



Hardware and Software
Engineered to Work Together



Java SE 8 Programming

Student Guide – Volume I
D84838GC10
Edition 1.0 | December 2014 | D87757

Learn more from Oracle University at oracle.com/education/

Authors

Anjana Shenoy
Michael Williams
Tom McGinn
Peter Fernandez

Technical Contributors and Reviewers

Pete Daly
Sravanti Tatiraju
Nick Ristuccia
Stuart Marks
Hiroshi Hiraga
Peter Hall
Matthew Slingsby
Marcus Hirt
Irene Rusman
Joanne Sun
Marilyn Beck
Joe A Boulenouar

Editors

Aju Kumar
Malavika Jinka
Arijit Ghosh
Anwesha Ray

Graphic Designer

Divya Thallap

Publishers

Giri Venugopal
Michael Sebastian
Veena Narasimhan

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

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 and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Contents

1 Introduction

Course Goals 1-2
Course Objectives 1-3
Audience 1-5
Prerequisites 1-6
Class Introductions 1-7
Course Environment 1-8
Java Programs Are Platform-Independent 1-9
Java Technology Product Groups 1-10
Java SE Platform Versions 1-11
Downloading and Installing the JDK 1-12
Java in Server Environments 1-13
The Internet of Things 1-14
The Java Community 1-15
The Java Community Process (JCP) 1-16
OpenJDK 1-17
Oracle Java SE Support 1-18
Additional Resources 1-19
Summary 1-20

2 Java Syntax and Class Review

Objectives 2-2
Java Language Review 2-3
Java Class Structure 2-4
A Simple Class 2-5
Java Naming Conventions 2-6
How to Compile and Run 2-7
How to Compile and Run: Example 2-8
Code Blocks 2-9
Primitive Data Types 2-10
Numeric Literals 2-11
Operators 2-12
Logical Operators 2-13
if else Statement 2-14
switch Statement 2-15

while Loop 2-16
do-while Loop 2-17
for Loop 2-18
Arrays and for-each Loop 2-19
Strings 2-20
String Operations: StringBuilder 2-21
A Simple Java Class: Employee 2-22
Methods 2-23
Creating an Instance of a Class 2-24
Constructors 2-25
package Statement 2-26
import Statements 2-27
Java Is Pass-By-Value 2-29
Pass-By-Value for Object References 2-30
Objects Passed as Parameters 2-31
Garbage Collection 2-32
Summary 2-33
Practice 2-1 Overview: Creating Java Classes 2-34
Quiz 2-35

3 Encapsulation and Subclassing

Objectives 3-2
Encapsulation 3-3
Encapsulation: Example 3-4
Encapsulation: Public and Private Access Modifiers 3-5
Encapsulation: Private Data, Public Methods 3-6
Employee Class Refined 3-7
Make Classes as Immutable as Possible 3-8
Method Naming: Best Practices 3-9
Encapsulation: Benefits 3-10
Creating Subclasses 3-11
Subclassing 3-12
Manager Subclass 3-13
Constructors in Subclasses 3-14
Using super 3-15
Constructing a Manager Object 3-16
Overloading Methods 3-17
Overloaded Constructors 3-18
Overloaded Constructors: Example 3-19
Single Inheritance 3-20
Summary 3-21

Practice 3-1 Overview: Creating Subclasses 3-22

Quiz 3-23

4 Overriding Methods, Polymorphism, and Static Classes

Objectives 4-2

Using Access Control 4-3

Protected Access Control: Example 4-4

Access Control: Good Practice 4-5

Overriding Methods 4-6

Invoking an Overridden Method 4-8

Virtual Method Invocation 4-9

Accessibility of Overriding Methods 4-10

Applying Polymorphism 4-11

Using the instanceof Keyword 4-13

Overriding Object methods 4-14

Object toString Method 4-15

Object equals Method 4-16

Overriding equals in Employee 4-17

Overriding Object hashCode 4-18

Methods Using Variable Arguments 4-19

Casting Object References 4-21

Upward Casting Rules 4-22

Downward Casting Rules 4-23

static Keyword 4-24

Static Methods 4-25

Using Static Variables and Methods: Example 4-26

Implementing Static Methods 4-27

Calling Static Methods 4-28

Static Variables 4-29

Defining Static Variables 4-30

Using Static Variables 4-31

Static Initializers 4-32

Static Imports 4-33

Design Patterns 4-34

Singleton Pattern 4-35

Singleton: Example 4-36

Immutable Classes 4-37

Example: Creating Immutable class in Java 4-38

Summary 4-39

Practice 4-1 Overview: Overriding Methods and Applying Polymorphism 4-40

Practice 4-2 Overview: Overriding Methods and Applying Polymorphism 4-41

Practice 4-3 Overview: Applying the Singleton Design Pattern 4-42
Quiz 4-43

5 Abstract and Nested Classes

Objectives 5-2
Modeling Business Problems with Classes 5-3
Enabling Generalization 5-4
Identifying the Need for Abstract Classes 5-5
Defining Abstract Classes 5-6
Defining Abstract Methods 5-7
Validating Abstract Classes 5-8
Final Methods 5-9
Final Classes 5-10
Final Variables 5-11
Declaring Final Variables 5-12
Nested Classes 5-13
Example: Member Class 5-14
Enumerations 5-15
Enum Usage 5-16
Complex Enums 5-17
Summary 5-19
Practice 5-1 Overview: Applying the Abstract Keyword 5-20
Practice 5-2 Overview: Using Inner Class As a Helper Class 5-21
Practice 5-3 Overview: Using Java Enumerations 5-22
Quiz 5-23

6 Interfaces and Lambda Expressions

Objectives 6-2
Java Interfaces 6-3
A Problem Solved by Interfaces 6-4
CrushedRock Class 6-5
The SalesCalcs Interface 6-6
Adding an Interface 6-7
Interface References 6-8
Interface Reference Usefulness 6-9
Interface Code Flexibility 6-10
default Methods in Interfaces 6-11
default Method: Example 6-12
static Methods in Interfaces 6-13
Constant Fields 6-14
Extending Interfaces 6-15

Implementing and Extending 6-16
Anonymous Inner Classes 6-17
Anonymous Inner Class: Example 6-18
String Analysis Regular Class 6-19
String Analysis Regular Test Class 6-20
String Analysis Interface: Example 6-21
String Analyzer Interface Test Class 6-22
Encapsulate the for Loop 6-23
String Analysis Test Class with Helper Method 6-24
String Analysis Anonymous Inner Class 6-25
String Analysis Lambda Expression 6-26
Lambda Expression Defined 6-27
What Is a Lambda Expression? 6-28
Lambda Expression Shorthand 6-31
Lambda Expressions as Variables 6-32
Summary 6-33
Practice 6-1: Implementing an Interface 6-34
Practice 6-2: Using Java Interfaces 6-35
Practice 6-3: Creating Lambda Expression 6-36
Quiz 6-37

7 Generics and Collections

Objectives 7-2
Topics 7-3
Generics 7-4
Simple Cache Class Without Generics 7-5
Generic Cache Class 7-6
Generics in Action 7-7
Generics with Type Inference Diamond 7-8
Collections 7-9
Collection Types 7-10
Collection Interfaces and Implementation 7-11
List Interface 7-12
ArrayList 7-13
Autoboxing and Unboxing 7-14
ArrayList Without Generics 7-15
Generic ArrayList 7-16
Generic ArrayList: Iteration and Boxing 7-17
Set Interface 7-18
TreeSet: Implementation of Set 7-19
Map Interface 7-20

Map Types 7-21
TreeMap: Implementation of Map 7-22
Deque Interface 7-23
Stack with Deque: Example 7-24
Ordering Collections 7-25
Comparable: Example 7-26
Comparable Test: Example 7-27
Comparator Interface 7-28
Comparator: Example 7-29
Comparator Test: Example 7-30
Summary 7-31
Practice 7-1 Overview: Counting Part Numbers by Using a HashMap 7-32
Practice 7-2 Overview: Implementing Stack by Using a Deque Object 7-33
Quiz 7-34

8 Collections, Streams, and Filters

Objectives 8-2
Collections, Streams, and Filters 8-3
The Person Class 8-4
Person Properties 8-5
Builder Pattern 8-6
Collection Iteration and Lambdas 8-7
RoboCallTest07: Stream and Filter 8-8
RobocallTest08: Stream and Filter Again 8-9
SalesTxn Class 8-10
Java Streams 8-11
The Filter Method 8-12
Method References 8-13
Method Chaining 8-14
Pipeline Defined 8-16
Summary 8-17
Practice Overview 8-18

9 Lambda Built-in Functional Interfaces

Objectives 9-2
Built-in Functional Interfaces 9-3
The java.util.function Package 9-4
Example Assumptions 9-5
Predicate 9-6
Predicate: Example 9-7
Consumer 9-8

Consumer: Example 9-9
Function 9-10
Function: Example 9-11
Supplier 9-12
Supplier: Example 9-13
Primitive Interface 9-14
Return a Primitive Type 9-15
Return a Primitive Type: Example 9-16
Process a Primitive Type 9-17
Process Primitive Type: Example 9-18
Binary Types 9-19
Binary Type: Example 9-20
Unary Operator 9-21
UnaryOperator: Example 9-22
Wildcard Generics Review 9-23
Summary 9-24
Practice Overview 9-25

10 Lambda Operations

Objectives 10-2
Streams API 10-3
Types of Operations 10-4
Extracting Data with Map 10-5
Taking a Peek 10-6
Search Methods: Overview 10-7
Search Methods 10-8
Optional Class 10-9
Lazy Operations 10-10
Stream Data Methods 10-11
Performing Calculations 10-12
Sorting 10-13
Comparator Updates 10-14
Saving Data from a Stream 10-15
Collectors Class 10-16
Quick Streams with Stream.of 10-17
Flatten Data with flatMap 10-18
Summary 10-19
Practice Overview 10-20

11 Exceptions and Assertions

Objectives 11-2

Error Handling 11-3
Exception Handling in Java 11-4
try-catch Statement 11-5
Exception Objects 11-6
Exception Categories 11-7
Handling Exceptions 11-8
finally Clause 11-9
try-with-resources Statement 11-10
Catching Multiple Exceptions 11-11
Declaring Exceptions 11-12
Handling Declared Exceptions 11-13
Throwing Exceptions 11-14
Custom Exceptions 11-15
Assertions 11-16
Assertion Syntax 11-17
Internal Invariants 11-18
Control Flow Invariants 11-19
Class Invariants 11-20
Controlling Runtime Evaluation of Assertions 11-21
Summary 11-22
Practice 11-1 Overview: Catching Exceptions 11-23
Practice 11-2 Overview: Extending Exception and Using throw and throws 11-24
Quiz 11-25

12 Java Date/Time API

Objectives 12-2
Why Is Date and Time Important? 12-3
Previous Java Date and Time 12-4
Java Date and Time API: Goals 12-5
Working with Local Date and Time 12-6
Working with LocalDate 12-7
LocalDate: Example 12-8
Working with LocalTime 12-9
LocalTime: Example 12-10
Working with LocalDateTime 12-11
LocalTimeDate: Example 12-12
Working with Time Zones 12-13
Daylight Savings Time Rules 12-14
Modeling Time Zones 12-15
Creating ZonedDateTime Objects 12-16
Working with ZonedDateTime Gaps/Overlaps 12-17

ZoneRules 12-18
Working Across Time Zones 12-19
Date and Time Methods 12-20
Date and Time Amounts 12-21
Period 12-22
Duration 12-23
Calculating Between Days 12-24
Making Dates Pretty 12-25
Using Fluent Notation 12-26
Summary 12-27
Practices 12-28

13 Java I/O Fundamentals

Objectives 13-2
Java I/O Basics 13-3
I/O Streams 13-4
I/O Application 13-5
Data Within Streams 13-6
Byte Stream InputStream Methods 13-7
Byte Stream OutputStream Methods 13-8
Byte Stream: Example 13-9
Character Stream Reader Methods 13-10
Character Stream Writer Methods 13-11
Character Stream: Example 13-12
I/O Stream Chaining 13-13
Chained Streams: Example 13-14
Console I/O 13-15
Writing to Standard Output 13-16
Reading from Standard Input 13-17
Channel I/O 13-18
Persistence 13-19
Serialization and Object Graphs 13-20
Transient Fields and Objects 13-21
Transient: Example 13-22
Serial Version UID 13-23
Serialization: Example 13-24
Writing and Reading an Object Stream 13-25
Serialization Methods 13-26
readObject: Example 13-27
Summary 13-28
Practice 13-1 Overview: Writing a Simple Console I/O Application 13-29

Practice 13-2 Overview: Serializing and Deserializing a ShoppingCart 13-30
Quiz 13-31

14 Java File I/O (NIO.2)

Objectives 14-2
New File I/O API (NIO.2) 14-3
Limitations of java.io.File 14-4
File Systems, Paths, Files 14-5
Relative Path Versus Absolute Path 14-6
Java NIO.2 Concepts 14-7
Path Interface 14-8
Path Interface Features 14-9
Path: Example 14-10
Removing Redundancies from a Path 14-11
Creating a Subpath 14-12
Joining Two Paths 14-13
Symbolic Links 14-14
Working with Links 14-15
File Operations 14-16
Checking a File or Directory 14-17
Creating Files and Directories 14-19
Deleting a File or Directory 14-20
Copying a File or Directory 14-21
Moving a File or Directory 14-22
List the Contents of a Directory 14-23
Walk the Directory Structure 14-24
BufferedReader File Stream 14-25
NIO File Stream 14-26
Read File into ArrayList 14-27
Managing Metadata 14-28
Symbolic Links 14-29
Summary 14-30
Practice Overview 14-31
Quiz 14-33

15 Concurrency

Objectives 15-2
Task Scheduling 15-3
Legacy Thread and Runnable 15-4
Extending Thread 15-5
Implementing Runnable 15-6

The java.util.concurrent Package	15-7
Recommended Threading Classes	15-8
java.util.concurrent.ExecutorService	15-9
Example ExecutorService	15-10
Shutting Down an ExecutorService	15-11
java.util.concurrent.Callable	15-12
Example Callable Task	15-13
java.util.concurrent.Future	15-14
Example	15-15
Threading Concerns	15-16
Shared Data	15-17
Problems with Shared Data	15-18
Nonshared Data	15-19
Atomic Operations	15-20
Out-of-Order Execution	15-21
The synchronized Keyword	15-22
synchronized Methods	15-23
synchronized Blocks	15-24
Object Monitor Locking	15-25
Threading Performance	15-26
Performance Issue: Examples	15-27
java.util.concurrent Classes and Packages	15-28
The java.util.concurrent.atomic Package	15-29
java.util.concurrent.CyclicBarrier	15-30
Thread-Safe Collections	15-32
CopyOnWriteArrayList: Example	15-33
Summary	15-34
Practice 15-1 Overview: Using the java.util.concurrent Package	15-35
Quiz	15-36

16 The Fork-Join Framework

Objectives	16-2
Parallelism	16-3
Without Parallelism	16-4
Naive Parallelism	16-5
The Need for the Fork-Join Framework	16-6
Work-Stealing	16-7
A Single-Threaded Example	16-8
java.util.concurrent.ForkJoinTask<V>	16-9
RecursiveTask Example	16-10
compute Structure	16-11

compute Example (Below Threshold)	16-12
compute Example (Above Threshold)	16-13
ForkJoinPool Example	16-14
Fork-Join Framework Recommendations	16-15
Summary	16-16
Practice 16-1 Overview: Using the Fork-Join Framework	16-17
Quiz	16-18

17 Parallel Streams

Objectives	17-2
Streams Review	17-3
Old Style Collection Processing	17-4
New Style Collection Processing	17-5
Stream Pipeline: Another Look	17-6
Styles Compared	17-7
Parallel Stream	17-8
Using Parallel Streams: Collection	17-9
Using Parallel Streams: From a Stream	17-10
Pipelines Fine Print	17-11
Embrace Statelessness	17-12
Avoid Statefulness	17-13
Streams Are Deterministic for Most Part	17-14
Some Are Not Deterministic	17-15
Reduction	17-16
Reduction Fine Print	17-17
Reduction: Example	17-18
A Look Under the Hood	17-24
Illustrating Parallel Execution	17-25
Performance	17-36
A Simple Performance Model	17-37
Summary	17-38
Practice	17-39

18 Building Database Applications with JDBC

Objectives	18-2
Using the JDBC API	18-3
Using a Vendor's Driver Class	18-4
Key JDBC API Components	18-5
Writing Queries and Getting Results	18-6
Using a ResultSet Object	18-7
CRUD Operations Using JDBC API: Retrieve	18-8

CRUD Operations Using JDBC: Retrieve	18-9
CRUD Operations Using JDBC API: Create	18-10
CRUD Operations Using JDBC API: Update	18-11
CRUD Operations Using JDBC API: Delete	18-12
SQLException Class	18-13
Closing JDBC Objects	18-14
try-with-resources Construct	18-15
Using PreparedStatement	18-16
Using PreparedStatement: Setting Parameters	18-17
Executing PreparedStatement	18-18
PreparedStatement: Using a Loop to Set Values	18-19
Using CallableStatement	18-20
Summary	18-21
Practice 18-1 Overview: Working with the Derby Database and JDBC	18-22
Quiz	18-23

19 Localization

Objectives	19-2
Why Localize?	19-3
A Sample Application	19-4
Locale	19-5
Properties	19-6
Loading and Using a Properties File	19-7
Loading Properties from the Command Line	19-8
Resource Bundle	19-9
Resource Bundle File	19-10
Sample Resource Bundle Files	19-11
Initializing the Sample Application	19-12
Sample Application: Main Loop	19-13
The printMenu Method	19-14
Changing the Locale	19-15
Sample Interface with French	19-16
Format Date and Currency	19-17
Displaying Currency	19-18
Formatting Currency with NumberFormat	19-19
Displaying Dates	19-20
Displaying Dates with DateTimeFormatter	19-21
Format Styles	19-22
Summary	19-23
Practice 19-1 Overview: Creating a Localized Date Application	19-24
Quiz	19-25

1

Introduction

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Course Goals

- This course covers the core APIs that you use to design object-oriented applications with Java. This course also covers writing database programs with JDBC.
- Use this course to further develop your skills with the Java language and prepare for the Oracle Certified Professional, Java SE 8 Programmer Exam.

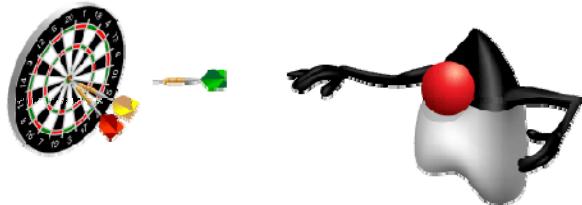


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Course Objectives

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

- Create Java technology applications that leverage the object-oriented features of the Java language, such as encapsulation, inheritance, and polymorphism
- Execute a Java application from the command line
- Create applications that use the Collections framework
- Search and filter collections by using Lambda Expressions
- Implement error-handling techniques by using exception handling



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Course Objectives

- Implement input/output (I/O) functionality to read from and write to data and text files and understand advanced I/O streams
- Manipulate files, directories, and file systems by using the NIO.2 specification
- Perform multiple operations on database tables, including creating, reading, updating, and deleting, by using the JDBC API
- Create high-performing multithreaded applications that avoid deadlock
- Use Lambda Expression concurrency features



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Audience

The target audience includes those who have:

- Completed the *Java SE 8 Fundamentals* course or have experience with the Java language, and can create, compile, and execute programs
- Experience with at least one programming language
- An understanding of object-oriented principles
- Experience with basic database concepts and a basic knowledge of SQL



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Prerequisites

To successfully complete this course, you must know how to:

- Compile and run Java applications
- Create Java classes
- Create object instances by using the `new` keyword
- Declare Java primitive and reference variables
- Declare Java methods by using return values and parameters
- Use conditional constructs such as `if` and `switch` statements
- Use looping constructs such as `for`, `while`, and `do` loops
- Declare and instantiate Java arrays
- Use the Java Platform, Standard Edition API Specification (Javadocs)



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Class Introductions

Briefly introduce yourself:

- Name
- Title or position
- Company
- Experience with Java programming and Java applications
- Reasons for attending



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Course Environment



Classroom PC

Core Apps

- JDK 8
- NetBeans 8.0

Additional Tools

- Firefox
- Java DB (Derby)
- Oracle Linux

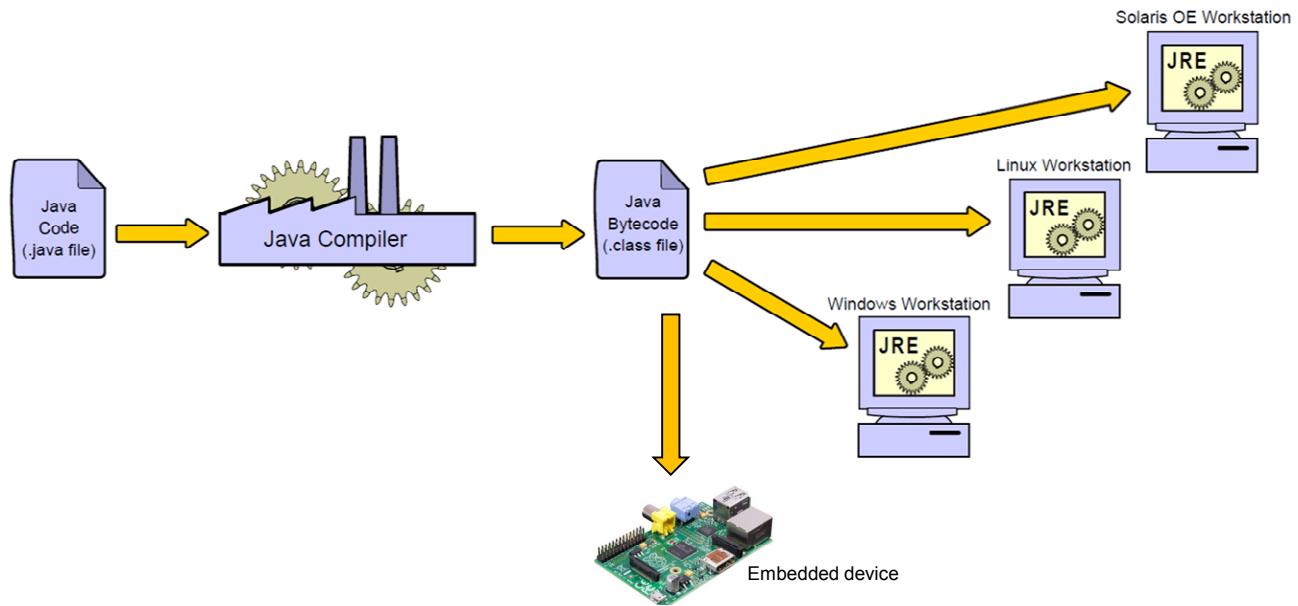
ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In this course, the following products are preinstalled for the lesson practices:

- **JDK 8:** The Java SE Development Kit includes the command-line Java compiler (`javac`) and the Java Runtime Environment (JRE), which supplies the `java` command needed to execute Java applications.
- **Firefox:** A web browser is used to view the HTML documentation (Javadoc) for the Java SE Platform libraries.
- **NetBeans 8.0:** The NetBeans IDE is a free and open-source software development tool for professionals who create enterprise, web, desktop, and mobile applications. NetBeans 7.0.1 fully supports the Java SE 7 Platform. Support is provided by Oracle's Development Tools Support offering.
- **Java DB:** Java DB is Oracle's supported distribution of the open-source Apache Derby 100% Java technology database. It is fully transactional, secure, easy-to-use, standards-based SQL, JDBC API, and Java EE, yet small (only 2.5 MB).
- **Oracle Linux:** Oracle's enterprise implementation of Linux. Compatible with RedHat Linux.

