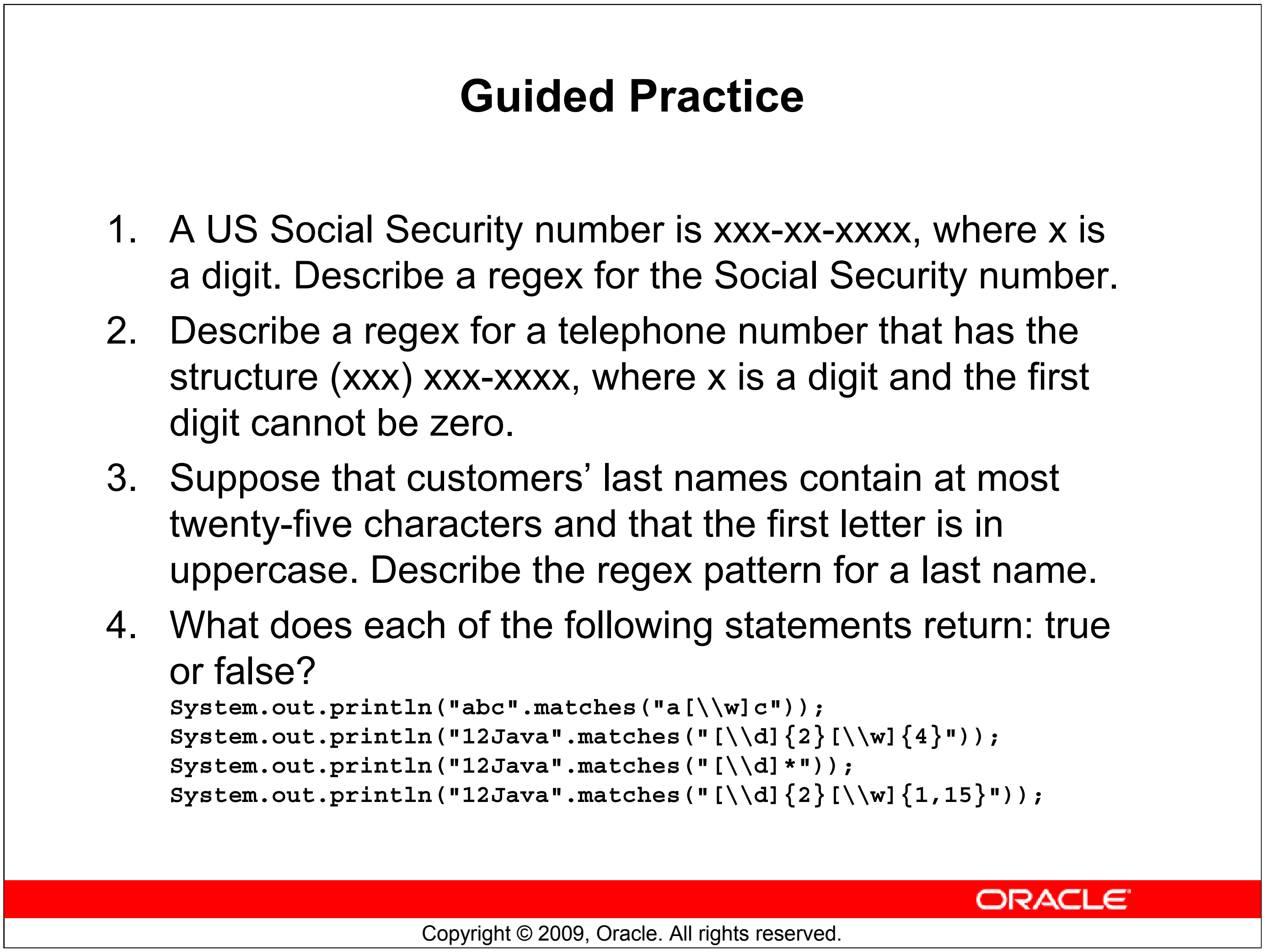
```
import java.util.regex.*;
public class Reg1 {
public static void main (String [] args) {
               Pattern email =
                        Pattern.compile("^\\S+@\\S+$");
               for (int j=0; j<args.length; j++) {
                      Matcher fit = email.matcher(args[j]);
                      if (fit.matches()) {
                              System.out.println (
                              "\"" +args[j] +
                              "\" IS a possible email
                                              address");
                      } else {
                              System.out.println (
                              "\"" + args[j] +
                              "\" is NOT an email address");
                }
```

Steps Involved in Matching (continued)

```
            }
        }
    }
```

Test the program by providing valid and invalid email addresses. To do this:

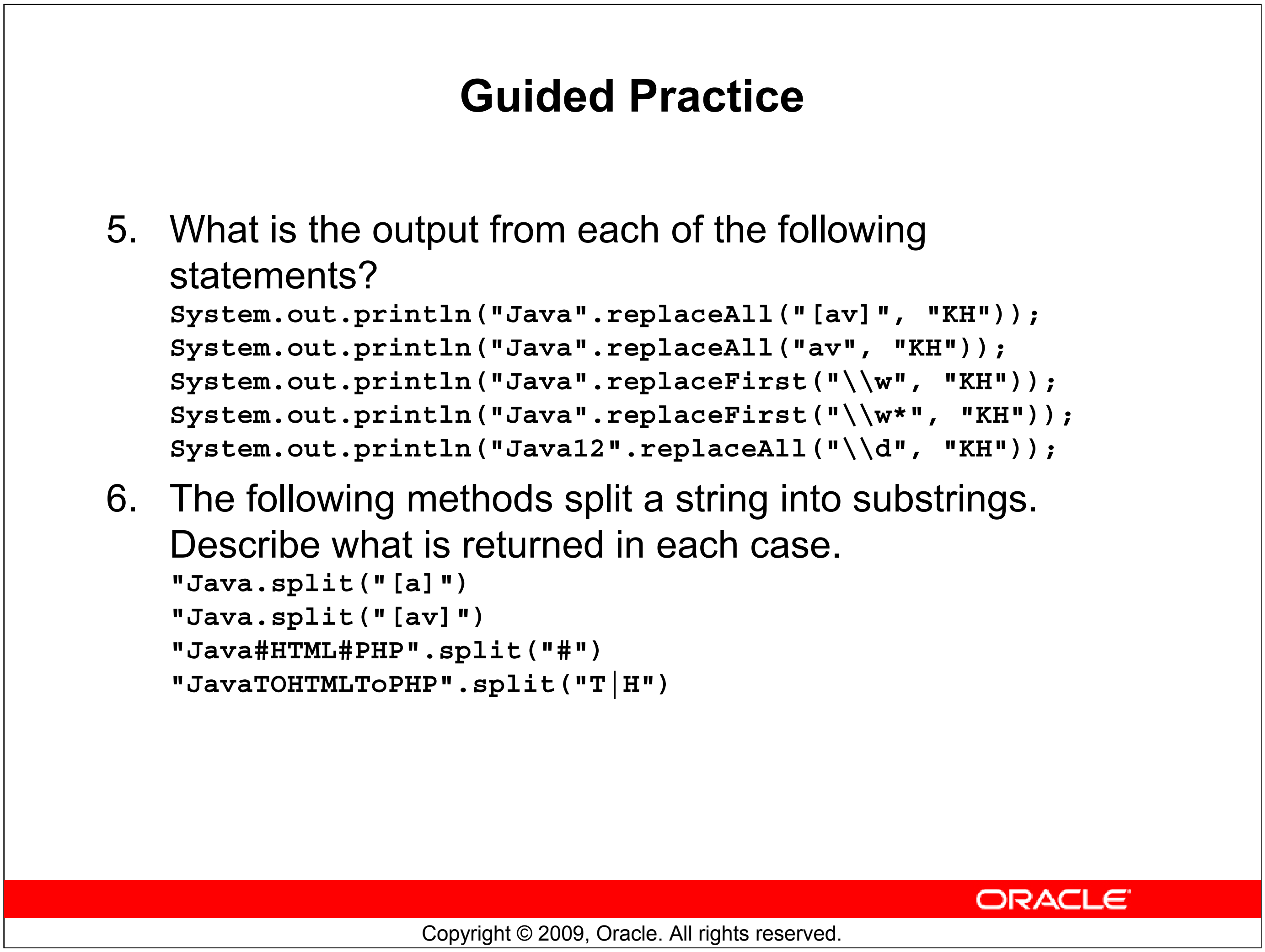
- Double-click the project, and select Run/Debug/Profile from the list at the left.
- On the Run/Debug/Profile page, click Edit (which is at the right of the Run Configurations box). Enter a valid or invalid email address in the Program Arguments field. Click OK twice.



Guided Practice

1. A US Social Security number is xxx-xx-xxxx, where x is a digit. Describe a regex for the Social Security number.
2. Describe a regex for a telephone number that has the structure (xxx) xxx-xxxx, where x is a digit and the first digit cannot be zero.
3. Suppose that customers' last names contain at most twenty-five characters and that the first letter is in uppercase. Describe the regex pattern for a last name.
4. What does each of the following statements return: true or false?

```
System.out.println("abc".matches("a[\\w]c"));
System.out.println("12Java".matches("[\\d]{2}[\\w]{4}"));
System.out.println("12Java".matches("[\\d]*"));
System.out.println("12Java".matches("[\\d]{2}[\\w]{1,15}"));
```



Guided Practice

5. What is the output from each of the following statements?

```
System.out.println("Java".replaceAll("[av]", "KH"));
System.out.println("Java".replaceAll("av", "KH"));
System.out.println("Java".replaceFirst("\\w", "KH"));
System.out.println("Java".replaceFirst("\\w*", "KH"));
System.out.println("Java12".replaceAll("\\d", "KH"));
```

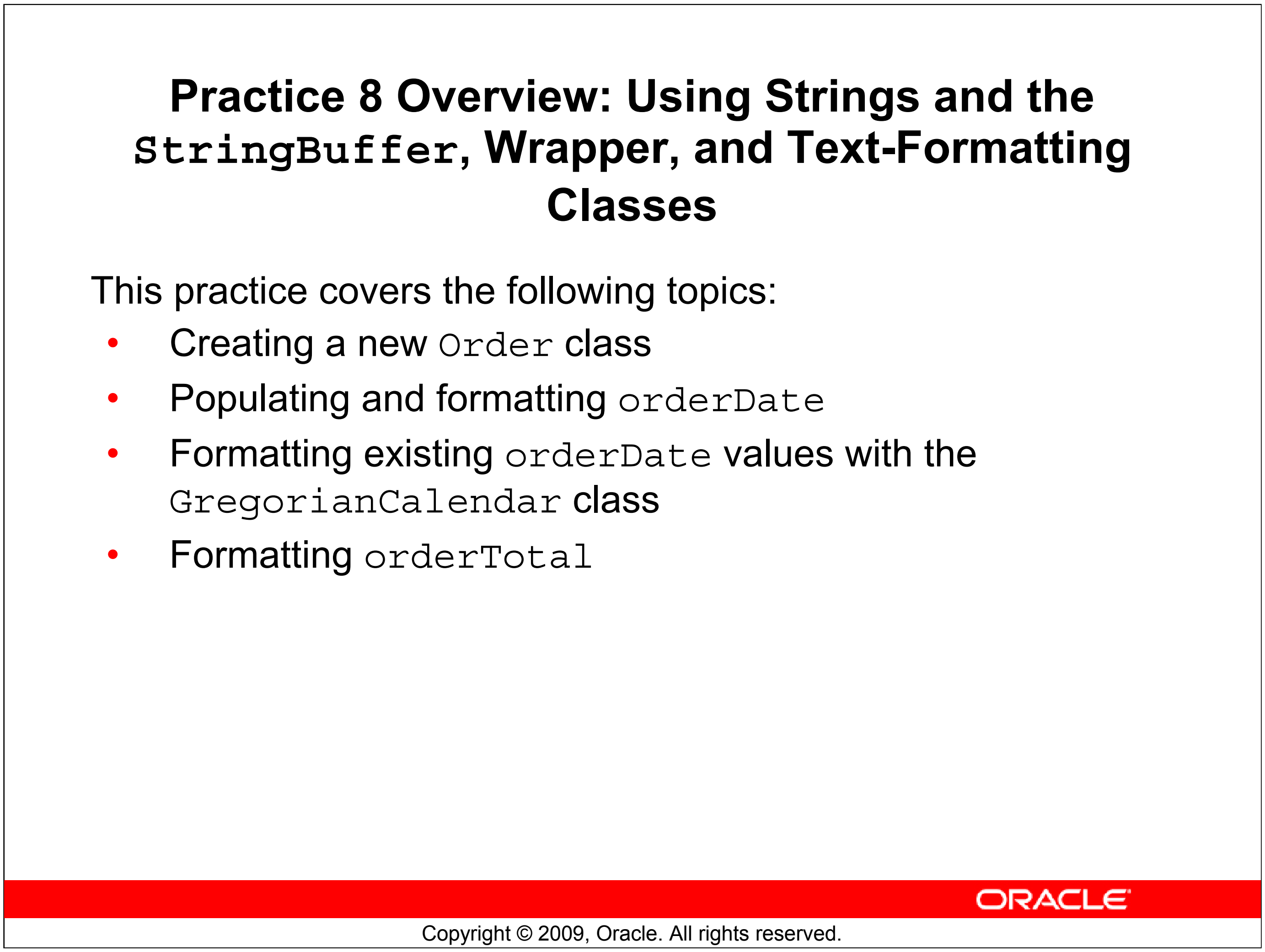
6. The following methods split a string into substrings. Describe what is returned in each case.

```
"Java.split("[a]")
"Java.split("[av]")
"Java#HTML#PHP".split("#")
"JavaTOHTMLToPHP".split("T|H")
```

Summary

In this lesson, you should have learned how to:

- Create strings in Java
- Perform operations on strings
- Compare `String` objects
- Use the conversion methods that are provided by the predefined wrapper classes
- Use the formatting classes
- Use regular expressions for matching, replacing, and splitting strings



Practice 8 Overview: Using Strings and the `StringBuffer`, Wrapper, and Text-Formatting Classes

This practice covers the following topics:

- Creating a new `Order` class
- Populating and formatting `orderDate`
- Formatting existing `orderDate` values with the `GregorianCalendar` class
- Formatting `orderTotal`

Practice 8 Overview: Using Strings and the `StringBuffer`, Wrapper, and Text-Formatting Classes

The goal of this practice is to modify the `Util` class to provide generic methods to support formatting the order details, such as presenting the total as a currency and controlling the date string format that is displayed. This should give you exposure to using some of the `java.text` formatting classes.

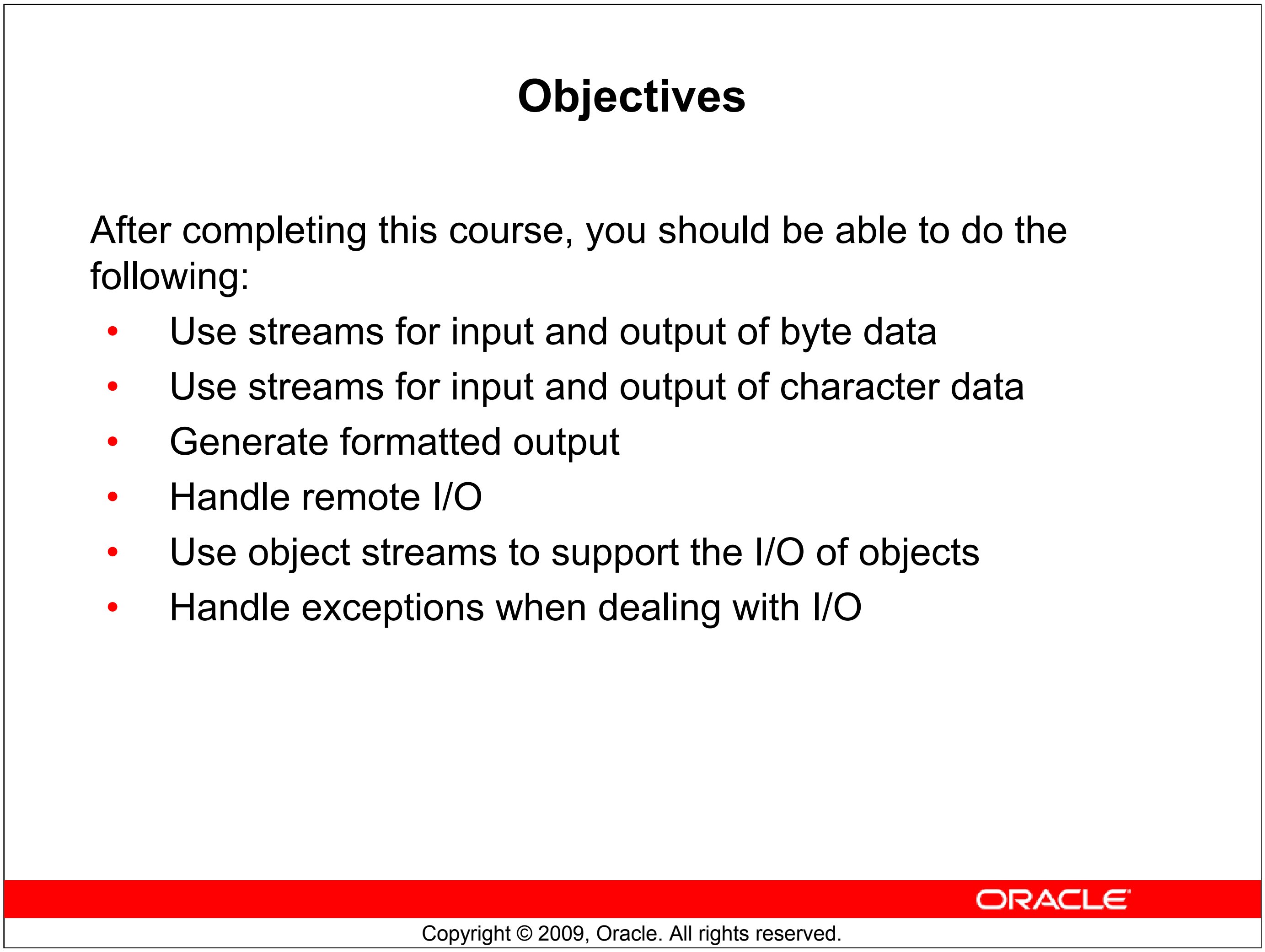
In this practice, you use the `GregorianCalendar` class. This class enables you to obtain a date value for a specific point in time. You can specify a date and time and see the behavior of your class respond to that specific date and time. The class can then be based on the values you enter rather than on the system date and time.

Note: If you have successfully completed the previous practice, continue using the same directory and files. If the compilation from the previous practice was unsuccessful and you want to move on to this practice, change to the `les08` directory, load the `OrderEntryApplicationLes08` application, and continue with this practice.

Viewing the model: To view the course application model up to this practice, load the OrderEntryApplicationLes08 application. In the Applications – Navigator node, expand `OrderEntryApplicationLes08 – OrderEntryProjectLes08 - Application Sources – oe`, and double-click the `UML Class Diagram1` entry. This diagram displays all the classes created up to this point in the course.

9

Using Streams for I/O



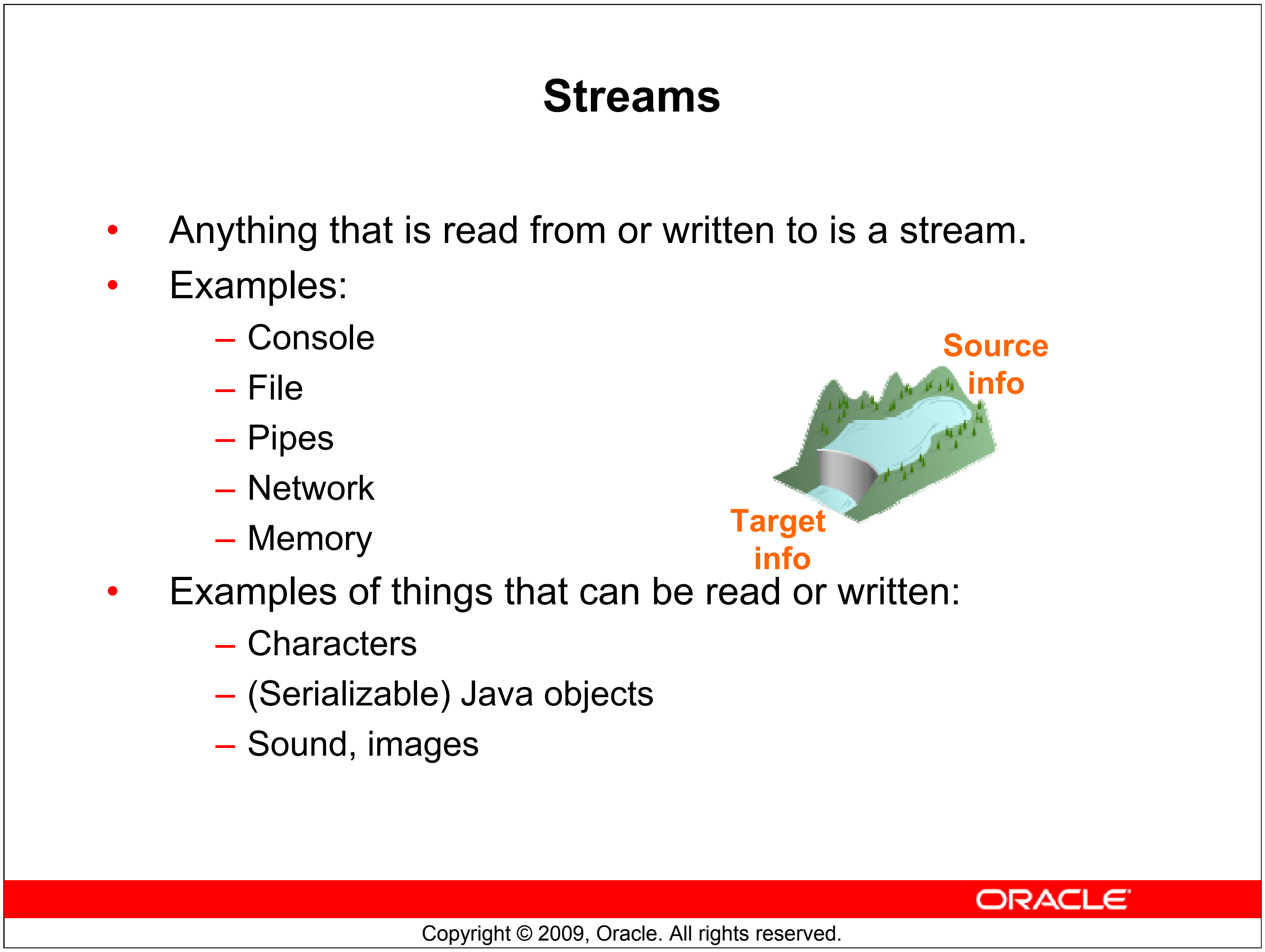
Objectives

After completing this course, you should be able to do the following:

- Use streams for input and output of byte data
- Use streams for input and output of character data
- Generate formatted output
- Handle remote I/O
- Use object streams to support the I/O of objects
- Handle exceptions when dealing with I/O

Lesson Objectives

This lesson discusses using streams for the input and output of byte and character data. It introduces the class hierarchies for byte and character I/O and provides examples of using some of the classes. You learn how to generate formatted output and how to use format specifiers to determine how items should be displayed. In addition, data and object streams and remote I/O are also covered.



Streams

- Anything that is read from or written to is a stream.
- Examples:
  - Console
  - File
  - Pipes
  - Network
  - Memory
- Examples of things that can be read or written:
  - Characters
  - (Serializable) Java objects
  - Sound, images

Source info
Target info

Streams

A stream is a sequence of data. It is an abstraction for anything one can read from or write to. The stream abstraction is necessary because the list of things your program can read from, or write to, is growing and dependent on technology. Earlier programs would want to read from a teletype or a card reader. Newer programs may want to write to an optical disk.

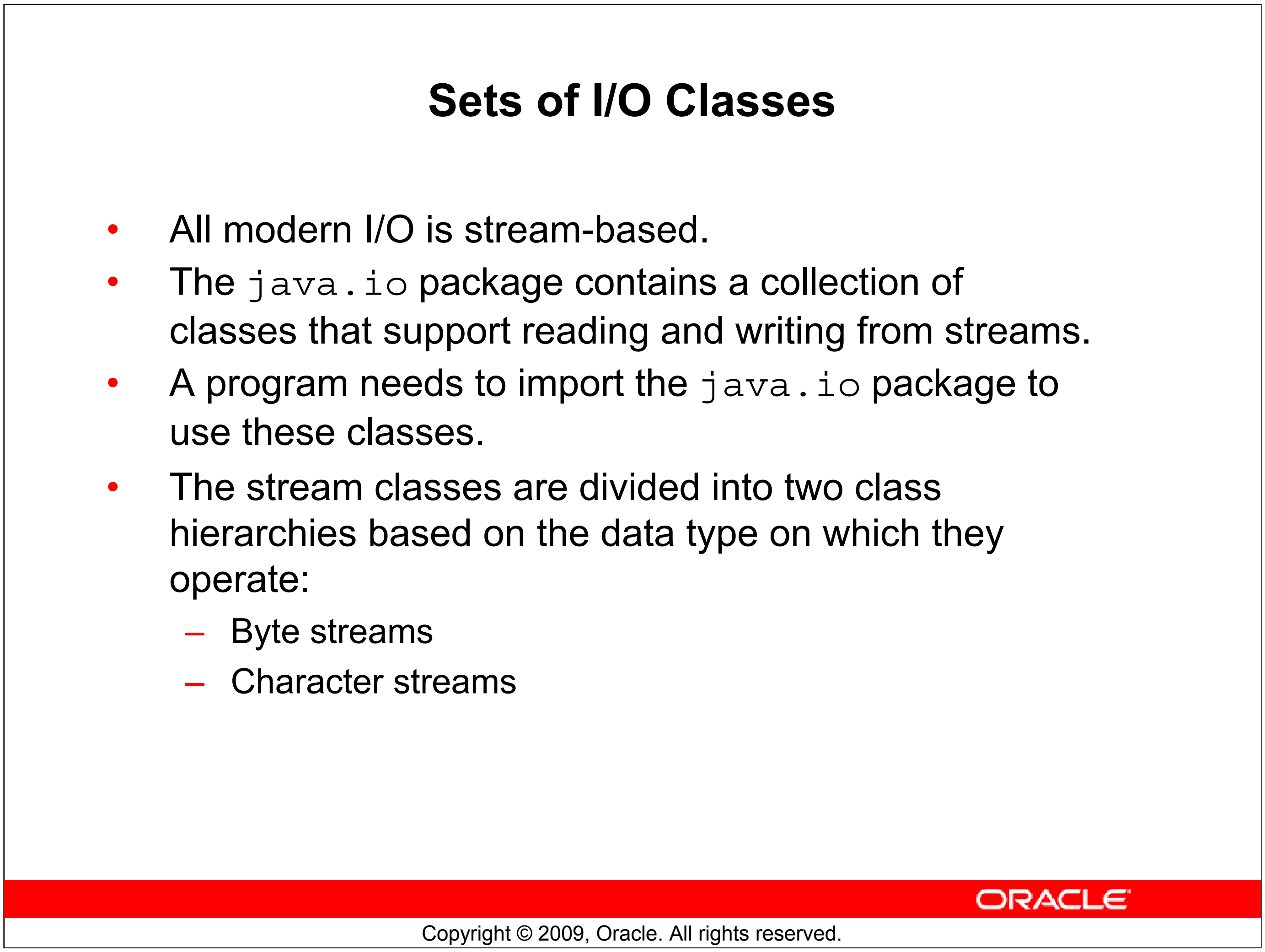
Examples of streams are listed in the slide. The most common stream to write to is the console. Other examples are files, pipes, the Internet, and the main memory.

What can you read and write to? Once again, the abstractions enable you to write to whatever you want: sound data, computational data, textual data, and so on.

Programming Model

No matter where the information is coming from or going to and no matter what type of data is being read or written, the algorithms for reading and writing data are almost always the same. You read (or write) as follows:

```
open a stream
while more data
    read (write) data
close the stream
```



Sets of I/O Classes

- All modern I/O is stream-based.
- The `java.io` package contains a collection of classes that support reading and writing from streams.
- A program needs to import the `java.io` package to use these classes.
- The stream classes are divided into two class hierarchies based on the data type on which they operate:
  - Byte streams
  - Character streams

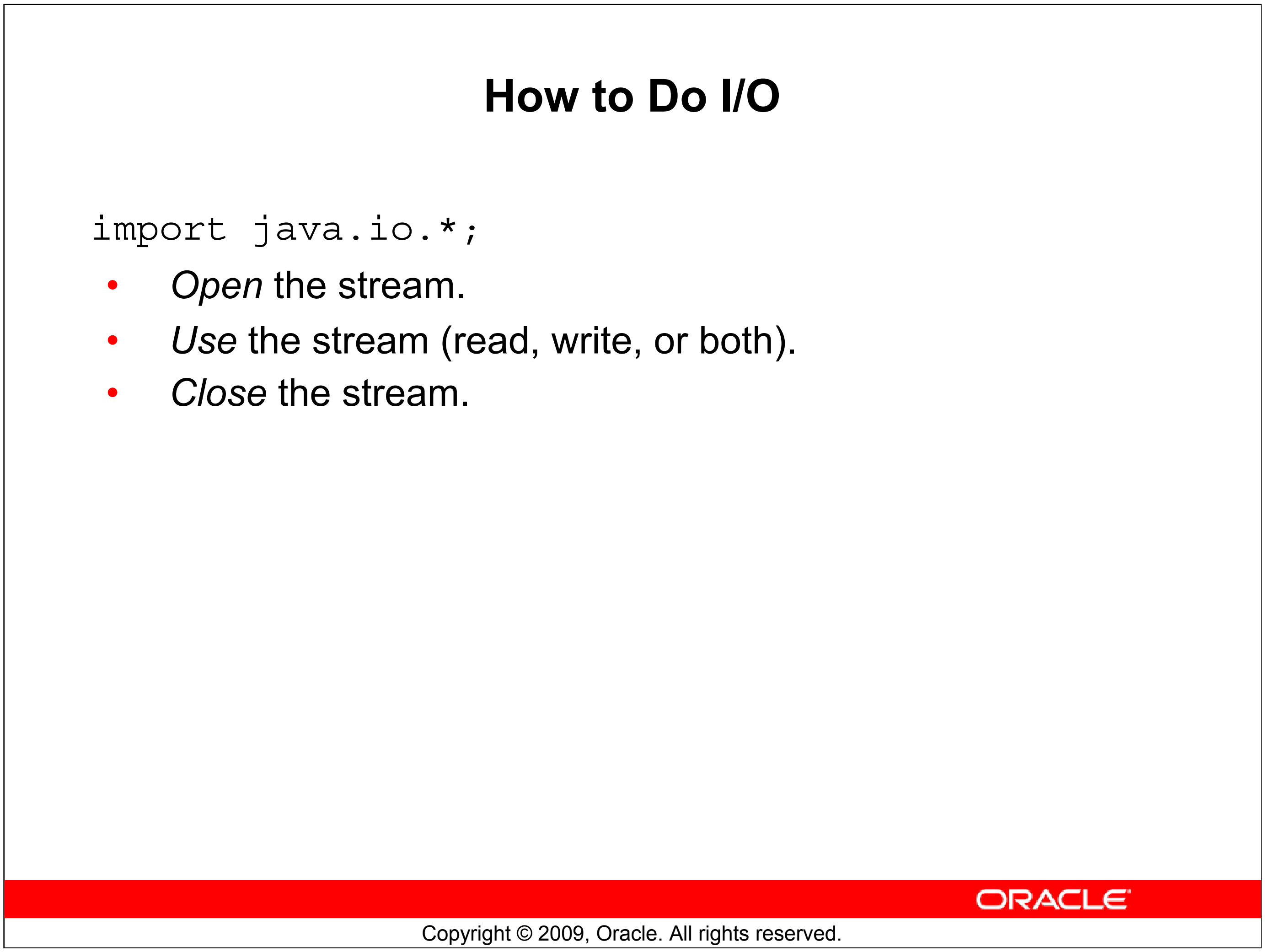
Sets of I/O Classes

Java implements the stream abstraction with several classes in the `java.io` package. Input is encapsulated in the `InputStream` class and output in the `OutputStream` class. These two abstract classes are what all objects should refer to when dealing with I/O in general. Java has several concrete subclasses of each to deal with the differences between disk files, network connections, and even memory buffers.