Java Programs Are Platform-Independent



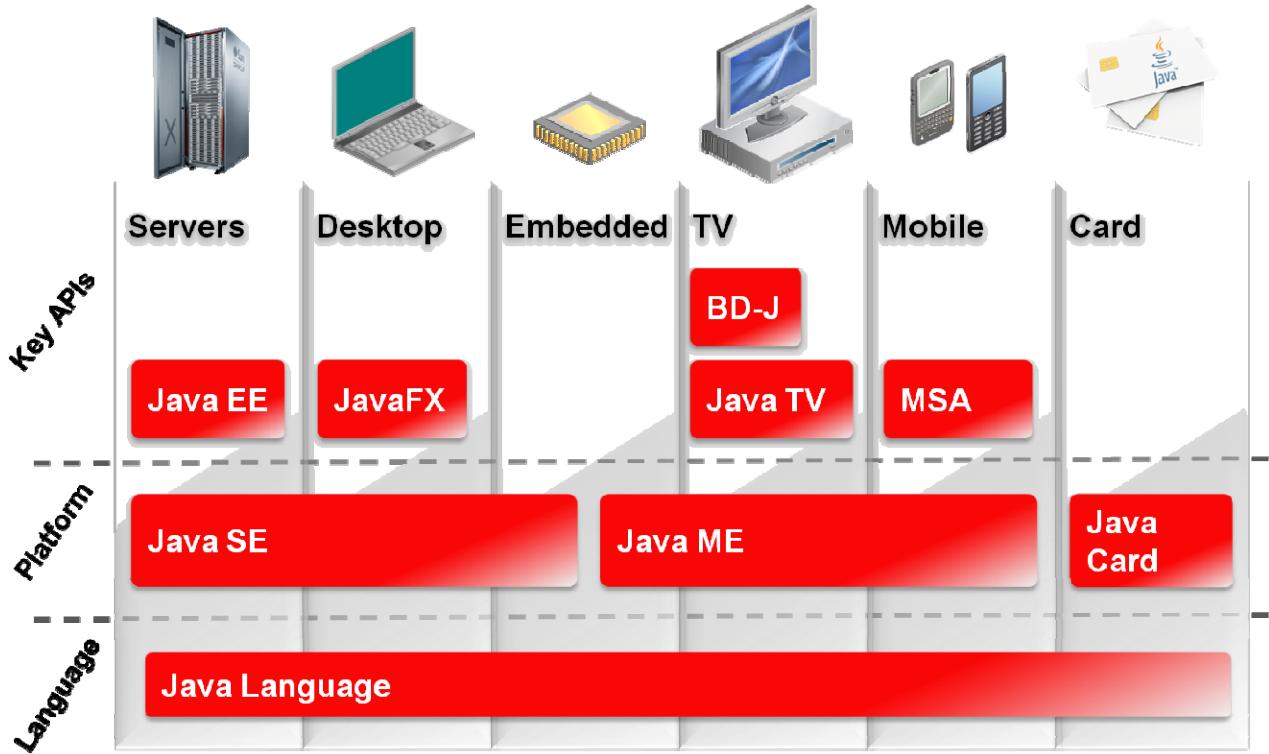
ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Platform-Independent Programs

Java technology applications are written in the Java programming language and compiled to Java bytecode. Bytecode is executed on the Java platform. The software that provides you with a runnable Java platform is called a Java Runtime Environment (JRE). A compiler, included in the Java SE Development Kit (JDK), is used to convert Java source code to Java bytecode.

Java Technology Product Groups



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Identifying Java Technology Groups

Oracle provides a complete line of Java technology products ranging from kits that create Java technology programs to emulation (testing) environments for consumer devices, such as cellular phones. As indicated in the graphic, all Java technology products share the foundation of the Java language. Java technologies, such as the Java Virtual Machine, are included (in different forms) in three different groups of products, each designed to fulfill the needs of a particular target market. The figure illustrates the three Java technology product groups and their target device types. Among other Java technologies, each edition includes a Software Development Kit (SDK) that allows programmers to create, compile, and execute Java technology programs on a particular platform:

- **Java Platform, Standard Edition (Java SE)**: Develops applets and applications that run within web browsers and on desktop computers, respectively. For example, you can use the Java SE Software Development Kit (SDK) to create a word-processing program for a personal computer. You can also use the Java SE to create an application that runs in a browser.

Note: Applets and applications differ in several ways. Primarily, applets are launched inside a web browser, whereas applications are launched within an operating system.

Java SE Platform Versions

Year	Developer Version (JDK)	Platform
1996	1.0	1
1997	1.1	1
1998	1.2	2
2000	1.3	2
2002	1.4	2
2004	1.5	5
2006	1.6	6
2011	1.7	7
2014	1.8	8

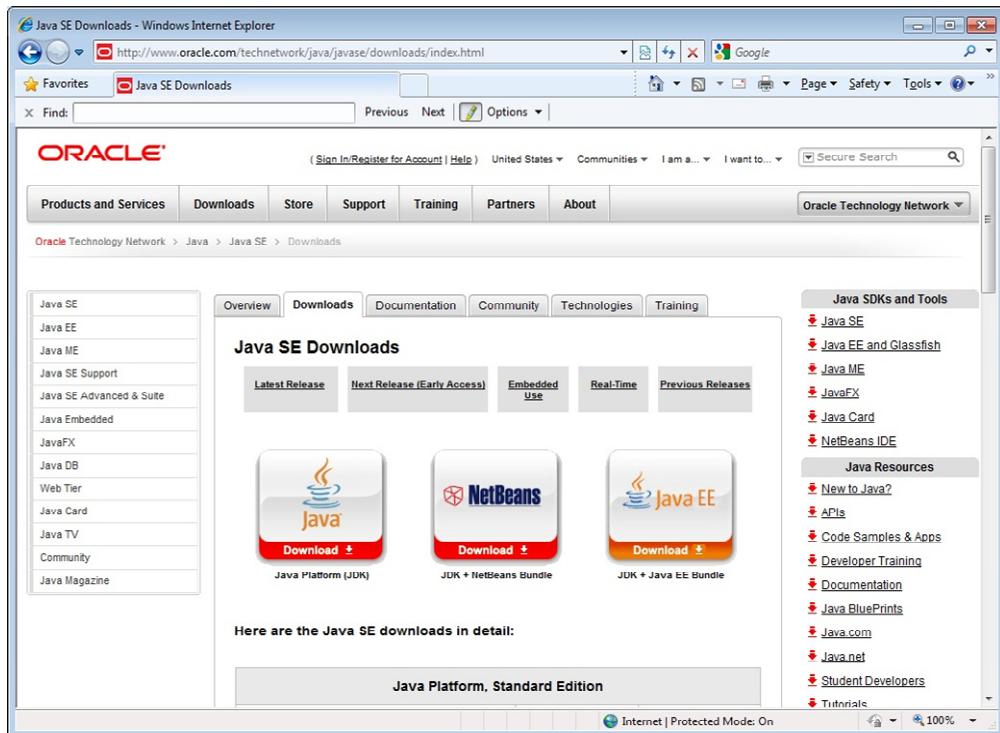


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

How to Detect Your Version

If Java SE is installed on your system, you can detect the version number by running `java -version`. Note that the `java` command is included with the Java Runtime Environment (JRE). As a developer, you also need a Java compiler, typically `javac`. The `javac` command is included in the Java SE Development Kit (JDK). Your operation system's PATH may need to be updated to include the location of `javac`.

Downloading and Installing the JDK



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

ORACLE

1. Go to <http://www.oracle.com/technetwork/java/javase/downloads/index.html>.
2. Choose the Java Platform, Standard Edition (Java SE) link.
3. Download the version that is appropriate for your operation system.
4. Follow the installation instructions.
5. Set your PATH.

Java in Server Environments



Java is common in enterprise environments:

- Oracle Fusion Middleware
 - Java application servers
 - GlassFish
 - WebLogic
- Database servers
 - MySQL
 - Oracle Database



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

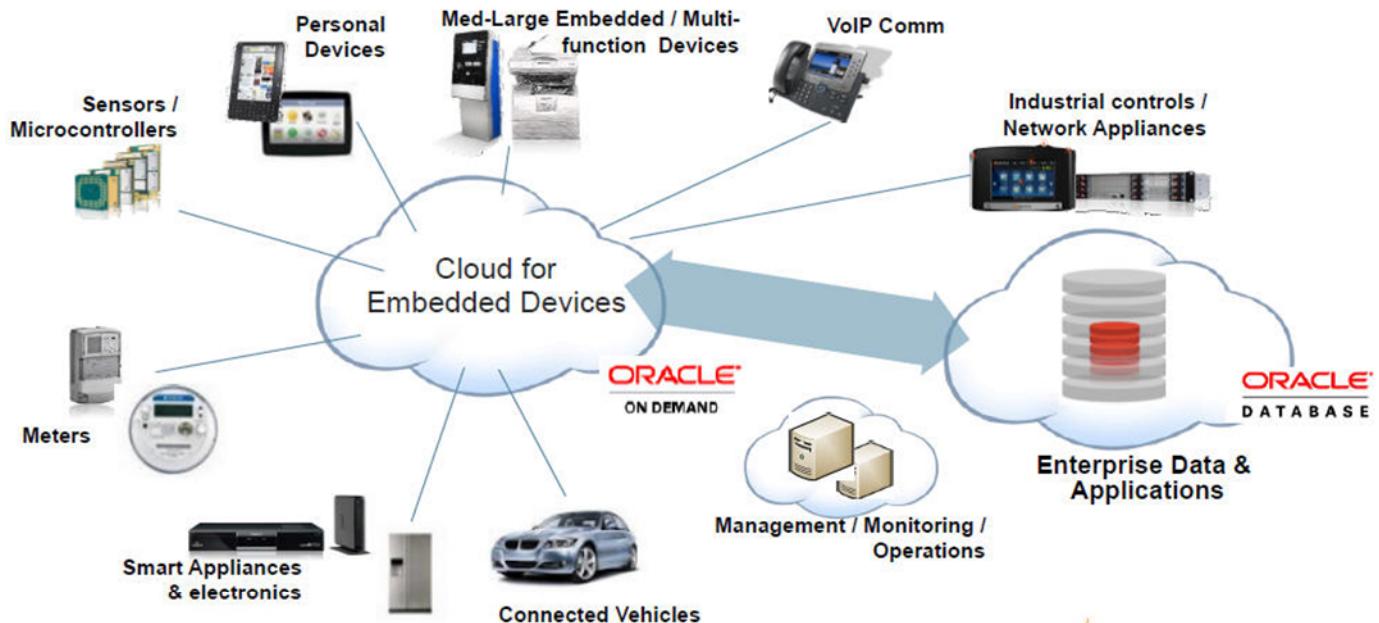
Enterprise Environments

In this course, you develop Java SE applications. There are standard patterns you need to follow when implementing Java SE applications, such as always creating a `main` method that may be different when implementing enterprise applications. Java SE is only the starting point in your path to becoming a Java developer. Depending on the needs of your organization, you may be required to develop applications that run inside Java EE application servers or other types of Java middleware.

Often, you will also need to manipulate information stored inside relational databases such as MySQL or Oracle Database. This course introduces you to the fundamentals of database programming.

The Internet of Things

Devices on the “edge” represent a huge growth opportunity.



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Embedded Devices

Java SE 8 and Java ME 8 now share a common code base, and both are available in reduced footprint formats for embedded devices. An embedded device is typically a small controller or sensor that is part of a larger device. Embedded applications provide the intelligence required to support electrical and/or mechanical functions of an embedded device.

Some estimates put the number of connected devices at the edge of the network at 50 billion by the year 2020.

The Java Community



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

What Is the Java Community?

At a very high level, *Java Community* is the term used to refer to the many individuals and organizations that develop, innovate, and use Java technology. This community includes developers as individuals, organizations, businesses, and open-source projects.

It is very common for you to download and use Java libraries from non-Oracle sources within the Java community. For instance, in this course, you use an Apache-developed JDBC library to access a relational database.

The Java Community Process (JCP)

The JCP is used to develop new Java standards:

- <http://jcp.org>
- Free download of all Java Specification Requests (JSRs)
- Early access to specifications
- Public review and feedback opportunities
- Open membership



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

JCP.next

The JCP produces the JSRs that outline the standards of the Java platform. The behavior of the JCP itself is also defined and improved through the JSR process. The JCP is evolving and its improvements are defined in JSR-348. JSR-348 introduces changes in the areas of transparency, participation, agility, and governance.

- **Transparency:** In the past, some aspects of the development of a JSR may have occurred behind closed doors. Transparent development is now the recommended practice.
- **Participation:** Individuals and Java User Groups are encouraged to become active in the JCP.
- **Agility:** Slow-moving JSRs are now actively discouraged.
- **Governance:** The SE and ME expert groups are merging into a single body.

OpenJDK

OpenJDK is the open-source implementation of Java:

- <http://openjdk.java.net/>
- GPL licensed open-source project
- JDK reference implementation
- Where new features are developed
- Open to community contributions
- Basis for Oracle JDK



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Why OpenJDK Is Important

Because it is open source, OpenJDK enables users to port Java to operating systems and hardware platforms of their choosing. Ports are underway for many platforms (besides those already supported) including FreeBSD, OpenBSD, NetBSD, and MacOS X.

Oracle Java SE Support

Java is available free of charge. However, Oracle does provide pay-for Java solutions:

- The Java SE Support Program provides updates for end-of-life Java versions.
- Oracle Java SE Advanced and Oracle Java SE Suite:
 - JRockit Mission Control
 - Memory Leak Detection
 - Low Latency GC (Suite)
 - JRockit Virtual Edition (Suite)



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Still Free

Java (Oracle JDK) is freely available at no cost. Oracle offers advanced commercial solutions at cost. The previously offered “Java for Business” program has been replaced by Oracle Java SE Support, which provides access to Oracle Premier Support and the Oracle Java SE Advanced and Oracle Java SE Suite binaries. For more information, visit <http://www.oracle.com/us/technologies/java/java-se-suite-394230.html>.

Additional Resources

Topic	Website
Education and Training	http://education.oracle.com
Product Documentation	http://www.oracle.com/technology/documentation
Product Downloads	http://www.oracle.com/technology/software
Product Articles	http://www.oracle.com/technology/pub/articles
Product Support	http://www.oracle.com/support
Product Forums	http://forums.oracle.com
Product Tutorials	http://www.oracle.com/technetwork/tutorials/index.html
Sample Code	https://www.samplecode.oracle.com
Oracle Technology Network for Java Developers	http://www.oracle.com/technetwork/java/index.html
Oracle Learning Library	http://www.oracle.com/goto/oll



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The table in the slide lists various web resources that are available for you to learn more about Java SE programming.

Summary

In this lesson, you should have learned about:

- The course objectives
- Software used in this course
- Java platforms (ME, SE, and EE)
- Java SE version numbers
- Obtaining a JDK
- The open nature of Java and its community
- Commercial support options for Java SE



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Java Syntax and Class Review

2

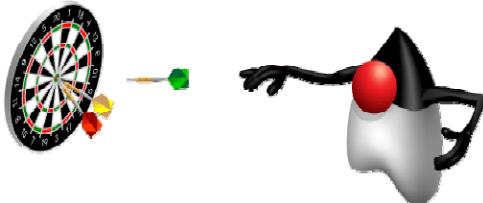


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Objectives

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

- Create simple Java classes
 - Create primitive variables
 - Use operators
 - Create and manipulate strings
 - Manage Flow Control:
 - Use `if-else` and `switch` statements
 - Iterate with loops: `while`, `do-while`, `for`, `enhanced for`
 - Create arrays
- Use Java fields, constructors, and methods
- Use `package` and `import` statements



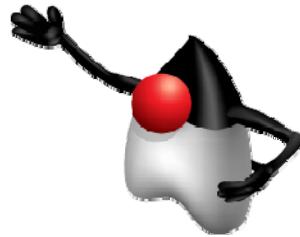
ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Java Language Review

This lesson is a review of fundamental Java and programming concepts. It is assumed that students are familiar with the following concepts:

- The basic structure of a Java class
- Program block and comments
- Variables
- Basic `if-else` and `switch` branching constructs
- Iteration with `for` and `while` loops



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Java Class Structure

```
package <package_name>;  
  
import <other_packages>;  
  
public class ClassName {  
    <variables(also known as fields)>;  
  
    <constructor(s)>;  
  
    <other methods>;  
}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

A Java class is described in a text file with a `.java` extension. In the example shown, the Java keywords are highlighted in bold.

- The `package` keyword defines where this class lives relative to other classes, and provides a level of access control. You use access modifiers (such as `public` and `private`) later in this lesson.
- The `import` keyword defines other classes or groups of classes that you are using in your class. The `import` statement helps to narrow what the compiler needs to look for when resolving class names used in this class.
- The `class` keyword precedes the name of this class. The name of the class and the file name must match when the class is declared `public` (which is a good practice). However, the keyword `public` in front of the `class` keyword is a modifier and is not required.
- Variables, or the data associated with programs (such as integers, strings, arrays, and references to other objects), are called *instance fields* (often shortened to *fields*).
- Constructors are functions called during the creation (instantiation) of an object (a representation in memory of a Java class).
- Methods are functions that can be performed on an object. They are also referred to as *instance methods*.

A Simple Class

A simple Java class with a `main` method:

```
public class Simple {  
  
    public static void main(String args[]) {  
  
    }  
}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

To run a Java program, you must define a `main` method in the Java class as shown in the slide. The `main` method is automatically called when the class is called from the command line.

Command-line arguments are passed to the program through the `args []` array.

Note: A method that is modified with the keyword `static` is invoked without a reference to a particular object. The class name is used instead. These methods are referred to as *class methods*. The `main` method is a special method that is invoked when this class is run by using the Java runtime.

Java Naming Conventions

```
1 public class CreditCard {  
2     public final int VISA = 5001;  
3     public String accountName;  
4     public String cardNumber;  
5     public Date expDate;  
6  
7     public double getCharges(){  
8         // ...  
9     }  
10  
11    public void disputeCharge(String chargeId, float amount){  
12        // ...  
13    }  
14}
```

Class names are nouns in upper camel case.

Constants should be declared in all uppercase letters

Variable names are short but meaningful in lower camel case.

Methods should be verbs, in lower camel case.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

- Class names should be nouns in mixed case, with the first letter uppercase and the first letter of each internal word capitalized. This approach is termed "upper camel case."
- Methods should be verbs in mixed case, with the first letter lowercase and the first letter of each internal word capitalized. This is termed "lower camel case."
- Variable names should be short but meaningful. The choice of a variable name should be mnemonic: designed to indicate to the casual observer the intent of its use.
- One-character variable names should be avoided except as temporary "throwaway" variables.
- Constants should be declared by using all uppercase letters.

Note: The keyword `final` is used to declare a variable whose value may only be assigned once. Once a `final` variable has been assigned, it always contains the same value. You will learn more about the keyword `final` in the lesson "Advanced Class Design."

For the complete *Code Conventions for the Java Programming Language* document, go to <http://www.oracle.com/technetwork/java/codeconv-138413.html>.

How to Compile and Run

Java class files must be compiled before running them.

To compile a Java source file, use the Java compiler (`javac`).

```
javac -cp <path to other classes> -d <compiler output path> <path to source>.java
```

- You can use the `CLASSPATH` environment variable to the directory above the location of the package hierarchy.
- After compiling the source `.java` file, a `.class` file is generated.
- To run the Java application, run it using the Java interpreter (`java`):

```
java -cp <path to other classes> <package name>.<classname>
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

CLASSPATH

The default value of the `classpath` is the current working directory (.); however, specifying the `CLASSPATH` variable or the `-cp` command line switch overrides this value.

The `CLASSPATH` variable is used by both the Java compiler and the Java interpreter (runtime).

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 on 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.

Note: The directory containing the root name of the package tree must be added to the `classpath`. Consider putting `classpath` information in the command window or even in the Java command, rather than hard-coding it in the environment.

How to Compile and Run: Example

- Assume that the class shown in the notes is in the directory test in the path /home/oracle:

```
$ javac HelloWorld.java
```

- To run the application, you use the interpreter and the class name:

```
$ java HelloWorld  
Hello World
```

- The advantage of an IDE like NetBeans is that management of the class path, compilation, and running the Java application are handled through the tool.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Example

Consider the following simple class in a file named `HelloWorld.java` in the test directory in the path /home/oracle:

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

Code Blocks

- Every class declaration is enclosed in a code block.
- Method declarations are enclosed in code blocks.
- Java fields and methods have block (or class) scope.
- Code blocks are defined in braces:

```
{ }
```

Example:

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



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Java fields (variables) and methods have a class scope defined by the opening left curly brace and ending at the closing right curly brace.

Class scope allows any method in the class to call or invoke any other method in the class.
Class scope also allows any method to access any field in the class.

Code blocks are always defined by using braces { }. A block is executed by executing each of the statements defined within the block in order from first to last (left to right).

The Java compiler discards unnecessary white space in a line of code. Line indentation is not required but makes the code much more readable. In this course, the line indentation is four spaces, which is the default line indentation used by the NetBeans IDE.

Primitive Data Types

Integer	Floating Point	Character	True False
byte short int long	float double	char	boolean
1, 2, 3, 42 7L 0xff 0b or 0B	3.0 22.0F .3337F 4.022E23	'a' \u0061' \n'	true false

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



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

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 of floating points: float and double.

double is called so because it provides double the precision of float. A float uses 32 bits to store data, whereas a double uses 64 bits.

Numeric Literals

- Any number of underscore characters (_) can appear between digits in a numeric field.
- This can improve the readability of your code.

```
long creditCardNumber = 1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
long hexBytes = 0xFF_EC_DE_5E;
long hexWords = 0xCAFE_BABE;
long maxLong = 0x7fff_ffff_ffff_ffffL;
byte nybbles = 0b0010_0101;
long bytes = 0b11010010_01101001_10010100_10010010;
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Rules for Literals

You can place underscores only between digits; you cannot place underscores in the following places:

- At the beginning or end of a number
- Adjacent to a decimal point in a floating point literal
- Prior to an F or L suffix
- In positions where a string of digits is expected

Note: The Java language is case-sensitive. In Java, the variable `creditCardNumber` is different from `CREDITCARDNUMBER`. Convention indicates that Java variables and method names use “lower camel case”—lowercase for the first letter of the first element of a variable name and uppercase for the first letter of subsequent elements.

Operators

- Simple assignment operator
 - = Simple assignment operator
- Arithmetic operators
 - + Additive operator (also used for String concatenation)
 - Subtraction operator
 - * Multiplication operator
 - / Division operator
 - % Remainder operator
- Unary operators
 - + Unary plus operator; indicates positive
 - Unary minus operator; negates an expression
 - ++ Increment operator; increments a value by 1
 - Decrement operator; decrements a value by 1
 - ! Logical complement operator; inverts the value of a boolean



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Because numbers have been introduced, the slide shows a list of common operators. Most are common to any programming language, and a description of each is provided in the slide.