The stream classes are divided into two class hierarchies based on the two kinds of Java I/O: byte and character.

- **Byte-oriented I/O** is intended for data that does not need to be read by humans. Avoiding the translation step to and from readability saves time and space. If you want to save a file for later reading by the same or another program, you usually want byte-oriented I/O.
- On the other hand, if the information needs to be read by people, you want **character I/O**. Java uses an international code called Unicode that requires 16 bits to represent a character. Unicode is sufficient to provide accurate alphabets for almost all of the world's languages.

Note: Computers do not differentiate between binary files and text files. All files are stored in binary format, and thus all files are essentially binary files. Text I/O is built upon binary I/O to provide a level of abstraction for character encoding and decoding.



How to Do I/O

```
import java.io.*;
```

- *Open* the stream.
- *Use* the stream (read, write, or both).
- *Close* the stream.

How to Do I/O

Opening a Stream

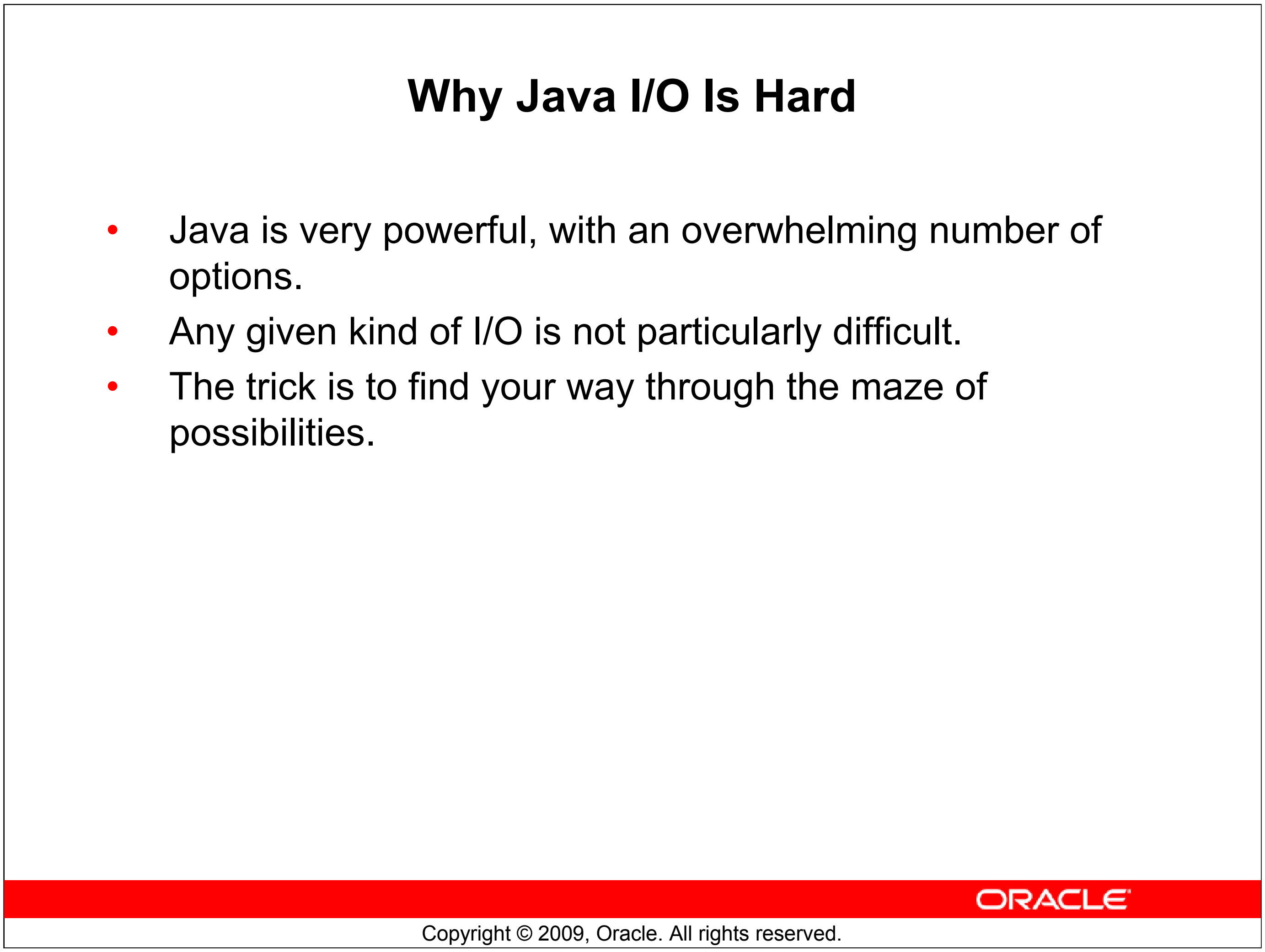
- There is data external to the program that you want to get; alternatively, you want to put data somewhere outside your program.
- When you open a stream, you are making a connection to that external place.
- After the connection is made, you forget about the external place and just use the stream.

Using a Stream

- Some streams can be used only for input, others only for output, still others for both.
- Using a stream means doing input to it, or output from it.
- But it is not usually that simple because you need to manipulate the data in some way as it comes in or goes out.

Closing a Stream

- A stream is an expensive resource.
- There is a limit on the number of streams that you can have open at one time.
- You should not have more than one stream open on the same file.
- You must close a stream before you can open it again.
- *Always close your streams!*

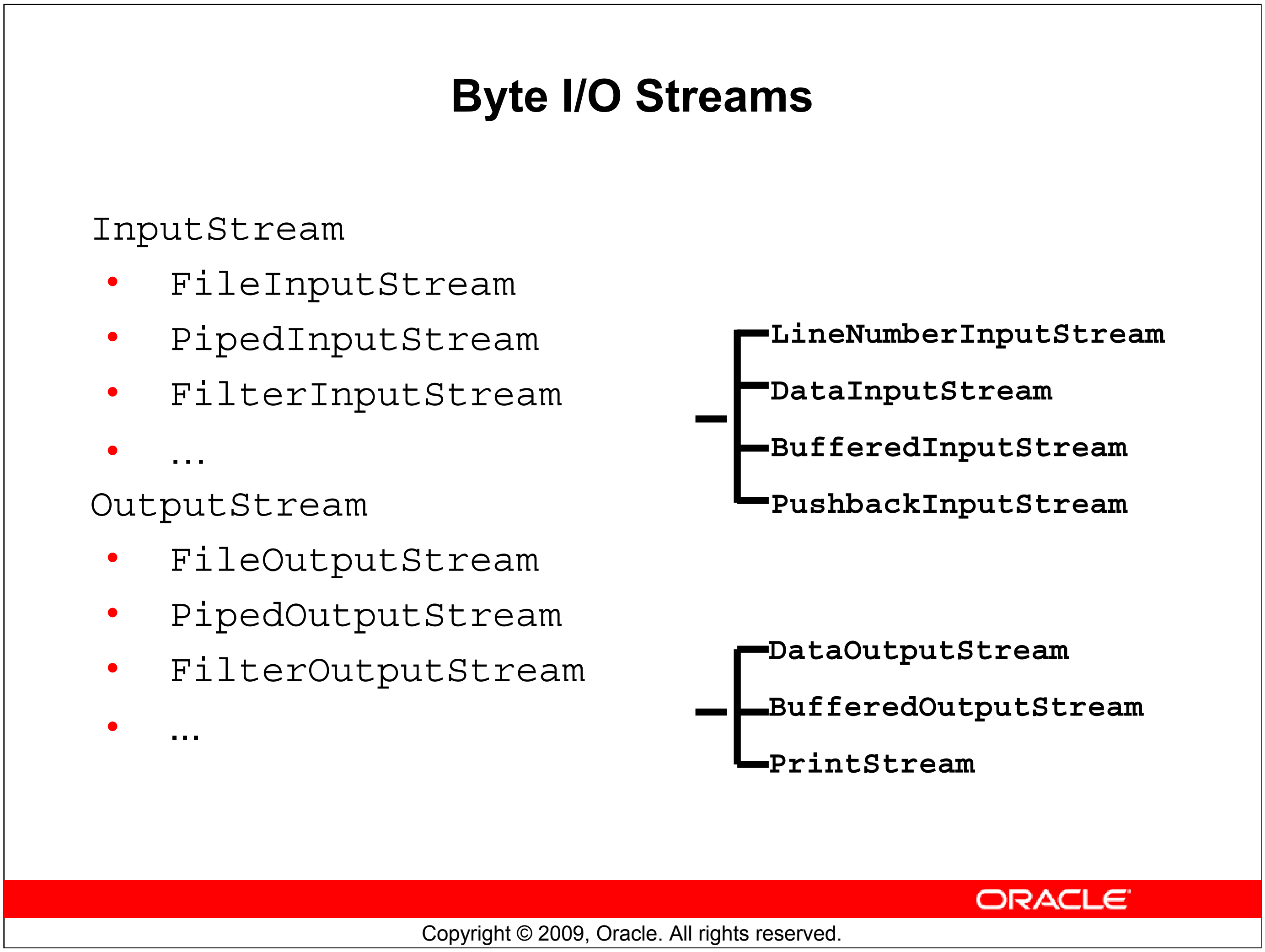


Why Java I/O Is Hard

- Java is very powerful, with an overwhelming number of options.
- Any given kind of I/O is not particularly difficult.
- The trick is to find your way through the maze of possibilities.

Why Java I/O Is Hard

The slides that follow are intended to help you find your way through the maze of possibilities!



Byte I/O Streams

`InputStream`

- `FileInputStream`
- `PipedInputStream`
- `FilterInputStream`
  - `LineNumberInputStream`
  - `DataInputStream`
  - `BufferedInputStream`
  - `PushbackInputStream`
- …

`OutputStream`

- `FileOutputStream`
- `PipedOutputStream`
- `FilterOutputStream`
  - `DataOutputStream`
  - `BufferedOutputStream`
  - `PrintStream`
- …

Byte I/O Streams

The design of the Java I/O classes is a good example of applying inheritance, where common operations are generalized in superclasses and subclasses provide specialized operations. The slide shows some of the classes for performing binary I/O. `InputStream` is the root for byte input classes and `OutputStream` is the root for byte output classes.

The descendants of the abstract class `InputStream` are:

```
ByteArrayInputStream
FileInputStream
FilterInputStream
     BufferedInputStream
     DataInputStream
     LineNumberInputStream  (deprecated – use LineNumberReader)
     PushbackInputStream
ObjectInputStream
PipedInputStream
```

Byte I/O Streams (continued)

SequenceInputStream

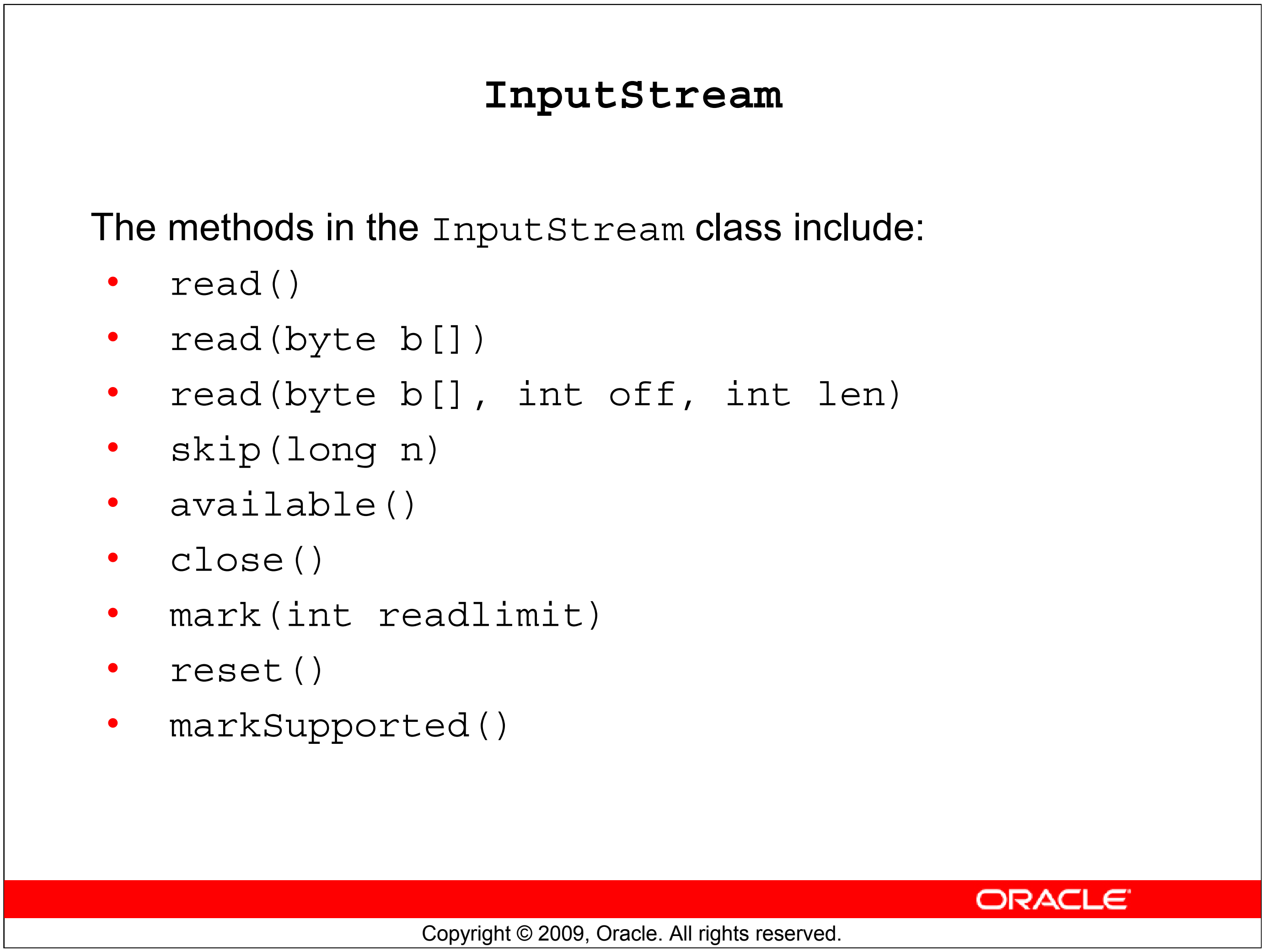
StringBufferInputStream (deprecated – use StringReader or ByteArrayInputStream)

The Java I/O classes are each meant to do one job well and to be used in combination to do complex tasks. As previously mentioned, the binary source of the stream can come from a file, a pipe, an array of bytes, or a Java object. You use the corresponding class to get the input information. For example, ByteArrayInputStream “reads” from a byte array; the FileInputStream class is used for reading information from a file.

The descendants of the abstract class OutputStream are:

- ByteArrayOutputStream
- FileOutputStream
- FilterOutputStream
  - BufferedOutputStream
  - DataOutputStream
  - PrintStream
- ObjectOutputStream
- PipedOutputStream

The mechanisms for output streams are identical to those for the input streams. You must use the class that corresponds to the type of target to which you want to write the byte information. A DataOutputStream provides encodings for the basic Java types. A PrintStream is the old way to convert to human readable form, but PrintWriter should generally be used instead now.



InputStream

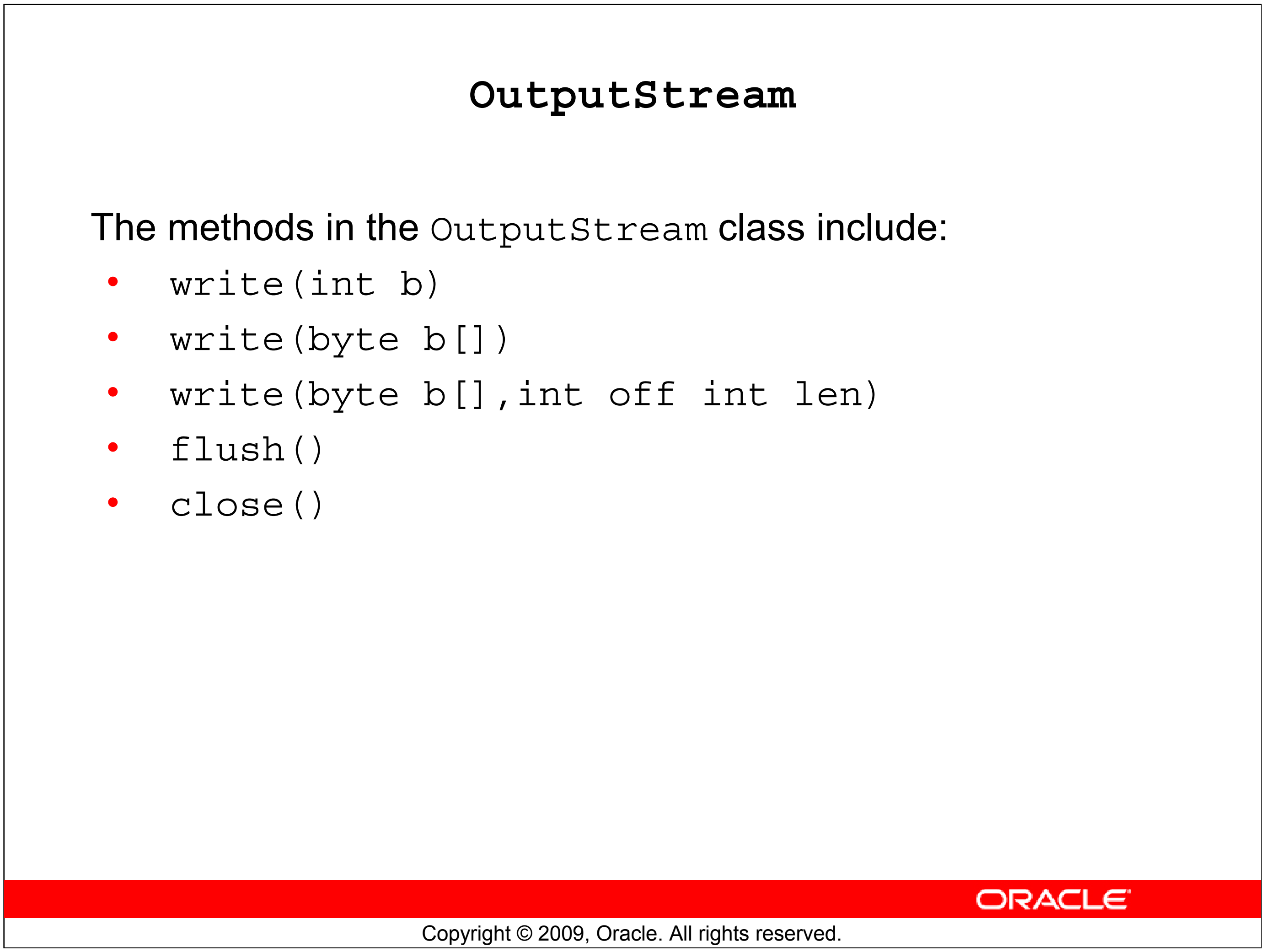
The methods in the `InputStream` class include:

- `read()`
- `read(byte b[])`
- `read(byte b[], int off, int len)`
- `skip(long n)`
- `available()`
- `close()`
- `mark(int readlimit)`
- `reset()`
- `markSupported()`

`InputStream`

`InputStream` is an abstract class that defines Java's model of streaming binary input. The `InputStream` class has the following methods:

- `read()` reads the next byte of data from the input stream.
- `read(byte b[])` attempts to read up to `b.length` bytes into `b` and returns the actual number that were successfully read.
- `read(byte b[], int off, int len)` attempts to read up to `len` bytes into `b` starting at `b[off]`, returning the number of bytes successfully read.
- `skip(long n)` skips over and discards `n` bytes of data from this input stream, returning the number of bytes skipped.
- `available()` returns the number of bytes of input currently available for reading.
- `close()` closes the input stream and releases any system resources associated with it.
- `mark(int readlimit)` places a mark at the current point in the input stream that will remain valid until `readlimit` bytes are read.
- `reset()` returns the input pointer to the previously set mark.
- `markSupported()` returns true if `mark/reset` are supported on this stream.



OutputStream

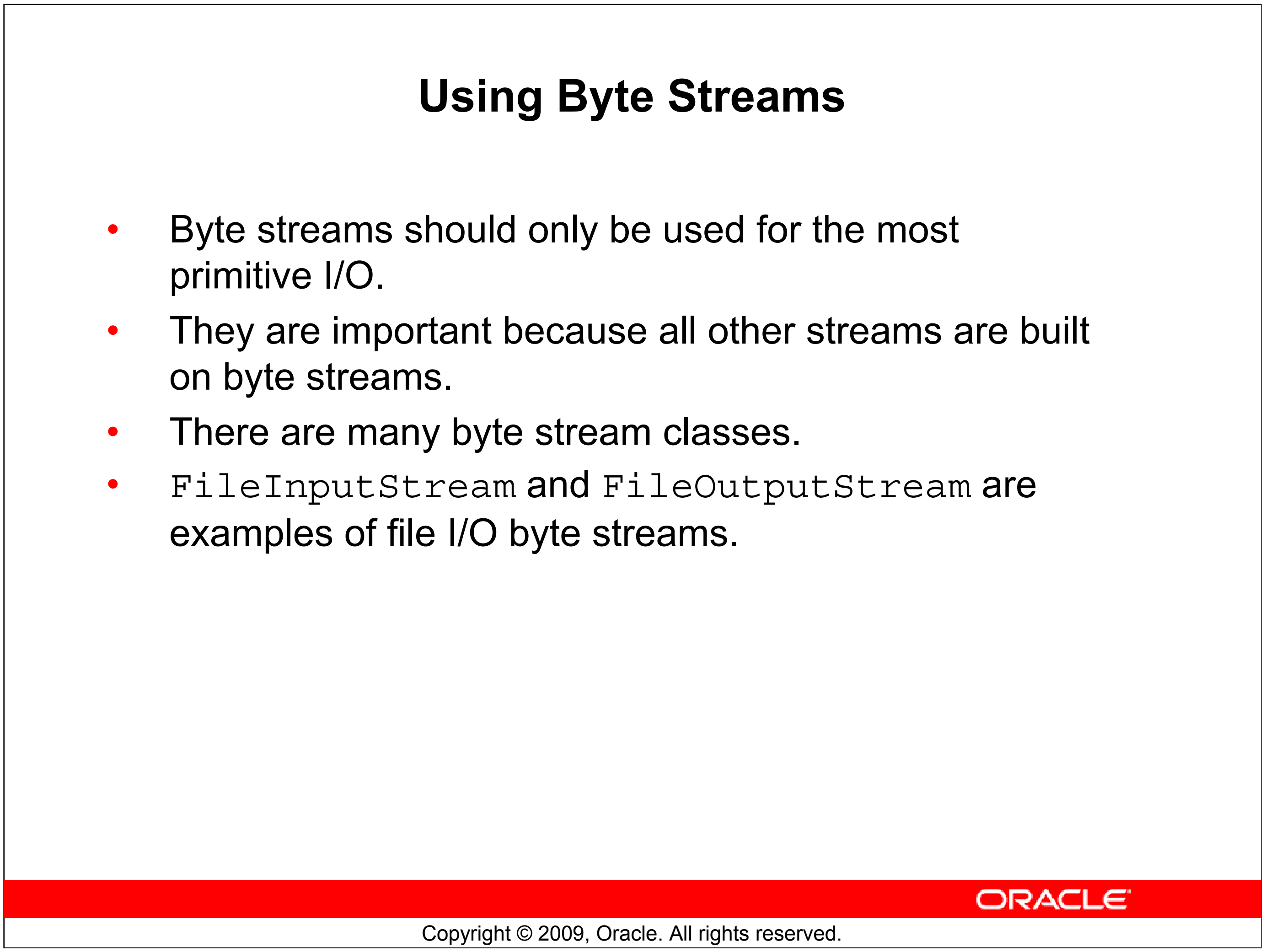
The methods in the `OutputStream` class include:

- `write(int b)`
- `write(byte b[])`
- `write(byte b[],int off int len)`
- `flush()`
- `close()`

OutputStream

`OutputStream` is an abstract class that defines Java's model of streaming binary output. The `OutputStream` class has the following methods:

- `write(int b)` writes a single byte to an output stream. Note that the parameter is an `int` that allows you to call write with expressions without having to cast them back to `byte`.
- `write(byte b[])` writes a complete array of bytes to an output stream.
- `write(byte b[], int off, int len)` writes a subrange of `len` bytes from the array `b`, beginning at `b[off]`.
- `flush()` finalizes the output state so that any buffers are cleared.
- `close()` closes the output stream and releases any system resources associated with it.



Using Byte Streams

- Byte streams should only be used for the most primitive I/O.
- They are important because all other streams are built on byte streams.
- There are many byte stream classes.
- `FileInputStream` and `FileOutputStream` are examples of file I/O byte streams.

Using Byte Streams

The following example uses `FileInputStream` and `FileOutputStream`. Other kinds of byte streams are used in much the same way; they differ mainly in the way they are constructed.

Byte Stream Example

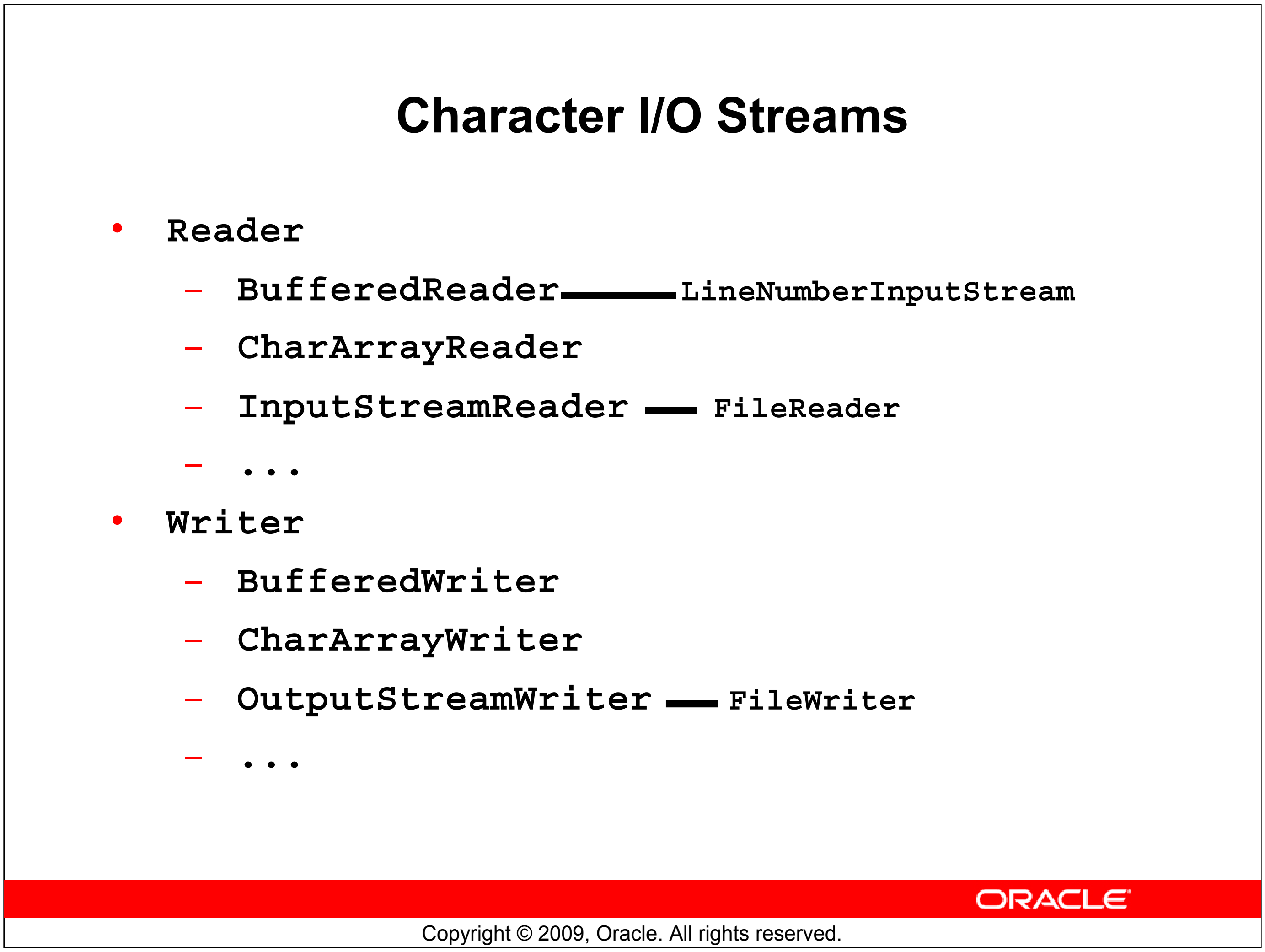
This example uses binary I/O to write ten byte values from 1 to 10 to a file named `temp.dat` and then reads them back from the file.

```
import java.io.*;
public class TestFileStream {
    public static void main(String[] args) throws IOException   {
    // Create an output stream to the file
    FileOutputStream output =
new FileOutputStream("temp.dat");
    // Output values to the file
```

Using Byte Streams (continued)

```
    for (int i = 1; i <= 10; i++)
       output.write(i);

    // Close the output stream
    output.close();

    //Create an input stream for the file
    FileInputStream input = new FileInputStream("temp.dat");

    // Read values from the file
    int value;
    while ((value = input.read()) != -1)
      System.out.print(value + " ");

    // Close the output stream
    input.close();
  }
}
```

Note: The expression `((value = input.read()) != -1)` reads a byte from `input.read()`, assigns it to `value` and checks whether it is `–1`. The input value of `–1` signifies the end of a file.



Character I/O Streams

- `Reader`
  - `BufferedReader` — `LineNumberInputStream`
  - `CharArrayReader`
  - `InputStreamReader` — `FileReader`
  - `...`
- `Writer`
  - `BufferedWriter`
  - `CharArrayWriter`
  - `OutputStreamWriter` — `FileWriter`
  - `...`

Character I/O Streams

The hierarchy of character I/O streams works in the same way as the one for byte I/O streams. In this case, all classes having to deal with input operations are inherited from the `Reader` class, and the classes that are concerned with output operations inherit from the `Writer` class.

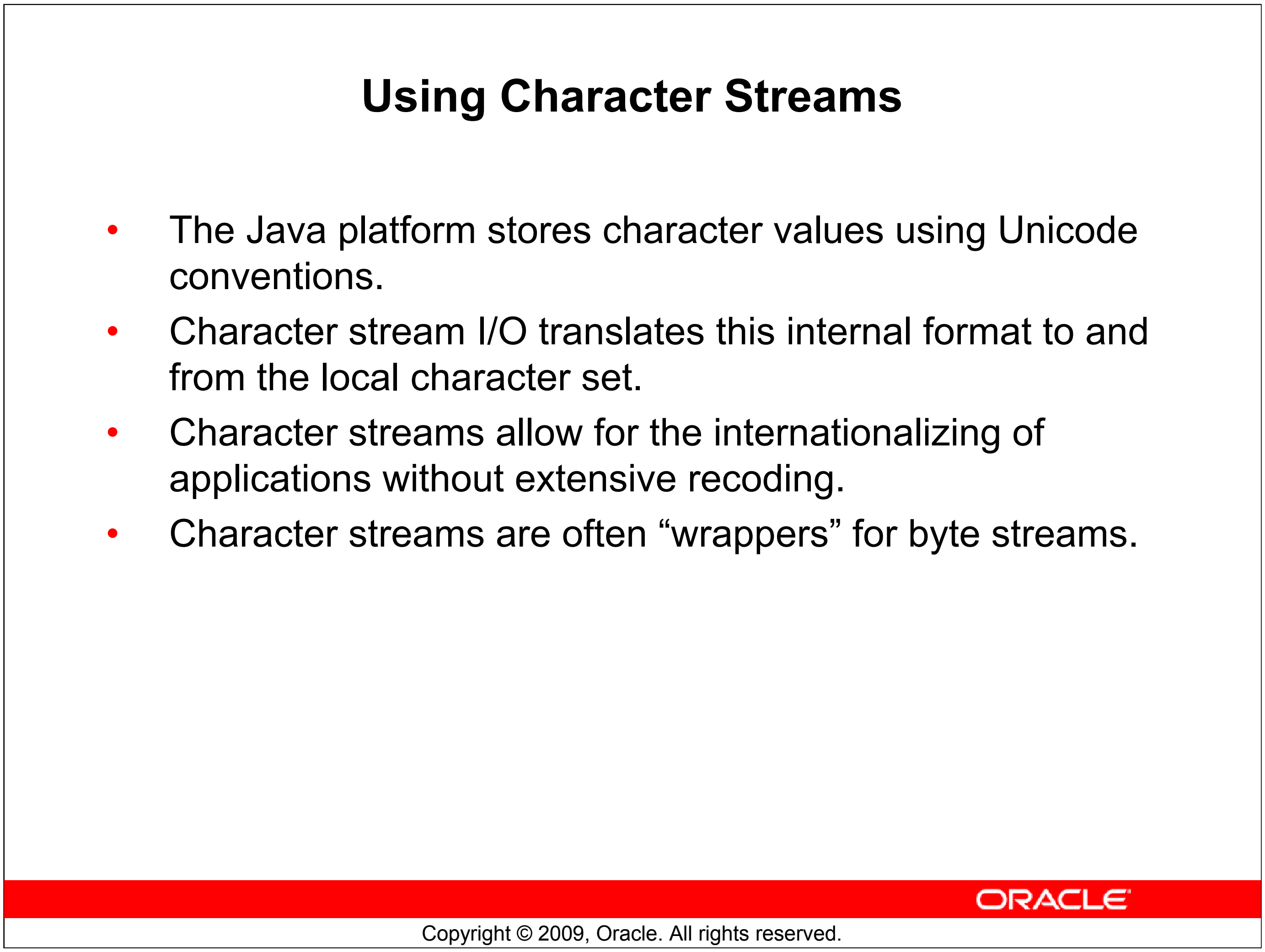
The abstract `Reader` class has the following descendants:

```
BufferedReader
     LineNumberReader
CharArrayReader
FilterReader (abstract)
     PushbackReader
InputStreamReader
     FileReader
PipedReader
StringReader
```

Character I/O Streams (continued)

Writers are used similarly. The descendants of the abstract class `Writer` are:

```
BufferedWriter
CharArrayWriter
FilterWriter (abstract)
OutputStreamWriter
     FileWriter
PipedWriter
PrintWriter
StringWriter
```



Using Character Streams

- The Java platform stores character values using Unicode conventions.
- Character stream I/O translates this internal format to and from the local character set.
- Character streams allow for the internationalizing of applications without extensive recoding.
- Character streams are often “wrappers” for byte streams.

Using Character Streams

For most applications, I/O with character streams is no more complicated than I/O with byte streams. Input and output done with stream classes automatically translates to and from the local character set. A program that uses character streams in place of byte streams adapts to the local character set and is ready for internationalization—all without extra effort by the programmer. Even if internationalization is not a priority, you can simply use the character stream classes without paying much attention to character set issues.

Character Stream Example

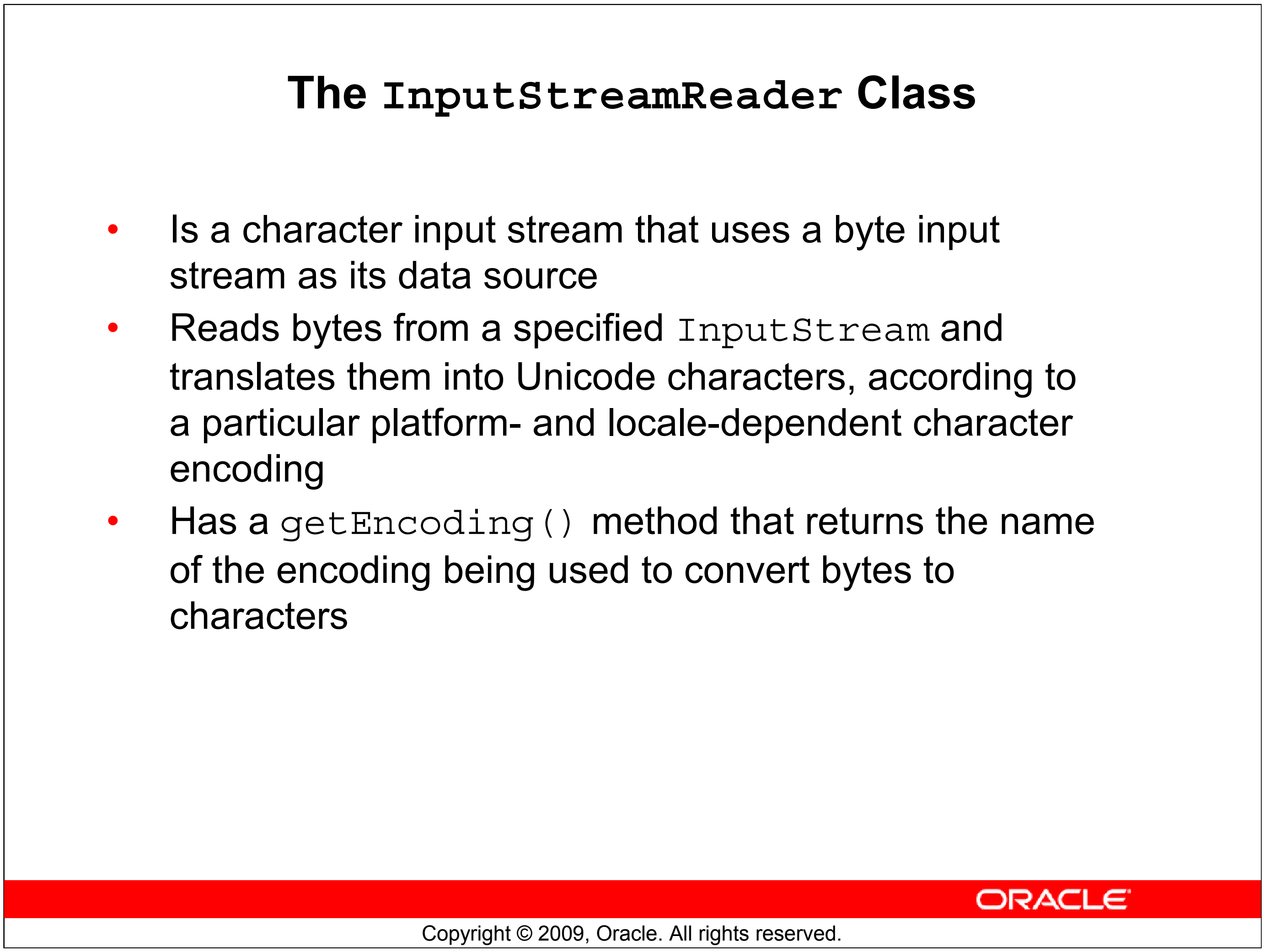
The following simple example uses the `FileWriter` class to write the string `"Hello World"` to a file called `hello.txt`.

Using Character Streams (continued)

```
import java.io.FileWriter;
public class Writer {
public static void main(String[] args) throws Exception {
String file = "hello.txt";
String text = "Hello World";
FileWriter fw = new FileWriter(file);
fw.write(text);
fw.close();
    }
}
```

Character streams are often “wrappers” for byte streams. The character stream uses the byte stream to perform the physical I/O, while the character stream handles translation between characters and bytes. There are two general-purpose, byte-to-character “bridge” streams: `InputStreamReader` and `OutputStreamWriter`. Use them to create character streams when there are no prepackaged character stream classes that meet your needs.

The `InputStreamReader` and `OutputStreamWriter` classes are discussed in the slides that follow.

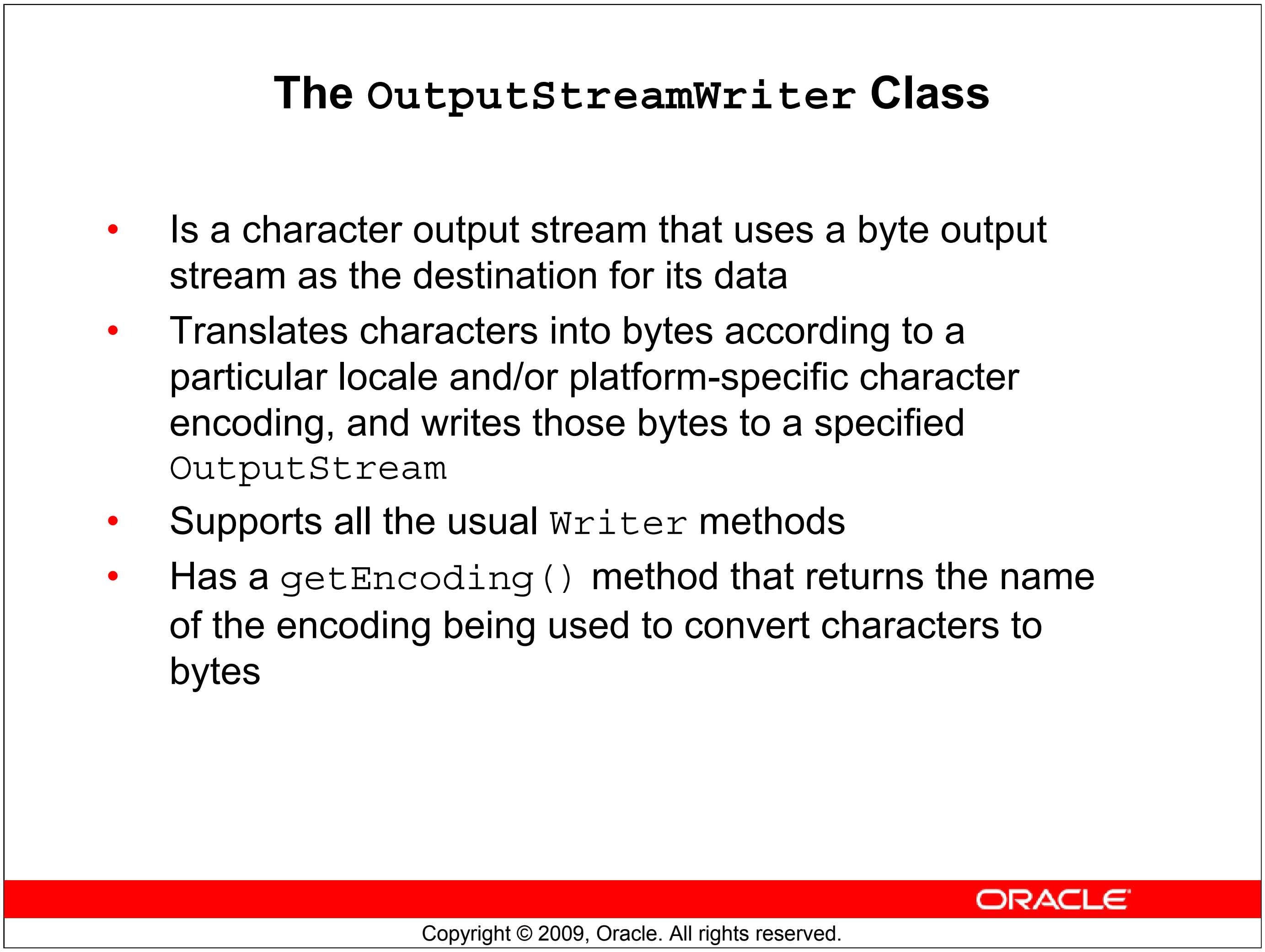


The `InputStreamReader` Class

- Is a character input stream that uses a byte input stream as its data source
- Reads bytes from a specified `InputStream` and translates them into Unicode characters, according to a particular platform- and locale-dependent character encoding
- Has a `getEncoding()` method that returns the name of the encoding being used to convert bytes to characters

The `InputStreamReader` Class

When you create an `InputStreamReader`, you specify an `InputStream` from which the `InputStreamReader` is to read bytes and, optionally, the name of the character encoding being used by those bytes. If you do not specify an encoding, the `InputStreamReader` uses the default encoding for the default locale—which is usually the correct thing to do.

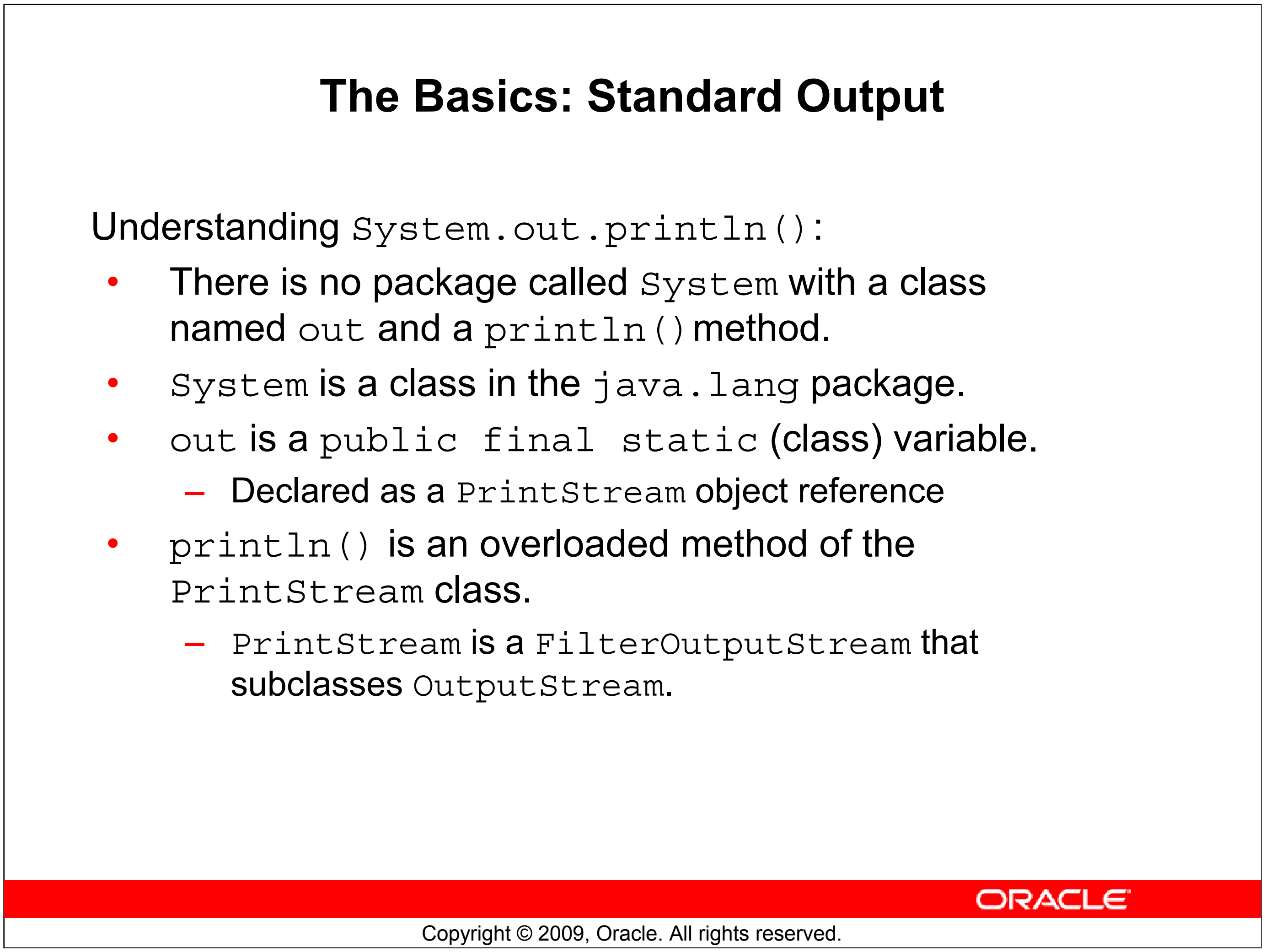


The `OutputStreamWriter` Class

- Is a character output stream that uses a byte output stream as the destination for its data
- Translates characters into bytes according to a particular locale and/or platform-specific character encoding, and writes those bytes to a specified `OutputStream`
- Supports all the usual `Writer` methods
- Has a `getEncoding()` method that returns the name of the encoding being used to convert characters to bytes

The `OutputStreamWriter` Class

When you create an `OutputStreamWriter`, you should specify the `OutputStream` to which it is to write bytes and, optionally, the name of the character encoding that should be used to convert characters to bytes. If you do not specify an encoding name, the `OutputStreamWriter` uses the default encoding for the default locale.



The Basics: Standard Output

Understanding `System.out.println()`:

- There is no package called `System` with a class named `out` and a `println()` method.
- `System` is a class in the `java.lang` package.
- `out` is a `public final static` (class) variable.
  - Declared as a `PrintStream` object reference
- `println()` is an overloaded method of the `PrintStream` class.
  - `PrintStream` is a `FilterOutputStream` that subclasses `OutputStream`.

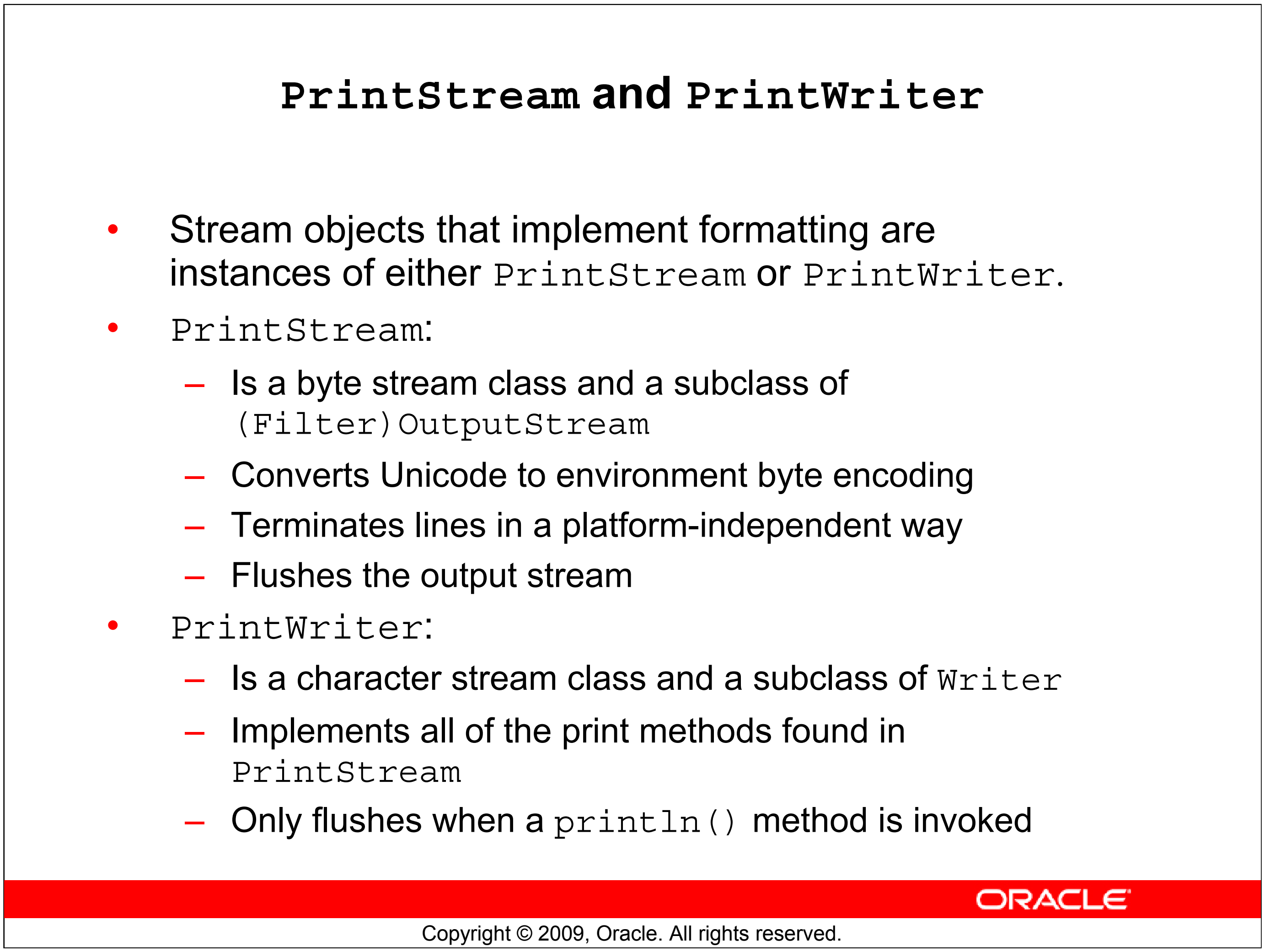
Understanding `System.out.println()`

The principal way in which your Java program has been exposed to the external environment in previous slides is through `System.out.println()`, with data sent to the console output. This is a very common use by beginning and advanced Java programmers.

Every Java program has a notion of a console to which both errors and output are redirected. If you run your program from an MS-DOS command shell by entering `java Classname`, the run-time environment binds that command window to the console and refers to the output as *standard out*.

Although Java syntax might make you think that there is a method called `println()` in the class called `out` in the package `System`, this is not the case.

Because `out` must be a variable referenced by `System.out`, it is a static variable of the `PrintStream` class. `println()` is a method in this class. The JVM auto initializes the variable and it is kept until the JVM stops. You can, therefore, refer to `System.out.println(...)`, `print(...)`, `flush(...)`, and so on.



`PrintStream` and `PrintWriter`

- Stream objects that implement formatting are instances of either `PrintStream` or `PrintWriter`.
- `PrintStream`:
  - Is a byte stream class and a subclass of `(Filter)OutputStream`
  - Converts Unicode to environment byte encoding
  - Terminates lines in a platform-independent way
  - Flushes the output stream
- `PrintWriter`:
  - Is a character stream class and a subclass of `Writer`
  - Implements all of the print methods found in `PrintStream`
  - Only flushes when a `println()` method is invoked

`PrintStream` and `PrintWriter`

You have already seen that the `OutputStream` class provides basic methods for low-level byte I/O operations. Using an `OutputStream` object is not very useful for printing text terminated by a new line in a platform-independent way. However, `PrintStream`, a subclass of `FilterOutputStream`, extends the basic I/O capabilities of `OutputStream` in the following ways:

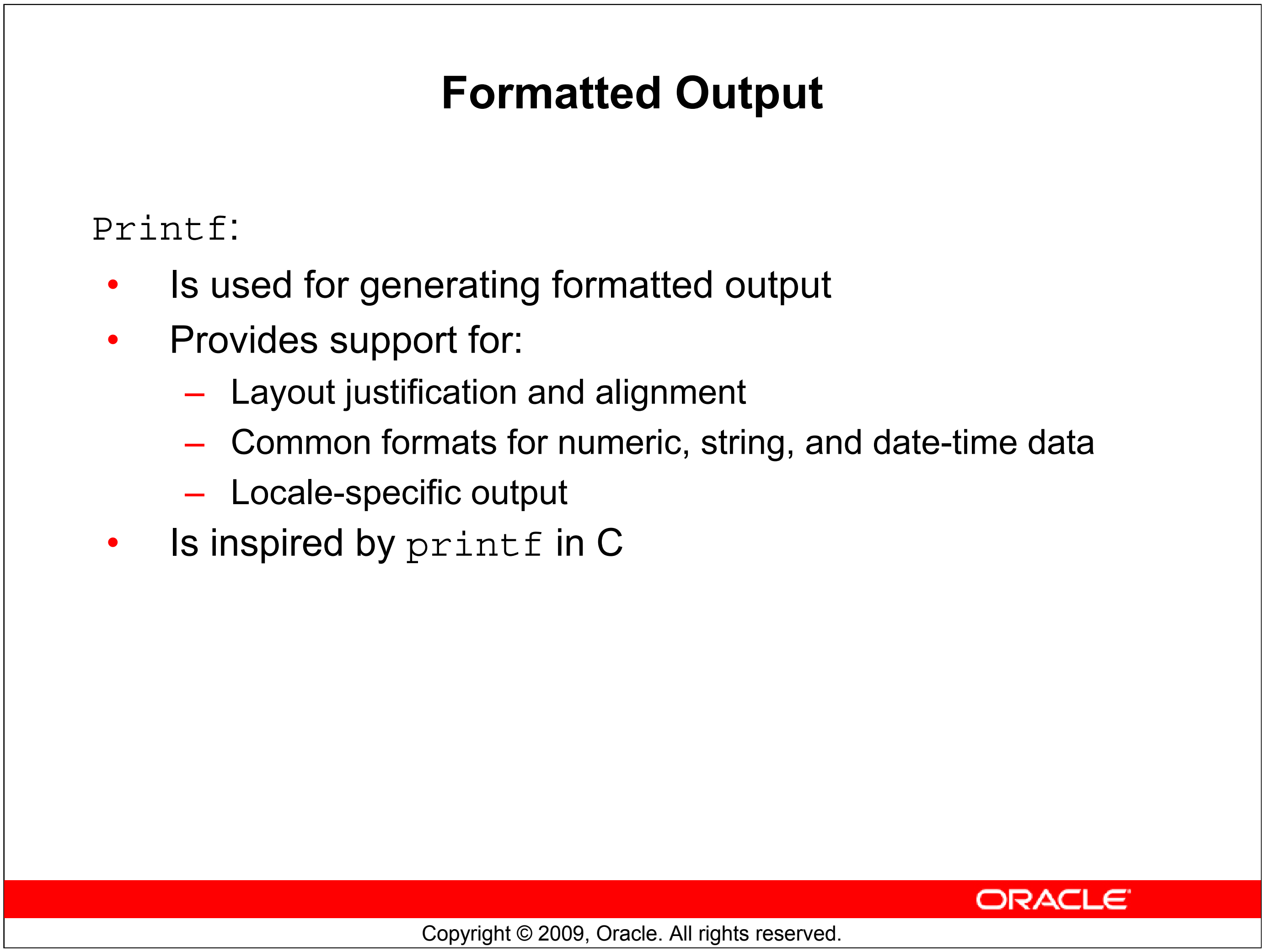
- It converts Java Unicode strings into the byte encoding of the environment so that you can see the text in a readable format.
- It terminates a line in a platform-independent way. In JDK releases before 1.2, new lines written were not platform independent.
- It flushes the stream. In general, there is no guarantee that data written will be visible immediately. Flushing the stream explicitly requests that the data be displayed *now*.

`PrintWriter` provides an implementation of the abstract `Writer` class specific to printing various primitive types and other objects to an output stream.

- It implements all of the print methods found in `PrintStream` but it does not contain methods for writing raw bytes, for which a program should use unencoded byte streams.

`PrintStream` and `PrintWriter` (continued)

- Unlike with the `PrintStream` class, automatic flushing, if enabled, will be done only when one of the `println()` methods is invoked, rather than whenever a newline character happens to be output. The `println()` methods use the platform's own notion of line separator rather than the newline character.
- None of the methods in this class throw an exception. However, errors can be detected by calling the `checkError()` method.



Formatted Output

`Printf`:

- Is used for generating formatted output
- Provides support for:
  - Layout justification and alignment
  - Common formats for numeric, string, and date-time data
  - Locale-specific output
- Is inspired by `printf` in C

Formatted Output

The `Formatter` class in Java SE 5.0 provides an interpreter for `printf`-style format strings. The class provides support for layout justification and alignment, common formats for numeric, string, and date-time data, and locale-specific output. Common Java types such as `byte`, `BigDecimal`, and `Calendar` are supported.

Formatted printing for the Java language is heavily inspired by C's `printf`. Although the format strings are similar to C, some customizations have been made to accommodate the Java language and exploit some of its features. Also, Java formatting is more strict than C's; for example, if a conversion is incompatible with a flag, an exception is thrown. In C, inapplicable flags are silently ignored. The format strings are thus intended to be recognizable to C programmers but not necessarily completely compatible with those in C.

The syntax to invoke the `printf()` method is:

```
System.out.printf(format, item1, item2, …itemk)
```

where format is a string that may consist of substrings and format specifiers.

Format Specifiers

- A format specifier specifies how an item should be displayed.
- An item may be a numeric value, a character, a Boolean value or a string.
- Each format specifier begins with a percent sign.

ORACLE

Format Specifiers

Frequently Used Specifiers

Specifier	Output	Example
`%b`	a Boolean value	true or false
`%c`	a character	“a”
`%d`	a decimal integer	200
`%f`	a floating-point number	45.460000
`%e`	a number in standard scientific notation	4.556000e+01
`%s`	a string	“Java is cool”

Format Specifiers (continued)

Format Specifier Example:

```
    int count = 5;
    double amount = 45.56;
    System.out.printf("Count is %d and amount is %f", count,
                                                   amount);
```

The output is:

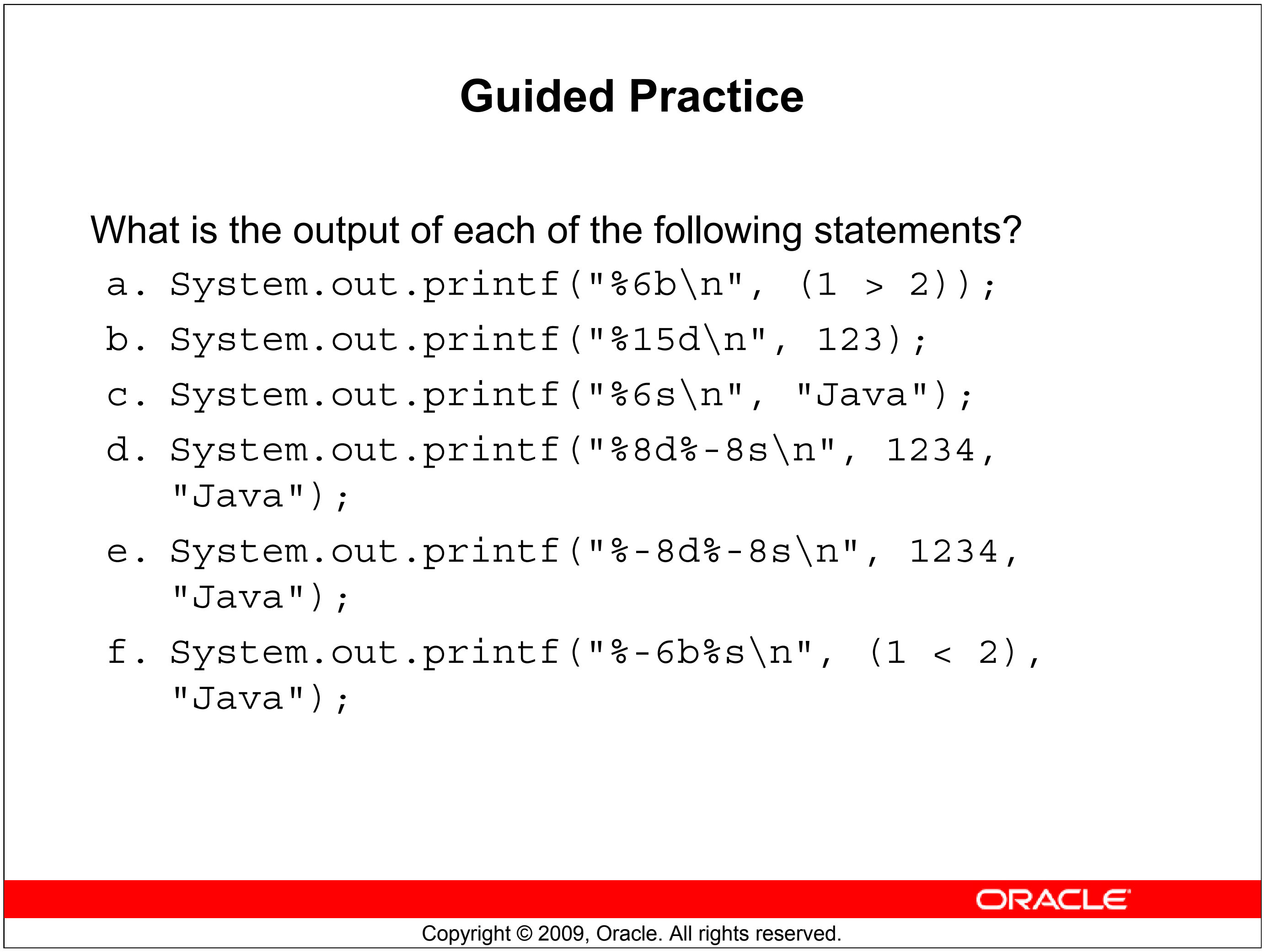
```
Count is 5 and amount is 45.560000
```

Items must match the specifier in order, in number, and in exact type. In the example, the specifier for `count` is `%d` and for `amount` is `%f`. By default, a floating point value is displayed with six digits after the decimal point.