The binary and bitwise operators have been omitted for brevity. For details about those operators, refer to the Java Tutorial:

<http://download.oracle.com/javase/tutorial/java/nutsandbolts/operators.html>

Note: Operators have definitive precedence. For the complete list, see the Java Tutorial link mentioned above.

Precedence can be overridden using parentheses.

Logical Operators

- Equality and relational operators
 - == Equal to
 - != Not equal to
 - > Greater than
 - >= Greater than or equal to
 - < Less than
 - <= Less than or equal to
- Conditional operators
 - && Conditional-AND
 - || Conditional-OR
 - ? : Ternary (shorthand for `if-then-else` statement)
- Type comparison operator
 - `instanceof` Compares an object to a specified type



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The slide shows a summary of the logic and conditional operators in Java.

if else Statement

```
1 public class IfElse {  
2  
3     public static void main(String args[]) {  
4         long a = 1;  
5         long b = 2;  
6  
7         if (a == b) {  
8             System.out.println("True");  
9         } else {  
10            System.out.println("False");  
11        }  
12    }  
13}  
14 }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The example in the slide demonstrates the syntax for an `if-else` statement in Java.

The `else` statement provides a secondary path of execution when an `if` clause evaluates to `false`.

The output from the code in the slide is `False`.

switch Statement

```
1 public class SwitchStringStatement {  
2     public static void main(String args[]) {  
3  
4         String color = "Blue";  
5         String shirt = " Shirt";  
6  
7         switch (color){  
8             case "Blue":  
9                 shirt = "Blue" + shirt;  
10                break;  
11            case "Red":  
12                shirt = "Red" + shirt;  
13                break;  
14            default:  
15                shirt = "White" + shirt;  
16        }  
17  
18        System.out.println("Shirt type: " + shirt);  
19    }  
20 }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

A `switch` statement is used to compare the value of a variable with multiple values. For each of these values, you can define a set of statements to execute. The `switch` statement evaluates its expression, then executes all statements that follow the matching `case` label. Each `break` statement terminates the enclosing `switch` statement. The `break` statements are necessary because without them, statements in `switch` blocks *fall through*: All statements after the matching `case` label are executed in sequence, regardless of the expression of subsequent `case` labels, until a `break` statement is encountered.

Note: Deciding whether to use `if-then-else` statements or a `switch` statement is based on readability and the expression that the statement is testing. An `if-then-else` statement can test expressions based on ranges of values or conditions, whereas a `switch` statement tests expressions based only on a single integer, character, enumerated value, or `String` object.

In Java SE 7 and later, you can use a `String` object in the `switch` statement's expression. The code snippet in the slide demonstrates using `Strings` in the `switch` statement.

while Loop

```
package com.example.review;

public class WhileTest {

    public static void main(String args[]) {
        int x = 10;
        while (x < 20) {
            System.out.print("value of x : " + x);
            x++;
            System.out.print("\n");
        }
    }
}
```

expression returning
boolean value



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The `while` loop performs a test and continues if the expression evaluates to `true`.
The `while` statement evaluates *expression*, which must return a boolean value. If the expression evaluates to `true`, the `while` statement executes the *statements* in the `while` block.
The `while` loop, shown in the slide, iterates through an array by using a counter. The output from the example in the slide:

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

There is also a `do-while` loop, where the test after the expression has run at least once. The difference between `do-while` and `while` is that `do-while` evaluates its expression at the bottom of the loop instead of the top. The output from the example in the slide:

value of x : 30

for Loop

```
1 public class ForLoop {  
2  
3     public static void main(String args[]){  
4  
5         for (int i = 0; i < 9; i++ ) {  
6             System.out.println("i: " + i);  
7         }  
8     }  
9 }  
10 }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The `for` statement provides a compact way to iterate over a range of values - it repeatedly loops until a particular condition is satisfied.

A counter is initialized and incremented for each step of the `for` loop shown in the slide. When the condition statement evaluates to false (when `i` is no longer less than 9), the loop exits. Here is the sample output for this program.

```
i: 0  
i: 1  
i: 2  
i: 3  
i: 4  
i: 5  
i: 6  
i: 7  
i: 8
```

Arrays and for-each Loop

```
1 public class ArrayOperations {
2     public static void main(String args[]){
3
4         String[] names = new String[3];
5
6         names[0] = "Blue Shirt";
7         names[1] = "Red Shirt";
8         names[2] = "Black Shirt";
9
10        int[] numbers = {100, 200, 300};
11
12        for (String name:names){
13            System.out.println("Name: " + name);
14        }
15
16        for (int number:numbers){
17            System.out.println("Number: " + number);
18        }
19    }
20 }
```

Arrays are objects.
Array objects have a
final field length.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

This class demonstrates how to define arrays in Java and iterate over arrays by using `for-each` loop. The first array, `names`, creates a `String` array and initializes each element separately. The second array, `numbers` is an array of integers. Each array is iterated through using the Java `for-each` construct. The enhanced `for` loop is used to make the `for` loop more compact and easy to read. It is used to access each successive value in a collection of values. It is commonly used to iterate over an array or a Collections class (for example, an `Array` or an `ArrayList`). The output of the class is shown here:

Name: Blue Shirt

Name: Red Shirt

Name: Black Shirt

Number: 100

Number: 200

Number: 300

Note: Arrays are also objects by default. All arrays support the methods of the class `Object`. You can always obtain the size of an array by using its `length` field.

Strings

```
1 public class Strings {  
2  
3     public static void main(String args[]) {  
4  
5         char letter = 'a';  
6  
7         String string1 = "Hello";  
8         String string2 = "World";  
9         String string3 = "";  
10        String dontDoThis = new String ("Bad Practice");  
11  
12        string3 = string1 + string2; // Concatenate strings  
13  
14        System.out.println("Output: " + string3 + " " + letter);  
15  
16    }  
17 }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The code in the slide demonstrates how text characters are represented in Java. Single characters can be represented with the `char` type. However, Java also includes a `String` type for representing multiple characters. Strings can be defined as shown in the slide and combined by using the "+" sign as a concatenation operator.

The output from the code in the slide is:

Output: HelloWorld a

Caution: The String class is immutable: After being created, the contents of an object of the class `String` can never be modified. The immutability of `String` objects helps the JVM to reuse `String` objects, reducing memory overhead and improving performance.

Strings should always be initialized by using the assignment operator "=" and text in quotation marks, as shown in the examples. The use of `new` to initialize a `String` is strongly discouraged. The reason is that "Bad Practice" in line 10 is a `String` literal of type `String`. Using the `new` keyword simply creates another instance functionally identical to the literal. If this statement appeared inside of a loop that was frequently invoked, a lot of needless `String` instances could be created.

String Operations: StringBuilder

```
public class StringOperations {  
  
    public static void main(String arg[]) {  
  
        StringBuilder sb = new StringBuilder("hello");  
        System.out.println("string sb: " + sb);  
        sb.append(" world");  
        System.out.println("string sb: " + sb);  
  
        sb.append("!").append(" are").append(" you?");  
        System.out.println("string sb: " + sb);  
  
        sb.insert(12, " How");  
        System.out.println("string sb: " + sb);  
  
        // Get length  
        System.out.println("Length: " + sb.length());  
  
        // Get SubString  
        System.out.println("Sub: " + sb.substring(0, 5));  
    }  
}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The slide demonstrates usage of the `StringBuilder` class and commonly used methods.

StringBuilder class

The `StringBuilder` objects are like `String` objects except that they may be modified (mutable). The `StringBuilder` class is used when dealing with larger strings or modifying the contents of a string often.

The example in this slide demonstrates modifying the `String` by using the `append` and `insert` methods and chaining by using the `append` method.

Output:

```
string sb: hello  
string sb: hello world  
string sb: hello world! are you?  
string sb: hello world! How are you?  
Length: 25  
Sub: hello
```

A Simple Java Class: Employee

A Java class is often used to represent a concept.

```
1 package com.example.domain;
2 public class Employee { class declaration
3     public int empId;
4     public String name;
5     public String ssn;
6     public double salary; } fields
7
8     public Employee () { a constructor
9 }
10
11    public int getEmpId () { a method
12        return empId;
13    }
14 }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

A Java class is often used to store or represent data for the construct that the class represents. For example, you could create a model (a programmatic representation) of an Employee. An Employee object defined by using this model contains values for empId, name, Social Security Number (ssn), and salary.

A constructor is used to create an instance of a class. Unlike methods, constructors do not declare a return type, and are declared with the same name as their class. Constructors can take arguments and you can declare more than one constructor.

Methods

When a class has data fields, a common practice is to provide methods for storing data (setter methods) and retrieving data (getter methods) from the fields.

```
1 package com.example.domain;
2 public class Employee {
3     public int empId;
4     // other fields...
5     public void setEmpId(int empId) {
6         this.empId = empId;
7     }
8     public int getEmpId() {
9         return empId;
10    }
11    // getter/setter methods for other fields...
12 }
```

Often a pair of methods
to set and get the
current field value.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Adding Instance Methods to the Employee Class

A common practice is to create a set of methods that manipulate field data: methods that set the value of each field, and methods that get the value of each field. These methods are called *accessors* (getters) and *mutators* (setters).

The convention is to use `set` and `get` plus the name of the field with the first letter of the field name capitalized (lower camel case). Most modern integrated development environments (IDEs) provide an easy way to automatically generate the accessor (getter) and mutator (setter) methods for you.

Notice that the set methods use the keyword `this`. The `this` keyword allows the compiler to distinguish between the field name of the class (`this`) and the parameter name being passed in as an argument. Without the keyword `this`, the net effect is that you are assigning a value to itself. (In fact, NetBeans provides a warning: "Assignment to self.")

In this simple example, you could use the `setName` method to change the employee name and the `setSalary` method to change the employee salary.

Note: The methods declared on this slide are called *instance* methods. They are invoked by using an instance of this class (described in the next slide).

Creating an Instance of a Class

To construct or create an instance (object) of the Employee class, use the new keyword.

```
/* In some other class, or a main method */
Employee emp = new Employee();
emp.empId = 101;    // legal if the field is public,
                   // but not good OO practice
emp.setEmpId(101); // use a method instead
emp.setName("John Smith");
emp.setSsn("011-22-3467");
emp.setSalary(120345.27);
```

Invoking an
instance method.

- In this fragment of Java code, you construct an instance of the Employee class and assign the reference to the new object to a variable called emp.
- Then you assign values to the Employee object.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Creating an instance of the Employee Class

- You need to allocate memory for the Employee object and call a constructor in the class to initialize all the instance variables of the class.
- To allocate memory, new operator is used and to initialize all the instance variables of the class, the constructor is invoked.
- An instance of an object is created when you use the new keyword with a constructor. All the fields declared in the class are provided with memory space and initialized to their default values.
- If the memory allocation and constructor are successful, a reference to the object is returned as a result. In the example in the slide, the reference is assigned to a variable called emp.
- After all the data fields are set with values, you have an instance of an Employee with an empId with a value of 101, name with the string John Smith, Social Security Number string (ssn) set to 011-22-3467, and salary with the value of 120,345.27.

Constructors

```
public class Employee {  
    public Employee() {  
    }  
}
```

A simple no-argument (no-arg) constructor

```
Employee emp = new Employee();
```

- A constructor is used to create an instance of a class.
- Constructors can take parameters.
- A constructor is declared with the same name as its class.



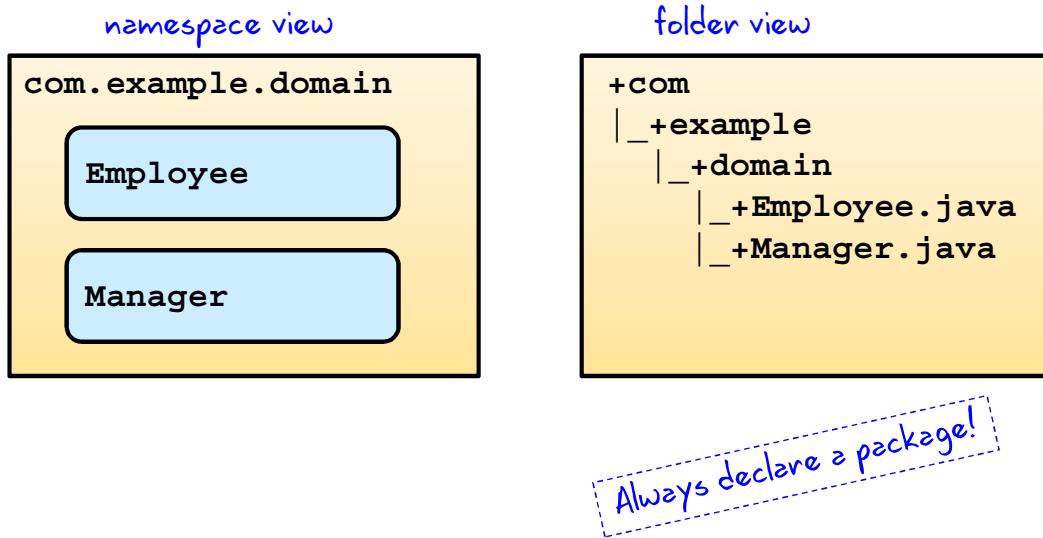
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Default and No-Arg Constructors

- Every class must have at least one constructor.
- If you do not provide any in a class's declaration, the compiler creates a **default constructor** that takes no arguments when it's invoked.
- The default constructor initializes the instance variables to the initial values specified in their declarations or to their default values (zero for primitive numeric types, false for boolean values, and null for references).
- If your class declares constructors, the compiler will not create a default constructor.
- In this case, you must declare a **no-arg** constructor if default initialization is required.
- Like a default constructor, a no-arg constructor is invoked with empty parentheses.

package Statement

- The package keyword is used in Java to group classes together.
- A package is implemented as a folder and, like a folder, provides a *namespace* to a class.



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Packages

In Java, a package is a group of (class) types. There can be only one package declaration for a file.

Packages create a namespace, a logical collection of things, like a directory hierarchy. Packages prevent name collision and also provide access control to classes. It is a good practice to always use a package declaration.

import Statements

The `import` keyword is used to identify classes you want to reference in your class.

- The `import` statement provides a convenient way to identify classes that you want to reference in your class.

```
import java.util.Date;
```

- You can import a single class or an entire package:

```
import java.util.*;
```

- You can include multiple `import` statements:

```
import java.util.Date;  
import java.util.Calendar;
```

- It is good practice to use the full package and class name rather than the wildcard `*` to avoid class name conflicts.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Imports

You could refer to a class using its fully qualified namespace in your applications, as in the following example:

```
java.util.Date date = new java.util.Date();
```

But that would require a lot of typing. Instead, Java provides the `import` statement to allow you to declare that you want to reference a class in another package.

Notes:

It is a good practice to use the specific, fully qualified package and class name to avoid confusion when there are two classes with the same name, as in the following example:

`java.sql.Date` and `java.util.Date`. The first is a `Date` class used to store a `Date` type in a database, and `java.util.Date` is a general purpose `Date` class. As it turns out, `java.sql.Date` is a subclass of `java.util.Date`.

Modern IDEs, like NetBeans and Eclipse, automatically search for and add import statements for you. In NetBeans, for example, use the `Ctrl + Shift + I` key sequence to fix imports in your code.

import Statements

- import statements follow the package declaration and precede the class declaration.
- An import statement is not required.
- By default, your class always imports `java.lang.*`
- You do not need to import classes that are in the same package:

```
package com.example.domain;  
import com.example.domain.Manager; // unused import
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

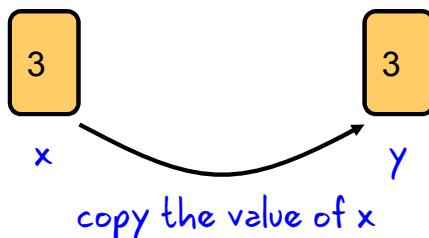
Java Is Pass-By-Value

The Java language (unlike C++) uses pass-by-value for all assignment operations.

- To visualize this with primitives, consider the following:

```
int x = 3;  
int y = x;
```

- The value of `x` is copied and passed to `y`:



- If `x` is later modified (for example, `x = 5;`), the value of `y` remains unchanged.

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The Java language uses pass-by-value for all assignment operations. This means that the argument on the right side of the equal sign is evaluated, and the value of the argument is assigned to the left side of the equal sign.

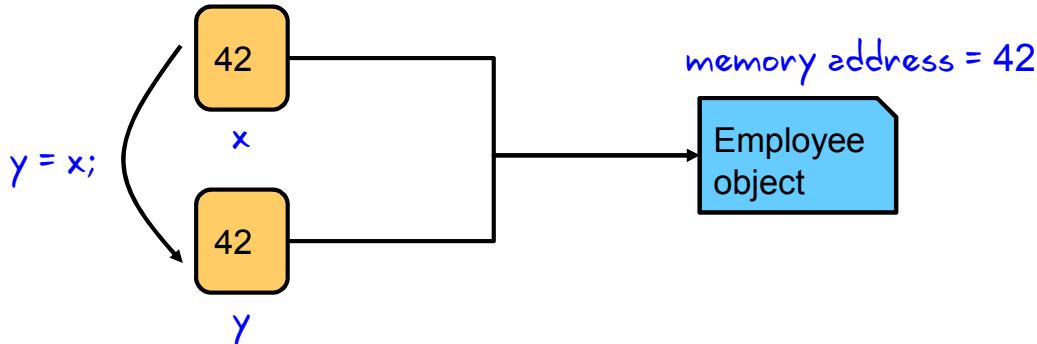
For Java primitives, this is straightforward. Java does not pass a reference to a primitive (such as an integer), but rather a copy of the value.

Pass-By-Value for Object References

For Java objects, the *value* of the right side of an assignment is a reference to memory that stores a Java object.

```
Employee x = new Employee();  
Employee y = x;
```

- The reference is some address in memory.



- After the assignment, the value of `y` is the same as the value of `x`: a reference to the same `Employee` object.

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

For Java objects, the value of an object reference is the memory pointer to the instance of the `Employee` object created.

When you assign the value of `x` to `y`, you are not creating a new `Employee` object, but rather a copy of the value of the reference.

Note: An object is a class instance or an array. The reference values (references) are pointers to these objects, and a special `null` reference, which refers to no object.

Objects Passed as Parameters

```
4 public class ObjectPassTest {  
5     public static void main(String[] args) {  
6         ObjectPassTest test = new ObjectPassTest();  
7         Employee x = new Employee();  
8         x.setSalary(120_000.00);  
9         test.foo(x);  
10        System.out.println ("Employee salary: "  
11                + x.getSalary());  
12    }  
13  
14    public void foo(Employee e) {  
15        e.setSalary(130_000.00);  
16        e = new Employee();  
17        e.setSalary(140_000.00);  
18    }  
}
```

salary set to 120_000

What will x.getSalary() return at the end of the main() method?



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

On line 6, a new `Employee` object is created and the reference to that object is assigned to the variable `x`. The `salary` field is set to `120_000` on line 8.

On line 9, the reference to `x` is passed to the `foo` method.

Line 15 of the `foo` method sets the `salary` value of `e` to `130_000`. Because `e` refers to the same reference as `x`, the `salary` value for `x` is now `130_000`.

Line 16 creates a new `Employee` object that changes the reference held in `e`. Now `x` is no longer referred to in the `foo` method.

Line 17 set the `salary` field, but `x` is unaffected as `e` no longer holds a reference to `x`.

The output from the program is:

Employee salary: 130000.0

Garbage Collection

When an object is instantiated by using the `new` keyword, memory is allocated for the object. The scope of an object reference depends on where the object is instantiated:

```
public void someMethod() {  
    Employee e = new Employee();  
    // operations on e  
}
```

Object e scope ends here.

- When `someMethod` completes, the memory referenced by `e` is no longer accessible.
- Java's garbage collector recognizes when an instance is no longer accessible and eligible for collection.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Garbage is automatically collected by the JVM. The timing of the collection depends upon many factors. A detailed discussion of garbage collection mechanics is beyond the scope of this lesson.

Summary

In this lesson, you should have learned how to:

- Create simple Java classes
 - Create primitive variables
 - Use Operators
 - Manipulate Strings
 - Use if-else and switch branching statements
 - Iterate with loops
 - Create arrays
- Use Java fields, constructors, and methods
- Use package and import statements



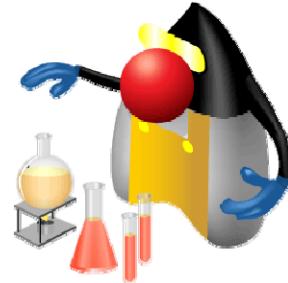
ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 2-1 Overview: Creating Java Classes

This practice covers the following topics:

- Creating a Java class using the NetBeans IDE
- Creating a Java class with a `main` method
- Writing code in the body of the `main` method to create an instance of the `Employee` object and print values from the class to the console
- Compiling and testing the application by using the NetBeans IDE



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Quiz

Which is the printed result in the following fragment?

```
public float average (int[] values) {  
    float result = 0;  
    for (int i = 1; i < values.length; i++)  
        result += values[i];  
    return (result/values.length);  
}  
// ... in another method in the same class  
int[] nums = {100, 200, 300};  
System.out.println (average(nums));
```

- a. 100.00
- b. 150.00
- c. 166.66667
- d. 200.00



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Answer: c

Arrays begin with an index of 0. This average method is only averaging the second through n^{th} values. Therefore, the result is $200+300/3 = 166.66667$. Change the for loop to $\text{int } = 0$; to calculate the average.

Quiz

In the following fragment, which two statements are false?

```
package com.oracle.test;  
public class BrokenClass {  
    public boolean valid = "false";  
    public String s = "A new string";  
    public int i = 40_000.00;  
    public BrokenClass() { }  
}
```

- a. An import statement is missing.
- b. The boolean valid is assigned a String.
- c. String s is created.
- d. BrokenClass method is missing a return statement.
- e. You need to create a new BrokenClass object.
- f. The integer value i is assigned a double.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Answer: b and f will cause compilation errors.

- a. An import statement is not required, unless the class uses classes outside of java.lang.
- d. BrokenClass() is a constructor.
- e. Construction of a BrokenClass instance would typically happen in another class.

Quiz

What is displayed when the following code snippet is compiled and executed?