Note: You can put a minus sign (-) in the specifier to indicate that the item is left-justified in the output within the specified field.

Examples of Specifying Width and Precision

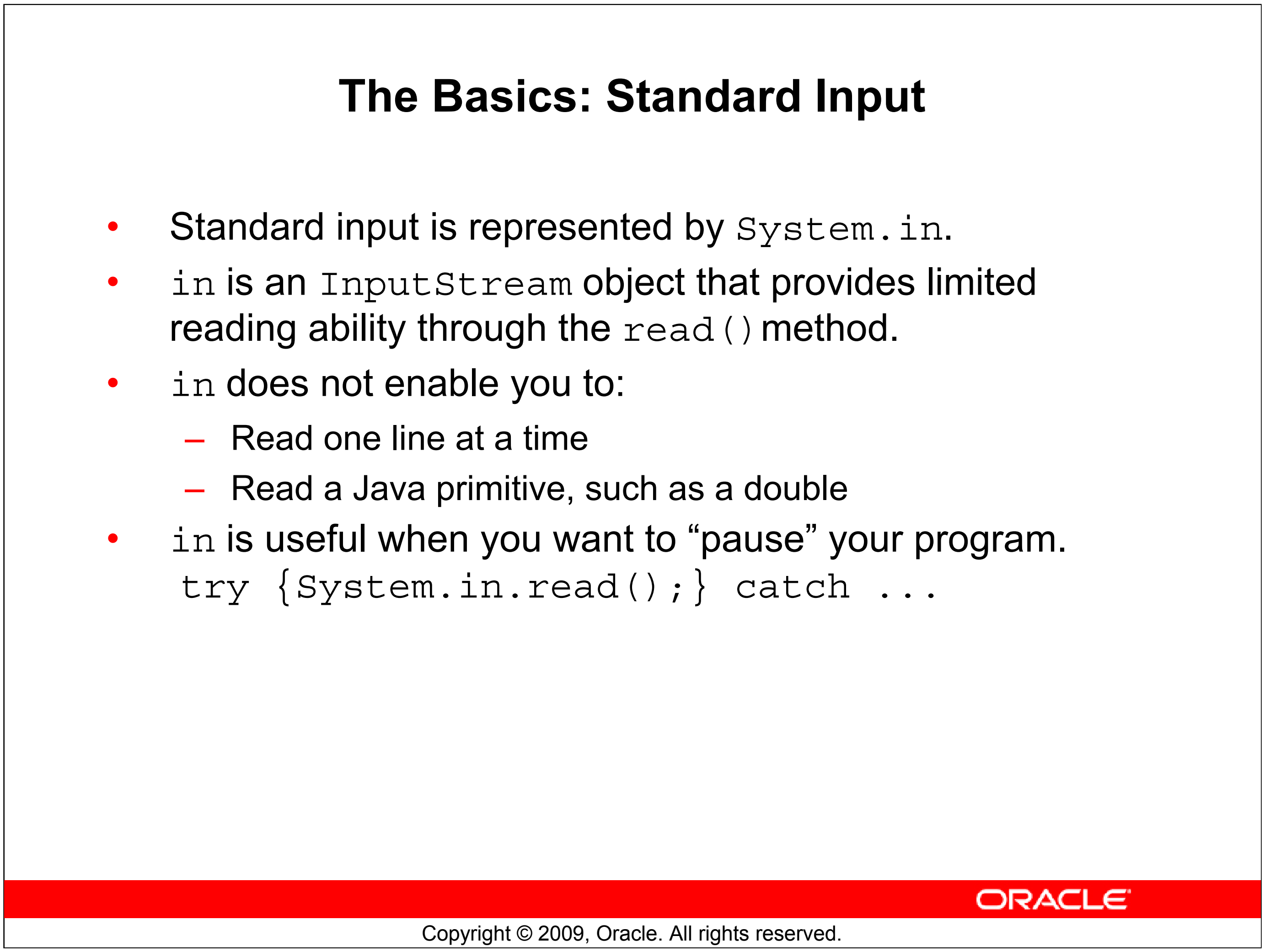
Example	Output
`%5c`	Output the character and add four spaces before the character item.
`%5d`	Output the integer with width at least 5. If the number of digits in the item is <5, add spaces before the number. If the number of digits in the item is >5, the width is automatically increased.
`%6b`	Output the Boolean value and add one space before the false value and two spaces before the true value.
`%12s`	Output the string with width at least 12 characters. If the string has less than 12 characters, add spaces before the string. If the string item has more than 12 characters, the width is automatically increased.



Guided Practice

What is the output of each of the following statements?

```
a. System.out.printf("%6b\n", (1 > 2));
b. System.out.printf("%15d\n", 123);
c. System.out.printf("%6s\n", "Java");
d. System.out.printf("%8d%-8s\n", 1234,
   "Java");
e. System.out.printf("%-8d%-8s\n", 1234,
   "Java");
f. System.out.printf("%-6b%s\n", (1 < 2),
   "Java");
```



The Basics: Standard Input

- Standard input is represented by `System.in`.
- `in` is an `InputStream` object that provides limited reading ability through the `read()` method.
- `in` does not enable you to:
  - Read one line at a time
  - Read a Java primitive, such as a double
- `in` is useful when you want to “pause” your program.

```
try {System.in.read();} catch ...
```

Wrapping Streams

To understand wrapping, you need to look no further than `System.in`, the equivalent of `System.out`, to read from the console. There is no direct correlation between the two in spite of the apparent similarity.

`in` is a static instance variable in `System`, just as `out` is an instance of the `OutputStream` class. Unlike `out`, `in` provides limited reading ability. For instance, you cannot read a line at a time using `in`. This implies that you cannot easily read a Java `double`, such as `3.1415`, entered at the command line. The main reason is that `in` lets you read only one byte at a time using the `read()` method. You would have to read first the `3`, then the decimal point, then `1`, and so on until you reached a white space in order to obtain the value entered by the user. To read more effectively, you must wrap `System.in` by another stream that provides more facilities (as shown in the following slide).

Wrapping Streams (continued)

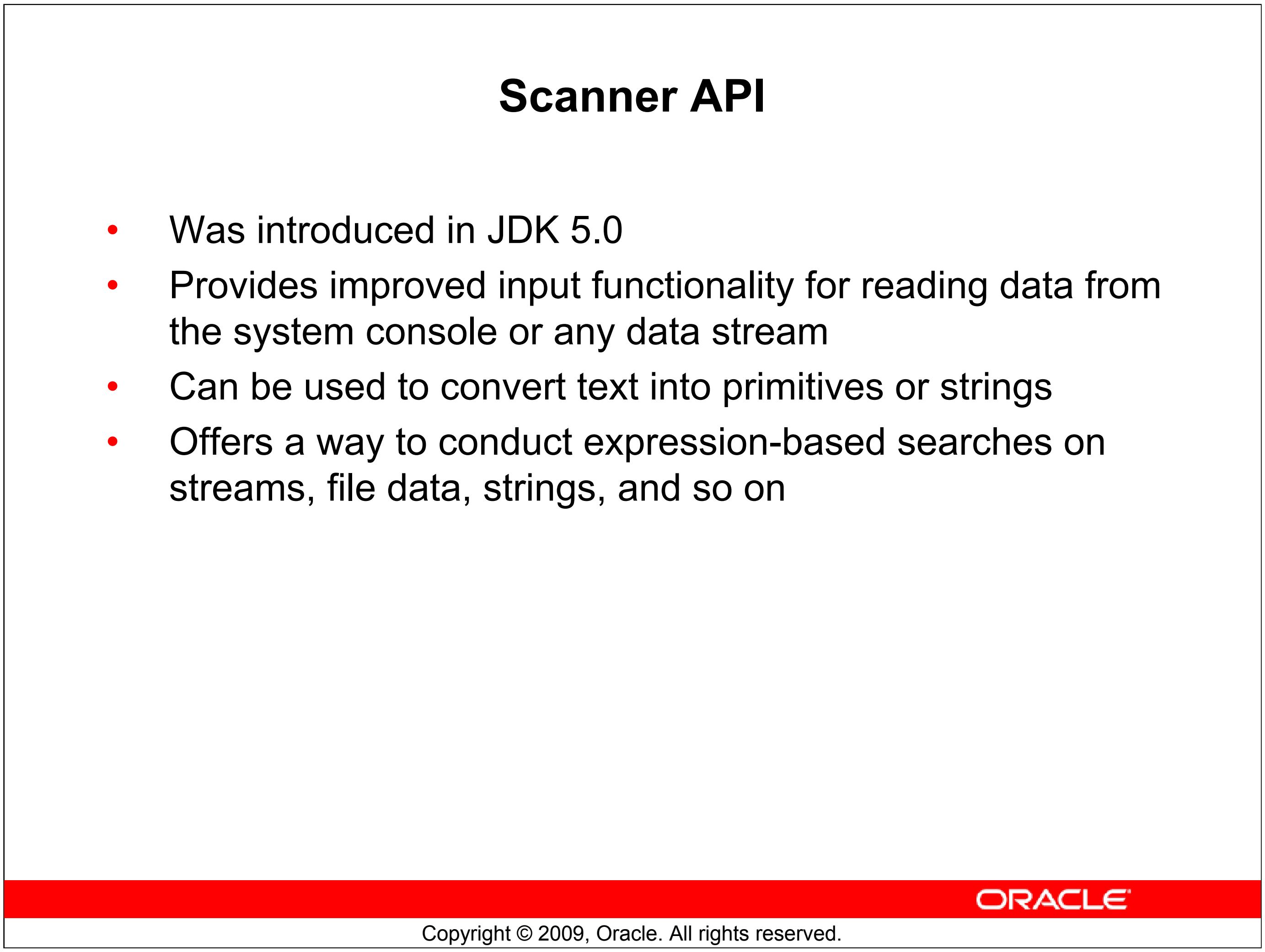
Using `System.in` Without Wrapping

`System.in`, however, is extremely useful for debugging purposes when you want your program to pause (for whatever reason). This is achieved by the following sequence:

```
    try { System.in.read(); } catch (IOException e) {...}
```

When used in this way, the `System.in` stream is “unwrapped.”

Note that methods in the `InputStream` class throw exceptions, unlike methods in the `PrintWriter` and `PrintStream` classes.



Scanner API

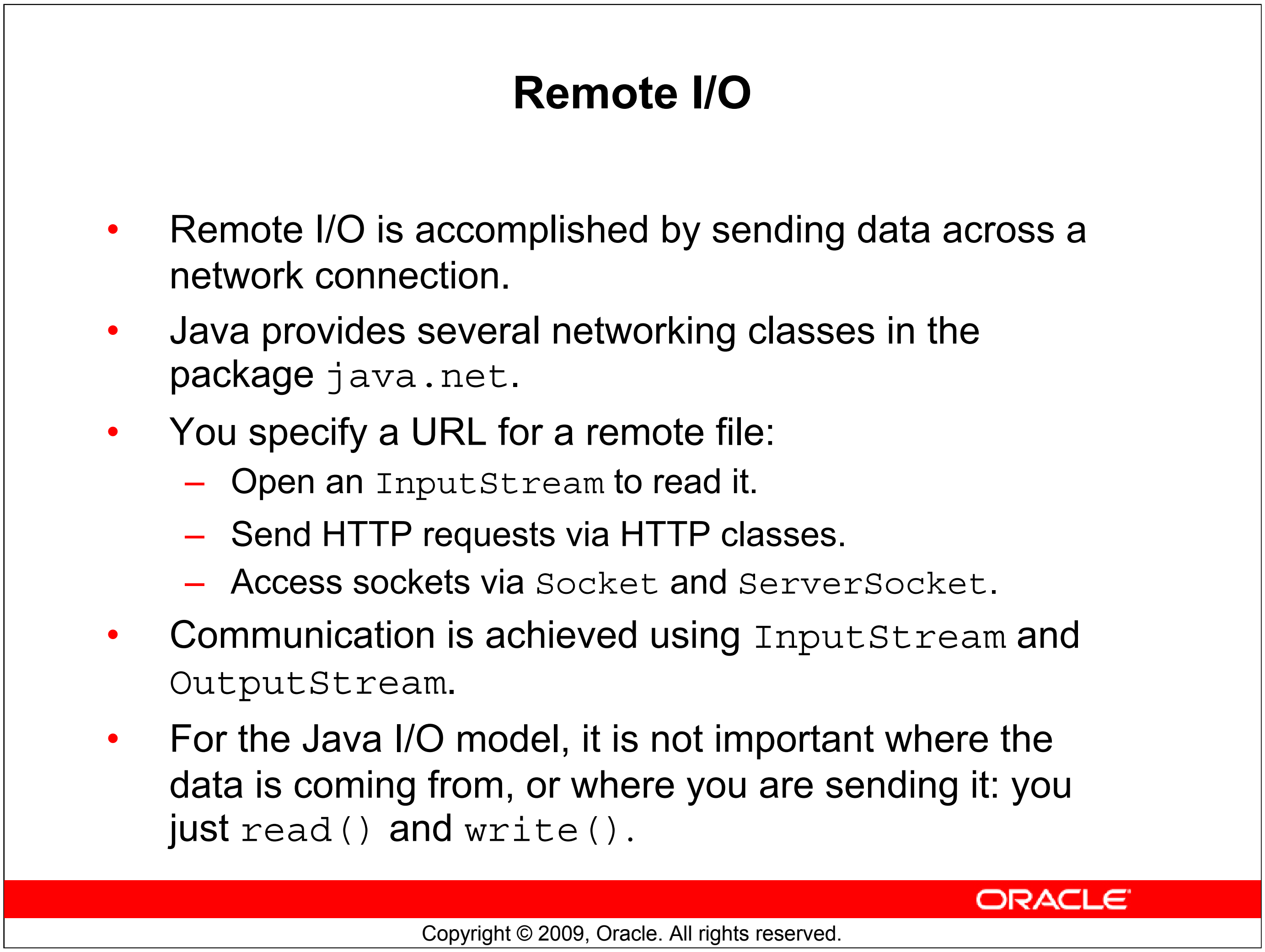
- Was introduced in JDK 5.0
- Provides improved input functionality for reading data from the system console or any data stream
- Can be used to convert text into primitives or strings
- Offers a way to conduct expression-based searches on streams, file data, strings, and so on

Scanner API

The Scanner API provides basic input functionality for reading data from the system console or any data stream. The following example reads a string from standard input and expects a following `int` value.

Scanner methods such as `next` and `nextInt` will block if no data is available. If you need to process more complex input, there are also pattern-matching algorithms available from the `java.util.Formatter` class.

```
Scanner s = new Scanner(System.in);
     String param = s.next();
     int value = s.nextInt();
     s.close();
```



Remote I/O

- Remote I/O is accomplished by sending data across a network connection.
- Java provides several networking classes in the package `java.net`.
- You specify a URL for a remote file:
  - Open an `InputStream` to read it.
  - Send HTTP requests via HTTP classes.
  - Access sockets via `Socket` and `ServerSocket`.
- Communication is achieved using `InputStream` and `OutputStream`.
- For the Java I/O model, it is not important where the data is coming from, or where you are sending it: you just `read()` and `write()`.

Remote I/O

Java is the first programming language to provide as much support for network I/O as it does for file I/O. Network I/O relies primarily on the basic `InputStream` and `OutputStream` methods, which you can wrap with any higher-level stream that suits your needs: buffering, cryptography, compression, or whatever your application requires.

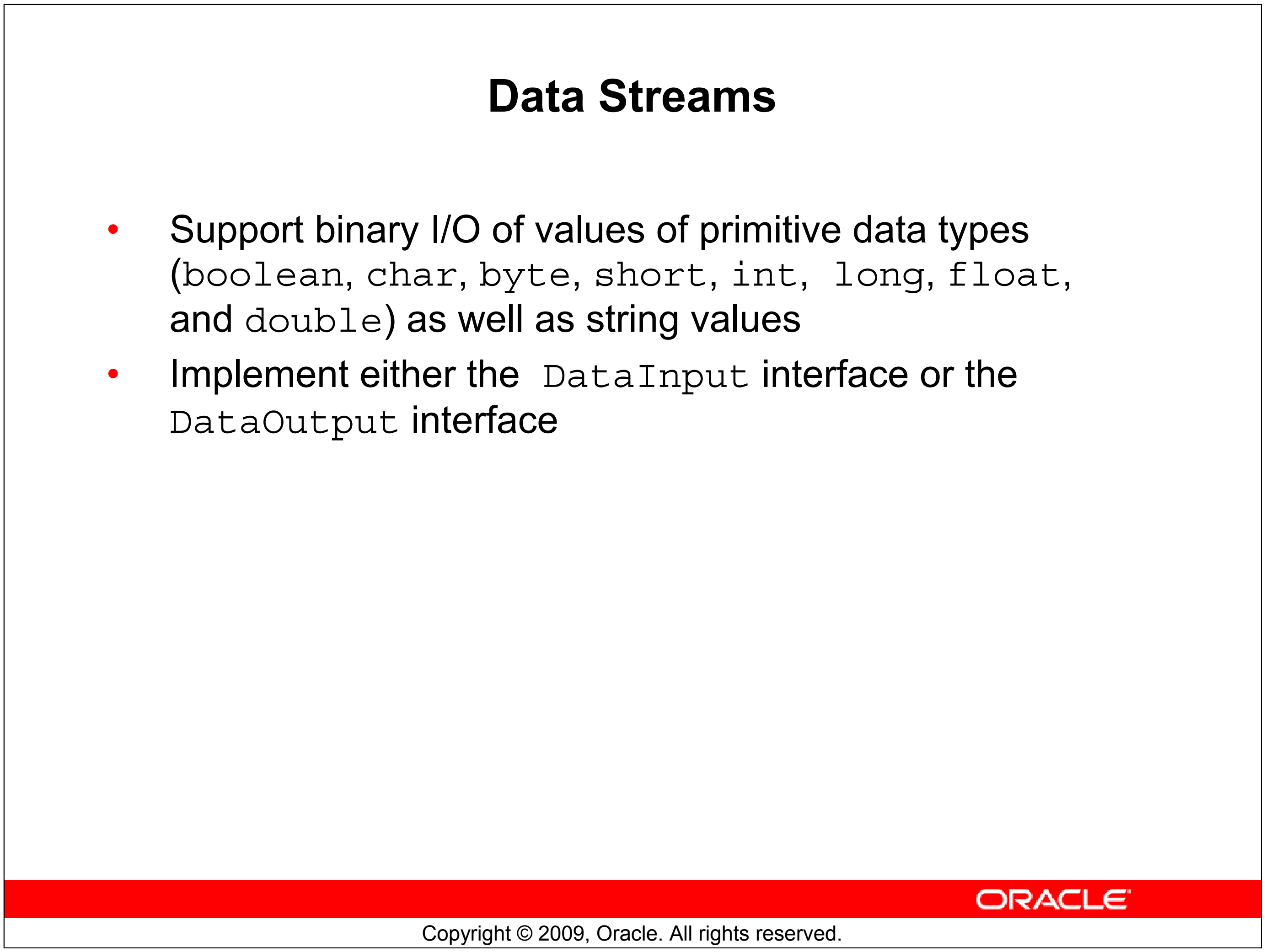
Network I/O Example

The following example reads the first ten lines of the JDeveloper and ADF 11g New Features page on OTN.

```
import java.net.*;
import java.io.*;

public class WebRead  {
  public static void main (String[] args) throws Exception {
    URL u = new
   URL("http://www.oracle.com/technology/products/jdev/
        collateral/papers/11/newfeatures/index.html");
    URLConnection c = u.openConnection();
```

Remote I/O (continued)

```
 InputStreamReader reader = new
      InputStreamReader(c.getInputStream());
 BufferedReader otnpage = new BufferedReader(reader);


 for (int i = 0, i<10, i++)
System.out.println(otnpage.readLine());
   }
}
```



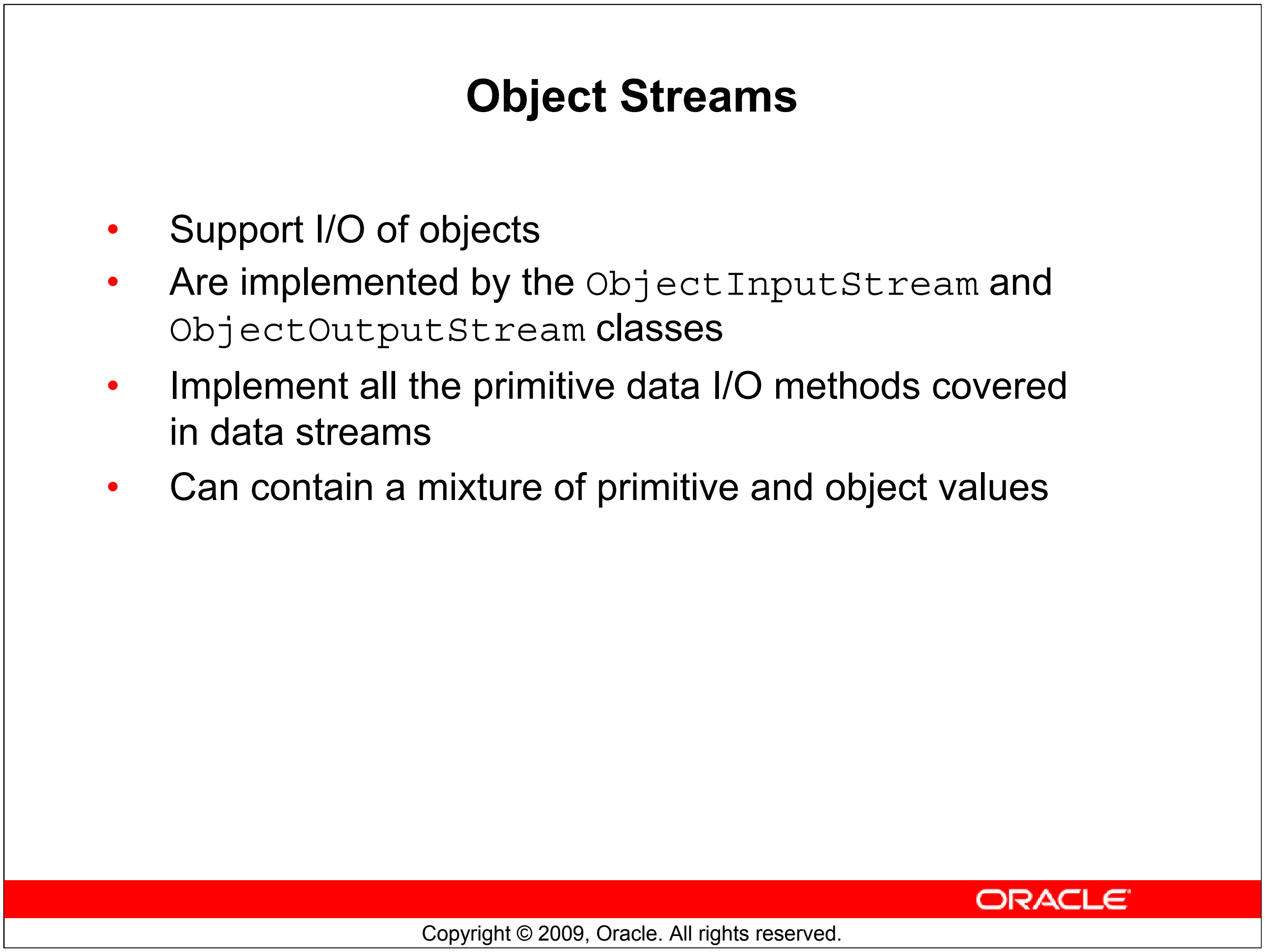
Data Streams

- Support binary I/O of values of primitive data types (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, and `double`) as well as string values
- Implement either the `DataInput` interface or the `DataOutput` interface

Data Streams

You can store primitive values by writing to a file with `PrintStream`; reading those values in requires that they be parsed from `Strings`, however, which is inefficient. A better way to deal with primitive values is to write binary data to the file. The Java core APIs define the interfaces `DataInput`() and `DataOutput ()` to provide methods that will write a binary representation of a primitive value. These interfaces define methods such as:

- `readInt()/writeInt()`
- `readFloat()/writeFloat()`
- `readUTF()/writeUTF()`

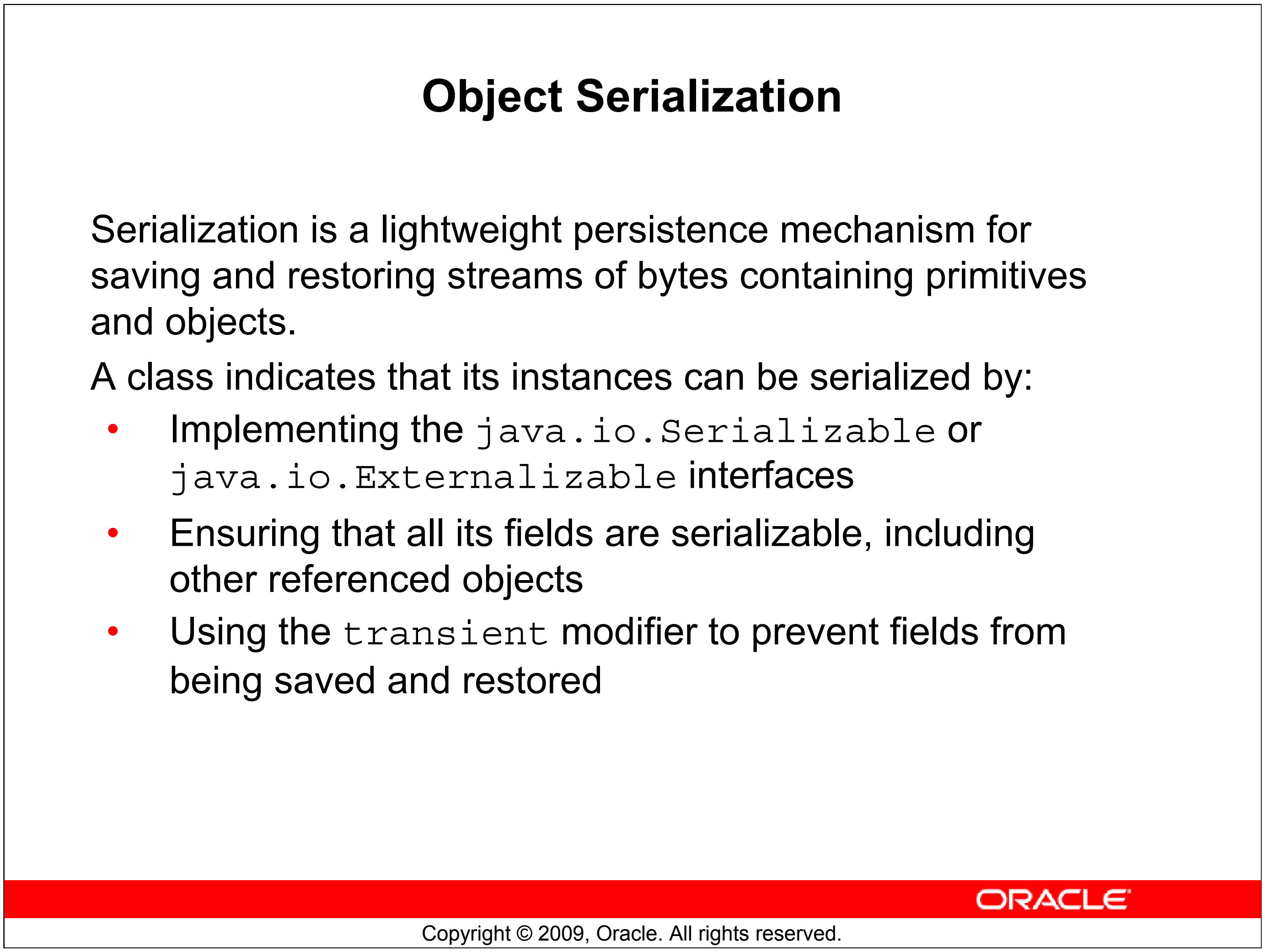


Object Streams

- Support I/O of objects
- Are implemented by the `ObjectInputStream` and `ObjectOutputStream` classes
- Implement all the primitive data I/O methods covered in data streams
- Can contain a mixture of primitive and object values

Object Streams

Just as data streams support I/O of primitive data types, object streams support I/O of objects. The object streams `ObjectInputStream` and `ObjectOutputStream` allow you to read and write objects: when you use `writeObject()` to write an object to an `ObjectOutputStream`, bytes representing the object—including all other objects that it references—are written to the stream. This process of transforming an object into a stream of bytes is called *serialization*. Because the serialized form is expressed in bytes, not characters, the object streams have no `Reader` or `Writer` forms. Object serialization is discussed in the next slides.



Object Serialization

Serialization is a lightweight persistence mechanism for saving and restoring streams of bytes containing primitives and objects.

A class indicates that its instances can be serialized by:

- Implementing the `java.io.Serializable` or `java.io.Externalizable` interfaces
- Ensuring that all its fields are serializable, including other referenced objects
- Using the `transient` modifier to prevent fields from being saved and restored

Object Serialization

Object serialization is the process of encoding an object and the objects it references into a stream of bytes. Object serialization also provides mechanisms for reconstructing the object and its referenced objects from the stream.

Serialization can be used for lightweight persistence (for example, permanent storage in file on disk) or for communication between distributed Java applications.

The object saved and the relationship it has with other objects (via object references) is called an object graph. When an object is saved and restored, the objects it references must also be maintained. By default, when an object is stored, all the objects that are reachable from that object are stored as well—that is, the object graph is stored.

For an object to be saved to and restored from a stream, its class can implement one of the following:

- `java.io.Serializable` interface
- `java.io.Externalizable` interface

Only the identity of the class of an `Externalizable` instance is written in the serialization stream. It is the responsibility of the class to save and restore the contents of its instances.

Object Serialization (continued)

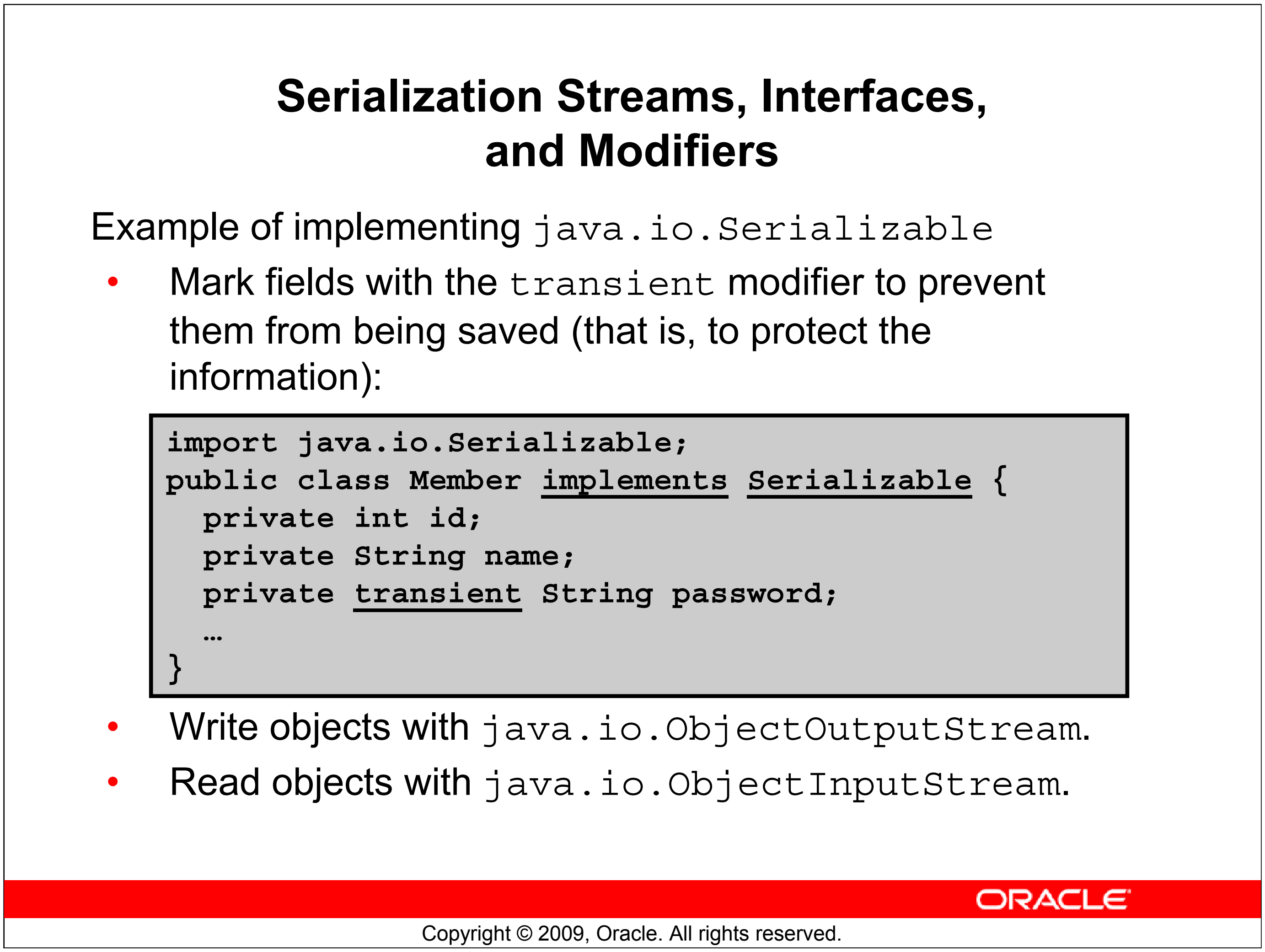
Serialization: Example

```
package serdemo;
import java.io.ObjectInputStream;
import java.io.FileInputStream;
import java.io.Serializable;
import java.io.ObjectOutputStream;
import java.io.FileOutputStream; public class
SerializationDemo  {
   public static void main(String[] args) {
   Person p1 = new Person("John", 'M', null);
   Person p2 = new Person("Mary", 'F', p1);
   p1.setSpouse(p2);
   try{
     ObjectOutputStream os = new ObjectOutputStream(new
FileOutputStream("person_graph.ser") );
     os.writeObject(p1); //entire object graph is written
     os.close();
     ObjectInputStream is = new ObjectInputStream(new
FileInputStream("person_graph.ser") );
     Person p3 = (Person) is.readObject(); //entire object graph
is read
     is.close();
     System.out.println(p3);
     Person p4 = p3.getSpouse(); //Object obtained from graph.
     System.out.println(p4);
   }
   catch(Exception ioe){
     ioe.printStackTrace();
   }
  }//end main()
}
class Person implements Serializable{
 private String name;
  private char gender;
  private Person spouse;
```

Object Serialization (continued)

Serialization: Example (continued)

```
  Person(){
  }
  Person(String name, char gender, Person spouse){
    setName(name);
    setGender(gender);
    setSpouse(spouse);
  }
   public String getName() {
    return name;
  }

  public void setName(String newName) {
    name = newName;
  }

  public char getGender() {
    return gender;
  }

  public void setGender(char newGender) {
    gender = newGender;
  }

  public Person getSpouse() {
    return spouse;
  }

  public void setSpouse(Person newSpouse) {
    spouse = newSpouse;
  }
  public String toString(){
    return "\nName=" + name
          + "\nGender=" + gender
          + "\nSpouse=" + getSpouse().getName();
  }
}
```



Serialization Streams, Interfaces, and Modifiers

Example of implementing `java.io.Serializable`

- Mark fields with the `transient` modifier to prevent them from being saved (that is, to protect the information):

```
import java.io.Serializable;
public class Member implements Serializable {
  private int id;
  private String name;
  private transient String password;
  …
}
```

- Write objects with `java.io.ObjectOutputStream`.
- Read objects with `java.io.ObjectInputStream`.

Serialization Streams, Interfaces, and Modifiers

As already stated, for an object to be serialized, its class must implement the `Serializable` interface (as shown in the slide) or the `Externalizable` interface. Implementing the `Serializable` interface does not require that you write any methods and acts as a marker to the Java serialization system that the object can be serialized.

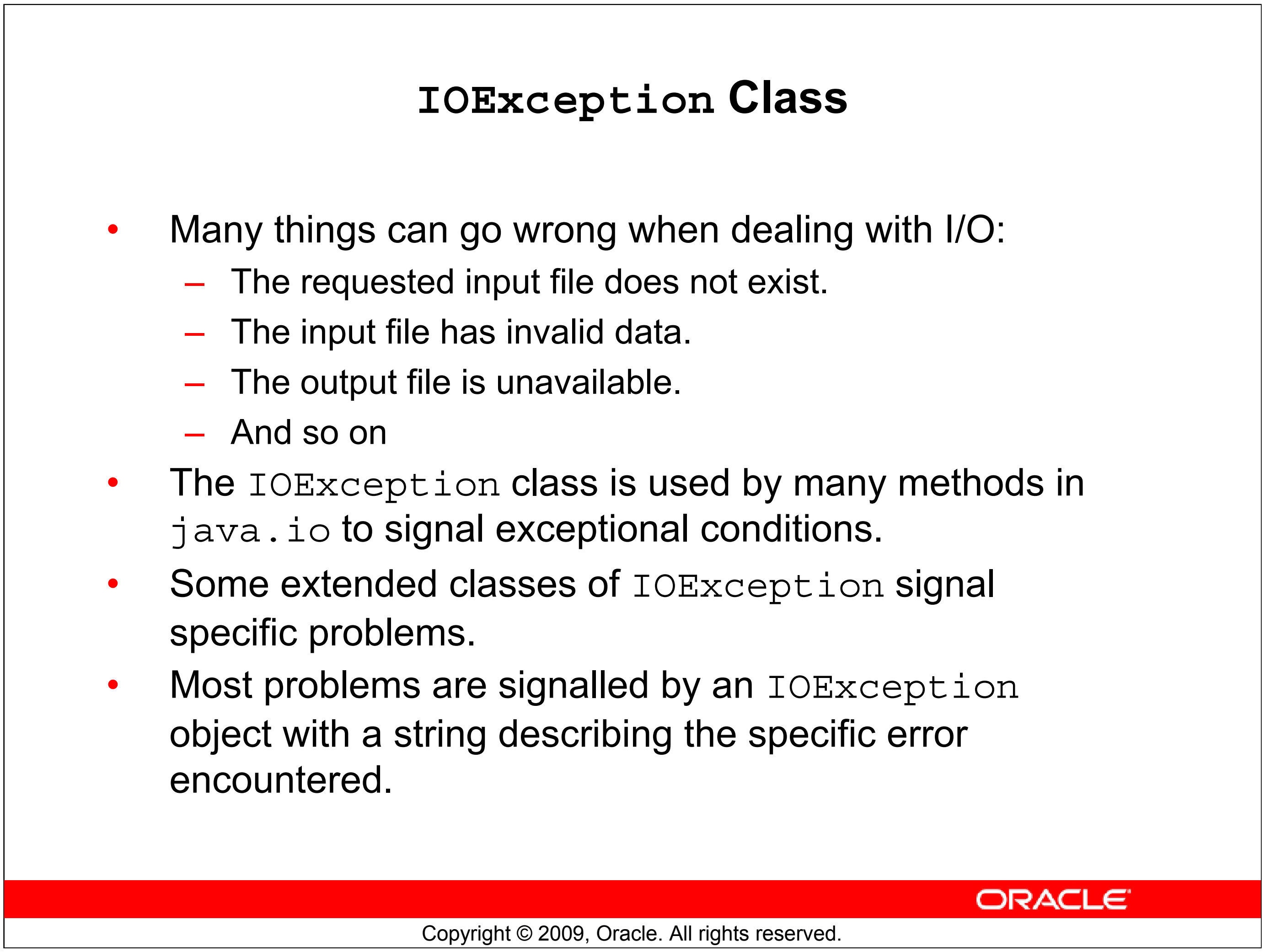
Use the `transient` modifier for variables whose values you do not want saved when the object contained is serialized (for example, to prevent sensitive information from being stored). However, if a variable in your serializable object references another object that is *not* serializable, that variable *must* be made transient; otherwise, serialization fails. The `java.io.NotSerializableException` exception is thrown if serialization fails.

Stream for Writing Objects

Writing objects to a stream is accomplished by using the `ObjectOutputStream` class, whose constructor accepts another `OutputStream`. Thus, wrapping another stream in an `ObjectOutputStream` is quite common. For example, wrapping a `FileOutputStream` in an `ObjectOutputStream` stores objects in a file.

Stream for Reading Objects

Reading objects from a stream is possible by using the `ObjectInputStream` class whose constructor accepts another `InputStream`.



IOException Class

- Many things can go wrong when dealing with I/O:
  - The requested input file does not exist.
  - The input file has invalid data.
  - The output file is unavailable.
  - And so on
- The IOException class is used by many methods in java.io to signal exceptional conditions.
- Some extended classes of IOException signal specific problems.
- Most problems are signalled by an IOException object with a string describing the specific error encountered.

IOException Class

The Java core libraries define several I/O exceptions, all of which extend a class named IOException. IOException directly extends Exception; that means that it is *not* a "run-time exception." As a result, you must *always* either provide trycatch blocks around *any* I/O code, or add a throws clause to the method that performs the I/O. Lesson 14 deals with throwing and catching exceptions.

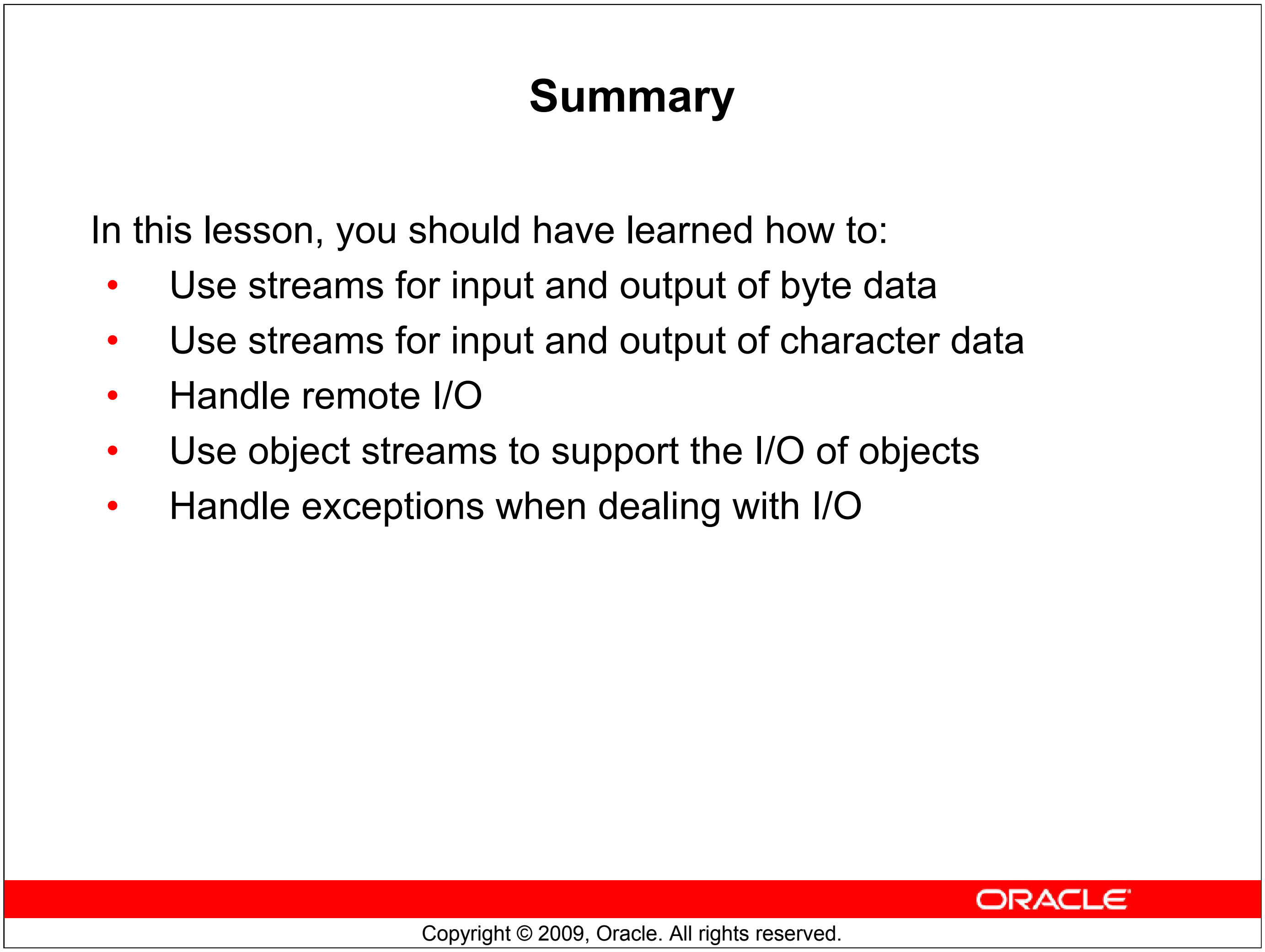
java.io Exceptions

- CharConversionException: A problem occurred while converting a char to one or more bytes, or vice versa.
- EOFException: The end of a stream was unexpectedly reached.
- FileNotFoundException: A file specified in a constructor to FileInputStream, FileOutputStream, FileReader, FileWriter, or RandomAccessFile does not exist.
- InterruptedIOException: An I/O operation was halted because the thread performing the operation was terminated.
- InvalidClassException: Thrown during a Serialization operation because a serialized object does not match the class that the run time sees, or because necessary types or constructors were not present

The `IOException` Class (continued)

`java.io` Exceptions (continued)

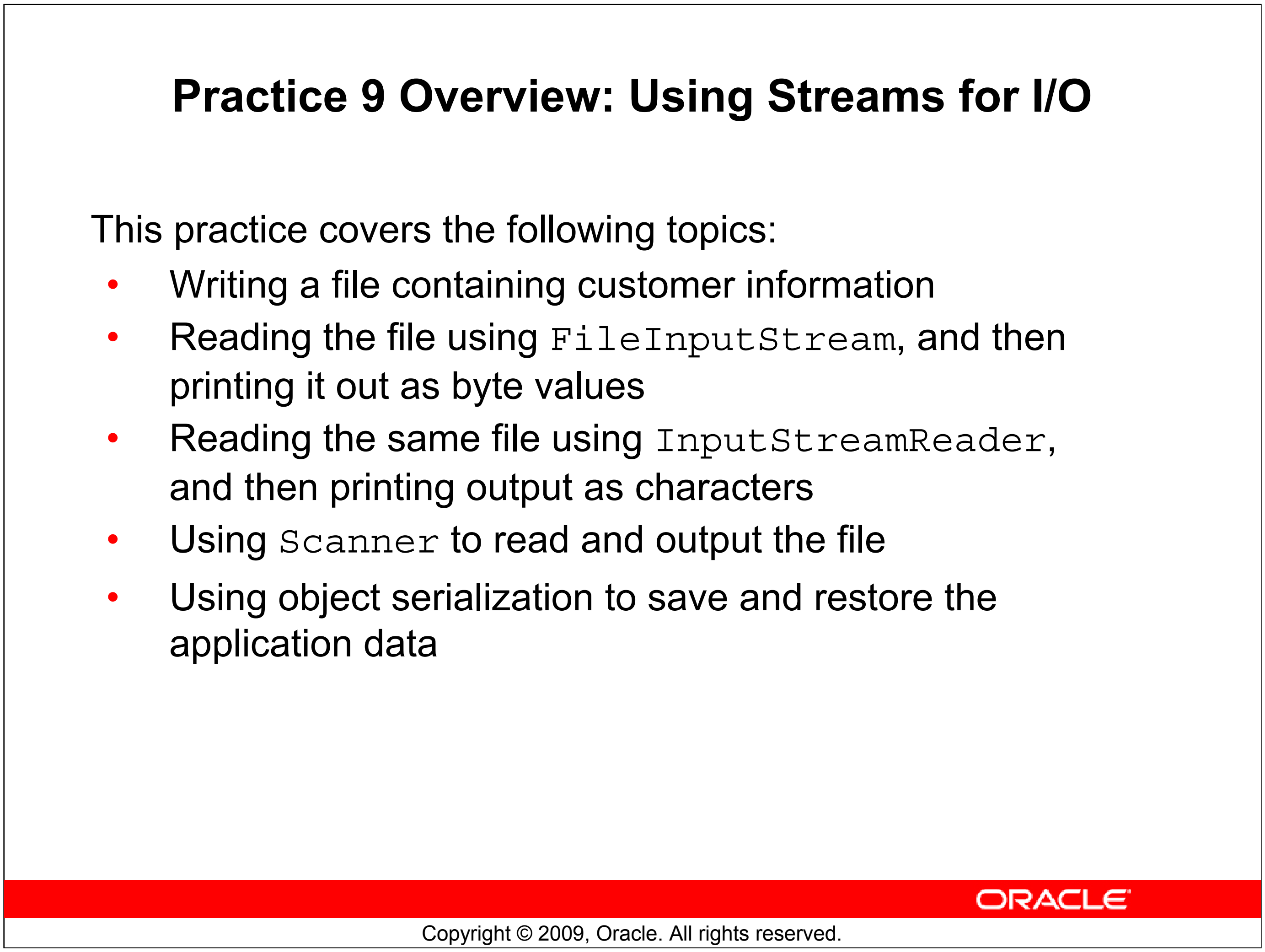
- `InvalidObjectException`: Thrown during serialization if a deserialized class fails validation tests.
- `IOException`: The general I/O exception class. All other I/O exceptions extend this.
- `NotActiveException`: Thrown if serialization support methods are called outside the scope of the valid serialization methods
- `NotSerializableException`: Thrown if an object that is being serialized does not implement `Serializable`
- `ObjectStreamException`: A common class for all exceptions related to serialization
- `OptionalDataException`: Thrown when unexpected data is encountered when trying to deserialize an object
- `StreamCorruptedException`: Thrown when a serialized object stream is corrupted and no other sense can be made of it
- `SyncFailedException`: Thrown if a file's output content could not be actually written to the physical file system
- `UnsupportedEncodingException`: A specific character encoding is not supported.
- `UTFDataFormatException`: An attempt to read a UTF-8 stream failed due to malformed UTF-8 content.
- `WriteAbortedException`: Thrown when *reading* a serialized object stream if the stream was only partially written because an `ObjectStreamException` was thrown when *writing* it



Summary

In this lesson, you should have learned how to:

- Use streams for input and output of byte data
- Use streams for input and output of character data
- Handle remote I/O
- Use object streams to support the I/O of objects
- Handle exceptions when dealing with I/O



Practice 9 Overview: Using Streams for I/O

This practice covers the following topics:

- Writing a file containing customer information
- Reading the file using `FileInputStream`, and then printing it out as byte values
- Reading the same file using `InputStreamReader`, and then printing output as characters
- Using `Scanner` to read and output the file
- Using object serialization to save and restore the application data

Practice 9 Overview: Using Streams for I/O

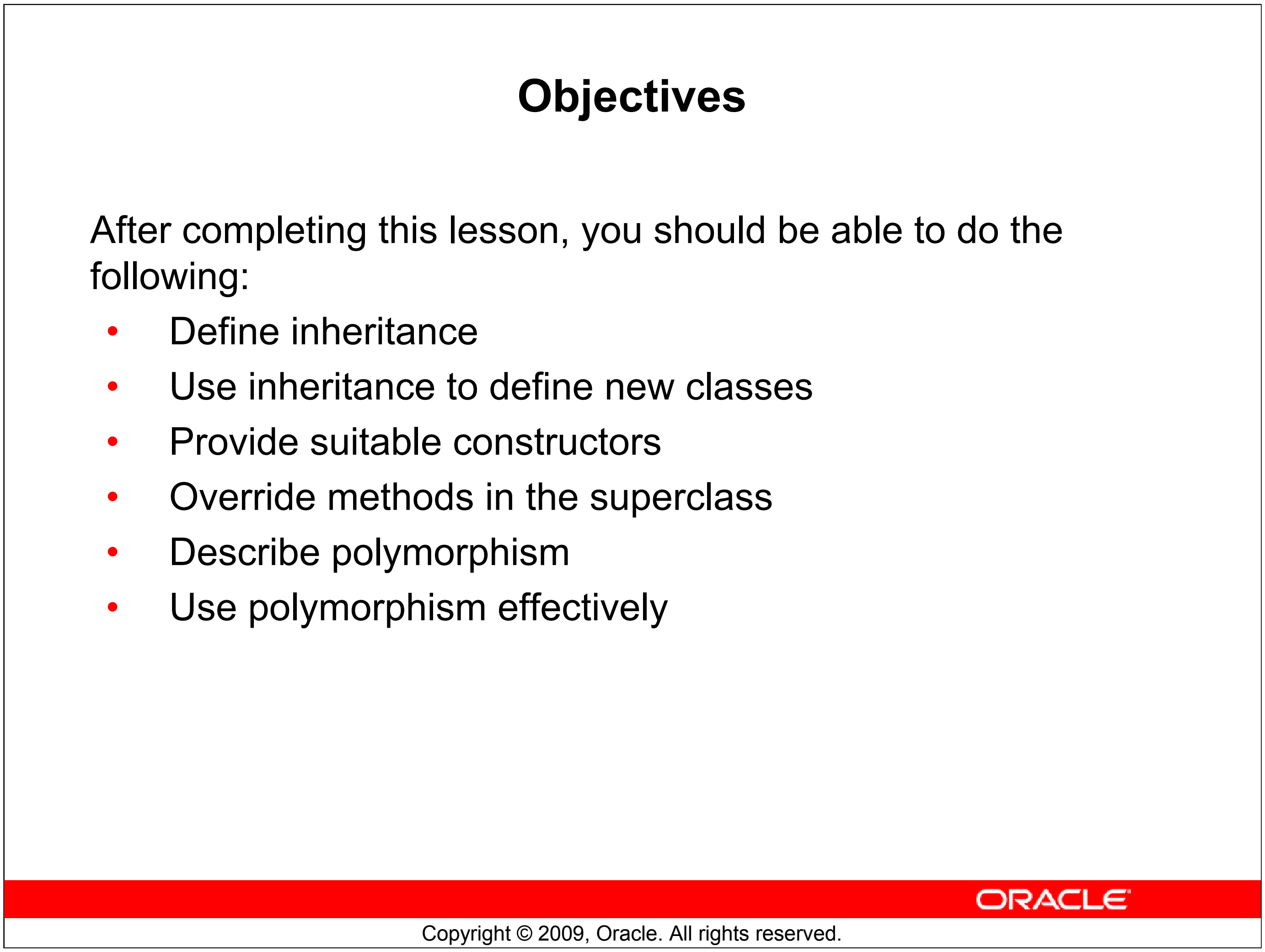
The goal of this practice is to use byte- and character-based streams to read and write application data. You also use Object Serialization to save and restore.

Note: If you have successfully completed the previous practice, continue using the same directory and files. If the compilation from the previous practice was unsuccessful and you want to move on to this practice, change to the `les09` directory, load the `OrderEntryApplicationLes09` application, and continue with this practice.

Viewing the model: To view the course application model up to this practice, load the `OrderEntryApplicationLes09` application. In the Applications – Navigator node, expand `OrderEntryApplicationLes09 – OrderEntryProjectLes09 - Application Sources – oe` and double-click the `UML Class Diagram1` entry. This diagram displays all the classes created up to this point in the course.

10

Inheritance and Polymorphism



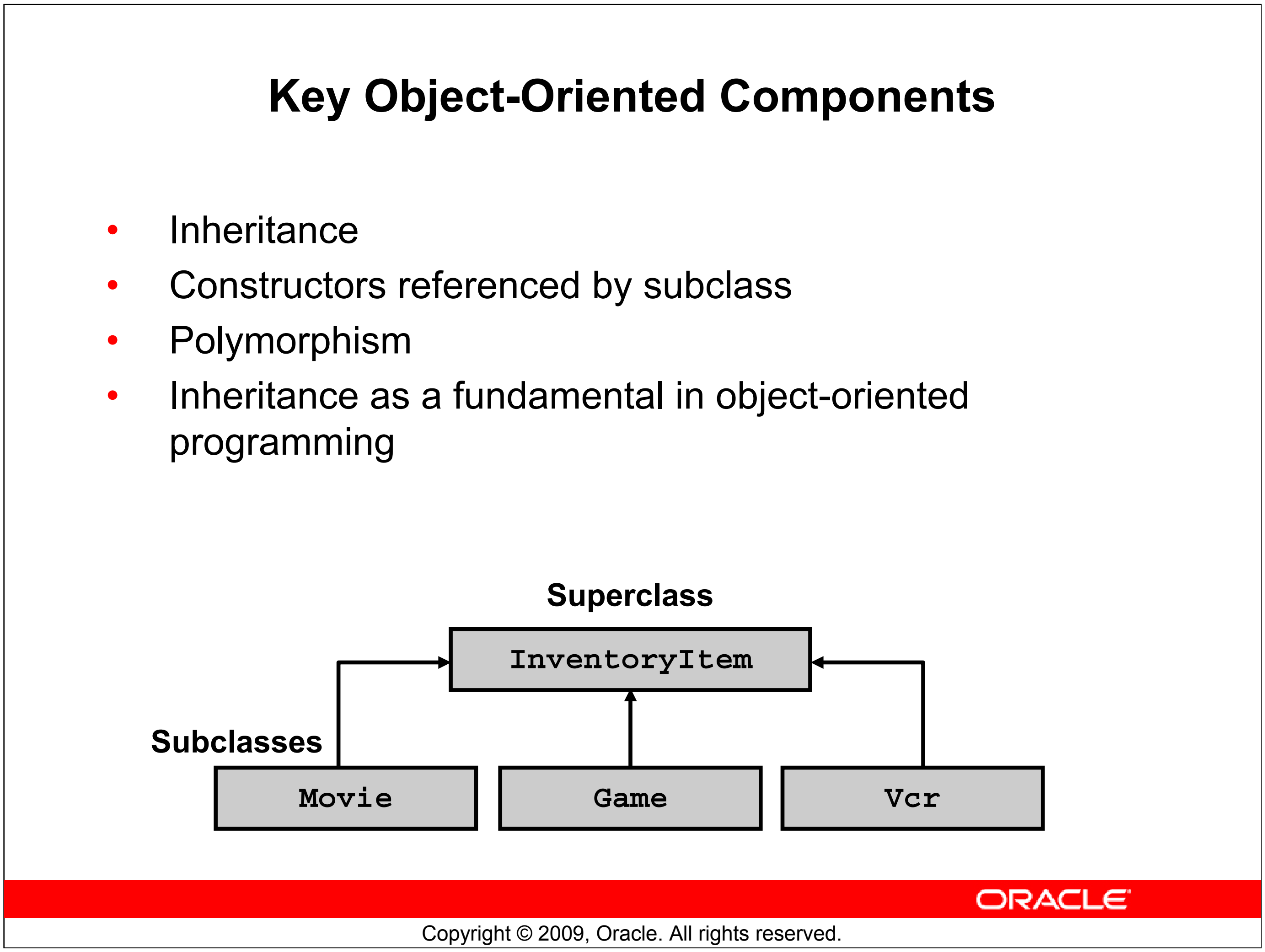
Objectives

After completing this lesson, you should be able to do the following:

- Define inheritance
- Use inheritance to define new classes
- Provide suitable constructors
- Override methods in the superclass
- Describe polymorphism
- Use polymorphism effectively

Lesson Objectives

Inheritance and polymorphism are two of the fundamental concepts of object-oriented programming, and both enable code reuse. When you have invested time in developing and subsequently testing your code, a well-designed hierarchy of classes is the basis for reusing that code. Polymorphism allows you to create clean, sensible, readable, and future-proof code. The lesson shows you how to apply these concepts to your Java applications.



Key Object-Oriented Components

- Inheritance
- Constructors referenced by subclass
- Polymorphism
- Inheritance as a fundamental in object-oriented programming

Superclass

`InventoryItem`

Subclasses

`Movie` `Game` `Vcr`

Key Object-Oriented Components

What Is Inheritance?

Inheritance defines a relationship between classes in which one class shares the data structure and behaviors of another class. Inheritance is a valuable technique because it enables and encourages software reuse by allowing you to create a new class based on the properties of an existing class. As a result, the developer is able to achieve greater productivity than would otherwise be possible.

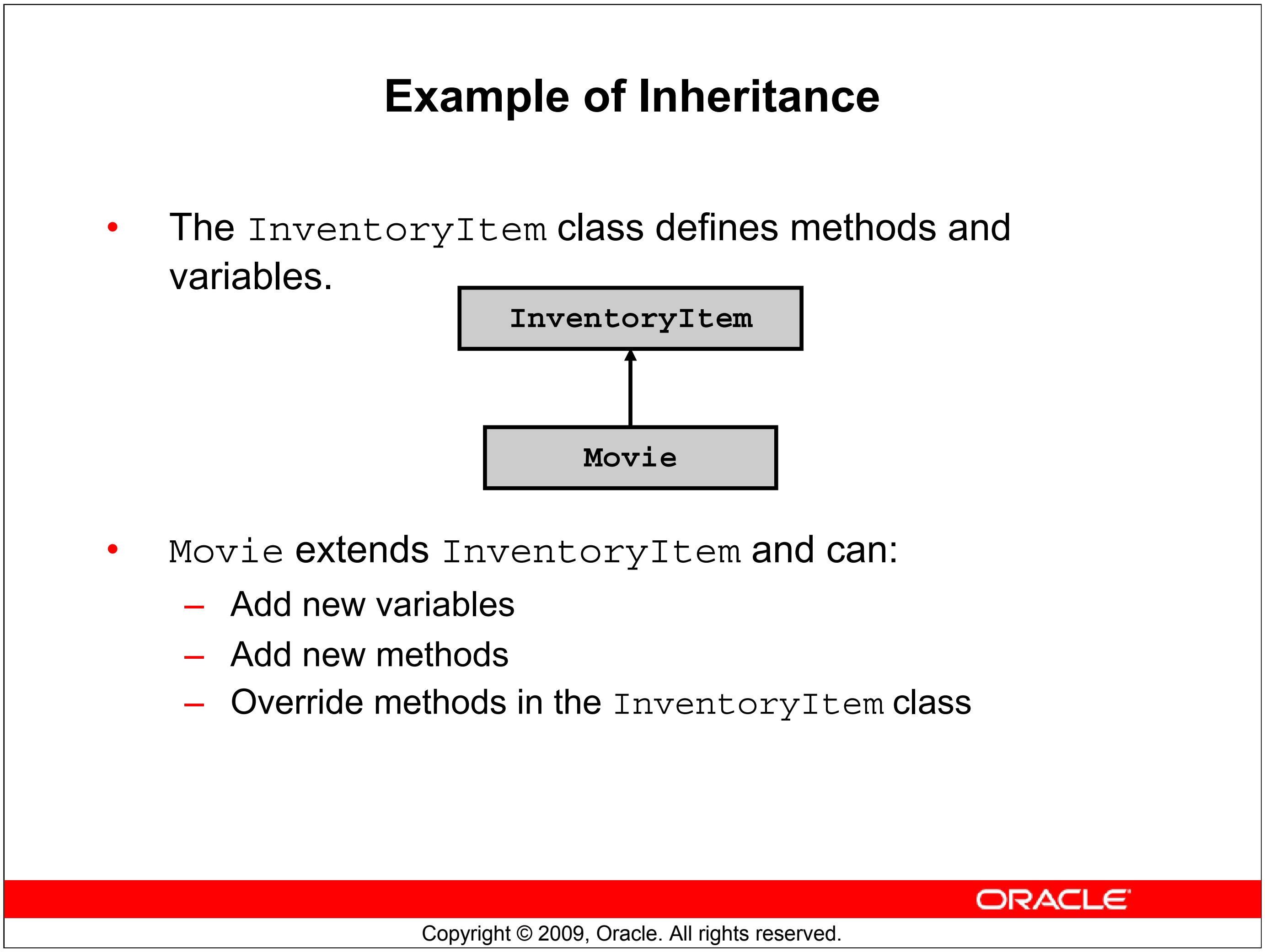
Inheritance and Constructors

You have already learned that constructors are blocks of code that are executed when an object of the class is created. When using an inheritance model, each subclass has access to the superclass’s constructor. Any common constructor code can be put in the superclass constructor and called by the subclass. This technique minimizes the need for duplicated code and provides for consistent object creation.