```
String s1 = new String("Test");
String s2 = new String("Test");
if (s1==s2)
    System.out.println("Same");
if (s1.equals(s2))
    System.out.println("Equals");
```

- a. Same
- b. Equals
- c. Same Equals
- d. Compiler Error



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Answer: b

Because Strings S1 and S2 are created by using a `new` keyword, their addresses are not the same. However, the values are the same and so “Equals” is printed.

Encapsulation and Subclassing

3

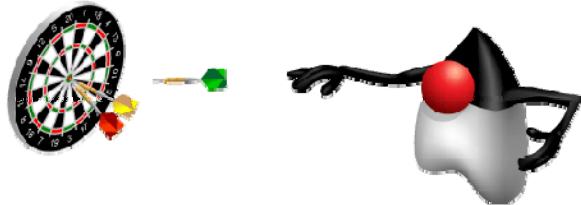
ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Objectives

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

- Use encapsulation in Java class design
- Model business problems by using Java classes
- Make classes immutable
- Create and use Java subclasses
- Overload methods



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Encapsulation

- Encapsulation is one of the four fundamental object-oriented programming concepts. The other three are inheritance, polymorphism, and abstraction.
- The term *encapsulation* means to enclose in a capsule, or to wrap something around an object to cover it.
- Encapsulation covers, or wraps, the internal workings of a Java object.
 - Data variables, or fields, are hidden from the user of the object.
 - Methods, the functions in Java, provide an explicit service to the user of the object but hide the implementation.
 - As long as the services do not change, the implementation can be modified without impacting the user.

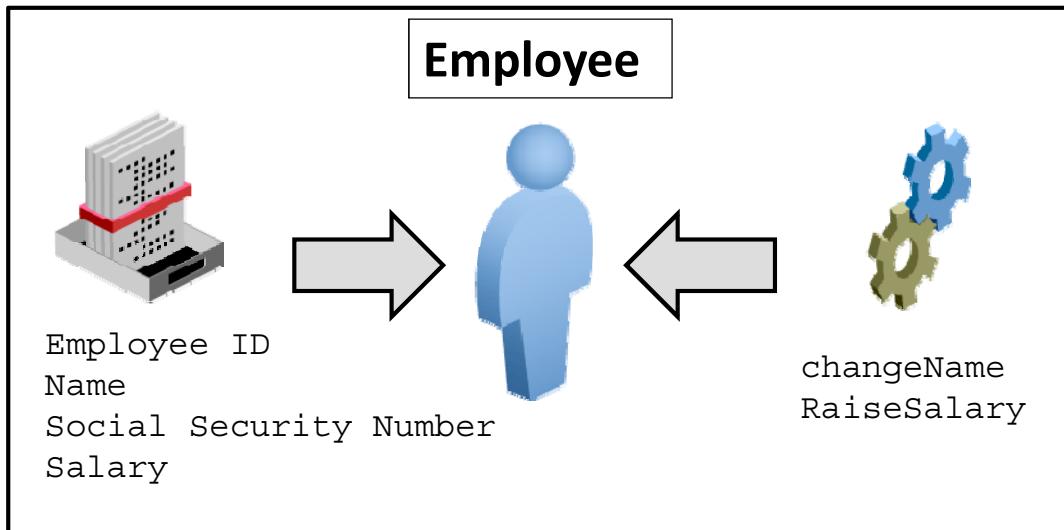
ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

An analogy for encapsulation is the steering wheel of a car. When you drive a car, whether it is your car, a friend's car, or a rental car, you probably never worry about how the steering wheel implements a right-turn or left-turn function. The steering wheel could be connected to the front wheels in a number of ways: ball and socket, rack and pinion, or some exotic set of servo mechanisms. As long as the car steers properly when you turn the wheel, the steering wheel encapsulates the functions you need—you do not have to think about the implementation.

Encapsulation: Example

What data and operations would you encapsulate in an object that represents an employee?



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

A Simple Model

Suppose that you are asked to create a model of a typical employee. What data might you want to represent in an object that describes an employee?

- **Employee ID:** You can use this as a unique identifier for the employee.
- **Name:** Humanizing an employee is always a good idea.
- **Social Security Number:** For United States employees only. You may want some other identification for non-U.S. employees.
- **Salary:** How much the employee makes is always good to record.

What operations might you allow on the employee object?

- **Change Name:** If the employee gets married or divorced, there could be a name change.
- **Raise Salary:** It increases based on merit.

After an employee object is created, you probably do not want to allow changes to the Employee ID or Social Security fields. Therefore, you need a way to create an employee without alterations except through the allowed methods.

Encapsulation: Public and Private Access Modifiers

- The `public` keyword, applied to fields and methods, allows any class in any package to access the field or method.
- The `private` keyword, applied to fields and methods, allows access only to other methods within the class itself.

```
Employee emp=new Employee();  
emp.salary=2000; // Compiler error- salary is a private field  
emp.raiseSalary(2000); //ok
```

- The `private` keyword can also be applied to a method to hide an implementation detail.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Java has three visibility modifiers: `public`, `private`, and `protected`.

Encapsulation: Private Data, Public Methods

One way to hide implementation details is to declare all of the fields **private**.

- The `Employee` class currently uses `public` access for all of its fields.
- To encapsulate the data, make the fields `private`.

```
public class Employee {  
  
    private int empId;  
    private String name;  
    private String ssn;  
    private double salary;  
  
    //... constructor and methods  
}
```

Declaring fields `private` prevents direct access to this data from a class instance.
// illegal!
`emp.salary = 1_000_000_000.00;`



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In Java, we accomplish encapsulation through the use of visibility modifiers. Declaring Java fields `private` makes it invisible outside of the methods in the class itself.

In this example, the fields `custID`, `name`, and `amount` are now marked `private`, making them invisible outside of the methods in the class itself.

Employee Class Refined

```
1 public class Employee {  
2     // private fields ...  
3     public Employee () {  
4     }  
5     // Remove all of the other setters  
6     public void changeName(String newName) {  
7         if (newName != null) {  
8             this.name = newName;  
9         }  
10    }  
11  
12    public void raiseSalary(double increase) {  
13        this.salary += increase;  
14    }  
15 }
```

Encapsulation step 2:
These method names
make sense in the
context of an
Employee.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The current setter methods in the class allow any class that uses an instance of `Employee` to alter the object's ID, salary, and SSN fields. From a business standpoint, these are not operations you would want on an employee. Once the employee is created, these fields should be immutable (no changes allowed).

The `Employee` model as defined in the slide titled “Encapsulation: Example” had only two operations: one for changing an employee name (as a result of a marriage or divorce) and one for increasing an employee's salary.

To refine the `Employee` class, the first step is to remove the setter methods and create methods that clearly identify their purpose. Here there are two methods, one to change an employee name (`setName`) and the other to increase an employee salary (`raiseSalary`).

Note that the implementation of the `setName` method tests the string parameter passed in to make sure that the string is not a null. The method can do further checking as necessary.

Make Classes as Immutable as Possible

```
1  public class Employee {  
2      // private fields ...  
3      // Create an employee object  
4      public Employee (int empId, String  
5                          name, String ssn,  
6                          double salary) {  
7          this.empId = empId;  
8          this.name = name;  
9          this.ssn = ssn;  
10         this.salary = salary;  
11     }  
12  
13     public void changeName(String newName) { ... }  
14  
15     public void raiseSalary(double increase) { ... }  
16 }
```

Encapsulation step 3:
Remove the no-arg
constructor; implement
a constructor to set
the value of all fields.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Good Practice: Immutability

Finally, because the class no longer has setter methods, you need a way to set the initial value of the fields. The answer is to pass each field value in the construction of the object. By creating a constructor that takes all of the fields as arguments, you can guarantee that an `Employee` instance is fully populated with data *before* it is a valid employee object. This constructor *replaces* the default constructor.

Granted, the user of your class could pass null values, and you need to determine if you want to check for those in your constructor. Strategies for handling those types of situations are discussed in later lessons.

Removing the setter methods and replacing the no-arg constructor also guarantees that an instance of `Employee` has immutable Employee ID and Social Security Number (SSN) fields.

Method Naming: Best Practices

Although the fields are now hidden by using private access, there are some issues with the current Employee class.

- The setter methods (currently public access) allow any other class to change the ID, SSN, and salary (up or down).
- The current class does not really represent the operations defined in the original Employee class design.
- Two best practices for methods:
 - Hide as many of the implementation details as possible.
 - Name the method in a way that clearly identifies its use or functionality.
- The original model for the Employee class had a Change Name and an Increase Salary operation.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Choosing Well-Intentioned Methods

Just as fields should clearly define the type of data that they store, methods should clearly identify the operations that they perform. One of the easiest ways to improve the readability of your code (Java code or any other) is to write method names that clearly identify what they do.

Encapsulation: Benefits

The benefits of using encapsulation are as follows:

- Protects an object from unwanted access by clients
- Prevents assigning undesired values for its variables by the clients, which can make the state of an object unstable
- Allows changing the class implementation without modifying the client interface



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Creating Subclasses

You created a Java class to model the data and operations of an Employee. Now suppose you wanted to specialize the data and operations to describe a Manager.

```
1 package com.example.domain;
2 public class Manager {
3     private int empId;
4     private String name;    wait a minute...
5     private String ssn;    this code looks very familiar....
6     private double salary;
7     private String deptName;
8     public Manager () {   }
9     // access and mutator methods...
10 }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Specialization by Using Java Subclassing

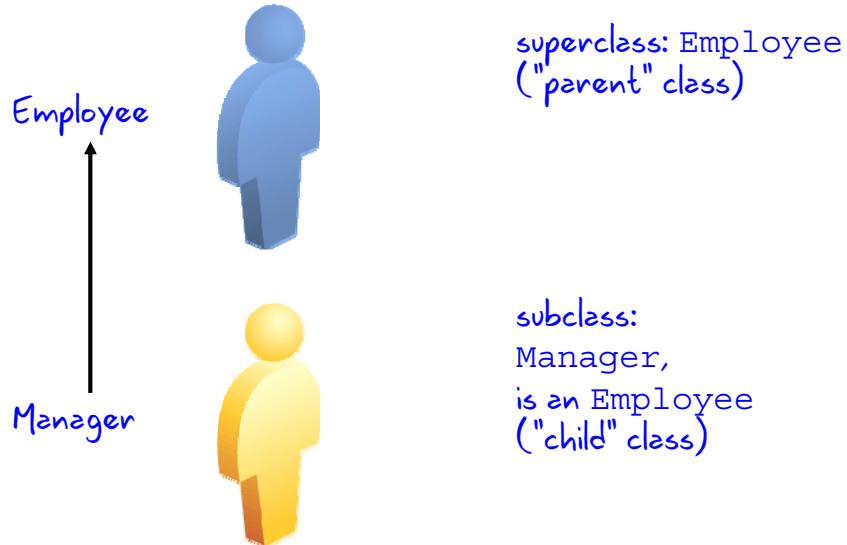
The Manager class shown here closely resembles the Employee class, but with some specialization. A Manager also has a department, with a department name. As a result, there are likely to be additional operations as well.

What this demonstrates is that a Manager is an Employee—but an Employee with additional features.

However, if we were to define Java classes this way, there would be a lot of redundant coding.

Subclassing

In an object-oriented language like Java, subclassing is used to define a new class in terms of an existing one.



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

A Simple Java Program

When an existing class is subclassed, the new class created is said to inherit the characteristics of the other class. This new class is called the *subclass* and is a specialization of the superclass. All the nonprivate fields and methods from the superclass are part of the subclass.

In this diagram, a Manager class gets all the nonprivate fields and all the public methods from Employee.

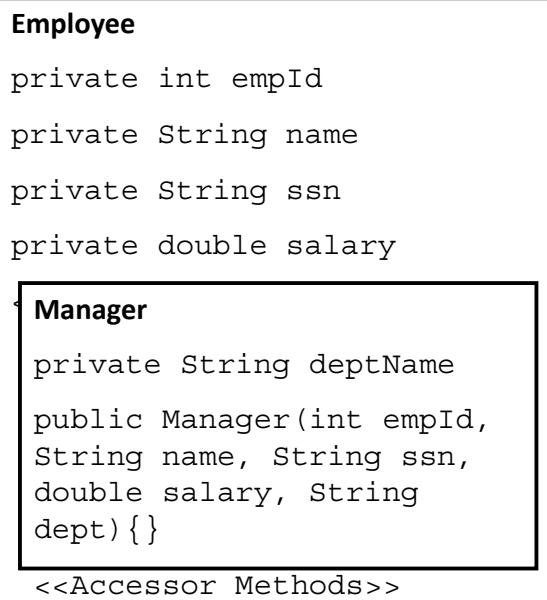
It is important to grasp that although Manager specializes Employee, a Manager is still an Employee.

Note: The term *subclass* is a bit of a misnomer. Most people think of the prefix “*sub*” as meaning “less.” However, a Java subclass is the sum of itself and its parent. When you create an instance of a subclass, the resulting in-memory structure contains all codes from the parent class, grandparent class, and so on all the way up the class hierarchy until you reach the class Object.

Manager Subclass

```
public class Manager extends Employee { }
```

The keyword **extends** creates the inheritance relationship:



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The code snippet in the slide demonstrates the Java syntax for subclassing.

The diagram in the slide demonstrates an inheritance relationship between the **Manager** class and, its parent, the **Employee** class.

- The **Manager** class, by extending the **Employee** class, inherits all of the non-private data fields and methods from **Employee**.
- Since a manager is also an employee, then it follows that **Manager** has all of the same attributes and operations of **Employee**.

Note: The **Manager** class declares its own constructor. Constructors are *not* inherited from the parent class. There are additional details about this in the next slide.

Constructors in Subclasses

Although a subclass inherits all of the methods and fields from a parent class, it does not inherit constructors. There are two ways to gain a constructor:

- Write your own constructor.
- Use the default constructor.
 - If you do not declare a constructor, a default no-arg constructor is provided for you.
 - If you declare your own constructor, the default constructor is no longer provided.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Constructors in Subclasses

Every subclass inherits the nonprivate fields and methods from its parent (superclass). However, the subclass does not inherit the constructor from its parent. It must provide a constructor.

The *Java Language Specification* includes the following description:

“Constructor declarations are not members. They are never inherited and therefore are not subject to hiding or overriding.”

Using `super`

To construct an instance of a subclass, it is often easiest to call the constructor of the parent class.

- In its constructor, Manager calls the constructor of Employee.
- The `super` keyword is used to call a parent's constructor.
- It must be the first statement of the constructor.
- If it is not provided, a default call to `super()` is inserted for you.
- The `super` keyword may also be used to invoke a parent's method or to access a parent's (nonprivate) field.

```
super (empId, name, ssn, salary);
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Constructing a Manager Object

Creating a Manager object is the same as creating an Employee object:

```
Manager mgr = new Manager (102, "Barbara Jones",
                           "107-99-9078", 109345.67, "Marketing");
```

- All of the Employee methods are available to Manager:

```
mgr.raiseSalary (10000.00);
```

- The Manager class defines a new method to get the Department Name:

```
String dept = mgr.getDeptName();
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Even though the Manager.java file does not contain all of the methods from the Employee.java class (explicitly), they are included in the definition of the object. Thus, after you create an instance of a Manager object, you can use the methods declared in Employee.

You can also call methods that are specific to the Manager class as well.

Overloading Methods

Your design may call for several methods in the same class with the same name but with different arguments.

```
public void print (int i)  
public void print (float f)  
public void print (String s)
```

- Java permits you to reuse a method name for more than one method.
- Two rules apply to overloaded methods:
 - Argument lists *must* differ.
 - Return types *can* be different.
- Therefore, the following is not legal:

```
public void print (int i)  
public String print (int i)
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You might want to design methods with the same intent (method name), like `print`, to print out several different types. You could design a method for each type:

```
printInt(int i)  
printFloat(float f)  
printString(String s)
```

But this would be tedious and not very object-oriented. Instead, you can create a reusable method name and just change the argument list. This process is called *overloading*.

With overloading methods, the argument lists must be different—in order, number, or type. And the return types can be different. However, two methods with the same argument list that differ only in return type are not allowed.

Overloaded Constructors

- In addition to overloading methods, you can overload constructors.
- The overloaded constructor is called based upon the parameters specified when the `new` is executed.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Overloaded Constructors: Example

```
public class Box {  
  
    private double length, width, height;  
  
    public Box() {  
        this.length = 1;  
        this.height = 1;  
        this.width = 1;  
    }  
  
    public Box(double length) {  
        this.width = this.length = this.height = length;  
    }  
  
    public Box(double length, double width, double height) {  
        this.length = length;  
        this.height = height;  
        this.width = width;  
        System.out.println("and the height of " + height + ".");  
    }  
  
    double volume() {  
        return width * height * length;  
    }  
}
```

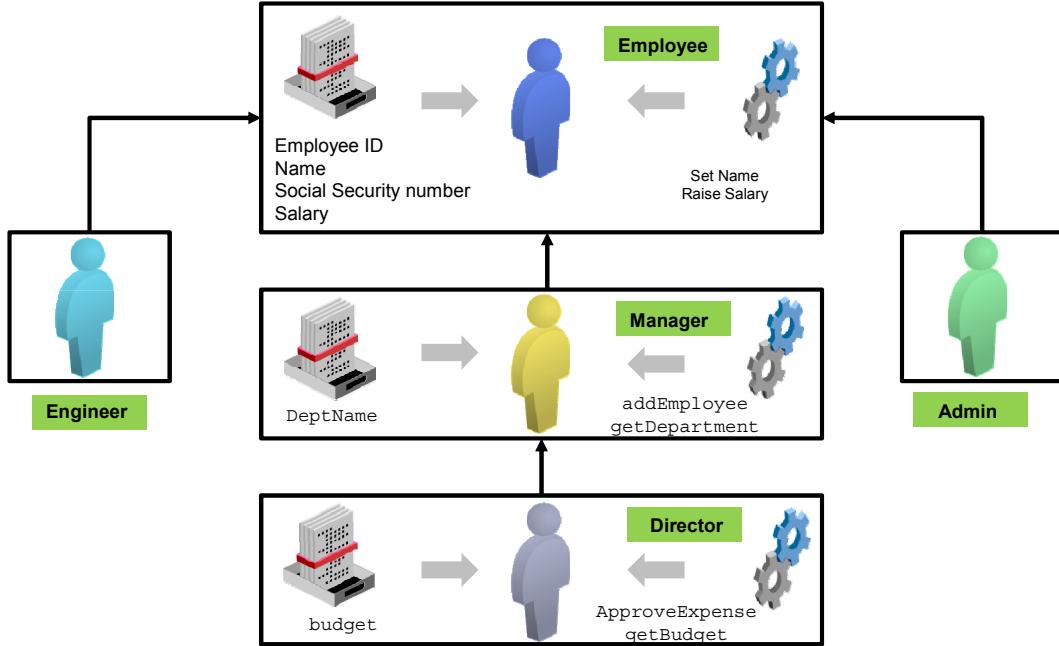


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The example in the slide demonstrates overloaded constructors: there are three overloaded constructors, which vary based on the no of arguments.

Single Inheritance

The Java programming language permits a class to extend only one other class. This is called *single inheritance*.



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Summary

In this lesson, you should have learned how to:

- Use encapsulation in Java class design
- Model business problems by using Java classes
- Make classes immutable
- Create and use Java subclasses
- Overload methods

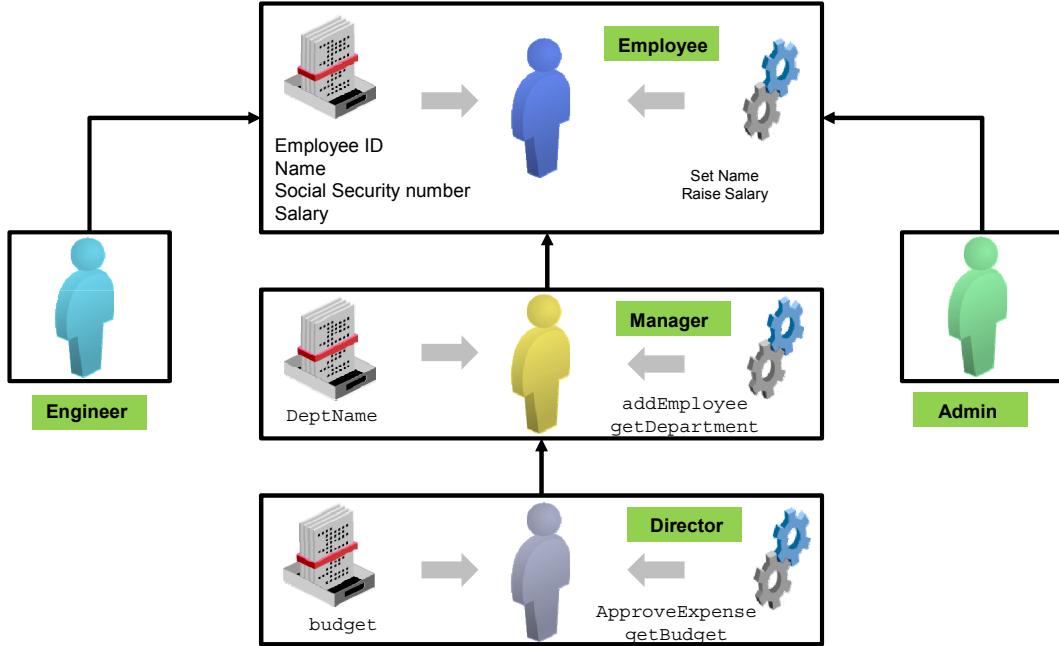


ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Single Inheritance

The Java programming language permits a class to extend only one other class. This is called *single inheritance*.



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Quiz

Which of the following declarations demonstrates the application of good Java naming conventions?

- a. public class repeat { }
- b. public void Screencoord (int x, int y) { }
- c. private int XCOORD;
- d. public int calcOffset (int xCoord, int yCoord) { }



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Quiz

What changes would you perform to make this class immutable? (Choose all that apply.)

```
public class Stock {  
    public String symbol;  
    public double price;  
    public int shares;  
    public double getStockValue() { }  
    public void setSymbol(String symbol) { }  
    public void setPrice(double price) { }  
    public void setShares(int number) { }  
}
```

- a. Make the **fields** symbol, shares, and price **private**.
- b. Remove setSymbol, setPrice, and setShares.
- c. Make the getStockValue **method** private.
- d. Add a constructor that takes symbol, shares, and price as arguments.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Overriding Methods, Polymorphism, and Static Classes



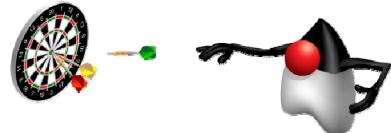
ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Objectives

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

- Use access levels: `private`, `protected`, `default`, and `public`
- Override methods
- Use virtual method invocation
- Use `varargs` to specify variable arguments
- Use the `instanceof` operator to compare object types
- Use upward and downward casts
- Model business problems by using the `static` keyword
- Implement the singleton design pattern



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Using Access Control

- You have seen the keywords `public` and `private`.
- There are four access levels that can be applied to data fields and methods.
- Classes can be default (no modifier) or `public`.

Modifier (keyword)	Same Class	Same Package	Subclass in Another Package	Universe
<code>private</code>	Yes			
<code>default</code>	Yes	Yes		
<code>protected</code>	Yes	Yes	Yes	
<code>public</code>	Yes	Yes	Yes	Yes



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The table in the slide illustrates access to a field or method marked with the access modifier in the left column.

The access modifier keywords shown in this table are `private`, `protected`, and `public`.

When a keyword is absent, the `default` access modifier is applied.

private: Provides the greatest control over access to fields and methods. With `private`, a data field or method can be accessed only within the same Java class.

default: Also called package level access. With `default`, a data field or method can be accessed within the same class or package. A default class cannot be subclassed outside its package.

protected: Provides access within the package and subclass. Fields and methods that use `protected` are said to be “subclass-friendly.” Protected access is extended to subclasses that reside in a package different from the class that owns the protected feature. As a result, `protected` fields or methods are actually more accessible than those marked with `default` access control.

public: Provides the greatest access to fields and methods, making them accessible anywhere: in the class, package, subclasses, and any other class.

Protected Access Control: Example

```
1 package demo;
2 public class Foo {
3     protected int result = 20; ←
4     int num= 25;           subclass-friendly declaration
5 }
```

```
1 package test;
2 import demo.Foo;
3 public class Bar extends Foo {
4     private int sum = 10;
5     public void reportSum () {
6         sum += result;
7         sum +=num;           ← compiler error
8     }
9 }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In this example, there are two classes in two packages. Class `Foo` is in the package `demo`, and declares a data field called `result` with a `protected` access modifier.