Key Object-Oriented Components (continued)

Polymorphism

Polymorphism describes the ability of Java to execute a specific method based on the object reference that is used in the call. Using this technique, you define a method in the superclass and override it in the appropriate subclass. You can now write method calls to the superclass, and if the method is overridden in a subclass, Java calls the appropriate method. This is a very powerful construct that you can use to define superclass methods before knowing the details of any subclasses.



Example of Inheritance

- The InventoryItem class defines methods and variables.

InventoryItem
Movie

- Movie extends InventoryItem and can:
  - Add new variables
  - Add new methods
  - Override methods in the InventoryItem class

`InventoryItem` Class

The InventoryItem class defines the attributes and methods that are relevant for all kinds of inventory items. The attributes and methods may include:

- Attributes, such as the date of purchase, purchase cost, and condition
- Methods, such as calculating a deposit, changing the condition, and setting the price

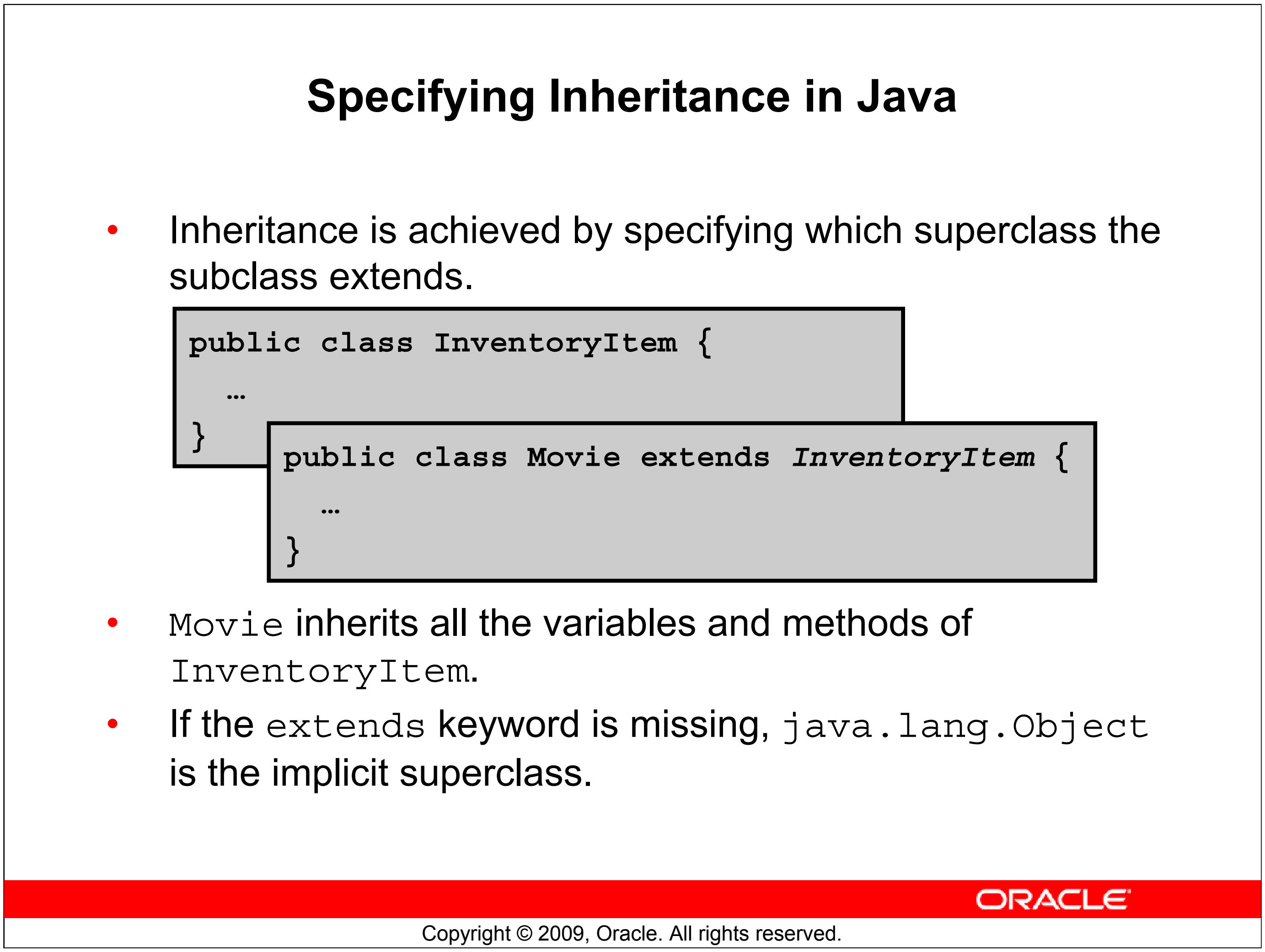
Dealing with Different Types of InventoryItem

Depending on what you are trying to do in your program, you may need to represent a specific type of inventory item in a particular way. You can use inheritance to define a separate subclass of InventoryItem for each different item type. For example, you may define classes such as Movie, Game, and Vcr.

Each subclass automatically inherits the attributes and methods of InventoryItem but can provide additional attributes and methods as necessary. For example, the Movie class may define the following additional attributes and methods:

- Attributes, such as the title of the movie, the director, and the running length
- Methods, such as reviewing the movie and setting the rating

Subclasses can also override a method from the superclass if they provide more specialized behavior for the method. In this example, Movie could override InventoryItem when calculating a deposit method. A movie may have an additional amount calculated into the deposit.



Specifying Inheritance in Java

- Inheritance is achieved by specifying which superclass the subclass extends.

```
public class InventoryItem {
  …
}
```

```
public class Movie extends InventoryItem {
  …
}
```

- `Movie` inherits all the variables and methods of `InventoryItem`.
- If the `extends` keyword is missing, `java.lang.Object` is the implicit superclass.

Specifying Inheritance in Java

When you define a subclass, you must provide code only for the features in the subclass that are different from those of the superclass. In a very real way, the subclass is extending the superclass. The syntax for specifying inheritance in Java makes use of the `extends` keyword, as in the following example:

```
public class InventoryItem {
  // Definition of the InventoryItem class
}
public class Movie extends InventoryItem {
  // Additional methods and attributes, to distinguish a
  // Movie from other types of InventoryItem
}
```

Characteristics of Inheritance in Java

If you have experience with another object-oriented language (such as C++), note that Java allows only single inheritance. In other words, a class can specify only one immediate superclass.

Specifying Inheritance in Java (continued)

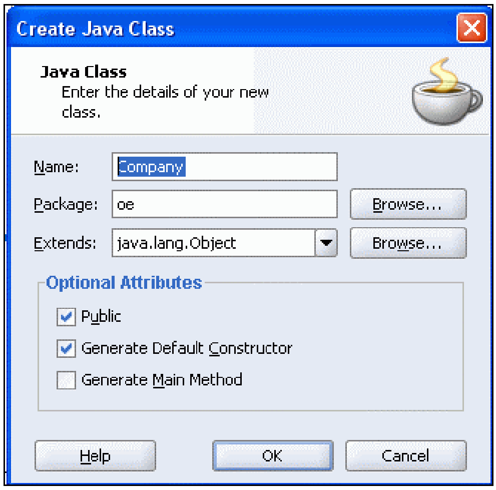
Also, remember that all classes in Java are automatically inherited from the root class called `Object`, which sits at the top of the inheritance tree. If a class does not specify an explicit superclass, as is the case with `InventoryItem` in the slide, the class is deemed to extend directly from `Object`, as if it were defined as follows:

```
  public class InventoryItem extends Object { …
```

The `java.lang.Object` class is the root class for all the classes in Java.

Defining Inheritance with Oracle JDeveloper

- When creating a new class, JDeveloper asks for its superclass:

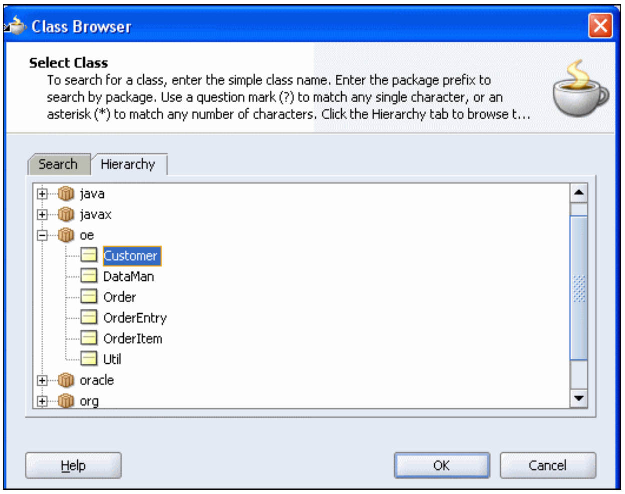


Create Java Class

Java Class
Enter the details of your new class.

Name: Company
Package: oe
Extends: java.lang.Object

Optional Attributes
☑ Public
☑ Generate Default Constructor
□ Generate Main Method



Class Browser

Select Class
To search for a class, enter the simple class name. Enter the package prefix to search by package. Use a question mark (?) to match any single character, or an asterisk (*) to match any number of characters. Click the Hierarchy tab to browse t...

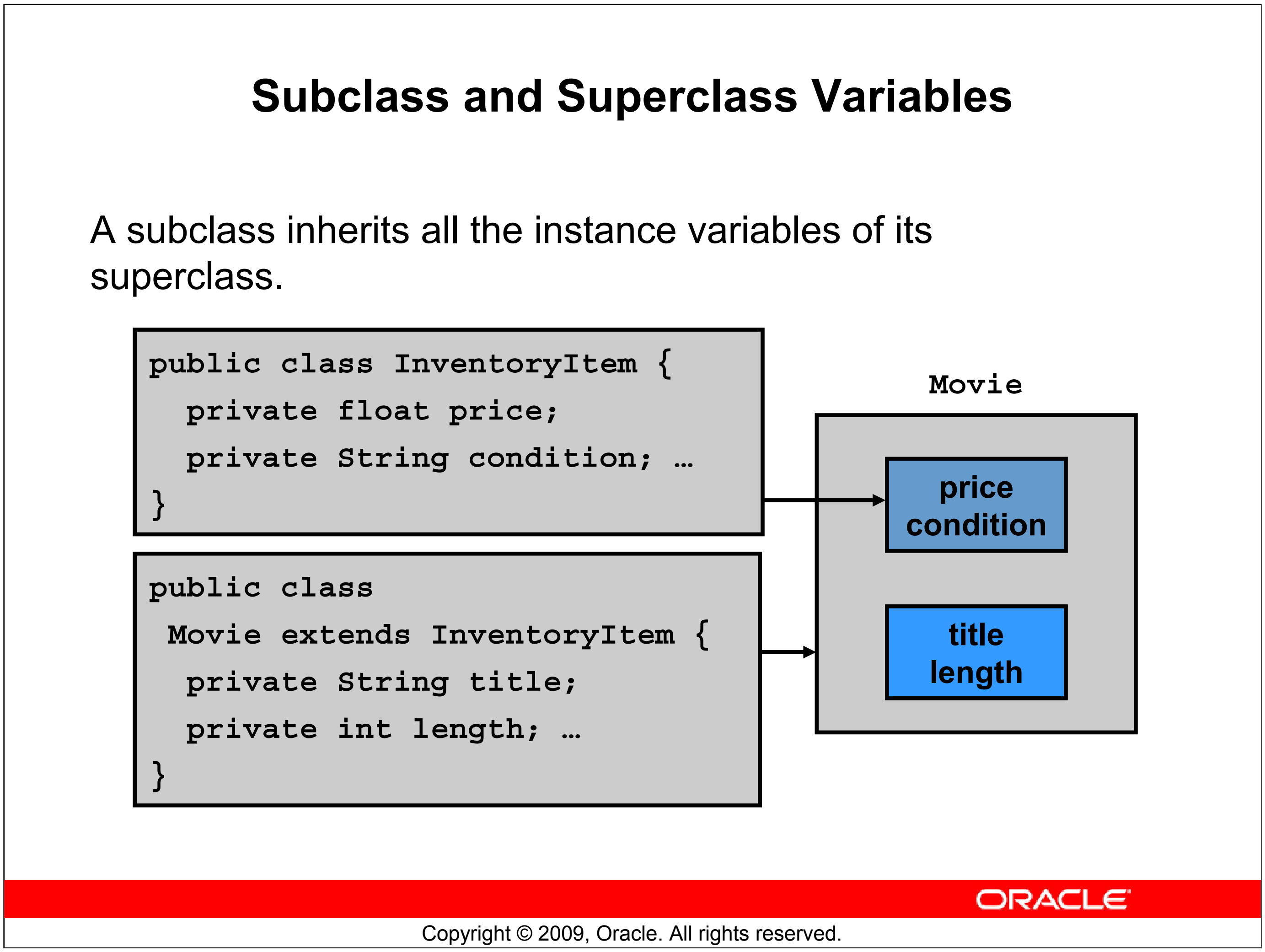
- JDeveloper generates the code automatically.

Defining Inheritance by Using Oracle JDeveloper

To define a new class that inherits from another class, follow these steps:

1. Select File > New from the Main menu bar.
2. A dialog box prompts you for the kind of feature you want to create. Select the General category, and double-click the Java Class icon. This launches the New Class dialog box.
3. Specify the name for your class, and browse to the name of the superclass that you want to extend. In the Browse window, JDeveloper brings up all packages (and the classes that are contained inside them) that it currently holds in memory. The default superclass name is `java.lang.Object` (that is, the `Object` class that is located in the `java.lang` package).
4. Click the OK button. JDeveloper generates a skeleton class to get you started, as follows:

```
package practice17;
public class Movie extends InventoryItem {
}
```



Subclass and Superclass Variables

A subclass inherits all the instance variables of its superclass.

```
public class InventoryItem {
  private float price;
  private String condition; …
}
```

```
public class
 Movie extends InventoryItem {
  private String title;
  private int length; …
}
```

Movie

price
condition

title
length

Subclass and Superclass Variables

The superclass defines the variables that are relevant for all kinds of inventory items, such as the purchase date and condition. The subclass `Movie` inherits these variables automatically and has to specify only the `Movie`-specific variables, such as the title.

What Does an Object Look Like?

If you create a plain `InventoryItem` object, it contains only the instance variables that are defined in `InventoryItem`:

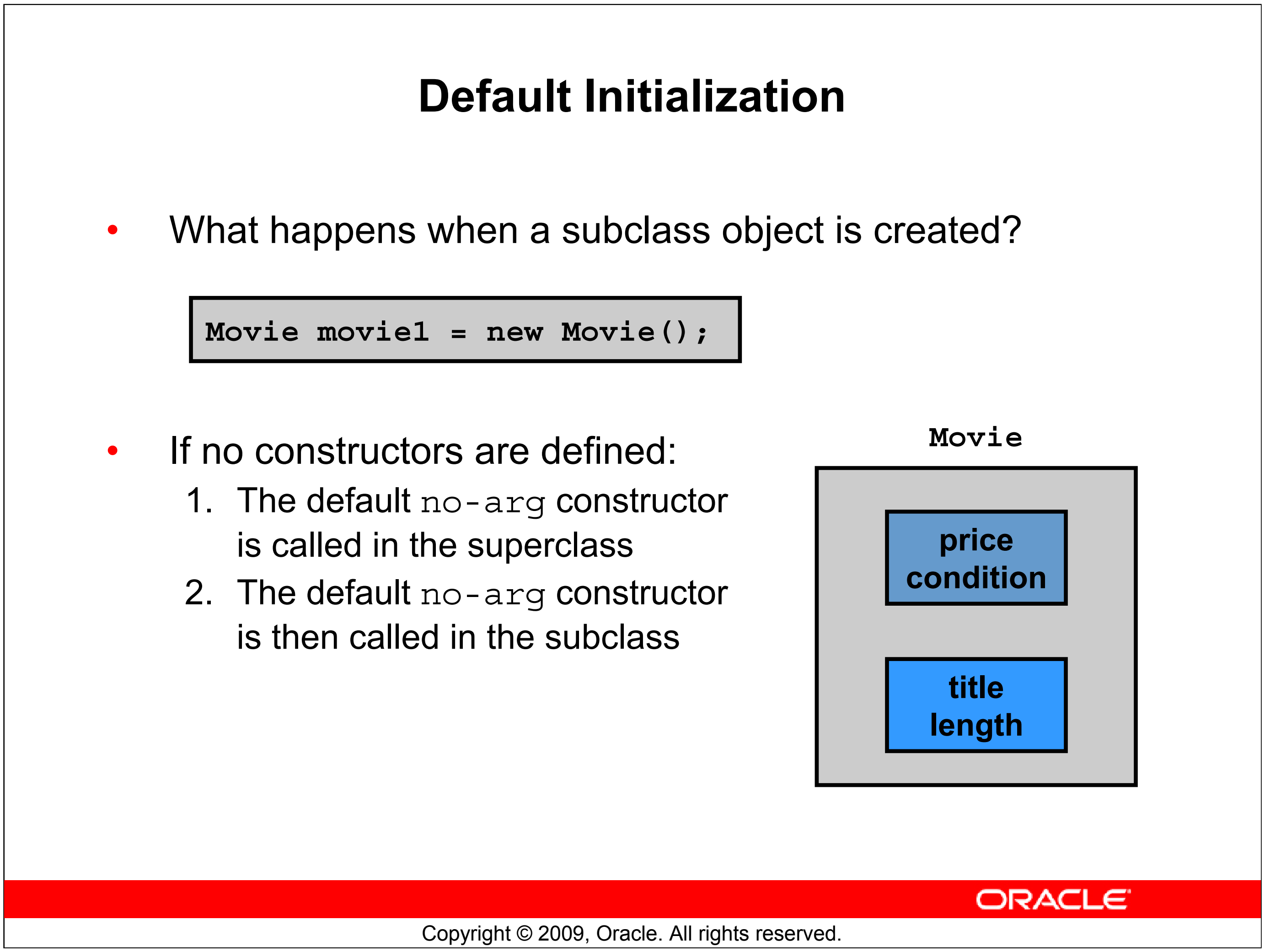
```
 InventoryItem i = new InventoryItem (…);
   //    an InventoryItem has a price and condition
```

However, if you create a `Movie` object, it contains four instance variables: the two inherited from `InventoryItem`, plus two added in `Movie`:

```
 Movie m = new Movie(…);
   // A Movie object has a price and condition, because a
   // Movie is a kind of InventoryItem.
   // The Movie object also has a title and length.
```

Declaring Instance Variables as `private`

Instance variables must normally be declared as `private`, which means that instances of subclasses inherit the values but cannot access them directly.



Default Initialization

- What happens when a subclass object is created?

```
Movie movie1 = new Movie();
```

- If no constructors are defined:
  1. The default `no-arg` constructor is called in the superclass
  2. The default `no-arg` constructor is then called in the subclass

Movie

price
condition

title
length

ORACLE

Default Provision of Constructors

A class does not inherit any constructors from its superclass. Therefore, the `InventoryItem` class has only the constructors explicitly declared in its definition or a default `no-arg` constructor if there are no other constructors.

What Happens When a Subclass Object Is Created?

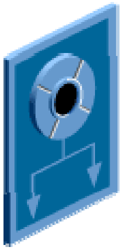
The example in the slide creates a `movie1` object. For the moment, assume that neither the `Movie` class nor the `InventoryItem` class provides any constructors; all they have is the default `no-arg` constructor that is provided automatically by Java.

What happens when a `movie1` object is created? Objects are always constructed from the top class down to the bottom class—that is, from the `Object` class down to the class that is being instantiated using `new`. This ensures that a constructor in a subclass can always rely on the proper construction of its superclass.

In the example, when you create a `movie1` object, by calling new Movie(), the movie constructor has to decide which superclass constructor to call. The `no-arg` constructor of `InventoryItem` is then called to initialize the `InventoryItem` instance variables with default values. The `price` is set to `0` and `condition` is set to its default: `excellent`. After the superclass is initialized, the `no-arg` constructor of `Movie` is then called to initialize the `title` and `length` instance variables with default values.

`Super()` Reference

- Refers to the base, super class
- Is useful for calling base class constructors
- Must be the first line in the derived class constructor



`Super()` Reference

The `super` reference is useful only when a class has an ancestor.

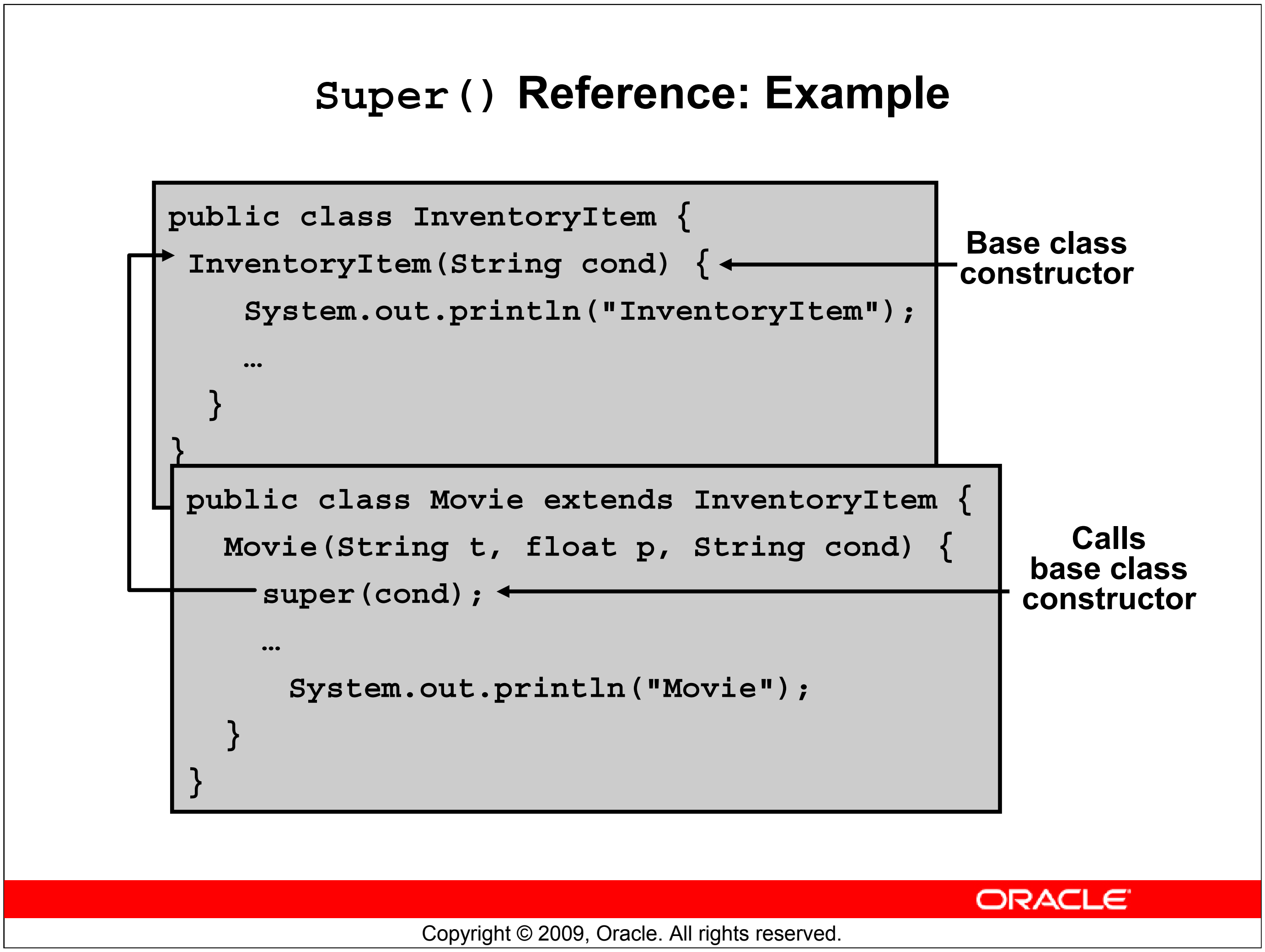
Calling Constructors

One of the more common uses of `super` is to invoke a constructor provided by the superclass, other than the default constructor. When the superclass was designed, it probably had a constructor to ensure proper initialization of any new objects. Because a subclass inherits all the superclass variables, they must be initialized for subclass objects as well.

The syntax rule is that `super()` must be the first line in the subclass constructor.

Add the super reference within the subclass constructor to access the superclass constructor:

```
-   subclass(…) {   // constructor for the subclass
-   super(…);       // call the superclass constructor
-   … ;               // subclass specific constructor code
-   }
```



`Super()` Reference: Example

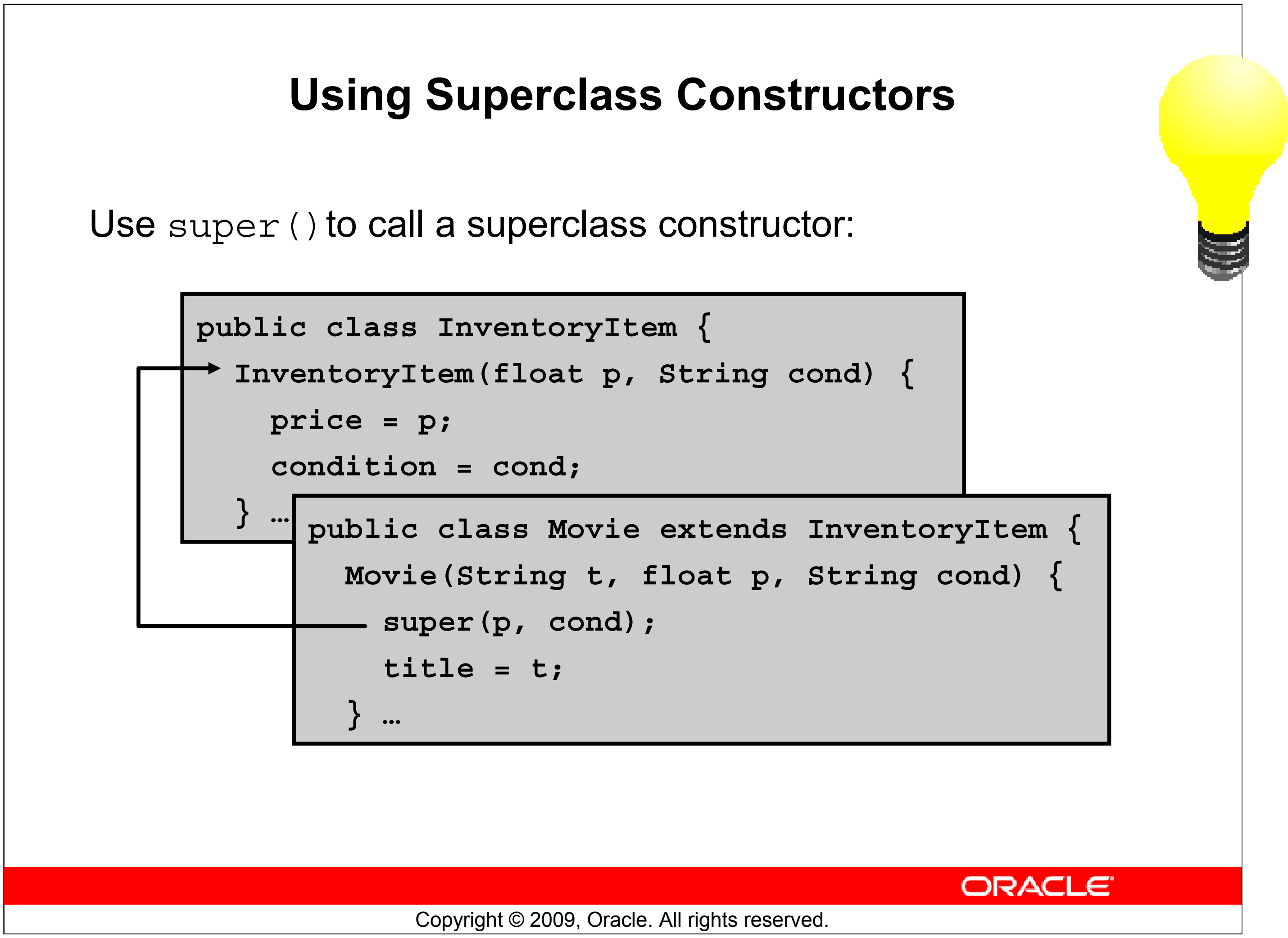
```
public class InventoryItem {
 InventoryItem(String cond) {
    System.out.println("InventoryItem");
    …
  }
}
```

```
public class Movie extends InventoryItem {
  Movie(String t, float p, String cond) {
    super(cond);
    …
      System.out.println("Movie");
  }
}
```

`Super()` Reference: Example

In the example, there are initialization routines that must happen for all inventory items. Those routines are placed in the `InventoryItem` constructor. These routines must be used regardless of the type of inventory item that is being constructed, whether it is a movie, game, or book.

There are also constructors in each of the subclasses to take care of subclass-specific routines. The `Movie` constructor reuses the `InventoryItem` constructor by referencing it with the `super` keyword. This statement is the first statement in the `Movie` constructor and may be followed by whatever other statements are necessary to fully construct a `Movie` object.



Using Superclass Constructors

Use `super()` to call a superclass constructor:

```
public class InventoryItem {
  InventoryItem(float p, String cond) {
    price = p;
    condition = cond;
  } …
```

```
public class Movie extends InventoryItem {
  Movie(String t, float p, String cond) {
    super(p, cond);
    title = t;
  } …
```

Nondefault Initialization with Inheritance

The superclass and subclass often have constructors that take arguments. For example, `InventoryItem` may have a constructor that takes arguments to initialize `price` and `condition`:

```
public InventoryItem (float p, String cond) {
  price = p;
  condition = cond;
}
```

Likewise, the `Movie` class may have a constructor that takes enough arguments to initialize its attributes. This is where things get interesting. A `Movie` object has three attributes, `price` and `condition`, which are inherited from `InventoryItem`, plus `title`, which is defined in `Movie` itself. The `Movie` constructor may therefore take three arguments:

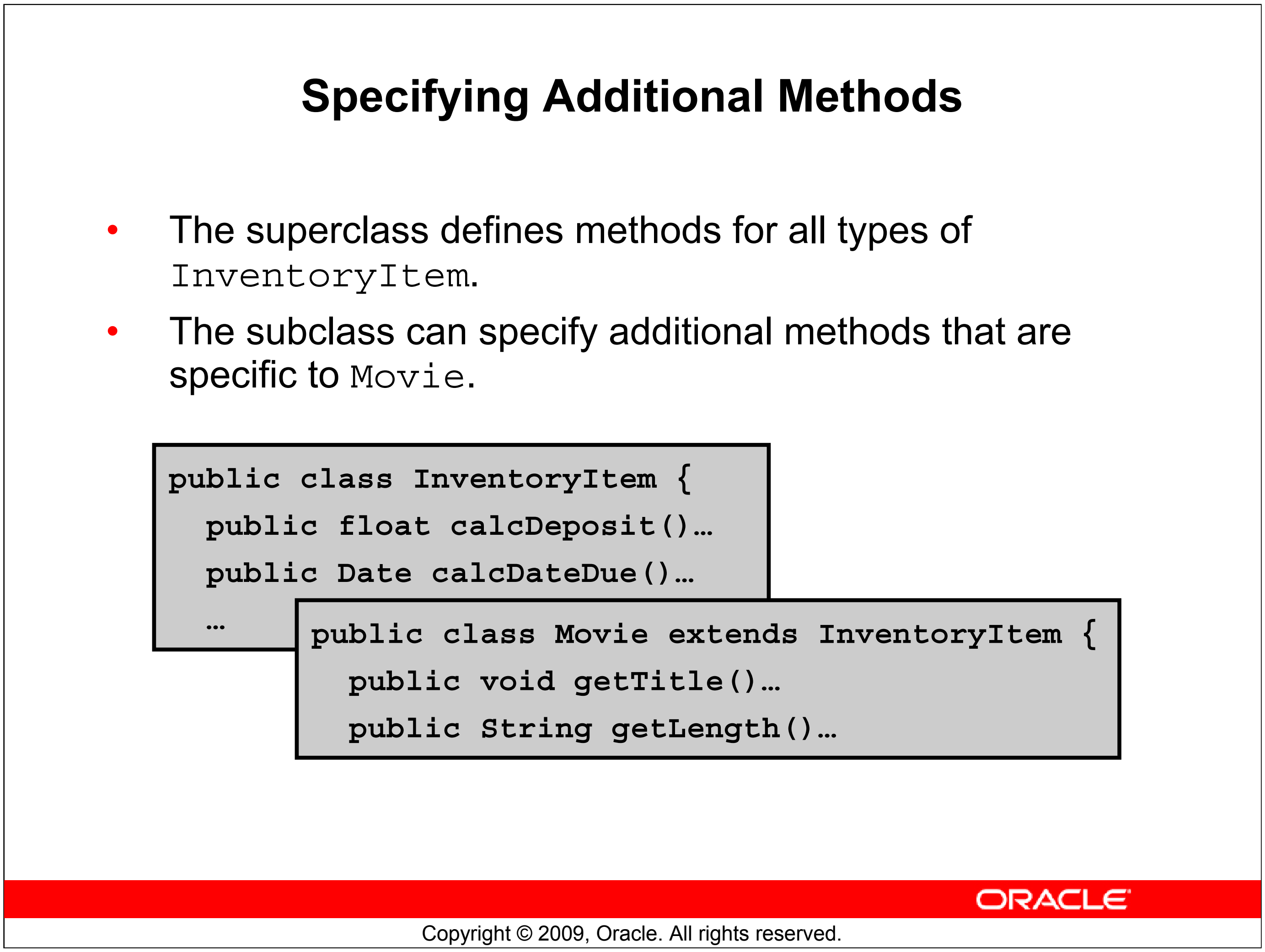
```
public Movie(float p, String cond, String t) { … }
```

Nondefault Initialization with Inheritance (continued)

Rather than initializing price and condition explicitly, all that the Movie constructor has to do is call the superclass constructor. This can be achieved by using the super keyword; the call to super(…) must be the first statement in the constructor.

```
public Movie(float p, String cond, String t) {
  super(p, cond);  // Call superclass constructor
  title = t;   // Initialize Movie-specific attributes
```

If you do not explicitly call super(…), the compiler calls the superclass no-arg constructor by default. If the superclass does not have a no-arg constructor, a compiler error occurs.



Specifying Additional Methods

- The superclass defines methods for all types of `InventoryItem`.
- The subclass can specify additional methods that are specific to `Movie`.

```
public class InventoryItem {
  public float calcDeposit()…
  public Date calcDateDue()…
  …
```

```
public class Movie extends InventoryItem {
  public void getTitle()…
  public String getLength()…
```

Methods in the Superclass and Subclass

The slide shows some of the methods that are declared in the superclass and the subclass. The superclass defines the methods that are relevant for all kinds of `InventoryItem`, such as the ability to calculate a deposit or the due date for the item. The subclass `Movie` inherits these methods from the superclass and must add only the `Movie`-specific methods, such as getting the title and getting the length.

Methods in the Superclass and Subclass (continued)

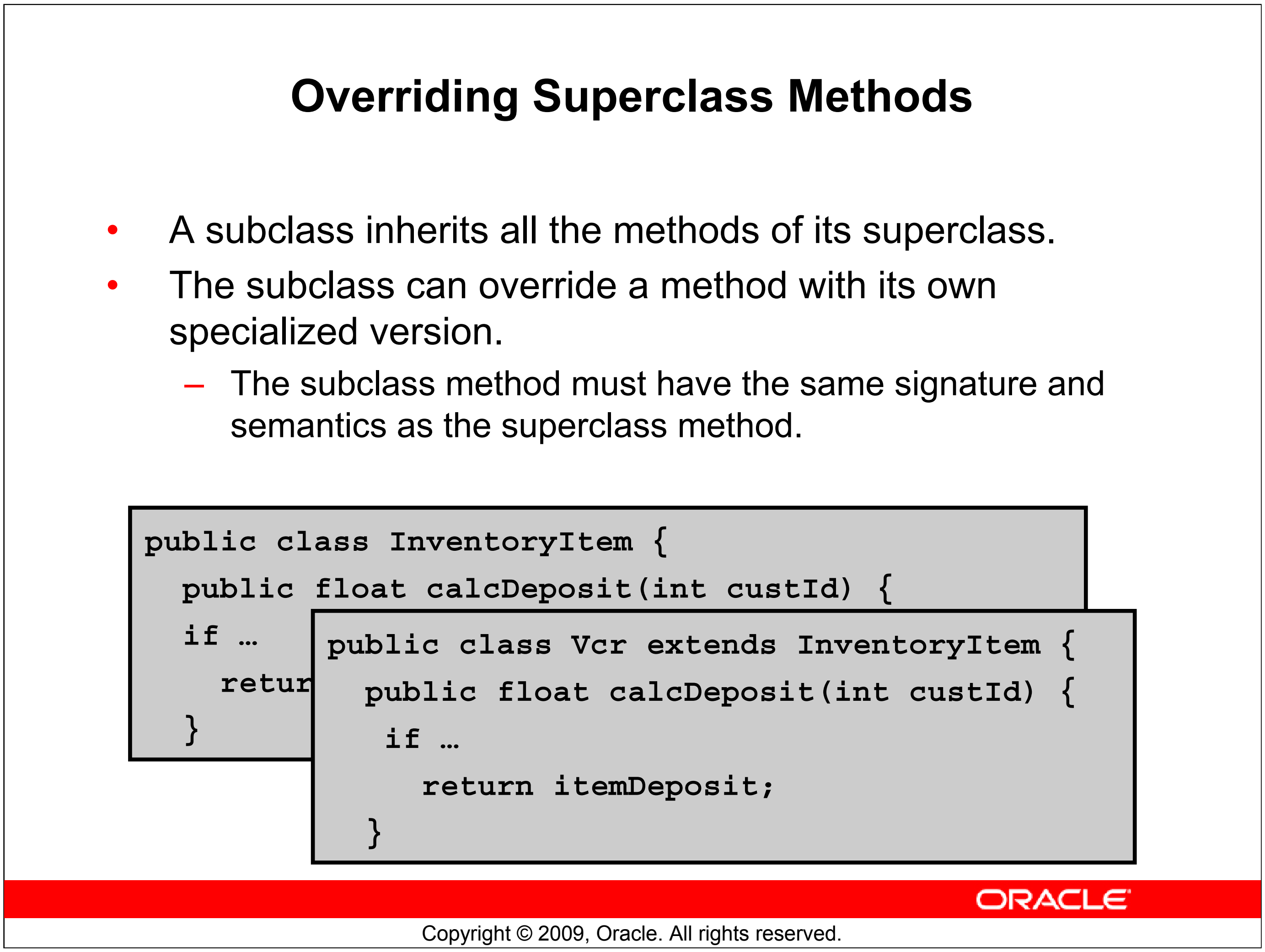
What Methods Can Be Called

When you create an object, you can call any of its `public` methods plus any `public` methods that are declared in its superclass. For example, if you create an `InventoryItem` object, you can call the `public` methods that are defined in `InventoryItem`, plus any `public` methods that are defined in its `Object` superclass:

```
InventoryItem i = new InventoryItem (…);
 i.getId();   // Call a public method in InventoryItem
 i.getClass(…);        // Call a public method in Object
```

If you create a `Movie` object, you can call any `public` methods that are defined in `Movie`, `InventoryItem`, or `Object`:

```
 Movie m = new movie(…); // Create a Movie object
 m.getTitle();          // Call a public method in Movie
 m.getId();   // Call a public method in InventoryItem
 m.getClass(…);         // Call a public method in Object
```



Overriding Superclass Methods

- A subclass inherits all the methods of its superclass.
- The subclass can override a method with its own specialized version.
  - The subclass method must have the same signature and semantics as the superclass method.

```
public class InventoryItem {
  public float calcDeposit(int custId) {
  if …
    retur
  }
```

```
public class Vcr extends InventoryItem {
  public float calcDeposit(int custId) {
   if …
     return itemDeposit;
  }
```

Overriding Superclass Methods

A subclass inherits all the methods of its superclass. However, a subclass can modify the behavior of a method in a superclass by overriding it, as shown in the slide.

To override a superclass method, the subclass defines a method with exactly the same signature and return type as a method somewhere above it in the inheritance hierarchy.

The method in the subclass effectively hides the method in the superclass. It is important to make sure that the method in the subclass has the same return type and signature as the one that it is overriding.

Which Method Is Called?

In the example in the slide, the `InventoryItem` class provides a `calcDeposit()` method, and the `Vcr` class overrides it with a more specialized version. If you create an `InventoryItem` object and call `calcDeposit()`, it calls the `InventoryItem` version of the method. If you create a `Vcr` object and call `calcDeposit()`, it calls the `Vcr` version of the method.

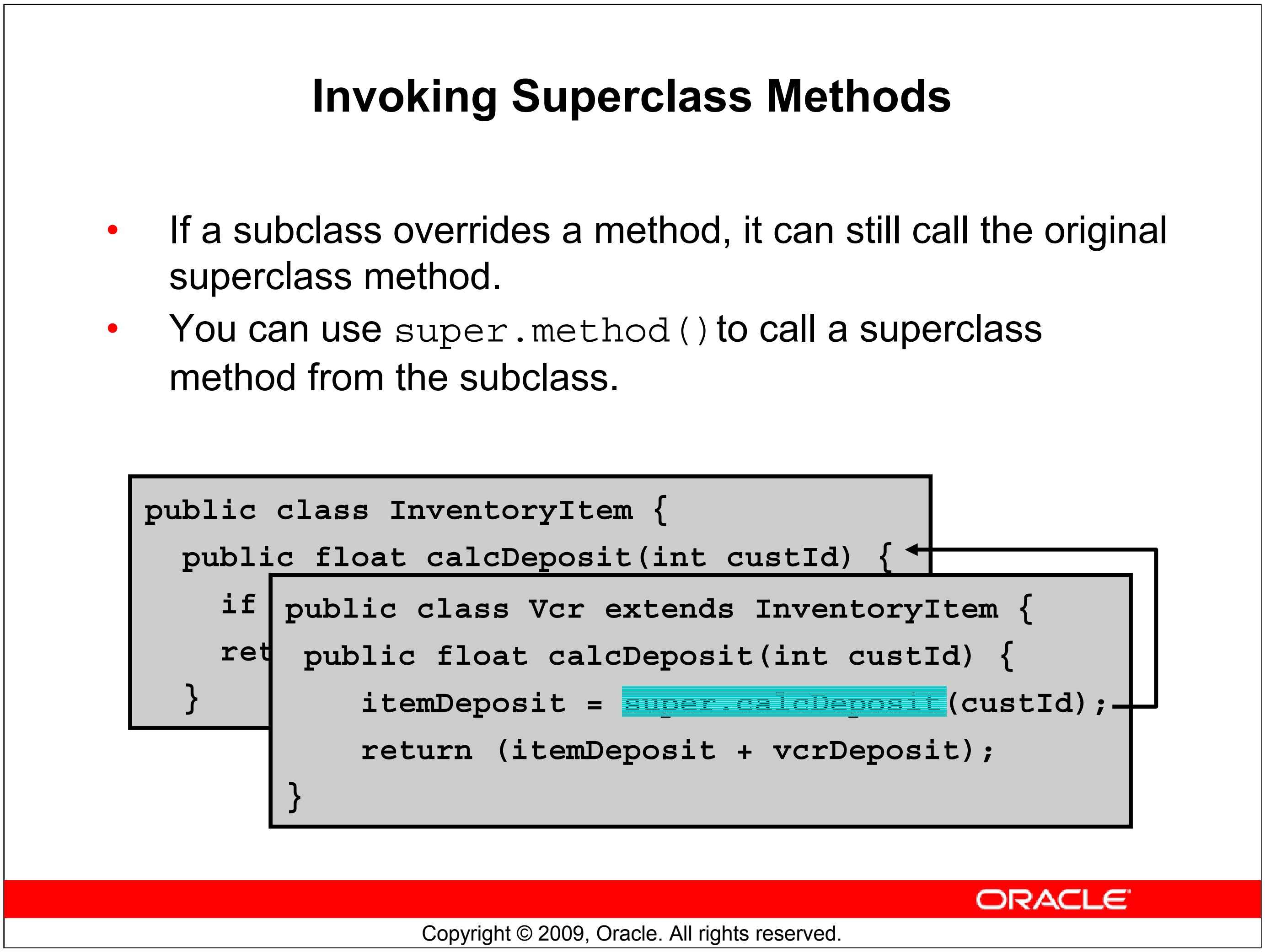
NOTE that overriding (i.e 'hiding') a superclass method can lead to code that is hard to maintain and reuse. You need to bear these kinds of maintenance issues in mind when designing your inheritance hierarchy.

Overriding Superclass Methods (continued)

Overriding and Overloading

Do not confuse “method overloading” with “method overriding”:

- Method overloading is a process by which you define multiple methods with different signatures. Overloaded methods are resolved at compile time, based on the arguments that you supply.
- Method overriding is a process by which you provide a method with exactly the same signature as a method in a superclass. Overridden methods are resolved at run time, unlike overloaded methods.



Invoking Superclass Methods

- If a subclass overrides a method, it can still call the original superclass method.
- You can use `super.method()` to call a superclass method from the subclass.

```
public class InventoryItem {
  public float calcDeposit(int custId) {
    if
    ret
  }
```

```
public class Vcr extends InventoryItem {
 public float calcDeposit(int custId) {
    itemDeposit = super.calcDeposit(custId);
    return (itemDeposit + vcrDeposit);
}
```

Invoking Superclass Methods

Calling an Overridden Method from the Client Program

As previously mentioned, when a subclass overrides a method in a superclass, it hides that method. For example, if the client program creates a `Vcr` object and calls the `calcDeposit()` method, it always executes the `Vcr` version of `calcDeposit()`:

```
  Vcr v = new Vcr(…); // Create a Vcr object
  v.calcDeposit(…);    // Executes Vcr calcDeposit() method
```

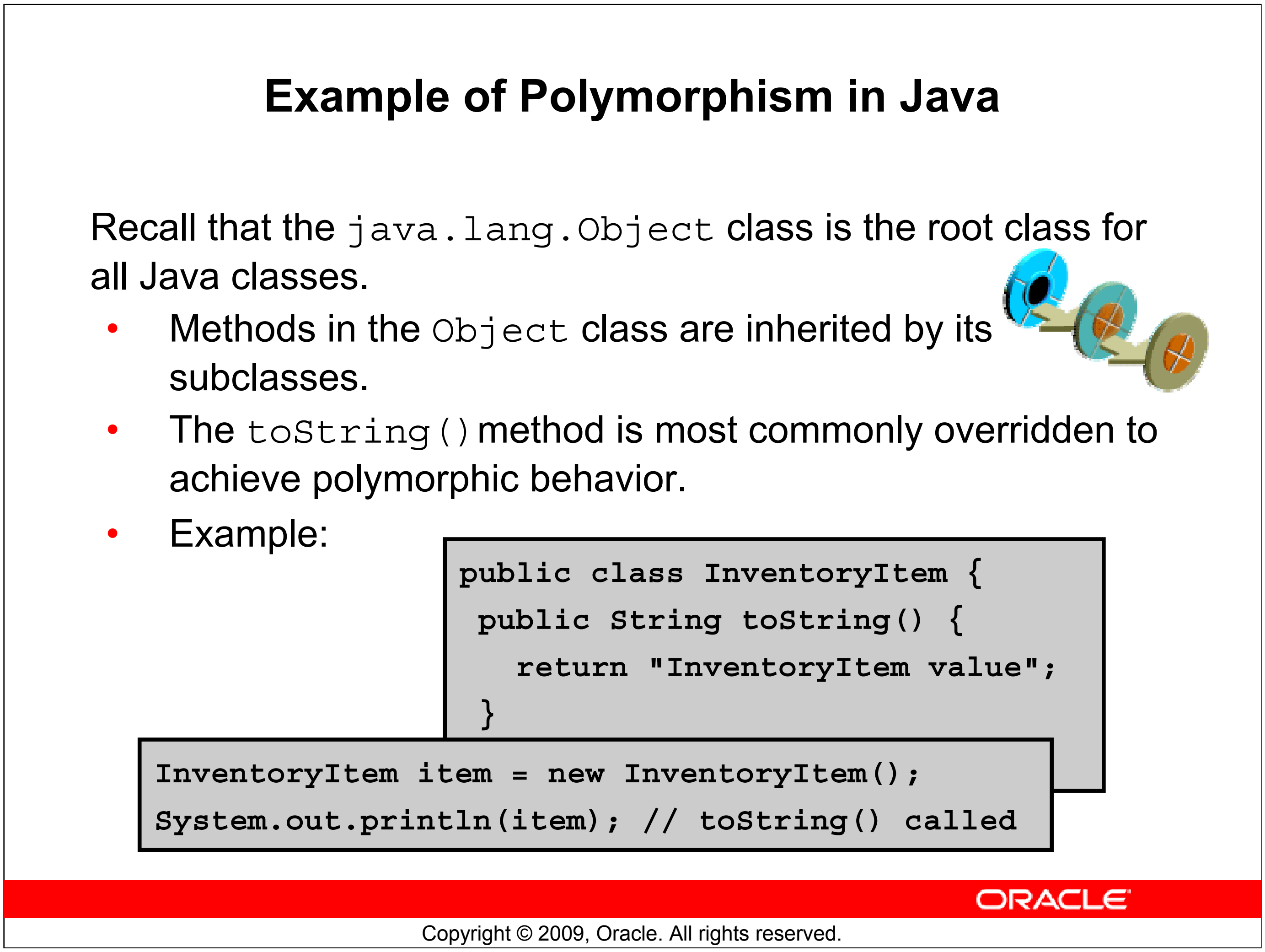
Calling an Overridden Method from the Subclass

Within the `Vcr` version of `calcDeposit()`, you can call the `InventoryItem` version of `calcDeposit()` that is defined in the superclass by using the `super` keyword. The `super` keyword is similar to `this`, except that it acts as a reference to the current object as an instance of its superclass.

Calling an overridden superclass method by using `super` helps to avoid duplicating the code that is contained in the overridden method; by reducing the amount of duplicate code, the code is more consistent and easier to maintain.

Here is an example of the syntax for overriding a method:

```
super.calcDeposit();
```



Example of Polymorphism in Java

Recall that the `java.lang.Object` class is the root class for all Java classes.

- Methods in the `Object` class are inherited by its subclasses.
- The `toString()` method is most commonly overridden to achieve polymorphic behavior.
- Example:

```
public class InventoryItem {
 public String toString() {
   return "InventoryItem value";
 }
```

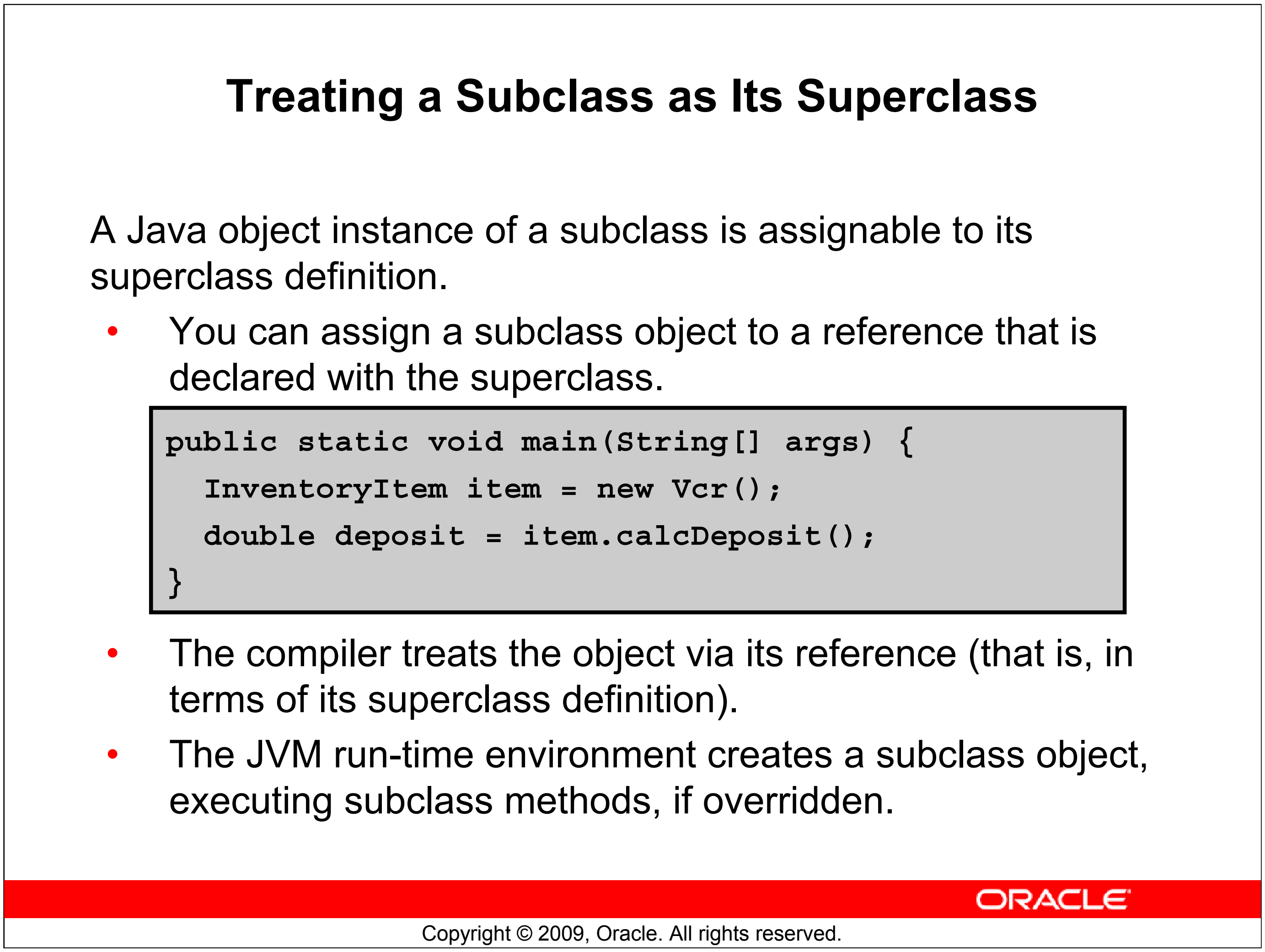
```
InventoryItem item = new InventoryItem();
System.out.println(item); // toString() called
```

Polymorphism and the `toString` Method

Polymorphism returns a string representation of the object. In general, the `toString` method returns a string that “textually represents” this object. The result must be a concise but informative representation that is easy to read. It is recommended that all subclasses override this method.

The `toString` method for the `Object` class returns a string consisting of the name of the class of which the object is an instance, the “@” character, and the unsigned hexadecimal representation of the hash code of the object. In other words, this method returns a string equal to the value of the following:

```
getClass().getName() + '@' + Integer.toHexString(hashCode)
```



Treating a Subclass as Its Superclass

A Java object instance of a subclass is assignable to its superclass definition.

- You can assign a subclass object to a reference that is declared with the superclass.

```
public static void main(String[] args) {
  InventoryItem item = new Vcr();
  double deposit = item.calcDeposit();
}
```

- The compiler treats the object via its reference (that is, in terms of its superclass definition).
- The JVM run-time environment creates a subclass object, executing subclass methods, if overridden.

Treating a Subclass as Its Superclass

Any Java subclass object can be assigned to an object reference variable that is declared as its superclass or as the same class as itself. The slide example shows that a `Vcr` object is assigned to the `item` object reference, which is declared as an `InventoryItem`. The `Vcr` must previously be declared as a class that extends `InventoryItem`. The Java compiler accepts this as valid syntax. This is necessary for polymorphism.

There are two ways to look at the code example: in compiler view and in run-time view.

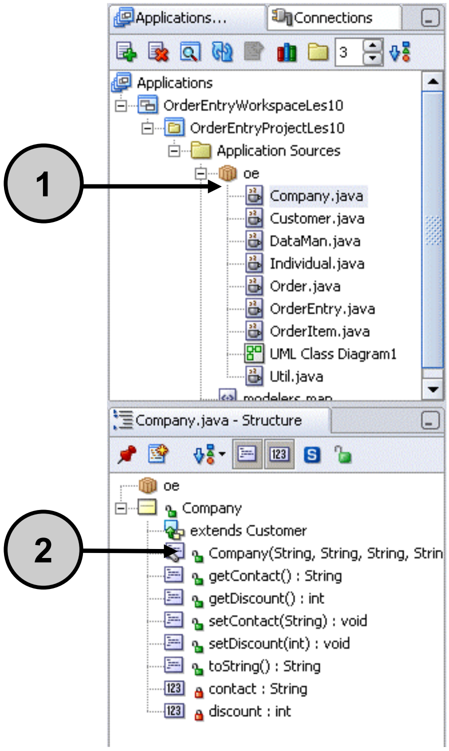
Compiler View

The compiler sees the `Vcr` object as if it were a “kind of” `InventoryItem`. Therefore, all methods that are called from the `item` object reference can only be those defined in the `InventoryItem` class because the item is defined as an `InventoryItem`. In essence, you are writing generic code to deal with common functionality of any kind of inventory item object.

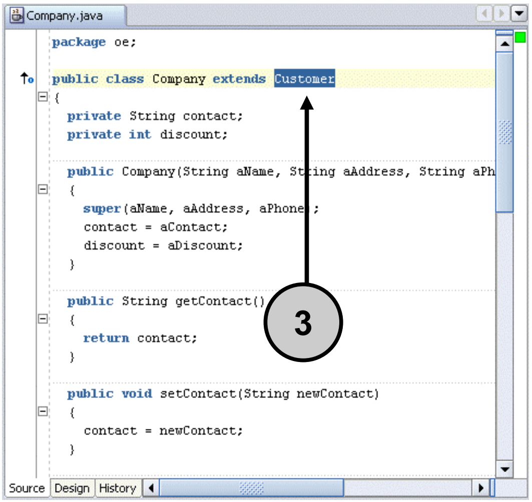
Run-Time View

At run time, the JVM dynamically creates the `Vcr` object. Thus, when you call a method such as `item.calcDeposit()`, it is the `Vcr`’s `calcDeposit()` method that is invoked if it overrides its superclass definition. Otherwise, the inherited method is called. The JVM uses a run-time type-checking mechanism to ensure that the call is valid; otherwise, it throws an exception.

Browsing Superclass References with Oracle JDeveloper



Oracle JDeveloper makes it easy to browse the contents of your superclass.



```
package oe;

public class Company extends Customer
{
  private String contact;
  private int discount;

  public Company(String aName, String aAddress, String aPh
  {
    super(aName, aAddress, aPhone);
    contact = aContact;
    discount = aDiscount;
  }

  public String getContact()
  {
    return contact;
  }

  public void setContact(String newContact)
  {
    contact = newContact;
  }
```

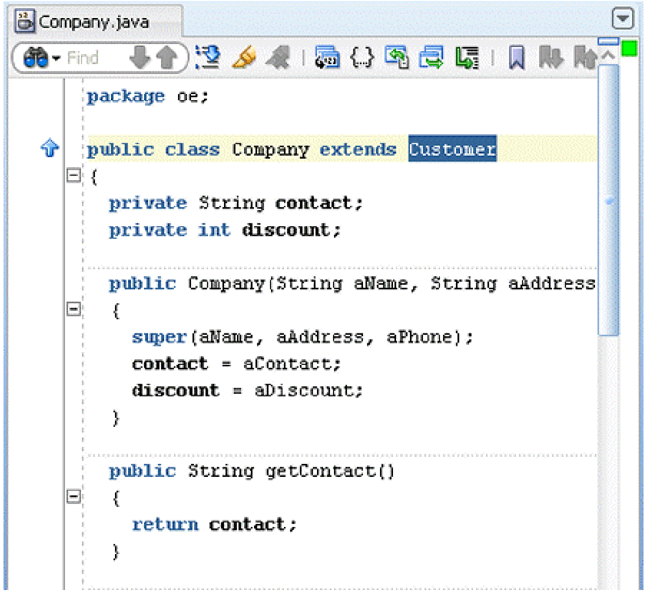
Browsing Superclass References with Oracle JDeveloper

You can use JDeveloper to browse the contents of any of your superclass references.

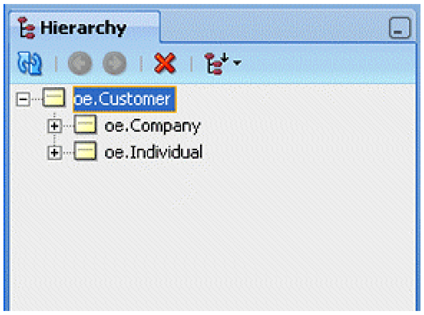
1. Select your class in the navigation pane.
   The structure pane at the upper left of the navigation pane lists all the classes in your project. Select the subclass that you want to start with, such as `Individual`.
2. Select the superclass in the structure pane.
   The structure pane at the lower left of the navigation pane lists all the methods, variables, and constructors for the current class. It also contains an icon to represent the superclass, which in this case is `Customer`.
3. View the superclass reference in the subclass code.
   Select the `extends` superclass text in the structure pane (in this case, `extends Customer`). The Code Editor displays the reference.

Hierarchy Browser

- Select the class and choose View Type Hierarchy from the context menu.



```
package oe;

public class Company extends Customer
{
  private String contact;
  private int discount;

  public Company(String aName, String aAddress
  {
    super(aName, aAddress, aPhone);
    contact = aContact;
    discount = aDiscount;
  }

  public String getContact()
  {
    return contact;
  }
```

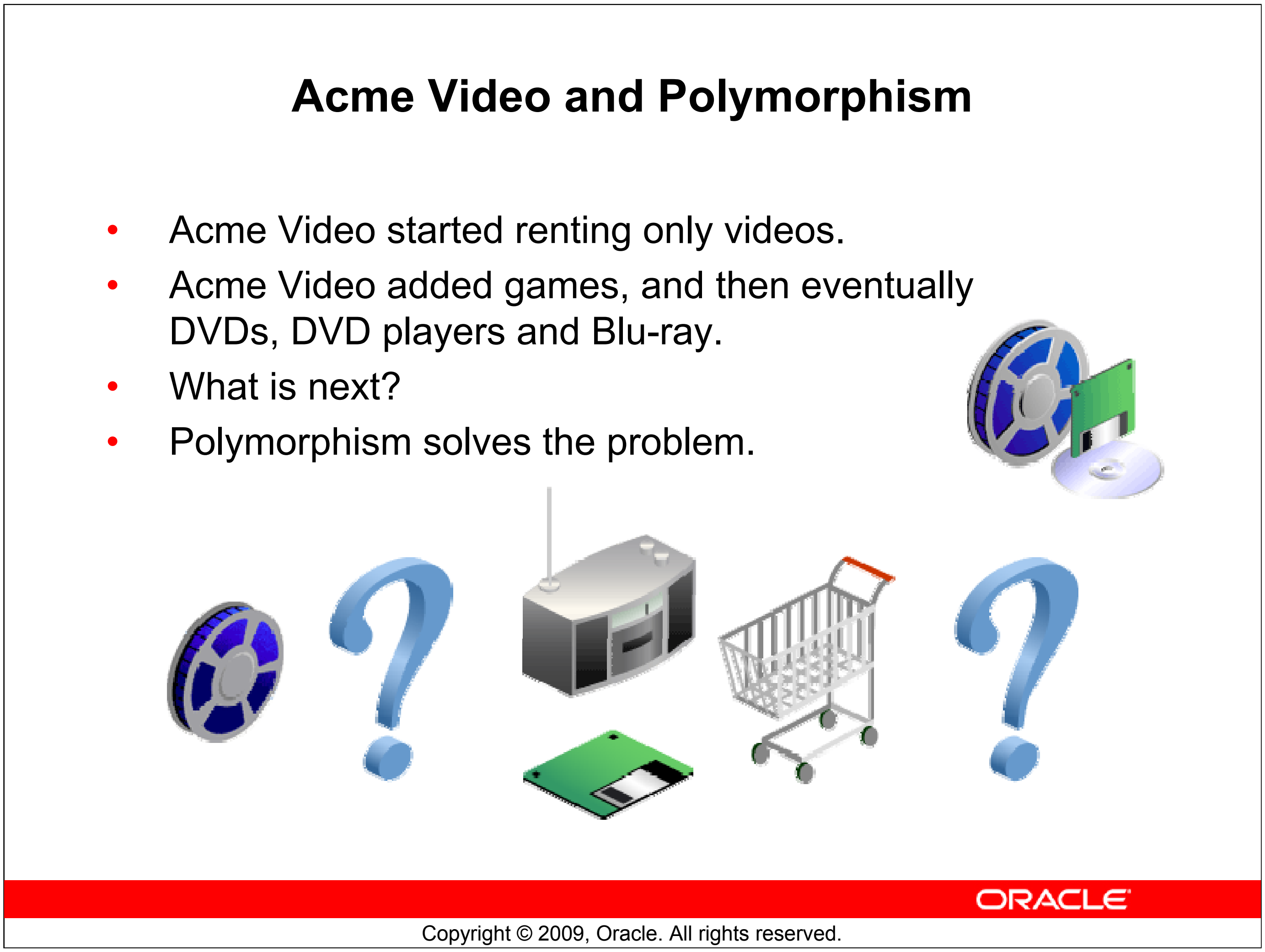


- oe.Customer
  - oe.Company
  - oe.Individual

- The Hierarchy window displays the class hierarchy.