In the class `Bar`, which extends `Foo`, there is a method, `reportSum`, that adds the value of `result` to `sum`. The method then attempts to add the value of `num` to `sum`. The field `num` is declared using the default modifier, and this generates a compiler error. Why?

Answer: The field `result`, declared as a `protected` field, is available to all subclasses—even those in a different package. The field `num` is declared as using default access and is only available to classes and subclasses declared in the same package.

This example is from the `JavaAccessExample` project.

Access Control: Good Practice

A good practice when working with fields is to make fields as inaccessible as possible, and provide clear intent for the use of fields through methods.

```
1 package demo;
2 public class Foo3 {
3     private int result = 20;
4     protected int getResult() {
5         return this.result;
6     }
7 }
```

```
1 package test;
2 import demo.Foo3;
3 public class Bar3 extends Foo3 {
4     private int sum = 10;
5     public void reportSum() {
6         sum += getResult();
7     }
8 }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

A slightly modified version of the example using the `protected` keyword is shown in the slide. If the idea is to limit access of the field `result` to classes within the package and the subclasses (package-protected), you should make the access explicit by defining a method purposefully written for package and subclass-level access.

Overriding Methods

Consider a requirement to provide a String that represents some details about the Employee class fields.

```
3 public class Employee {  
4     private int empId;  
5     private String name;  
14    // Lines omitted  
15  
16    public String getDetails() {  
17        return "ID: " + empId + " Name: " + name;  
18    }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Although the Employee class has getters to return values for a print statement, it might be nice to have a utility method to get specific details about the employee. Consider a method added to the Employee class to print details about the Employee object.

In addition to adding fields or methods to a subclass, you can also modify or change the existing behavior of a method of the parent (superclass).

You may want to specialize this method to describe a Manager object.

Overriding Methods

In the Manager class, by creating a method with the same signature as the method in the Employee class, you are *overriding* the getDetails method:

```
3 public class Manager extends Employee {  
4     private String deptName;  
17    // Lines omitted  
18  
19    @Override  
20    public String getDetails() {  
21        return super.getDetails () +  
22            " Dept: " + deptName;  
23    }
```

A subclass can invoke a parent method by using the `super` keyword.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

When a method is overridden, it replaces the method in the superclass (parent) class.

This method is called for any Manager instance.

A call of the form `super.getDetails()` invokes the `getDetails` method of the parent class.

Note: If, for example, a class declares two public methods with the same name, and a subclass overrides one of them, the subclass still inherits the other method.

Invoking an Overridden Method

- Using the previous examples of Employee and Manager:

```
5  public static void main(String[] args) {  
6      Employee e = new Employee(101, "Jim Smith",  
7          "011-12-2345", 100_000.00);  
8      Manager m = new Manager(102, "Joan Kern",  
9          "012-23-4567", 110_450.54, "Marketing");  
10  
11     System.out.println(e.getDetails());  
12     System.out.println(m.getDetails());  
13 }
```

- The correct `getDetails` method of each class is called:

```
ID: 101 Name: Jim Smith  
ID: 102 Name: Joan Kern Dept: Marketing
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

During run time, the Java Virtual Machine invokes the `getDetails` method of the appropriate class. If you comment out the `getDetails` method in the `Manager` class shown in the previous slide, what happens when `m.getDetails()` is invoked?

Answer: Recall that methods are inherited from the parent class. So, at run time, the `getDetails` method of the parent class (`Employee`) is executed.

Virtual Method Invocation

- What happens if you have the following?

```
5  public static void main(String[] args) {  
6      Employee e = new Manager(102, "Joan Kern",  
7          "012-23-4567", 110_450.54, "Marketing");  
8  
9      System.out.println(e.getDetails());  
10 }
```

- During execution, the object's runtime type is determined to be a Manager object:

```
ID: 102 Name: Joan Kern Dept: Marketing
```

- At run time, the method that is executed is referenced from a Manager object.
- This is an aspect of polymorphism called *virtual method invocation*.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Compiler Versus Runtime Behavior

The important thing to remember is the difference between the compiler (which checks that each method and field is accessible based on the strict definition of the class) and the behavior associated with an object determined at run time.

This distinction is an important and powerful aspect of polymorphism: The behavior of an object is determined by its runtime reference.

Because the object you created was a Manager object, at runtime, when the getDetails method was invoked, the runtime reference is to the getDetails method of a Manager class, even though the variable e is of the type Employee.

This behavior is referred to as *virtual method invocation*.

Note: If you are a C++ programmer, you get this behavior in C++ only if you mark the method by using the C++ keyword `virtual`.

Accessibility of Overriding Methods

The overriding method cannot be less accessible than the method in the parent class.

```
public class Employee {  
    //... other fields and methods  
    public String getDetails() { ... }  
}
```

```
3 public class BadManager extends Employee {  
4     private String deptName;  
5     // lines omitted  
20    @Override  
21    private String getDetails() { // Compile error  
22        return super.getDetails () +  
23            " Dept: " + deptName;  
24    }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

To override a method, the name and the order of arguments must be identical.

By changing the access of the Manager `getDetails` method to `private`, the **BadManager class will not compile.**

Applying Polymorphism

Suppose that you are asked to create a new class that calculates a bonus for employees based on their salary and their role (employee, manager, or engineer):

```
3 public class BadBonus {  
4     public double getBonusPercent(Employee e) {  
5         return 0.01;  
6     }  
7  
8     public double getBonusPercent(Manager m) {  
9         return 0.03;  
10    }  
11  
12    public double getBonusPercent(Engineer e) {  
13        return 0.01;  
14    }  
// Lines omitted
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Design Problem

What is the problem in the example in the slide? Each method performs the calculation based on the type of employee passed in, and returns the bonus amount.

Consider what happens if you add two or three more employee types. You would need to add three additional methods, and possibly replicate the code depending upon the business logic required to compute shares.

Clearly, this is not a good way to treat this problem. Although the code will work, this is not easy to read and is likely to create much duplicate code.

Applying Polymorphism

A good practice is to pass parameters and write methods that use the most generic possible form of your object.

```
public class GoodBonus {  
    public static double getBonusPercent(Employee e) {  
        // Code here  
    }  
}
```

```
// In the Employee class  
public double calcBonus() {  
    return this.getSalary() * GoodBonus.getBonusPercent(this);  
}
```

- One method will calculate the bonus for every type.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Use the Most Generic Form

A good practice is to design and write methods that take the most generic form of your object possible.

In this case, `Employee` is a good base class to start from. But how do you know what object type is passed in? You learn the answer in the next slide.

Using the instanceof Keyword

The Java language provides the `instanceof` keyword to determine an object's class type at run time.

```
3 public class GoodBonus {  
4     public static double getBonusPercent(Employee e) {  
5         if (e instanceof Manager) {  
6             return 0.03;  
7         }else if (e instanceof Director) {  
8             return 0.05;  
9         }else {  
10            return 0.01;  
11        }  
12    }  
13 }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In the `GoodBonus` class, the `getBonusPercent` method uses the `instanceof` operator to determine what type of `Employee` was passed to the method.

Overriding Object methods

The root class of every Java class is `java.lang.Object`.

- All classes will subclass `Object` by default.
- You do not have to declare that your class extends `Object`. The compiler does that for you.

```
public class Employee { //... }
```

is equivalent to

```
public class Employee extends Object { //... }
```

- The root class contains several nonfinal methods, but there are three that are important to consider overriding:
 - `toString`, `equals`, and `hashCode`



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Object `toString` Method

The `toString` method returns a `String` representation of the object.

```
Employee e = new Employee (101, "Jim Kern", ...)  
System.out.println (e);
```

- You can use `toString` to provide instance information:

```
public String toString () {  
    return "Employee id: " + empId + "\n"+  
           "Employee name:" + name;  
}
```

- This is a better approach to getting details about your class than creating your own `getDetails` method.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The `println` method is overloaded with a number of parameter types. When you invoke `System.out.println(e)`, the method that takes an `Object` parameter is matched and invoked. This method in turn invokes the `toString()` method on the object instance.

Note: Sometimes you may want to be able to print out the name of the class that is executing a method. The `getClass()` method is an `Object` method used to return the `Class` object instance, and the `getName()` method provides the fully qualified name of the runtime class. `getClass().getName(); // returns the name of this class instance`. These methods are in the `Object` class.

Object equals Method

The `Object.equals` method compares only object references.

- If there are two objects `x` and `y` in any class, `x` is equal to `y` if and only if `x` and `y` refer to the same object.
- Example:

```
Employee x = new Employee (1,"Sue","111-11-1111",10.0);
Employee y = x;
x.equals (y); // true
Employee z = new Employee (1,"Sue","111-11-1111",10.0);
x.equals (z); // false!
```

- Because what we really want is to test the contents of the `Employee` object, we need to override the `equals` method:

```
public boolean equals (Object o) { ... }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The `equals` method of `Object` determines (by default) only if the values of two object references point to the same object. Basically, the test in the `Object` class is simply as follows:

If `x == y`, return true.

For an object (like the `Employee` object) that contains values, this comparison is not sufficient, particularly if we want to make sure there is one and only one employee with a particular ID.

Overriding equals in Employee

An example of overriding the `equals` method in the `Employee` class compares every field for equality:

```
1 @Override
2 public boolean equals (Object o) {
3     boolean result = false;
4     if ((o != null) && (o instanceof Employee)) {
5         Employee e = (Employee)o;
6         if ((e.empId == this.empId) &&
7             (e.name.equals(this.name)) &&
8             (e.ssn.equals(this.ssn)) &&
9             (e.salary == this.salary)) {
10            result = true;
11        }
12    }
13 }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

This simple `equals` test first tests to make sure that the object passed in is not null, and then tests to make sure that it is an instance of an `Employee` class (all subclasses are also employees, so this works). Then the `Object` is cast to `Employee`, and each field in `Employee` is checked for equality.

Note: For string types, you should use the `equals` method to test the strings character by character for equality.

@Override annotation

This annotation is used to instruct compiler that method annotated with `@Override` is an overridden method from super class or interface. When this annotation is used the compiler check is to make sure you actually are overriding a method when you think you are. This way, if you make a common mistake of misspelling a method name or not correctly matching the parameters, you will be warned that your method does not actually override as you think it does. Secondly, it makes your code easier to understand when you are overriding methods.

Overriding Object hashCode

The general contract for `Object` states that if two objects are considered equal (using the `equals` method), then integer `hashCode` returned for the two objects should also be equal.

```
1 @Override //generated by NetBeans
2 public int hashCode() {
3     int hash = 7;
4     hash = 83 * hash + this.empId;
5     hash = 83 * hash + Objects.hashCode(this.name);
6     hash = 83 * hash + Objects.hashCode(this.ssn);
7     hash = 83 * hash + (int)
(Double.doubleToLongBits(this.salary) ^
(Double.doubleToLongBits(this.salary) >>> 32));
8     return hash;
9 }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Overriding hashCode

The Java documentation for the `Object` class states:

"... It is generally necessary to override the `hashCode` method whenever this method [`equals`] is overridden, so as to maintain the general contract for the `hashCode` method, which states that equal objects must have equal hash codes."

The `hashCode` method is used in conjunction with the `equals` method in hash-based collections, such as `HashMap`, `HashSet`, and `Hashtable`.

This method is easy to get wrong, so you need to be careful. The good news is that IDEs such as NetBeans can generate `hashCode` for you.

To create your own hash function, the following will help approximate a reasonable hash value for equal and unequal instances:

- 1) Start with a nonzero integer constant. Prime numbers result in fewer hashcode collisions.
- 2) For each field used in the `equals` method, compute an `int` hash code for the field. Notice that for the `Strings`, you can use the `hashCode` of the `String`.
- 3) Add the computed hash codes together.
- 4) Return the result.

Methods Using Variable Arguments

A variation of method overloading is when you need a method that takes any number of arguments of the same type:

```
public class Statistics {  
    public float average (int x1, int x2) {}  
    public float average (int x1, int x2, int x3) {}  
    public float average (int x1, int x2, int x3, int x4) {}  
}
```

- These three overloaded methods share the same functionality. It would be nice to collapse these methods into one method.

```
Statistics stats = new Statistics ();  
float avg1 = stats.average(100, 200);  
float avg2 = stats.average(100, 200, 300);  
float avg3 = stats.average(100, 200, 300, 400);
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Methods with a Variable Number of the Same Type

One case of overloading is when you need to provide a set of overloaded methods that differ in the number of the same type of arguments. For example, suppose you want to have methods to calculate an average. You may want to calculate averages for 2, 3, or 4 (or more) integers.

Each of these methods performs a similar type of computation—the average of the arguments passed in, as in this example:

```
public class Statistics {  
    public float average(int x1, int x2) { return (x1 + x2) / 2; }  
    public float average(int x1, int x2, int x3) {  
        return (x1 + x2 + x3) / 3;  
    }  
    public float average(int x1, int x2, int x3, int x4) {  
        return (x1 + x2 + x3 + x4) / 4;  
    }  
}
```

Java provides a convenient syntax for collapsing these three methods into just one and providing for any number of arguments.

Methods Using Variable Arguments

- Java provides a feature called *varargs* or *variable arguments*.

```
1 public class Statistics {  
2     public float average(int... nums) {  
3         int sum = 0;  
4         for (int x : nums) { // iterate int array nums  
5             sum += x;  
6         }  
7         return ((float) sum / nums.length);  
8     }  
9 }
```

The varargs notation treats the `nums` parameter as an array.

- Note that the `nums` argument is actually an array object of type `int []`. This permits the method to iterate over and allow any number of elements.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Using Variable Arguments

The `average` method shown in the slide takes any number of integer arguments. The notation `(int... nums)` converts the list of arguments passed to the `average` method into an array object of type `int`.

Note: Methods that use varargs can also take no parameters—an invocation of `average()` is legal. You will see varargs as optional parameters in use in the NIO.2 API in the lesson titled “Java File I/O.” To account for this, you could rewrite the `average` method in the slide as follows:

```
public float average(int... nums) {  
    int sum = 0; float result = 0;  
    if (nums.length > 0) {  
        for (int x : nums) // iterate int array nums  
            sum += x;  
        result = (float) sum / nums.length;  
    }  
    return (result);  
}
```

Casting Object References

After using the `instanceof` operator to verify that the object you received as an argument is a subclass, you can access the full functionality of the object by casting the reference:

```
4  public static void main(String[] args) {  
5      Employee e = new Manager(102, "Joan Kern",  
6          "012-23-4567", 110_450.54, "Marketing");  
7  
8      if (e instanceof Manager){  
9          Manager m = (Manager) e;  
10         m.setDeptName("HR");  
11         System.out.println(m.getDetails());  
12     }  
13 }
```

Without the cast to `Manager`, the `setDeptName` method would not compile.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Although a generic superclass reference is useful for passing objects around, you may need to use a method from the subclass.

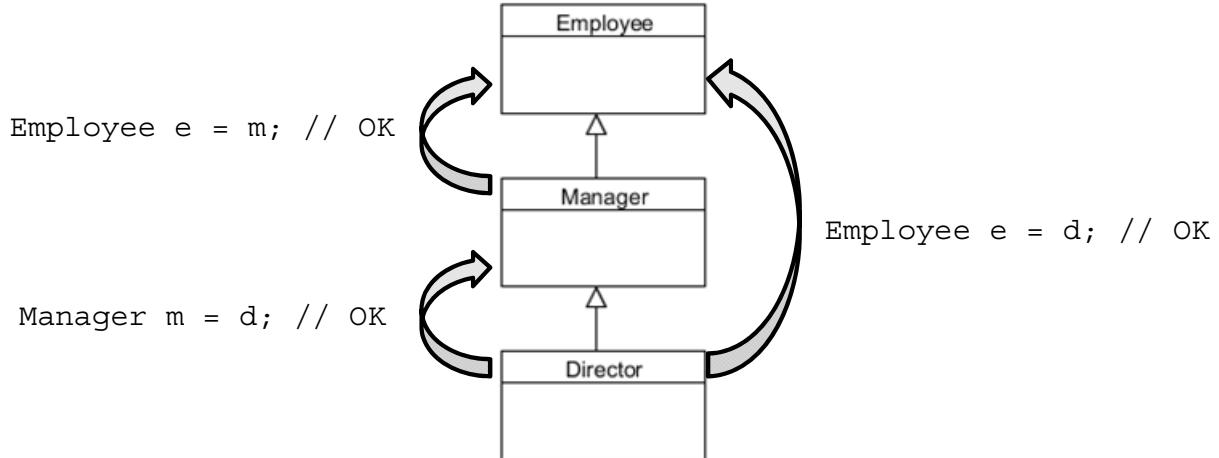
In the slide, for example, you need the `setDeptName` method of the `Manager` class. To satisfy the compiler, you can cast a reference from the generic superclass to the specific class.

However, there are rules for casting references. You see these in the next slide.

Upward Casting Rules

Upward casts are always permitted and do not require a cast operator.

```
Director d = new Director();  
Manager m = new Manager();
```



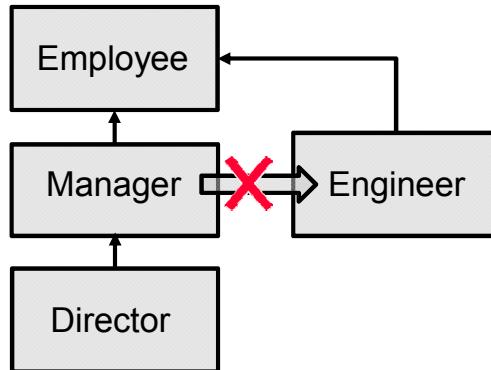
ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Downward Casting Rules

For downward casts, the compiler must be satisfied that the cast is possible.

```
5     Employee e = new Manager(102, "Joan Kern",
6             "012-23-4567", 110_450.54, "Marketing");
7
8     Manager m = (Manager)e; // ok
9     Engineer eng = (Manager)e; // Compile error
10    System.out.println(m.getDetails());
```



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Downward Casts

With a downward cast, the compiler simply determines if the cast is possible; if the cast down is to a subclass, then it is possible that the cast will succeed.

Note that at run time, the cast results in a `java.lang.ClassCastException` if the object reference is of a superclass and not of the class type or a subclass.

Finally, any cast that is outside the class hierarchy will fail, such as the cast from a `Manager` instance to an `Engineer`. A `Manager` and an `Engineer` are both employees, but a `Manager` is not an `Engineer`.

static Keyword

The `static` modifier is used to declare fields and methods as class-level resources.

Static class members:

- Can be used without object instances
- Are used when a problem is best solved without objects
- Are used when objects of the same type need to share fields
- Should *not* be used to bypass the object-oriented features of Java unless there is a good reason



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Static Methods

Static methods are methods that can be called even if the class they are declared in has not been instantiated.

Static methods:

- Are called class methods
- Are useful for APIs that are not object oriented
 - `java.lang.Math` contains many static methods
- Are commonly used in place of constructors to perform tasks related to object initialization
- Cannot access nonstatic members within the same class



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Using Static Variables and Methods: Example

```
3 public class A01MathTest {  
4     public static void main(String[] args) {  
5         System.out.println("Random: " + Math.random() * 10);  
6         System.out.println("Square root: " + Math.sqrt(9.0));  
7         System.out.println("Rounded random: " +  
8             Math.round(Math.random()*100));  
9         System.out.println("Abs: " + Math.abs(-9));  
10    }  
11 }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The `java.lang.Math` class contains methods and constants for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric.

The methods and constants in the `Math` class are all static, so you call them directly from the class as the examples in the slide demonstrates.

Here is some sample output from this example:

Random: 7.064916553599695

Square root: 3.0

Rounded random: 35

Abs: 9

Implementing Static Methods

- Use the static keyword before the method
- The method has parameters and return types like normal

```
3 import java.time.LocalDate;
4
5 public class StaticHelper {
6
7     public static void printMessage(String message) {
8         System.out.println("Messsage for " +
9             LocalDate.now() + ":" + message);
10    }
11
12 }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Static methods or class methods may be without an instantiating an object.

Static Method Limitations

Static methods can be used before any instances of their enclosing class have been created. If a class contains both static and instance components, the instance components cannot be accessed for a static context.

Calling Static Methods

```
double d = Math.random();  
StaticHelper.printMessage("Hello");
```

When calling static methods, you should:

- Qualify the location of the method with a class name if the method is located in a different class than the caller
 - Not required for methods within the same class
- Avoid using an object reference to call a static method



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Static Variables

Static variables are variables that can be accessed even if the class they are declared in has not been instantiated.

Static variables are:

- Called class variables
- Limited to a single copy per JVM
- Useful for containing shared data
 - Static methods store data in static variables.
 - All object instances share a single copy of any static variables.
- Initialized when the containing class is first loaded



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Class Loading

Application developer-supplied classes are typically loaded on demand (first use). Static variables are initialized when their enclosing class is loaded. An attempt to access a static class member can trigger the loading of a class.

Defining Static Variables

```
4 public class StaticCounter {  
5     private static int counter = 0;  
6  
7     public static int getCount() {  
8         return counter;  
9     }  
10  
11    public static void increment() {  
12        counter++;  
13    }  
14 }
```

Only one copy in memory



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

A static variable is defined when the `static` keyword precedes the type definition for a variable. Static variables are useful for containing shared data, all object instances share a single copy of any static variables.

Using Static Variables

```
double p = Math.PI;

5  public static void main(String[] args) {
6      System.out.println("Start: " + StaticCounter.getCount());
7      StaticCounter.increment();
8      StaticCounter.increment();
9      System.out.println("End: " + StaticCounter.getCount());
10 }
```

When accessing static variables, you should:

- Qualify the location of the variable with a class name if the variable is located in a different class than the caller
 - Not required for variables within the same class
- Avoid using an object reference to access a static variable



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Object References to Static Members

Just as using object references to static methods should be avoided, you should also avoid using object references to access static variables. Using a `private` access level prevents direct access to static variables.

Sample output:

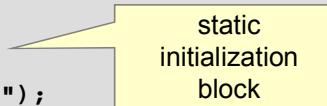
Start: 0

End: 2

Static Initializers

- Static initializer block is a code block prefixed by the static keyword.

```
3 public class A04StaticInitializerTest {  
4     private static final boolean[] switches = new boolean[5];  
5  
6     static{  
7         System.out.println("Initializing...");  
8         for (int i=0; i<5; i++){  
9             switches[i] = true;  
10        }  
11    }  
12  
13    public static void main(String[] args) {  
14        switches[1] = false; switches[2] = false;  
15        System.out.print("Switch settings: ");  
16        for (boolean curSwitch:switches){  
17            if (curSwitch){System.out.print("1");}  
18            else {System.out.print("0");}  
19        }  
}
```



static initialization block



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Here are some key facts about static initialization blocks:

- They are executed only once when the class is loaded.
- They are used to initialize static variables.
- A class can contain one or more static initializer blocks.
- They can appear anywhere in the class body.
- The blocks are called in the order that they appear in the source code.

Consider using static initializers when nontrivial code is required to initialize static variables.

Static Imports

A static import statement makes the static members of a class available under their simple name.

- Given either of the following lines:

```
import static java.lang.Math.random;  
import static java.lang.Math.*;
```

- Calling the `Math.random()` method can be written as:

```
public class StaticImport {  
    public static void main(String[] args) {  
        double d = random();  
    }  
}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Overusing static import can negatively affect the readability of your code. Avoid adding multiple static imports to a class.

Design Patterns

Design patterns are:

- Reusable solutions to common software development problems
- Documented in pattern catalogs
 - *Design Patterns: Elements of Reusable Object-Oriented Software*, written by Erich Gamma et al. (the “Gang of Four”)
- A vocabulary used to discuss design



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Design Pattern Catalogs

Pattern catalogs are available for many programming languages. Most of the traditional design patterns apply to any object-oriented programming language. One of the most popular books, *Design Patterns: Elements of Reusable Object-Oriented Software*, uses a combination of C++, Smalltalk, and diagrams to show possible pattern implementations. Many Java developers still reference this book because the concepts translate to any object-oriented language.

You learn more about design patterns and other Java best practices in the *Java Design Patterns* course.

Singleton Pattern

The singleton design pattern details a class implementation that can be instantiated only once.

```
public class SingletonClass {  
    1  private static final SingletonClass instance =  
        new SingletonClass();  
  
    2  private SingletonClass() {}  
  
    public static SingletonClass getInstance() {  
        3      return instance;  
    }  
}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Implementing the Singleton Pattern

A singleton is a class for which only one instance can be created and it provides a global point of access to this instance.

Singletons are useful to provide a unique source of data or functionality to other Java Objects. For example, you may use a singleton to access your data model from within your application or to define logger which the rest of the application can use.

To implement the singleton design pattern:

1. Use a static reference to point to the single instance. Making the reference final ensures that it will never reference a different instance.
2. Add a single private constructor to the singleton class. The private modifier allows only "same class" access, which prohibits any attempts to instantiate the singleton class except for the attempt in step 1.
3. A public factory method returns a copy of the singleton reference. This method is declared static to access the static field declared in step 1. Step 1 could use a public variable, eliminating the need for the factory method. Factory methods provide greater flexibility (for example, implementing a per-thread singleton solution) and are typically used in most singleton implementations.
4. In singleton pattern, it restricts the creation of instance until requested first time(Lazy initialization). To obtain a singleton reference, call the getInstance method:

```
SingletonClass ref = SingletonClass.getInstance();
```

Singleton: Example

```
3 public final class DbConfigSingleton {  
4     private final String hostName;  
5     private final String dbName;  
6     //Lines omitted  
10    private static final DbConfigSingleton instance =  
11        new DbConfigSingleton();  
12  
13    private DbConfigSingleton(){  
14        // Values loaded from file in practice  
15        hostName = "dbhost.example.com";  
16        // Lines omitted  
20    }  
21  
22    public static DbConfigSingleton getInstance() {  
23        return instance;  
24    }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Singletons are great for storing data shared by an entire application. In this example, some database configuration data is stored in a Singleton.

Immutable Classes

Immutable class:

- It is a class whose object state cannot be modified once created.
- Any modification of the object will result in another new immutable object.
- Example: Objects of `Java.lang.String`, any change on existing string object will result in another string; for example, replacing a character or creating substrings will result in new objects.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Rules to create a immutable class in Java:

- State of immutable object can not be modified after construction. Any modification would result in a new immutable object.
- Declare the class as final so it cannot be extended.
- All fields are declared private so that direct access is not allowed.
- Setter methods are not provided for variables.
- All fields of immutable class should be final.
- All the fields are initialized via a constructor.
- Object should be final in order to restrict subclass from altering immutability of parent class.

Example: Creating Immutable class in Java

```
public final class Contacts {  
    private final String firstName;  
    private final String lastName;  
  
    public Contacts(String fname, String lname) {  
        this.firstName = fname;  
        this.lastName = lname;  
    }  
    public String getFirstName() {  
        return firstName;  
    }  
    public String getLastName() {  
        return lastName;  
    }  
  
    public String toString() {  
        return firstName + " - " + lastName + " - " + lastName;  
    }  
}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The example in the slide demonstrates an immutable class in Java where all fields of class remain immutable and the class is also made final to avoid risk of immutability through inheritance.

Benefits of immutable classes in Java

1. Immutable objects are by default thread-safe, and can be shared without synchronization in concurrent environment.
2. Immutable object boost performance of Java application by reducing synchronization in code.
3. Reusability, you can cache immutable object and reuse them, much like String literals and Integers.

Note: If Immutable class has many optional and mandatory fields, you can also use Builder Design Pattern to make a class Immutable in Java.

Summary

In this lesson, you should have learned how to:

- Use access levels: `private`, `protected`, `default`, and `public`.
- Override methods
- Use virtual method invocation
- Use `varargs` to specify variable arguments
- Use the `instanceof` operator to compare object types
- Use upward and downward casts
- Model business problems by using the `static` keyword
- Implement the singleton design pattern



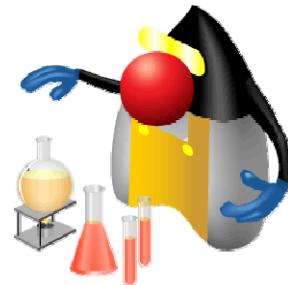
ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 4-1 Overview: Overriding Methods and Applying Polymorphism

This practice covers the following topics:

- Modifying the Employee, Manager, and Director classes; overriding the `toString()` method
- Creating an `EmployeeStockPlan` class with a grant stock method that uses the `instanceof` keyword



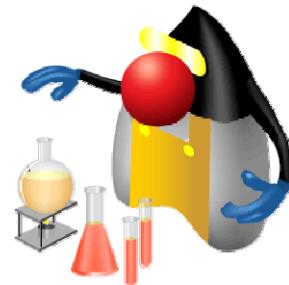
ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 4-2 Overview: Overriding Methods and Applying Polymorphism

This practice covers the following topics:

- Fixing compilation errors caused due to casting
- Identifying runtime exception caused due to improper casting

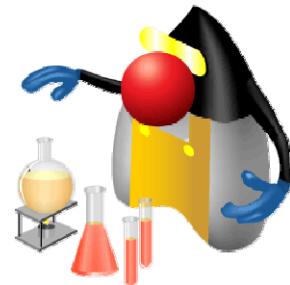


ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 4-3 Overview: Applying the Singleton Design Pattern

This practice covers using the `static` and `final` keywords and refactoring an existing application to implement the singleton design pattern.



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Quiz

Suppose that you have an Account class with a withdraw() method, and a Checking class that extends Account that declares its own withdraw() method. What is the result of the following code fragment?

```
1 Account acct = new Checking();  
2 acct.withdraw(100);
```

- a. The compiler complains about line 1.
- b. The compiler complains about line 2.
- c. Runtime error: incompatible assignment (line 1)
- d. Executes withdraw method from the Account class
- e. Executes withdraw method from the Checking class



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Quiz

Suppose that you have an Account class and a Checking class that extends Account. The body of the if statement in line 2 will execute.

```
1 Account acct = new Checking();  
2 if (acct instanceof Checking) { // will this block run? }
```

- a. True
- b. False



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Quiz

Suppose that you have an Account class and a Checking class that extends Account. You also have a Savings class that extends Account. What is the result of the following code?

```
1 Account acct1 = new Checking();  
2 Account acct2 = new Savings();  
3 Savings acct3 = (Savings)acct1;
```

- a. acct3 contains the reference to acct1.
- b. A runtime ClassCastException occurs.
- c. The compiler complains about line 2.
- d. The compiler complains about the cast in line 3.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Abstract and Nested Classes

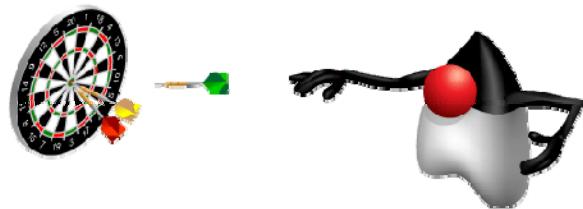
ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Objectives

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

- Design general-purpose base classes by using abstract classes
- Construct abstract Java classes and subclasses
- Apply the `final` keyword in Java
- Distinguish between top-level and nested classes



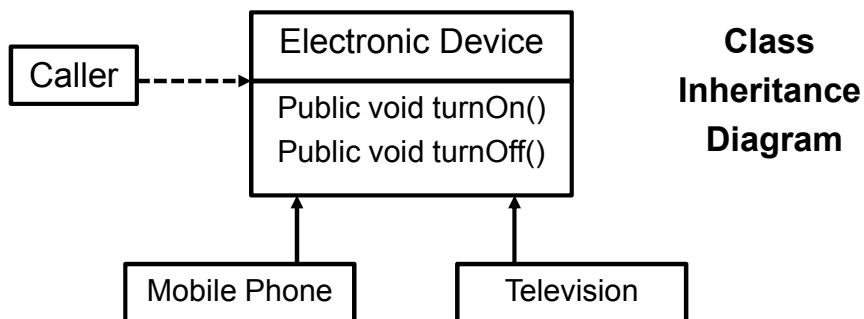
ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Modeling Business Problems with Classes

Inheritance (or subclassing) is an essential feature of the Java programming language. Inheritance provides code reuse through:

- **Method inheritance:** Subclasses avoid code duplication by inheriting method implementations.
- **Generalization:** Code that is designed to rely on the most generic type possible is easier to maintain.



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Class Inheritance

When designing an object-oriented solution, you should attempt to avoid code duplication. One technique to avoid duplication is to create library methods and classes. Libraries function as a central point to contain often reused code. Another technique to avoid code duplication is to use class inheritance. When there is a shared base type identified between two classes, any shared code may be placed in a parent class.

When possible, use object references of the most generic base type possible. In Java, generalization and specialization enable reuse through method inheritance and virtual method invocation (VMI). VMI, sometimes called “late-binding,” enables a caller to dynamically call a method as long as the method has been declared in a generic base type.

Enabling Generalization

Coding to a common base type allows for the introduction of new subclasses with little or no modification of any code that depends on the more generic base type.

```
ElectronicDevice dev = new Television();
dev.turnOn(); // all ElectronicDevices can be turned on
```

Always use the most generic reference type possible.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Coding for Generalization

Always use the most generic reference type possible. Java IDEs may contain refactoring tools that assist in changing existing references to a more generic base type.

Identifying the Need for Abstract Classes

Subclasses may not need to inherit a method implementation if the method is specialized.

```
public class Television extends ElectronicDevice {  
  
    public void turnOn() {  
        changeChannel(1);  
        initializeScreen();  
    }  
    public void turnOff() {}  
  
    public void changeChannel(int channel) {}  
    public void initializeScreen() {}  
  
}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Method Implementations

When sibling classes have a common method, it is typically placed in a parent class. Under some circumstances, however, the parent class's implementation will always need to be overridden with a specialized implementation.

In these cases, inclusion of the method in a parent class has both advantages and disadvantages. It allows the use of generic reference types, but developers can easily forget to supply the specialized implementation in the subclasses.

Defining Abstract Classes

A class can be declared as abstract by using the `abstract` class-level modifier.

```
public abstract class ElectronicDevice { }
```

- An abstract class can be subclassed.

```
public class Television extends ElectronicDevice { }
```

- An abstract class **cannot** be instantiated.

```
ElectronicDevice dev = new ElectronicDevice(); // error
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Declaring a class as abstract prevents any instances of that class from being created. It is a compile-time error to instantiate an abstract class. An abstract class is typically extended by a child class and may be used as a reference type.

Defining Abstract Methods

A method can be declared as abstract by using the `abstract` method-level modifier.

```
public abstract class ElectronicDevice {  
  
    public abstract void turnOn();  
    public abstract void turnOff();  
}
```

No braces

An abstract method:

- Cannot have a method body
- Must be declared in an abstract class
- Is overridden in subclasses

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Inheriting Abstract Methods

When a child class inherits an abstract method, it is inheriting a method signature but no implementation. For this reason, no braces are allowed when defining an abstract method. An abstract method is a way to guarantee that any child class will contain a method with a matching signature.

Note: An abstract method can take arguments and return values. For example:

```
abstract double calculateArea(double dim1, double dim2);
```

Validating Abstract Classes

The following additional rules apply when you use abstract classes and methods:

- An abstract class may have any number of abstract and nonabstract methods.
- When inheriting from an abstract class, you must do either of the following:
 - Declare the child class as abstract.
 - Override all abstract methods inherited from the parent class. Failure to do so will result in a compile-time error.

```
error: Television is not abstract and does not override  
abstract method turnOn() in ElectronicDevice
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Making Use of Abstract Classes

While it is possible to avoid implementing an abstract method by declaring child classes as abstract, this only serves to delay the inevitable. Applications require nonabstract methods to create objects. Use abstract methods to outline functionality required in child classes.

Final Methods

A method can be declared `final`. Final methods may not be overridden.

```
public class MethodParentClass {  
    public final void printMessage() {  
        System.out.println("This is a final method");  
    }  
}
```

```
public class MethodChildClass extends MethodParentClass {  
    // compile-time error  
    public void printMessage() {  
        System.out.println("Cannot override method");  
    }  
}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Performance Myths

There is little to no performance benefit when you declare a method as `final`. Methods should be declared as `final` only to disable method overriding.

Final Classes

A class can be declared `final`. Final classes may not be extended.

```
public final class FinalParentClass { }
```

```
// compile-time error
public class ChildClass extends FinalParentClass { }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Final Variables

The `final` modifier can be applied to variables.

Final variables may not change their values after they are initialized.

Final variables can be:

- Class fields
 - Final fields with compile-time constant expressions are constant variables.
 - Static can be combined with final to create an always-available, never-changing variable.
- Method parameters
- Local variables

Note: Final references must always reference the same object, but the contents of that object may be modified.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Benefits and Drawbacks of Final Variables

Bug Prevention

Final variables can never have their values modified after they are initialized. This behavior functions as a bug-prevention mechanism.

Thread Safety

The immutable nature of final variables eliminates any of the concerns that come with concurrent access by multiple threads.

Final Reference to Objects

A `final` object reference only prevents a reference from pointing to another object. If you are designing immutable objects, you must prevent the object's fields from being modified. Final references also prevent you from assigning a value of `null` to the reference. Maintaining an object's references prevents that object from being available for garbage collection.

Declaring Final Variables

```
public class VariableExampleClass {  
    private final int field;  
    public static final int JAVA_CONSTANT = 10;  
  
    public VariableExampleClass() {  
        field = 100;  
    }  
  
    public void changeValues(final int param) {  
        param = 1; // compile-time error  
        final int localVar;  
        localVar = 42;  
        localVar = 43; // compile-time error  
    }  
}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Final Fields

Initializing

Final fields (instance variables) must be either of the following:

- Assigned a value when declared
- Assigned a value in every constructor

Static and Final Combined

A field that is both static and final is considered a constant. By convention, constant fields use identifiers consisting of only uppercase letters and underscores.

Nested Classes

A nested class is a class declared within the body of another class. Nested classes:

- Have multiple categories
 - **Inner classes**
 - Member classes
 - Local classes
 - Anonymous classes
 - **Static nested classes**
- Are commonly used in applications with GUI elements
- Can limit utilization of a "helper class" to the enclosing top-level class



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

An **inner** nested class is considered part of the outer class and inherits access to all the private members of the outer class.

A **static** nested class is not an inner class, but its declaration appears similar with an additional `static` modifier on the nested class. Static nested classes can be instantiated before the enclosing outer class and, therefore, are denied access to all nonstatic members of the enclosing class.

Note: Anonymous classes are covered in detail in the lesson titled “Interfaces and Lambda Expressions.”

Reasons to Use Nested Classes

The following information is obtained from

<http://download.oracle.com/javase/tutorial/java/javaOO/nested.html>.

- **Logical Grouping of Classes**
 - If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.
- **Increased Encapsulation**
 - Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.
- **More Readable, Maintainable Code**
 - Nesting small classes within top-level classes places the code closer to where it is used.

Example: Member Class

```
public class BankEMICalculator {  
    private String CustomerName;  
    private String AccountNo;  
    private double loanAmount;  
    private double monthlypayment;  
    private EMICalculatorHelper helper = new EMICalculatorHelper();  
  
    /*Setters ad Getters*/  
  
    private class EMICalculatorHelper {  
        int loanTerm = 60;  
        double interestRate = 0.9;  
        double interestpermonth=interestRate/loanTerm;  
  
        protected double calcMonthlyPayment(double loanAmount)  
        {  
            double EMI= (loanAmount * interestpermonth) / ((1.0) - ((1.0) /  
Math.pow(1.0 + interestpermonth, loanTerm)));  
            return(Math.round(EMI));  
        }  
    }  
}
```

Inner class,
EMICalculatorHelper

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The example in the slide demonstrates an inner class, EMICalculatorHelper, which is defined in the BankEMICalculator class.

Enumerations

Java includes a typesafe enum to the language.

Enumerations (enums):

- Are created by using a variation of a Java class
- Provide a compile-time range check

```
public enum PowerState {  
    OFF,  
    ON,  
    SUSPEND;  
}
```

These are references to the only three PowerState objects that can exist.

An enum can be used in the following way:

```
Computer comp = new Computer();  
comp.setState(PowerState.SUSPEND);
```

This method takes a PowerState reference .

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Compile-Time Range Checking

In the example in the slide, the compiler performs a compile-time check to ensure that only valid PowerState instances are passed to the `setState` method. No range checking overhead is incurred at runtime.

Enum Usage

Enums can be used as the expression in a switch statement.

```
public void setState(PowerState state) {  
    switch(state) {  
        case OFF:  
            //...  
    }  
}
```

PowerState.OFF



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Complex Enums

Enums can have fields, methods, and private constructors.

```
public enum PowerState {  
    OFF("The power is off"),  
    ON("The usage power is high"),  
    SUSPEND("The power usage is low");  
  
    private String description;  
    private PowerState(String d) {  
        description = d;  
    }  
    public String getDescription() {  
        return description;  
    }  
}
```

Call a PowerState constructor to initialize the public static final OFF reference.

The constructor may not be public or protected.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Enum Constructors

You may not instantiate an enum instance with `new`.

Complex Enums

- Here is the complex enum in action.

```
public class ComplexEnumsMain {  
  
    public static void main(String[] args) {  
        Computer comp = new Computer();  
        comp.setState(PowerState.SUSPEND);  
        System.out.println("Current state: " +  
            comp.getState());  
        System.out.println("Description: " +  
            comp.getState().getDescription());  
    }  
}
```

- Output

```
Current state: SUSPEND  
Description: The power usage is low
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Enum Usage

When sent to be printed, the default behavior for an enumeration is to print the current value. Typically, an additional call to the enumeration's methods is necessary to get information stored in the enumeration.

Summary

In this lesson, you should have learned how to:

- Design general-purpose base classes by using abstract classes
- Construct abstract Java classes and subclasses
- Apply the `final` keyword in Java
- Distinguish between top-level and nested classes



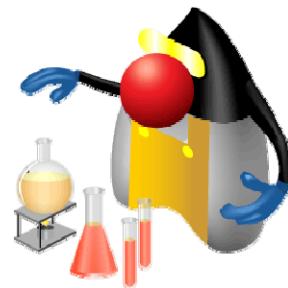
ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 5-1 Overview: Applying the Abstract Keyword

This practice covers the following topics:

- Identifying potential problems that can be solved using abstract classes
- Refactoring an existing Java application to use abstract classes and methods

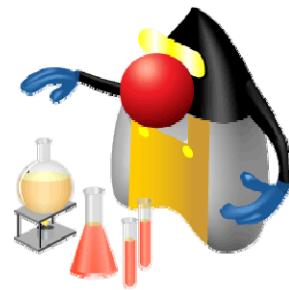


ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 5-2 Overview: Using Inner Class As a Helper Class

This practice covers using an inner class as a helper class to perform some calculations in an Employee class.

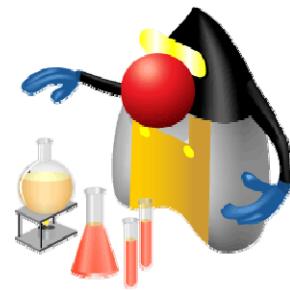


ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 5-3 Overview: Using Java Enumerations

This practice covers taking an existing application and refactoring the code to use an enum.



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Quiz

Which two of the following should an abstract method not have to compile successfully?

- a. A return value
- b. A method implementation
- c. Method parameters
- d. private access



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Quiz

Which of the following nested class types are inner classes?

- a. Anonymous
- b. Local
- c. Static
- d. Member



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Quiz

A final field (instance variable) can be assigned a value either when declared or in all constructors.

- a. True
- b. False



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Interfaces and Lambda Expressions



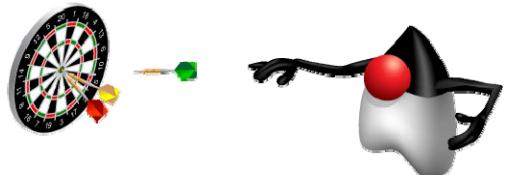
ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Objectives

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

- Define a Java interface
- Choose between interface inheritance and class inheritance
- Extend an interface
- Define a lambda expression



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Java Interfaces

Java interfaces are used to define abstract types. Interfaces:

- Are similar to abstract classes containing only public abstract methods
- Outline methods that must be implemented by a class
 - Methods must not have an implementation {braces}.
- Can contain constant fields
- Can be used as a reference type
- Are an essential component of many design patterns



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In Java, an interface outlines a contract for a class. The contract outlined by an interface mandates the methods that must be implemented in a class. Classes implementing the contract must fulfill the entire contract or be declared abstract.

A Problem Solved by Interfaces

Given: A company sells an assortment of products, very different from each other, and needs a way to access financial data in a similar manner.

- Products include:
 - Crushed Rock
 - Measured in pounds
 - Red Paint
 - Measured in gallons
 - Widgets
 - Measured by Quantity
- Need to calculate per item
 - Sales price
 - Cost
 - Profit



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

How can interfaces solve this problem? All these products are sold by the same company but are measured in different ways.

CrushedRock Class

- The CrushedRock class before interfaces

```
public class CrushedRock {  
    private String name;  
    private double salesPrice = 0;  
    private double cost = 0;  
    private double weight = 0; // In pounds  
  
    public CrushedRock(double salesPrice, double cost,  
double weight) {  
        this.salesPrice = salesPrice;  
        this.cost = cost;  
        this.weight = weight;  
    }  
}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Notice this class has a `weight` field. However, the `RedPaint` class has a `gallons` field and the `Widget` class has a `quantity` field. Calculating the required data for the three classes is similar but different for each.

The SalesCalcs Interface

- The SalesCalcs interface specifies the types of calculations required for our products.
 - Public, top-level interfaces are declared in their own .java file.

```
public interface SalesCalcs {  
    public String getName();  
    public double calcSalesPrice();  
    public double calcCost();  
    public double calcProfit();  
}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The SalesCalcs interface specifies what methods must be implemented by a class. The method signature specifies what is passed in and what is returned.

Rules for Interfaces

Access Modifiers

All methods in an interface are `public`, even if you forget to declare them as `public`. You may not declare methods as `private` or `protected` in an interface.

Abstract Modifier

Because all methods are implicitly `abstract`, it is redundant (but allowed) to declare a method as `abstract`. Because all interface methods are abstract, you may not provide any method implementation, not even an empty set of braces.

Implement Multiple Interfaces

A class can implement more than one interface in a comma-separated list at the end of the class declaration.

Adding an Interface

- The updated CrushedRock class implements SalesCalcs.

```
public class CrushedRock implements SalesCalcs{  
    private String name = "Crushed Rock";  
    ... // a number of lines not shown  
    @Override  
    public double calcCost(){  
        return this.cost * this.weight;  
    }  
  
    @Override  
    public double calcProfit(){  
        return this.calcSalesPrice() - this.calcCost();  
    }  
}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

On the class declaration line, the `implements` keyword specifies the `SalesCalcs` interface for this class. Each of the methods specified by `SalesCalcs` must be implemented. However, how the methods are implemented may differ from class to class. The only requirement is that the method signature matches. This allows the cost or sales price calculations to differ between classes.