Hierarchy Browser

You can also inspect the hierarchy of subtypes and supertypes of a class or interface when working in the Java Source Editor. The Hierarchy window displays the hierarchy of the selected class or interface.



Acme Video and Polymorphism

- Acme Video started renting only videos.
- Acme Video added games, and then eventually DVDs, DVD players and Blu-ray.
- What is next?
- Polymorphism solves the problem.

Acme Video and Polymorphism

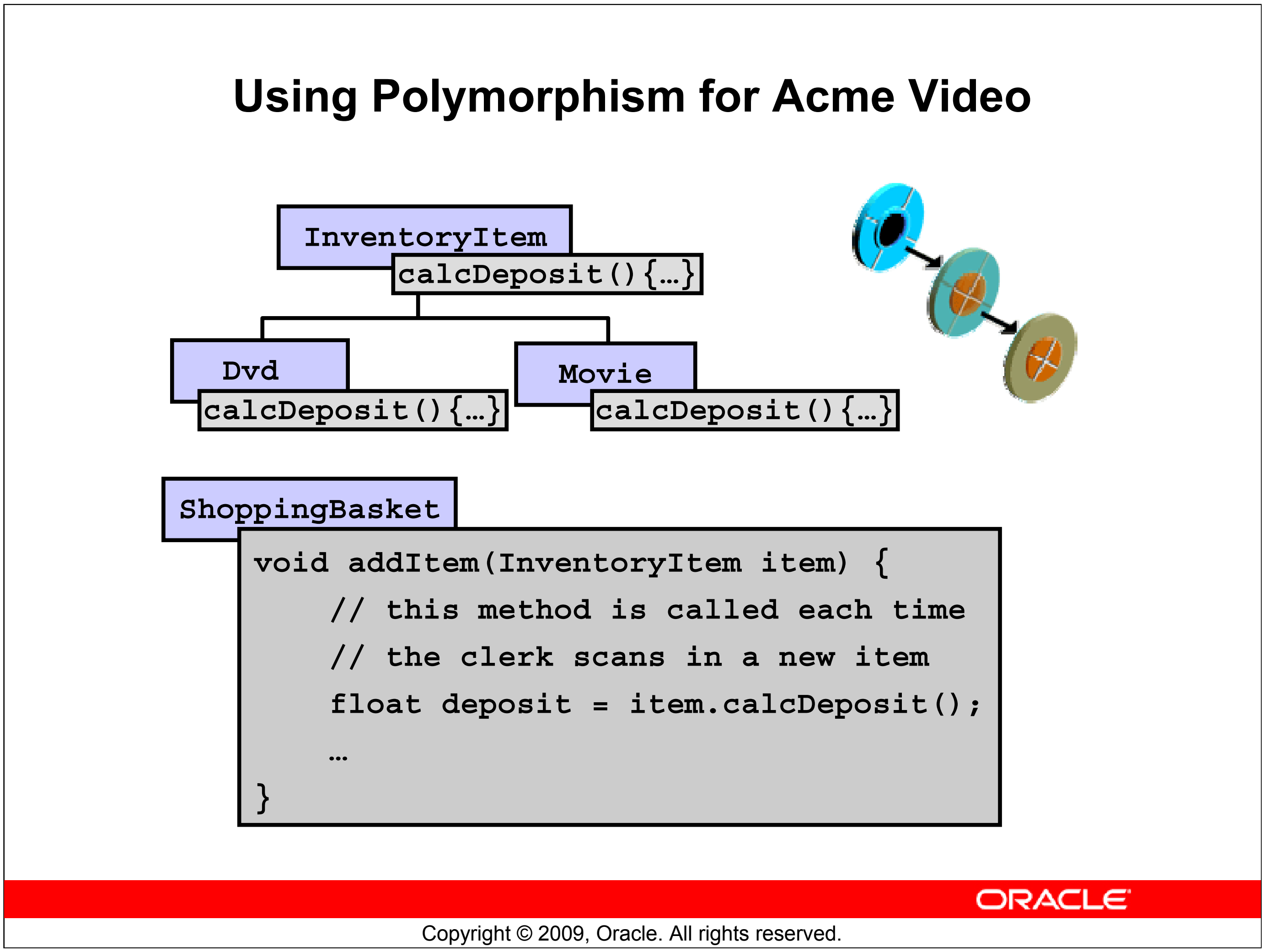
Acme Video started as a simple video rental business that only rented videos. As business began to improve, Acme Video decided to branch out and add video games, then DVDs to its inventory. It soon started getting equipment requests for DVD players and for Blu-ray.

Each of the different items that Acme is now renting has unique properties, and Acme handles each type in a slightly different manner. For example, it requires a deposit on the DVD players and Blu-ray devices but not on videos and games. The deposit is based on the type of equipment and the customer. Regular, established customers with good credit are not required to leave a deposit, whereas new customers are.

When the customer checks items out, Acme must determine the price of the items as well as any required deposit. Its application must be flexible enough to accept new types of items without having to change or recompile existing code each time its business is expanded. It accomplishes this goal by using Java’s polymorphic abilities.

Acme designed the `ShoppingBasket` class to simply accept and process inventory items, whatever type they may be. It then allows Java to determine the type of the item and to call the correct methods based on that type.

By using this technique, Acme can add as many new item types as required without having to change or recompile existing code.



Using Polymorphism for Acme Video

```
void addItem(InventoryItem item) {
    // this method is called each time
    // the clerk scans in a new item
    float deposit = item.calcDeposit();
    …
}
```

Using Polymorphism for Acme Video

When Acme designed its video rental application, it did not know all the types of inventory items that would be rented in the long term. In non-object-oriented programming, this would create a problem to be solved by modifying code each time a new type is added.

In Java, you can use polymorphism to solve the problem as follows. The `calcDeposit()` method in the `InventoryItem` class is overridden in the `Dvd` and `Movie` classes to provide object-specific calculation logic. The `ShoppingBasket` class includes an `addItem(InventoryItem item)` method that calls the `calcDeposit()` method by using an `InventoryItem` object.

At run time, Java interrogates the argument to determine its actual object type and determines whether the type has an overriding method. If it does, Java uses the subclass method in place of the superclass method.

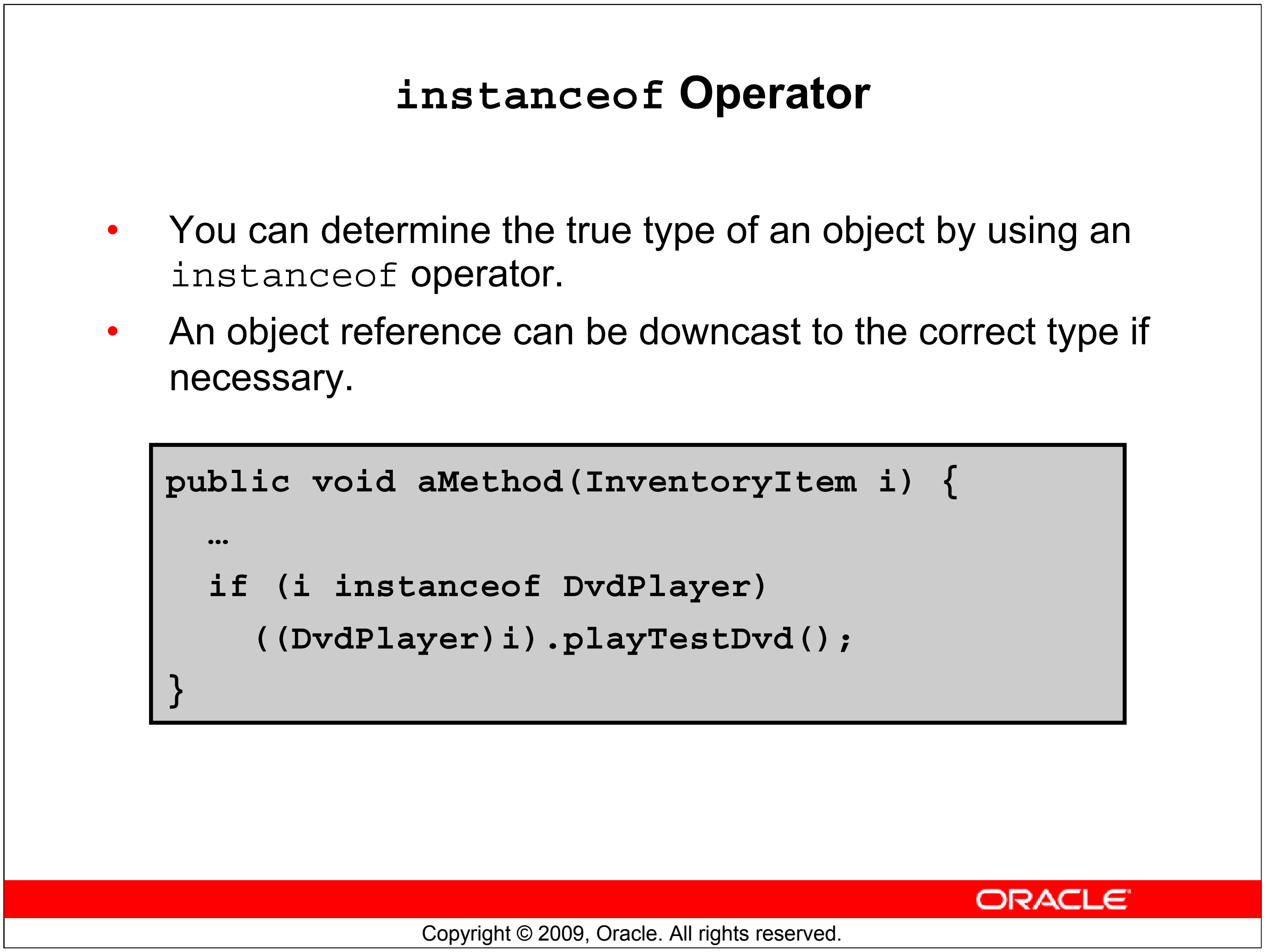
For example, if `movie` is a variable of the `Movie` type and `DvdPlayer` is a variable of the `DvdPlayer` type:

```
addItem(movie); // calls the Movie version of calcDeposit()
addItem(dvdPlayer);   // calls the DvdPlayer version of
   calcDeposit()
```

Using Polymorphism for Acme Video (continued)

The `addItem` method accepts any kind of `InventoryItem` object, including the plug-compatible subclass objects.

The significance is that the `ShoppingBasket` and `InventoryItem` classes do not need to change as new `InventoryItem` types are added to the business. The object-oriented code continues to work.



`instanceof` Operator

- You can determine the true type of an object by using an `instanceof` operator.
- An object reference can be downcast to the correct type if necessary.

```
public void aMethod(InventoryItem i) {
  …
  if (i instanceof DvdPlayer)
    ((DvdPlayer)i).playTestDvd();
}
```

`instanceof` Operator

You can use the `instanceof` operator to determine the type of an object at run time. This operator is useful in situations where you need to call some subclass-specific operation on an object but you must first verify that the object is of the correct type.

The syntax of the `instanceof` operator is as follows:

```
        objectRef instanceof className
```

The `instanceof` operator returns a `boolean` value. If the object that is referred to by `objectRef` is an instance of the specified `className` or one of its subclasses, the `instanceof` operator returns `true`. Otherwise, it returns `false`.

Example

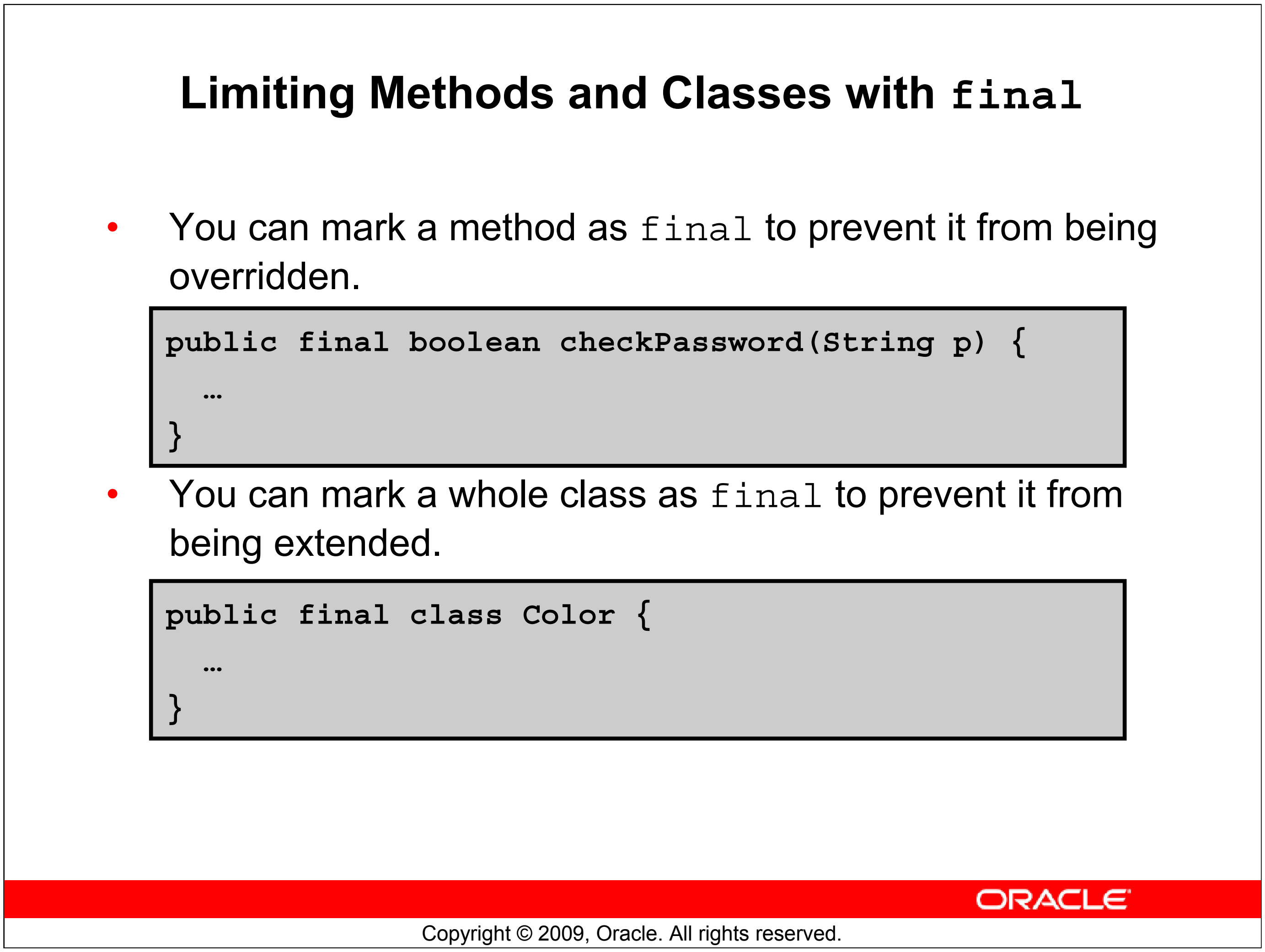
The method in the slide takes an object reference whose compile-time type is declared as `InventoryItem`. However, at run time, the object that is passed into the method may be any kind of `InventoryItem`, such as `Dvd`, `Movie`, or `Game`.

Inside the method, you use `instanceof` to test whether you have a `Dvd` object. If so, you convert the compile-time type of the object reference into the `Dvd` type, and then call a `Dvd`-specific method. This is often called *downcasting*.

instanceof Operator (continued)

Downcasting

The downcast is necessary in this example. Without it, the compiler allows you to call only those methods that are defined in the InventoryItem class. However, you must use downcasting sparingly. There are usually alternative designs that eliminate the need for excessive downcasting.



Limiting Methods and Classes with `final`

- You can mark a method as `final` to prevent it from being overridden.

```
public final boolean checkPassword(String p) {
  …
}
```

- You can mark a whole class as `final` to prevent it from being extended.

```
public final class Color {
  …
}
```

Limiting Methods and Classes with `final`

`final` Methods

Methods and classes are made `final` for two primary reasons: security and optimization.

If a method is performing some vital operation, such as identity validation or authorization checking, it must be declared `final` to prevent anyone from overriding the method and circumventing your security checks. Many of the methods that are defined in `java.net` classes are `final`.

`final` Classes

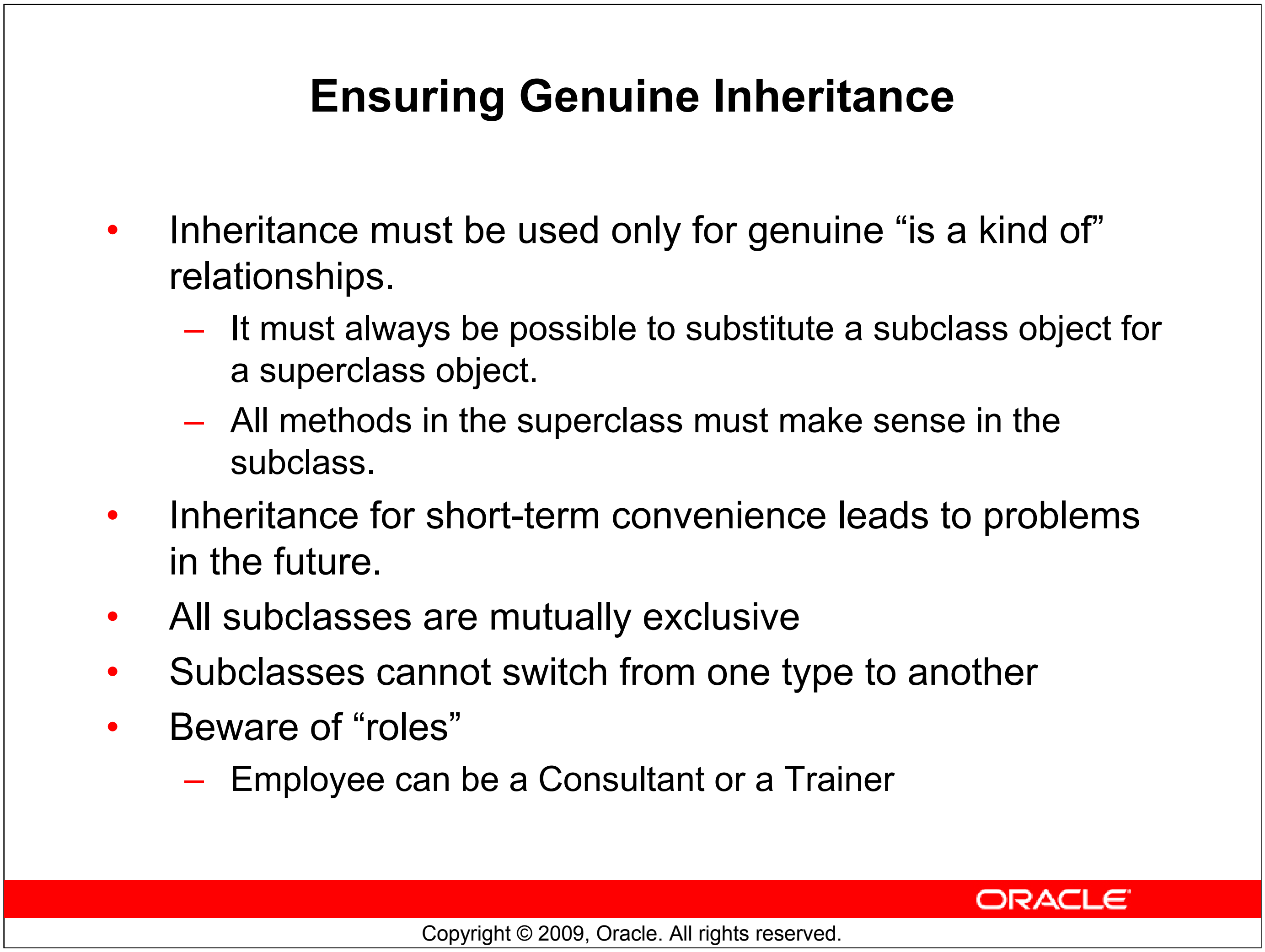
If you declare a class as `final`, it can never be extended by another class. This is a strong design statement that the class is sufficient to cater to all current and future requirements. The implication is clear: You never need to think about inheriting from this class. For example, the `Color` class in `java.awt` is declared `final`.

Limiting Methods and Classes with `final` (continued)

`final` Classes and `final` Methods

`final` classes enable the compiler to produce more efficient code. Because a `final` class cannot be extended, if the compiler encounters an object reference of that type and you call a method using that object reference, then the compiler does not need to perform run-time method binding to cater to any subclasses that may have overridden the method. Instead, the compiler can perform static binding—that is, the compiler can decide which method to call and avoid the overhead of run-time polymorphic lookup.

This is true for individual `final` methods as well. If you call a `final` method anywhere in your program, the compiler can call that method statically without determining whether the method may be overridden by a subclass.



Ensuring Genuine Inheritance

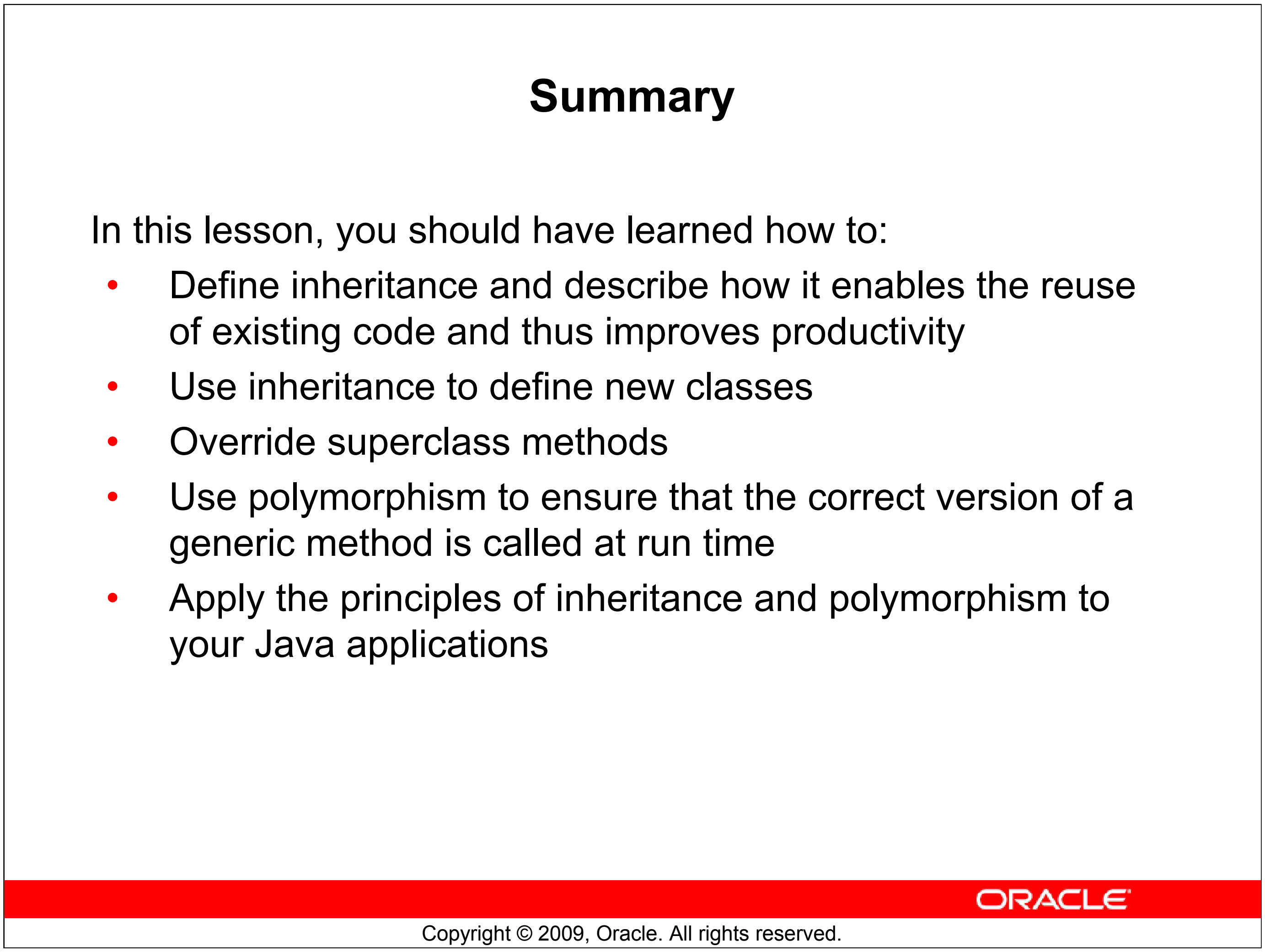
- Inheritance must be used only for genuine “is a kind of” relationships.
  - It must always be possible to substitute a subclass object for a superclass object.
  - All methods in the superclass must make sense in the subclass.
- Inheritance for short-term convenience leads to problems in the future.
- All subclasses are mutually exclusive
- Subclasses cannot switch from one type to another
- Beware of “roles”
  - Employee can be a Consultant or a Trainer

Ensuring Genuine Inheritance

Use inheritance only to model a genuine “is a kind of” relationship. In other words, do not use inheritance unless all the inherited methods apply to the subclass. If you cannot substitute a subclass object for a superclass object, you do not have a genuine “is a kind of” relationship. In this case, the classes may be related—but not related hierarchically.

If you do use inheritance, exploit the polymorphic nature of the instance methods in the inheritance hierarchy. For example, if you find that you need to test for the type of an object in an inheritance tree, use polymorphism to avoid having to write separate code to handle objects of each class. This maximizes the reusability of your code and makes your code easier to maintain in the future.

Consider carefully how to model “roles”: for example an Employee may be a Consultant or a Trainer. This is not inheritance, these are job roles that the employee plays.



Summary

In this lesson, you should have learned how to:

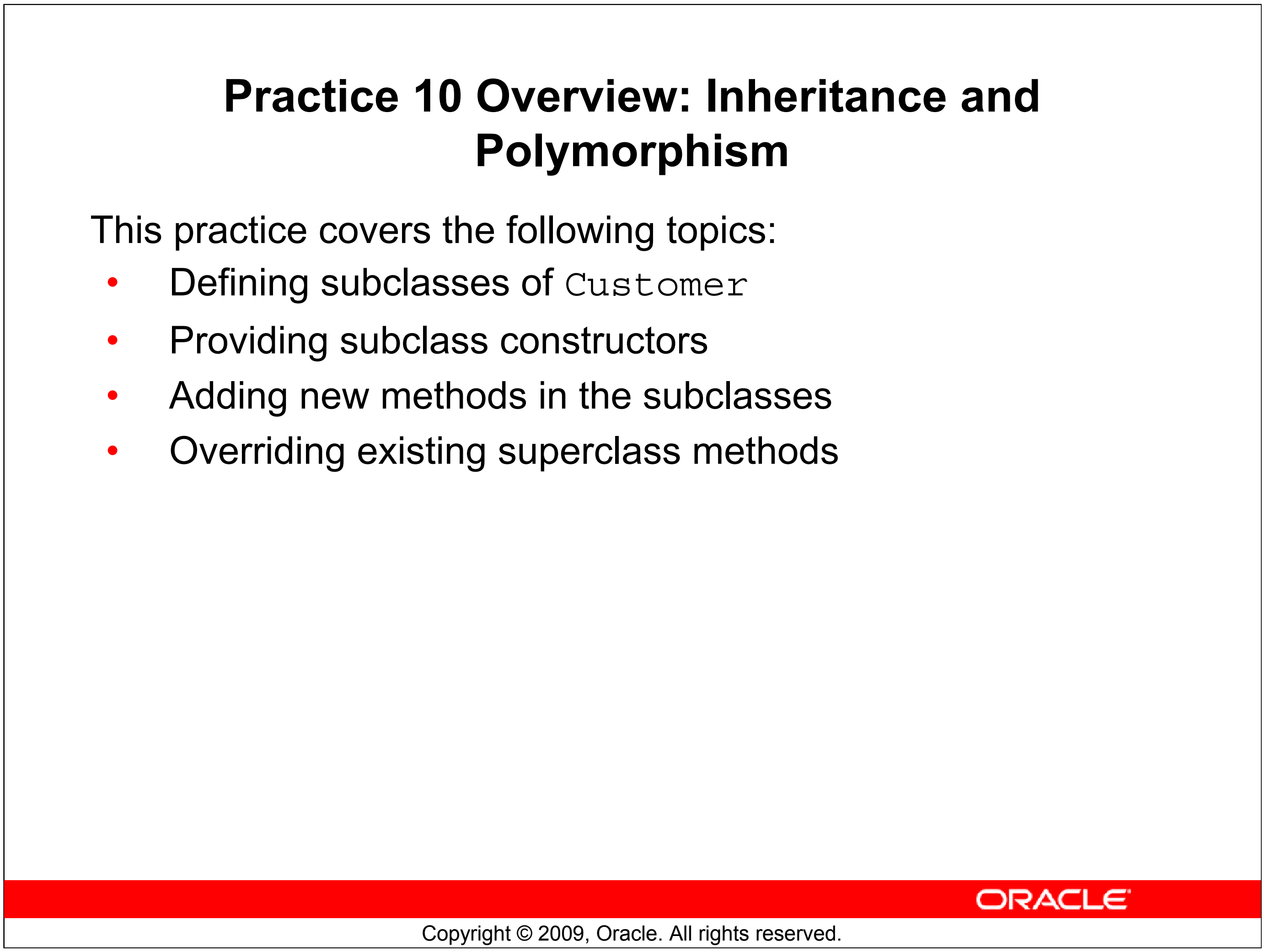
- Define inheritance and describe how it enables the reuse of existing code and thus improves productivity
- Use inheritance to define new classes
- Override superclass methods
- Use polymorphism to ensure that the correct version of a generic method is called at run time
- Apply the principles of inheritance and polymorphism to your Java applications

Summary

Inheritance and polymorphism are fundamental principles of object-oriented programming.

Inheritance describes a relationship between classes in which one class shares the data structure and behaviors of another class. It encourages the reuse of code by allowing you to create a new class (subclass) based on the properties of an existing class (superclass). You use the `extends` keyword to create a subclass based on a superclass.

Polymorphism allows a method to have multiple implementations that are selected based on the type of object that is passed into the method invocation. Acme Video has implemented the principle of polymorphism, in that the `calcDeposit()` method in the `InventoryItem` class is overridden in the `Dvd` and `Movie` classes (but is called in the `DvdPlayer` class) to provide object-specific calculation logic.



Practice 10 Overview: Inheritance and Polymorphism

This practice covers the following topics:

- Defining subclasses of `Customer`
- Providing subclass constructors
- Adding new methods in the subclasses
- Overriding existing superclass methods

Practice 10 Overview: Inheritance and Polymorphism

The goal of this practice is to understand how to create subclasses in Java and use polymorphism with inheritance through the `Company` and `Individual` subclasses of the `Customer` class. You refine the subclasses, override some methods, and add some new attributes by using the Class Editor in JDeveloper.

Note: If you have successfully completed the previous practice, continue using the same directory and files. If the compilation from the previous practice was unsuccessful and you want to move on to this practice, change to the `les10` directory, load the `OrderEntryApplicationLes10` application, and continue with this practice.

Viewing the model: To view the course application model up to this practice, load the `OrderEntryApplicationLes10` application. In the Applications – Navigator node, expand `OrderEntryApplicationLes10 – OrderEntryProjectLes10 - Application Sources – oe` and double-click the `UML Class Diagram1` entry. This diagram displays all the classes created up to this point in the course.