Interface References

- Any class that implements an interface can be referenced by using that interface.
- Notice how the `calcSalesPrice` method can be referenced by the `CrushedRock` class or the `SalesCalcs` interface.

```
CrushedRock rock1 = new CrushedRock(12, 10, 50);
SalesCalcs rock2 = new CrushedRock(12, 10, 50);
System.out.println("Sales Price: " +
rock1.calcSalesPrice());
System.out.println("Sales Price: " +
rock2.calcSalesPrice());
```

- Output

```
Sales Price: 600.0
Sales Price: 600.0
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Because `CrushedRock` implements `SalesCalcs`, a `SalesCalcs` reference can be used to access the data of a `CrushedRock` object.

Interface Reference Usefulness

- Any class implementing an interface can be referenced by using that interface. For example:

```
SalesCalcs[] itemList = new SalesCalcs[5];  
ItemReport report = new ItemReport();  
  
itemList[0] = new CrushedRock(12.0, 10.0, 50.0);  
itemList[1] = new CrushedRock(8.0, 6.0, 10.0);  
itemList[2] = new RedPaint(10.0, 8.0, 25.0);  
itemList[3] = new Widget(6.0, 5.0, 10);  
itemList[4] = new Widget(14.0, 12.0, 20);  
  
System.out.println("==Sales Report==");  
for(SalesCalcs item:itemList){  
    report.printItemData(item);  
}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Because all three classes share a common interface, a list of different classes like the above can be created and processed in the same way.

Interface Code Flexibility

- A utility class that references the interface can process any implementing class.

```
public class ItemReport {  
    public void printItemData(SalesCalcs item) {  
        System.out.println(" -- " + item.getName() + " Report-  
-");  
        System.out.println("Sales Price: " +  
item.calcSalesPrice());  
        System.out.println("Cost: " + item.calcCost());  
        System.out.println("Profit: " + item.calcProfit());  
    }  
}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Instead of having to write a method to print the data from each class, the interface reference allows you to retrieve the data from all three classes.

default Methods in Interfaces

Java 8 has added **default** methods as a new feature:

```
public interface SalesCalcs {  
    ... // A number of lines omitted  
    public default void printItemReport() {  
        System.out.println("--" + this.getName() + " Report--");  
        System.out.println("Sales Price: " + this.calcSalesPrice());  
        System.out.println("Cost: " + this.calcCost());  
        System.out.println("Profit: " + this.calcProfit());  
    }  
}
```

default methods:

- Are declared by using the keyword **default**
- Are fully implemented methods within an interface
- Provide useful inheritance mechanics



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Java 8 adds **default** methods as a new feature. Using the **default** keyword allows you to provide fully implemented methods to all implementing classes. The above example shows how the item report, which was implemented as a separate class earlier, can be fully implemented as a **default** method. Now all three classes automatically get a fully implemented **printItemReport** method.

The feature was added to simplify the development of APIs that rely heavily on interfaces. Before, simply adding a new method breaks all implementing and extended classes. Now, **default** methods can be added or changed without harming API hierarchies.

default Method: Example

Here is an updated version of the item report using `default` methods.

```
SalesCalcs[] itemList = new SalesCalcs[5];  
  
itemList[0] = new CrushedRock(12, 10, 50);  
itemList[1] = new CrushedRock(8, 6, 10);  
itemList[2] = new RedPaint(10, 8, 25);  
itemList[3] = new Widget(6, 5, 10);  
itemList[4] = new Widget(14, 12, 20);  
  
System.out.println("==Sales Report==");  
for(SalesCalcs item:itemList){  
    item.printItemReport();  
}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Printing the report now involves simply calling the `printItemReport` method.

static Methods in Interfaces

Java 8 allows **static** methods in an interface. So it is possible to create helper methods like the following.

```
public interface SalesCalcs {  
    ... // A number of lines omitted  
    public static void printItemArray(SalesCalcs[] items) {  
        System.out.println(reportTitle);  
        for(SalesCalcs item:items){  
            System.out.println("--" + item.getName() + " Report--");  
            System.out.println("Sales Price: " +  
item.calcSalesPrice());  
            System.out.println("Cost: " + item.calcCost());  
            System.out.println("Profit: " + item.calcProfit());  
        }  
    }  
}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

This is a convenience feature. Now you can include a helper method, like the above, in an interface instead of in a separate class.

Here is an example of calling the method:

```
SalesCalcs.printItemArray(itemList);
```

Constant Fields

Interfaces can have constant fields.

```
public interface SalesCalcs {  
    public static final String reportTitle="\\n==Static  
    List Report==";  
    ... // A number of lines omitted
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Constant fields are permitted in an interface. When you declare a field in an interface, it is implicitly `public`, `static`, and `final`. You may redundantly specify these modifiers.

Extending Interfaces

- Interfaces can extend interfaces:

```
public interface WidgetSalesCalcs extends SalesCalcs{  
    public String getWidgetType();  
}
```

- So now any class implementing WidgetSalesCalc must implement all the methods of SalesCalcs in addition to the new method specified here.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Implementing and Extending

- Classes can extend a parent class and implement an interface:

```
public class WidgetPro extends Widget implements  
WidgetSalesCalcs{  
    private String type;  
  
    public WidgetPro(double salesPrice, double cost, long  
quantity, String type){  
        super(salesPrice, cost, quantity);  
        this.type = type;  
    }  
  
    public String getWidgetType(){  
        return type;  
    }  
}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Extends First

If you use both `extends` and `implements`, `extends` must come first.

Anonymous Inner Classes

- Define a class in place instead of in a separate file
- Why would you do this?
 - Logically group code in one place
 - Increase encapsulation
 - Make code more readable
- StringAnalyzer interface

```
public interface StringAnalyzer {  
    public boolean analyze(String target, String  
        searchStr);  
}
```

- A single method interface
 - **Functional Interface**
- Takes two strings and returns a boolean



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

An interface like this with a single method is called a **Functional Interface**.

Anonymous Inner Class: Example

- Example method call with concrete class

```
20 // Call concrete class that implements StringAnalyzer  
21 ContainsAnalyzer contains = new ContainsAnalyzer();  
22  
23 System.out.println("==Contains==");  
24 Z03Analyzer.searchArr(strList01, searchStr, contains);
```

- Anonymous inner class example

```
22 Z04Analyzer.searchArr(strList01, searchStr,  
23     new StringAnalyzer(){  
24         @Override  
25         public boolean analyze(String target, String  
26             searchStr){  
27             return target.contains(searchStr);  
28     });
```

- The class is created in place.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The example shows how an anonymous inner class can be substituted for an object.

Here is the source code for ContainsAnalyzer:

```
public class ContainsAnalyzer implements StringAnalyzer {  
    public boolean analyze(String target, String searchStr) {  
        return target.contains(searchStr);  
    }  
}
```

Note that the anonymous inner class specifies no name but implements almost exactly the same code. The syntax is a little complicated as the class is defined where a parameter variable would normally be.

The slides that follow explain some of the advantages of this approach.

String Analysis Regular Class

- Class analyzes an array of strings given a search string
 - Print strings that contain the search string
 - Other methods could be written to perform similar string test
- Regular Class Example method

```
1 package com.example;
2
3 public class AnalyzerTool {
4     public boolean arrContains(String sourceStr, String
5         searchStr) {
6         return sourceStr.contains(searchStr);
7     }
8 }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The method takes the source string and searches for text matching the text in the search string. If there is a match, `true` is returned. If no match is found, `false` is returned.

Additional methods could be written to compare lengths or determine if the string starts with the search string. Only one method is implemented for simplicity.

String Analysis Regular Test Class

- Here is the code to test the class, Z01Analyzer

```
4  public static void main(String[] args) {  
5      String[] strList =  
6          {"tomorrow","toto","to","timbukto","the","hello","heat"};  
7      String searchStr = "to";  
8      System.out.println("Searching for: " + searchStr);  
9  
10     // Create regular class  
11     AnalyzerTool analyzeTool = new AnalyzerTool();  
12  
13     System.out.println("==Contains==");  
14     for(String currentStr:strList){  
15         if  (analyzeTool.arrContains(currentStr, searchStr)) {  
16             System.out.println("Match: " + currentStr);  
17         }  
18     }  
19 }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

This first class is pretty standard. The test array is passed to `foreach` loop where the method is used to print out matching words. Here is the output:

```
==Contains==  
Match: tomorrow  
Match: toto  
Match: to  
Match: timbukto
```

String Analysis Interface: Example

- What about using an interface?

```
3 public interface StringAnalyzer {  
4     public boolean analyze(String sourceStr, String  
5         searchStr);  
6 }  
7 }
```

- `StringAnalyzer` is a single method functional interface.
- Replacing the previous example and implementing the interface looks like this:

```
3 public class ContainsAnalyzer implements StringAnalyzer {  
4     @Override  
5     public boolean analyze(String target, String searchStr) {  
6         return target.contains(searchStr);  
7     }  
8 }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In the example, a switch is made to use an interface instead of just a plain class. Notice that `StringAnalyzer` is a functional interface because it has only one method. Other than the addition of the `implements` clause, the class is unchanged.

String Analyzer Interface Test Class

```
4  public static void main(String[] args) {  
5      String[] strList =  
6          {"tomorrow", "toto", "to", "timbukto", "the", "hello", "heat"};  
7      String searchStr = "to";  
8      System.out.println("Searching for: " + searchStr);  
9  
10     // Call concrete class that implements StringAnalyzer  
11     ContainsAnalyzer contains = new ContainsAnalyzer();  
12  
13     System.out.println("====Contains====");  
14     for(String currentStr:strList){  
15         if (contains.analyze(currentStr, searchStr)){  
16             System.out.println("Match: " + currentStr);  
17         }  
18     }  
19 }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The change to an interface does not change this test class much. The only difference is that a different class is used to perform the string testing. Also, if additional tests need to be performed, this would require additional `foreach` loops and a separate class for each test condition. Arguably, this might be a step back.

However, there are advantages of this approach.

Encapsulate the for Loop

- An improvement to the code is to encapsulate the forloop:

```
3 public class Z03Analyzer {  
4  
5     public static void searchArr(String[] strList, String  
6         searchStr, StringAnalyzer analyzer){  
7         for(String currentStr:strList){  
8             if (analyzer.analyze(currentStr, searchStr)){  
9                 System.out.println("Match: " + currentStr);  
10            }  
11        }  
12    }  
// A number of lines omitted
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

By encapsulating the forloop into a static helper method, only one loop is needed to process any sort of string test using the StringAnalyzer interface. The searchArr method remains unchanged in all the examples that follow.

Note the parameters for the method:

1. The string array
2. The search string
3. A class that implements the StringAnalyzer interface

String Analysis Test Class with Helper Method

- With the helper method, the main method shrinks to this:

```
13  public static void main(String[] args) {  
14      String[] strList01 =  
15          {"tomorrow", "toto", "to", "timbukto", "the", "hello", "heat"};  
16      String searchStr = "to";  
17      System.out.println("Searching for: " + searchStr);  
18  
19      // Call concrete class that implements StringAnalyzer  
20      ContainsAnalyzer contains = new ContainsAnalyzer();  
21  
22      System.out.println("====Contains====");  
23      Z03Analyzer.searchArr(strList01, searchStr, contains);  
24  }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Now the array can be searched and the results displayed with the single call on line 23.

String Analysis Anonymous Inner Class

- Create anonymous inner class for third argument.

```
19     // Implement anonymous inner class
20     System.out.println("==Contains==");
21     Z04Analyzer.searchArr(strList01, searchStr,
22         new StringAnalyzer(){
23             @Override
24             public boolean analyze(String target, String
25             searchStr){
26                 return target.contains(searchStr);
27             }
28         });
29 }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In this example, the third argument to the method call is an anonymous inner class. Notice that the class structure is the same as that of the `ContainsAnalyzer` in the previous example. Using the approach, the code is stored in the calling class. In addition, the logic for the `analyze` method can easily be changed depending on the circumstances. However, there are a few drawbacks.

1. The syntax is a little complicated. The entire class definition is included between the parentheses of the argument list.
2. Because there is no class name, when the code is compiled, a class file will be generated and a number assigned for the class. This is not a problem if there is only one anonymous inner class. However, when there is more than one in multiple classes, it is difficult to figure out which class file goes with which source file.

String Analysis Lambda Expression

- Use lambda expression for the third argument.

```
13  public static void main(String[] args) {  
14      String[] strList =  
15          {"tomorrow", "toto", "to", "timbukto", "the", "hello", "heat"};  
16      String searchStr = "to";  
17      System.out.println("Searching for: " + searchStr);  
18  
19      // Lambda Expression replaces anonymous inner class  
20      System.out.println("==Contains==");  
21      Z05Analyzer.searchArr(strList, searchStr,  
22          (String target, String search) -> target.contains(search));  
23  }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

With Java 8, a lambda expression can be substituted for an anonymous inner class. Notice some key facts.

- **The lambda expression has two arguments:** Just like in the two previous examples, the lambda expression uses the same arguments as the analyze method.
- **The lambda expression returns a boolean:** Just like in the two previous examples, a boolean is returned just like the analyze method.

In fact, the lambda expression, anonymous inner class, and concrete class are all essentially equivalent. A lambda expression is a new way to express code logic by using a functional interface as the base.

Lambda Expression Defined

Argument List	Arrow Token	Body
(int x, int y)	->	x + y

Basic Lambda examples

```
(int x, int y) -> x + y
```

```
(x, y) -> x + y
```

```
(x, y) -> { System.out.println(x + y); }
```

```
(String s) -> s.contains("word")
```

```
s -> s.contains("word")
```



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The lambda expression is an argument list, the arrow token, and then a block or expression. When a code block is used, multiple statements could be included in the block. The parameter types can be specified or inferred.

What Is a Lambda Expression?

```
(t,s) -> t.contains(s)
```

ContainsAnalyzer.java

```
public class ContainsAnalyzer implements StringAnalyzer {  
    public boolean analyze(String target, String searchStr) {  
        return target.contains(searchStr);  
    }  
}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Compare a lambda expression to an implementation of the `StringAnalyzer` interface.

What Is a Lambda Expression?

ContainsAnalyzer.java

```
public class ContainsAnalyzer implements StringAnalyzer {  
    public boolean analyze(String target, String searchStr) {  
        return target.contains(searchStr);  
    }  
}
```

(t, s) -> t.contains(s)

Both have parameters

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

ORACLE

In the slide, a lambda expression is compared to an implementation of the `StringAnalyzer` interface. In essence, they are equivalent because the lambda expression can be substituted for an anonymous inner class or the implementing class. All three cases rely on the `StringAnalyzer` interface as the underlying plumbing.

What Is a Lambda Expression?

ContainsAnalyzer.java

```
public class ContainsAnalyzer implements StringAnalyzer {  
    public boolean analyze(String target, String searchStr) {  
        return target.contains(searchStr);  
    }  
}
```

(t, s) -> t.contains(s)

Both have parameters

Both have a body with
one or more
statements



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In the slide, a lambda expression is compared to an implementation of the `StringAnalyzer` interface. In essence, they are equivalent because the lambda expression can be substituted for an anonymous inner class or the implementing class. All three cases rely on the `StringAnalyzer` interface as the underlying plumbing.

Lambda Expression Shorthand

- Lambda expressions using shortened syntax

```
20    // Use short form Lambda
21    System.out.println("==Contains==");
22    Z06Analyzer.searchArr(strList01, searchStr,
23        (t, s) -> t.contains(s));
24
25    // Changing logic becomes easy
26    System.out.println("==Starts With==");
27    Z06Analyzer.searchArr(strList01, searchStr,
28        (t, s) -> t.startsWith(s));
```

- The searchArr method arguments are:

```
public static void searchArr(String[] strList, String
    searchStr, StringAnalyzer analyzer)
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The first example in the slide is equivalent to the previous code example. The only difference is that lambda expression shorthand has been used. The type of the arguments is inferred from the context the lambda expression is used in. So the compiler knows that the signature for the StringAnalyzer.analyze is:

```
public boolean analyze(String sourceStr, String searchStr);
```

Thus, two strings are passed in and a boolean is returned.

Notice the second example on line 28. Now it becomes trivial to change the logic for a functional interface.

Lambda Expressions as Variables

- Lambda expressions can be treated like variables.
- They can be assigned, passed around, and reused.

```
19     // Lambda expressions can be treated like variables
20     StringAnalyzer contains = (t, s) -> t.contains(s);
21     StringAnalyzer startsWith = (t, s) -> t.startsWith(s);
22
23     System.out.println("==Contains==");
24     Z07Analyzer.searchArr(strList, searchStr,
25         contains);
26
27     System.out.println("==Starts With==");
28     Z07Analyzer.searchArr(strList, searchStr,
29         startsWith);
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Summary

In this lesson, you should have learned how to:

- Define a Java interface
- Choose between interface inheritance and class inheritance
- Extend an interface
- Define a Lambda Expression



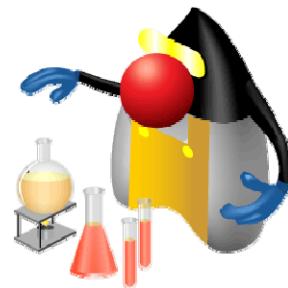
ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 6-1: Implementing an Interface

This practice covers the following topics:

- Writing an interface
- Implementing an interface
- Creating references of an interface type
- Casting to interface types



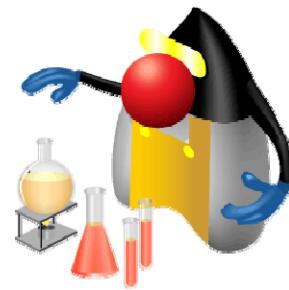
ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 6-2: Using Java Interfaces

This practice covers the following topics:

- Updating the banking application to use an interface
- Using interfaces to implement accounts



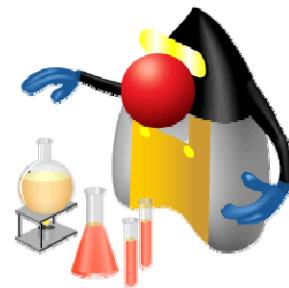
ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 6-3: Creating Lambda Expression

This practice covers the following topics:

- Performing string analysis using lambda expressions
- Practicing writing lambda expressions for the StringAnalyzer interface



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Quiz

All methods in an interface are:

- a. final
- b. abstract
- c. private
- d. volatile



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Quiz

When a developer creates an anonymous inner class, the new class is typically based on which one of the following?

- a. enums
- b. Executors
- c. Functional interfaces
- d. Static variables



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Quiz

Which is true about the parameters passed into the following lambda expression?

(t, s) -> t.contains(s)

- a. Their type is inferred from the context.
- b. Their type is executed.
- c. Their type must be explicitly defined.
- d. Their type is undetermined.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.



Generics and Collections



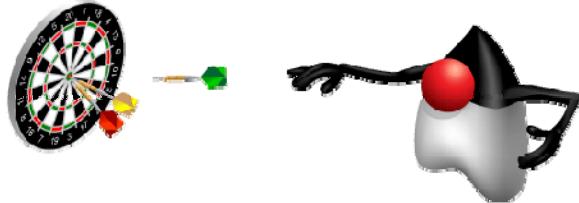
ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Objectives

After completing this lesson, you should be able to:

- Create a custom generic class
- Use the type inference diamond to create an object
- Create a collection without using generics
- Create a collection by using generics
- Implement an `ArrayList`
- Implement a `TreeSet`
- Implement a `HashMap`
- Implement a `Deque`
- Order collections



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Topics

- Generics
 - Generics with Type Inference Diamond
- Collections
 - Collection Types
 - List Interface
 - ArrayList Implementation
 - Autoboxing and Unboxing
 - Set Interface
 - Map Interface
 - Deque Interface
 - Ordering Collections
 - Comparable Interface
 - Comparator Interface



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Generics

- Provide flexible type safety to your code
- Move many common errors from run time to compile time
- Provide cleaner, easier-to-write code
- Reduce the need for casting with collections
- Are used heavily in the Java Collections API



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Simple Cache Class Without Generics

```
public class CacheString {  
    private String message;  
    public void add(String message){  
        this.message = message;  
    }  
  
    public String get(){  
        return this.message;  
    }  
}
```

```
public class CacheShirt {  
    private Shirt shirt;  
  
    public void add(Shirt shirt){  
        this.shirt = shirt;  
    }  
  
    public Shirt get(){  
        return this.shirt;  
    }  
}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The two examples in the slide show very simple caching classes. Even though each class is very simple, a separate class is required for any object type.

Generic Cache Class

```
1 public class CacheAny <T>{  
2  
3     private T t;  
4  
5     public void add(T t) {  
6         this.t = t;  
7     }  
8  
9     public T get() {  
10        return this.t;  
11    }  
12 }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

To create a generic version of the `CacheAny` class, a variable named `T` is added to the class definition surrounded by angle brackets. In this case, `T` stands for “type” and can represent any type. As the example in the slide shows, the code has changed to use `t` instead of a specific type of information. This change allows the `CacheAny` class to store any type of object.

`T` was chosen not by accident but by convention. Specific letters are commonly used with generics.

Note: You can use any identifier you want. The following values are merely strongly suggested.

Here are the conventions:

- `T`: Type
- `E`: Element
- `K`: Key
- `V`: Value
- `S, U`: Used if there are second types, third types, or more

Generics in Action

Compare the type-restricted objects to their generic alternatives.

```
1 public static void main(String args[]) {  
2     CacheString myMessage = new CacheString(); // Type  
3     CacheShirt myShirt = new CacheShirt(); // Type  
4  
5     //Generics  
6     CacheAny<String> myGenericMessage = new CacheAny<String>();  
7     CacheAny<Shirt> myGenericShirt = new CacheAny<Shirt>();  
8  
9     myMessage.add("Save this for me"); // Type  
10    myGenericMessage.add("Save this for me"); // Generic  
11  
12 }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Note how the one generic version of the class can replace any number of type-specific caching classes. The `add()` and `get()` functions work exactly the same way. In fact, if the `myMessage` declaration is changed to generic, no changes need to be made to the remaining code.

The example code can be found in the Generics project in the `TestCacheAny.java` file.

Generics with Type Inference Diamond

- Syntax:
 - There is no need to repeat types on the right side of the statement.
 - Angle brackets indicate that type parameters are mirrored.
- Simplifies generic declarations
- Saves typing

```
//Generics  
CacheAny<String> myMessage = new CacheAny<>();  
}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The type inference diamond is a new feature in JDK 7. In the generic code, notice how the right-side type definition is always equivalent to the left-side type definition. In JDK 7, you can use the diamond to indicate that the right type definition is equivalent to the left. This helps to avoid typing redundant information over and over again.

Example: TestCacheAnyDiamond.java

Note: In a way, it works in an opposite way from a “normal” Java type assignment. For example, `Employee emp = new Manager();` makes `emp` object an instance of `Manager`.

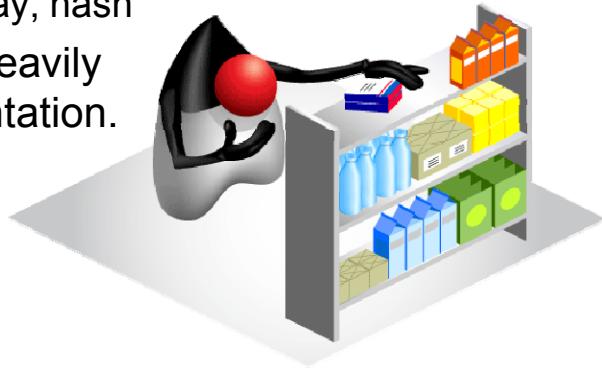
But in the case of generics:

```
ArrayList<Manager> managementTeam = new ArrayList<>();
```

The left side of the expression (rather than the right side) determines the type.

Collections

- A collection is a single object designed to manage a group of objects.
 - Objects in a collection are called *elements*.
 - *Primitives are not allowed in a collection.*
- Various collection types implement many common data structures:
 - Stack, queue, dynamic array, hash
- The Collections API relies heavily on generics for its implementation.



ORACLE®

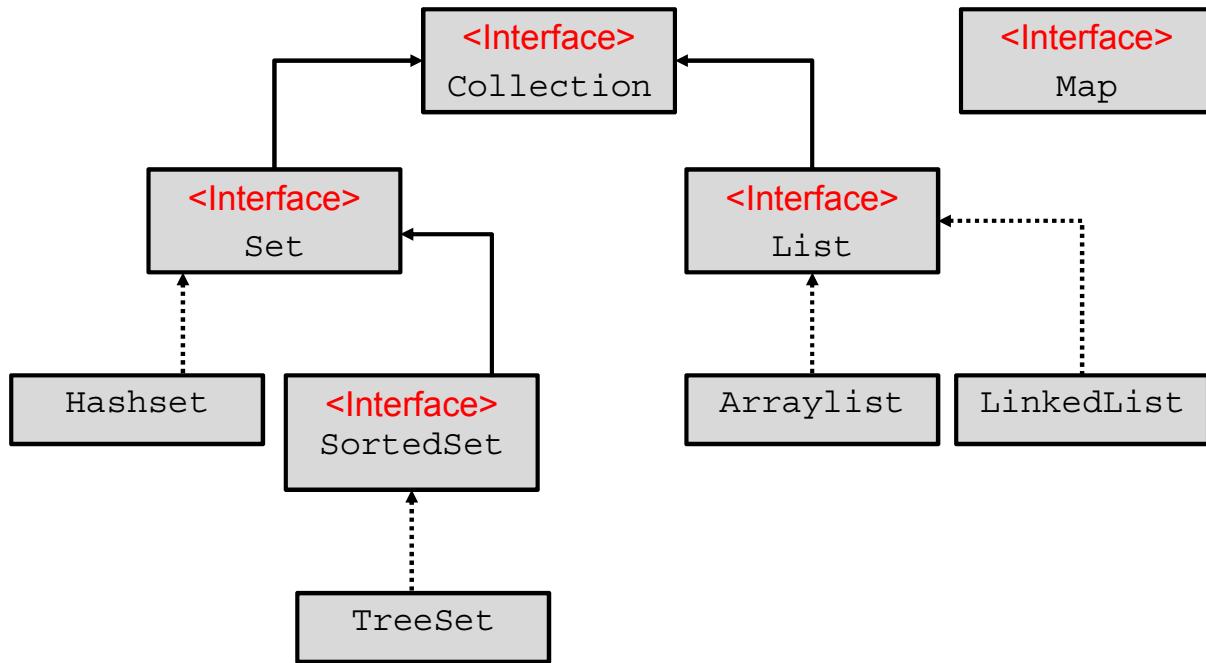
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

A collection is a single object that manages a group of objects. Objects in the collection are called *elements*. Various collection types implement standard data structures including stacks, queues, dynamic arrays, and hashes. All the collection objects have been optimized for use in Java applications.

Note: The Collections classes are all stored in the `java.util` package. The `import` statements are not shown in the following examples, but the `import` statements are required for each collection type:

- `import java.util.List;`
- `import java.util.ArrayList;`
- `import java.util.Map;`

Collection Types



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The diagram in the slide shows the Collection framework. The framework is made up of a set of interfaces for working with a group (collection) of objects.

Characteristics of the Collection Framework

List, Set, and Map are interfaces in Java and many concrete implementations of them are available in the Collections API.

Note: The Map interface is a separate inheritance tree and is discussed later in the lesson.

Collection Interfaces and Implementation

Interface	Implementation		
List	ArrayList	LinkedList	
Set	TreeSet	HashSet	LinkedHashSet
Map	HashMap	HashTable	TreeMap
Deque	ArrayDeque		



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The table in the slide shows the commonly used interfaces and their popular implementation.

List Interface

- List defines generic list behavior.
 - Is an ordered collection of elements
- List behaviors include:
 - Adding elements at a specific index
 - Getting an element based on an index
 - Removing an element based on an index
 - Overwriting an element based on an index
 - Getting the size of the list
- List allows duplicate elements.



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

ArrayList

- Is an implementation of the List interface
 - The list automatically grows if elements exceed initial size.
- Has a numeric index
 - Elements are accessed by index.
 - Elements can be inserted based on index.
 - Elements can be overwritten.
- Allows duplicate items

```
List<Integer> partList = new ArrayList<>(3);  
    partList.add(new Integer(1111));  
    partList.add(new Integer(2222));  
    partList.add(new Integer(3333));  
    partList.add(new Integer(4444)); // ArrayList auto grows  
    System.out.println("First Part: " + partList.get(0)); //  
First item  
    partList.add(0, new Integer(5555)); // Insert an item by  
index
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Autoboxing and Unboxing

- Simplifies syntax
- Produces cleaner, easier-to-read code

```
1 public class AutoBox {  
2     public static void main(String[] args) {  
3         Integer intObject = new Integer(1);  
4         int intPrimitive = 2;  
5  
6         Integer tempInteger;  
7         int tempPrimitive;  
8  
9         tempInteger = new Integer(intPrimitive);  
10        tempPrimitive = intObject.intValue();  
11  
12        tempInteger = intPrimitive; // Auto box  
13        tempPrimitive = intObject; // Auto unbox
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Lines 9 and 10 show a traditional method for moving between objects and primitives. Lines 12 and 13 show boxing and unboxing.

Autoboxing and Unboxing

Autoboxing and unboxing are Java language features that enable you to make sensible assignments without formal casting syntax. Java provides the casts for you at compile time.

Note: Be careful when using autoboxing in a loop. There is a performance cost to using this feature.

ArrayList Without Generics

```
1  public class OldStyleArrayList {  
2      public static void main(String args[]) {  
3          List partList = new ArrayList(3);  
4  
5          partList.add(new Integer(1111));  
6          partList.add(new Integer(2222));  
7          partList.add(new Integer(3333));  
8          partList.add("Oops a string!");  
9  
10         Iterator elements = partList.iterator();  
11         while (elements.hasNext()) {  
12             Integer partNumberObject = (Integer)elements.next(); // error!  
13             int partNumber = partNumberObject.intValue();  
14  
15             System.out.println("Part number: " + partNumber);  
16         }  
17     }  
18 }
```

Java example using syntax prior to Java 1.5

Runtime error:
ClassCastException



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In the example in the slide, a part number list is created by using an `ArrayList`. There is no type definition when using syntax prior to Java version 1.5. So any type can be added to the list as shown on line 8. It is up to the programmer to know what objects are in the list and in what order. If the list was only for `Integer` objects, a runtime error would occur on line 12.

On lines 10–16, with a nongeneric collection, an `Iterator` is used to iterate through the list of items. Notice that a lot of casting is required to get the objects back out of the list so you can print the data.

In the end, there is a lot of needless “syntactic sugar” (extra code) working with collections in this way.

If the line that adds the `String` to the `ArrayList` is commented out, the program produces the following output:

```
Part number: 1111  
Part number: 2222  
Part number: 3333
```

Generic ArrayList

```
1  public class GenericArrayList {  
2      public static void main(String args[]) {  
3          List<Integer> partList = new ArrayList<>(3);  
4  
5          partList.add(new Integer(1111));  
6          partList.add(new Integer(2222));  
7          partList.add(new Integer(3333));  
8          partList.add("Bad Data"); // compiler error now  
9  
10         Iterator<Integer> elements = partList.iterator();  
11         while (elements.hasNext()) {  
12             Integer partNumberObject = elements.next();  
13             int partNumber = partNumberObject.intValue();  
14  
15             System.out.println("Part number: " + partNumber);  
16         }  
17     }  
18 }
```

No cast required.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

With generics, things are much simpler. When the `ArrayList` is initialized on line 3, any attempt to add an invalid value (line 8) results in a compile-time error.

Note: On line 3, the `ArrayList` is assigned to a `List` type. Using this style enables you to swap out the `List` implementation without changing other code.

Generic ArrayList: Iteration and Boxing

```
for (Integer partNumberObj:partList) {  
    int partNumber = partNumberObj; // Demos auto unboxing  
    System.out.println("Part number: " + partNumber);  
}
```

- The enhanced `for` loop, or `for-each` loop, provides cleaner code.
- No casting is done because of autoboxing and unboxing.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Using the `for-each` loop is much easier and provides much cleaner code. No casts are done because of the autounboxing feature of Java.

Set Interface

- A Set is an interface that contains only unique elements.
- A Set has no index.
- Duplicate elements are not allowed.
- You can iterate through elements to access them.
- TreeSet provides sorted implementation.



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The main difference between List and Set in Java is that List is an ordered collection, which allows duplicates, whereas Set is an unordered collection, which does not allow duplicates.

TreeSet: Implementation of Set

```
1 public class SetExample {  
2     public static void main(String[] args) {  
3         Set<String> set = new TreeSet<>();  
4  
5         set.add("one");  
6         set.add("two");  
7         set.add("three");  
8         set.add("three"); // not added, only unique  
9  
10        for (String item:set){  
11            System.out.println("Item: " + item);  
12        }  
13    }  
14 }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The example in the slide uses a `TreeSet`, which sorts the items in the `set`. If the program is run, the output is as follows:

```
Item: one  
Item: three  
Item: two
```

Map Interface

- A collection that stores multiple key-value pairs
 - Key: Unique identifier for each element in a collection
 - Value: A value stored in the element associated with the key
- Called “associative arrays” in other languages

Key	Value
101	Blue Shirt
102	Black Shirt
103	Gray Shirt

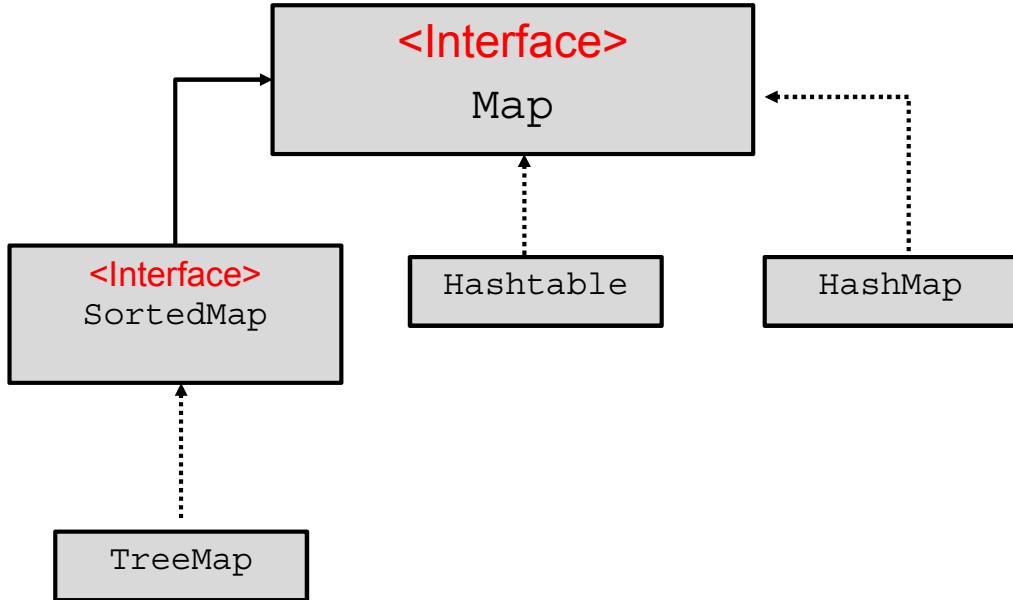


ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

A Map is good for tracking things such as part lists and their descriptions (as shown in the slide).

Map Types



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The `Map` interface does not extend the `Collection` interface because it represents mappings and not a collection of objects. Some of the key implementation classes include:

- `TreeMap`: A map where the keys are automatically sorted
- `Hashtable`: A classic associative array implementation with keys and values. `Hashtable` is synchronized.
- `HashMap`: An implementation just like `Hashtable` except that it accepts null keys and values. Also, it is not synchronized.

TreeMap: Implementation of Map

```
1 public class MapExample {  
2     public static void main(String[] args){  
3         Map <String, String> partList = new TreeMap<>();  
4         partList.put("S001", "Blue Polo Shirt");  
5         partList.put("S002", "Black Polo Shirt");  
6         partList.put("H001", "Duke Hat");  
7  
8         partList.put("S002", "Black T-Shirt"); // Overwrite value  
9         Set<String> keys = partList.keySet();  
10  
11         System.out.println("==== Part List ====");  
12         for (String key:keys) {  
13             System.out.println("Part#: " + key + " " +  
14                             partList.get(key));  
15         }  
16     }  
17 }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

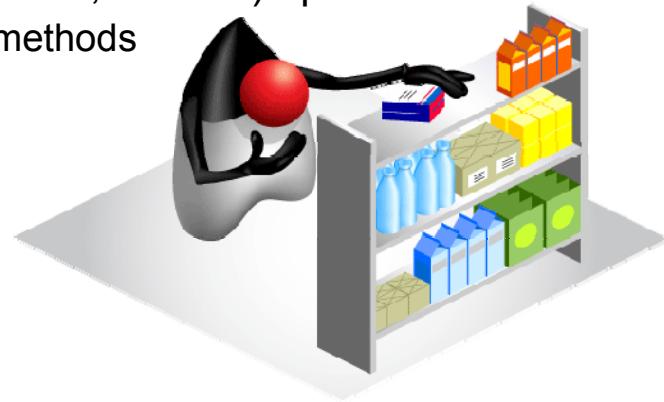
The example in the slide shows how to create a Map and perform standard operations on it.
The output from the program is:

```
==== Part List ====  
Part#: H002 Duke Hat  
Part#: S001 Blue Polo Shirt  
Part#: S002 Black T-Shirt
```

Deque Interface

A collection that can be used as a stack or a queue

- It means a “double-ended queue” (and is pronounced “deck”).
- A queue provides FIFO (first in, first out) operations:
 - `add(e)` and `remove()` methods
- A stack provides LIFO (last in, first out) operations:
 - `push(e)` and `pop()` methods



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Deque is a child interface of Collection (just like Set and List).

A queue is often used to track asynchronous message requests so they can be processed in order. A stack can be very useful for traversing a directory tree or similar structures.

A Deque is a “doubled-ended queue.” Essentially this means that a Deque can be used as a queue (first in, first out [FIFO] operations) or as a stack (last in, first out [LIFO] operations).

Stack with Deque: Example

```
1 public class TestStack {  
2     public static void main(String[] args){  
3         Deque<String> stack = new ArrayDeque<>();  
4         stack.push("one");  
5         stack.push("two");  
6         stack.push("three");  
7  
8         int size = stack.size() - 1;  
9         while (size >= 0 ) {  
10             System.out.println(stack.pop());  
11             size--;  
12         }  
13     }  
14 }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Ordering Collections

- The Comparable and Comparator interfaces are used to sort collections.
 - Both are implemented by using generics.
- Using the Comparable interface:
 - Overrides the `compareTo` method
 - Provides only one sort option
- The Comparator interface:
 - Is implemented by using the `compare` method
 - Enables you to create multiple Comparator classes
 - Enables you to create and use numerous sorting options



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The Collections API provides two interfaces for ordering elements: Comparable and Comparator.

- Comparable: Is implemented in a class and provides a single sorting option for the class
- Comparator: Enables you to create multiple sorting options. You plug in the designed option whenever you want

Both interfaces can be used with sorted collections, such as TreeSet and TreeMap.

Comparable: Example

```
1 public class ComparableStudent implements Comparable<ComparableStudent>{
2     private String name; private long id = 0; private double gpa = 0.0;
3
4     public ComparableStudent(String name, long id, double gpa) {
5         // Additional code here
6     }
7     public String getName(){ return this.name; }
8         // Additional code here
9
10    public int compareTo(ComparableStudent s){
11        int result = this.name.compareTo(s.getName());
12        if (result > 0) { return 1; }
13        else if (result < 0){ return -1; }
14        else { return 0; }
15    }
16 }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The example in the slide implements the `Comparable` interface and its `compareTo` method. Notice that because the interface is designed by using generics, the angle brackets define the class type that is passed into the `compareTo` method. The `if` statements are included to demonstrate the comparisons that take place. You can also merely return a result.

The returned numbers have the following meaning.

- **Negative number:** `s` comes before the current element.
- **Positive number:** `s` comes after the current element.
- **Zero:** `s` is equal to the current element.

In cases where the collection contains equivalent values, replace the code that returns zero with additional code that returns a negative or positive number.

Comparable Test: Example

```
public class TestComparable {  
    public static void main(String[] args){  
        Set<ComparableStudent> studentList = new TreeSet<>();  
  
        studentList.add(new ComparableStudent("Thomas Jefferson", 1111, 3.8));  
        studentList.add(new ComparableStudent("John Adams", 2222, 3.9));  
        studentList.add(new ComparableStudent("George Washington", 3333, 3.4));  
  
        for(ComparableStudent student:studentList){  
            System.out.println(student);  
        }  
    }  
}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In the example in the slide, an `ArrayList` of `ComparableStudent` elements is created. After the list is initialized, it is sorted by using the `Comparable` interface. The output of the program is as follows:

Name: George Washington ID: 3333 GPA:3.4

Name: John Adams ID: 2222 GPA:3.9

Name: Thomas Jefferson ID: 1111 GPA:3.8

Note: The `ComparableStudent` class has overridden the `toString()` method.

Comparator Interface

- Is implemented by using the `compare` method
- Enables you to create multiple Comparator classes
- Enables you to create and use numerous sorting options



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The example in the next slide shows how to use `Comparator` with an unsorted interface such as `ArrayList` by using the `Collections` utility class.

Comparator: Example

```
public class StudentSortName implements Comparator<Student>{  
    public int compare(Student s1, Student s2){  
        int result = s1.getName().compareTo(s2.getName());  
        if (result != 0) { return result; }  
        else {  
            return 0; // Or do more comparing  
        }  
    }  
}
```

```
public class StudentSortGpa implements Comparator<Student>{  
    public int compare(Student s1, Student s2){  
        if (s1.getGpa() < s2.getGpa()) { return 1; }  
        else if (s1.getGpa() > s2.getGpa()) { return -1; }  
        else { return 0; }  
    }  
}
```

Here the compare logic is reversed and results in descending order.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The example in the slide shows the `Comparator` classes that are created to sort based on Name and GPA. For the name comparison, the `if` statements have been simplified.

Comparator Test: Example

```
1 public class TestComparator {
2     public static void main(String[] args){
3         List<Student> studentList = new ArrayList<>(3);
4         Comparator<Student> sortName = new StudentSortName();
5         Comparator<Student> sortGpa = new StudentSortGpa();
6
7         // Initialize list here
8
9         Collections.sort(studentList, sortName);
10        for(Student student:studentList){
11            System.out.println(student);
12        }
13
14        Collections.sort(studentList, sortGpa);
15        for(Student student:studentList){
16            System.out.println(student);
17        }
18    }
19 }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The example in the slide shows how the two `Comparator` objects are used with a collection.

Note: Some code has been commented out to save space.

Notice how the `Comparator` objects are initialized on lines 4 and 5. After the `sortName` and `sortGpa` variables are created, they can be passed to the `sort()` method by name.

Running the program produces the following output.

```
Name: George Washington  ID: 3333  GPA:3.4
Name: John Adams      ID: 2222  GPA:3.9
Name: Thomas Jefferson  ID: 1111  GPA:3.8
Name: John Adams      ID: 2222  GPA:3.9
Name: Thomas Jefferson  ID: 1111  GPA:3.8
Name: George Washington  ID: 3333  GPA:3.4
```

Notes:

- The `Collections` utility class provides a number of useful methods for various collections. Methods include `min()`, `max()`, `copy()`, and `sort()`.
- The `Student` class has overridden the `toString()` method.

Summary

In this lesson, you should have learned how to:

- Create a custom generic class
- Use the type inference diamond to create an object
- Create a collection without using generics
- Create a collection by using generics
- Implement an `ArrayList`
- Implement a `Set`
- Implement a `HashMap`
- Implement a `Deque`
- Order collections



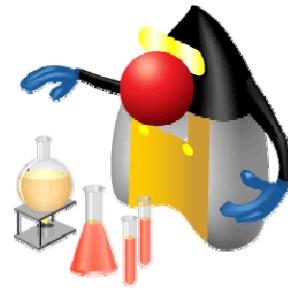
ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 7-1 Overview: Counting Part Numbers by Using a HashMap

This practice covers the following topics:

- Creating a map to store a part number and count
- Creating a map to store a part number and description
- Processing the list of parts and producing a report

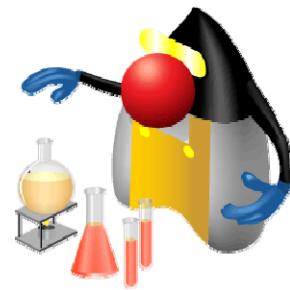


ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 7-2 Overview: Implementing Stack by Using a Deque Object

This practice covers using the Deque object to implement a stack.



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Quiz

Which of the following is *not* a conventional abbreviation for use with generics?

- a. T: Table
- b. E: Element
- c. K: Key
- d. V: Value



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Quiz

Which interface would you use to create multiple sort options for a collection?

- a. Comparable
- b. Comparison
- c. Comparator
- d. Comparinator



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

8

Collections, Streams, and Filters

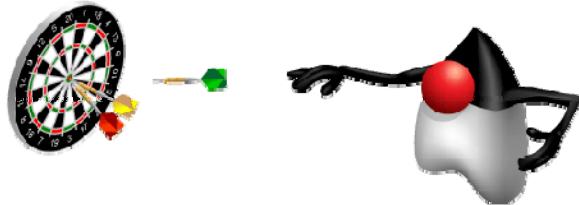
ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Objectives

After completing this lesson, you should be able to:

- Describe the Builder pattern
- Iterate through a collection by using lambda syntax
- Describe the Stream interface
- Filter a collection by using lambda expressions
- Call an existing method by using a method reference
- Chain multiple methods
- Define pipelines in terms of lambdas and collections



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Collections, Streams, and Filters

- Iterate through collections using forEach
- Streams and Filters



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The Person Class

- Person class
 - Attributes like name, age, address, etc.
- Class created by using the Builder pattern
 - Generates a collection persons for examples
- RoboCall Example
 - An app for contacting people via mail, phone, email
 - Given a list of people query for certain groups
 - Used for test and demo
- Groups queried for
 - Drivers: Persons over the age of 16
 - Draftees: Male persons between 18 and 25 years old
 - Pilots: Persons between 23 and 65 years old



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The RoboCall App

The RoboCall app is an application for automating the communication with groups of people. It can contact individuals by phone, email, or regular mail. In this example, the app will be used to contact three groups of people.

Drivers: Persons over the age of 16

Draftees: Male persons between the ages of 18 and 25

Pilots (specifically commercial pilots): Persons between the ages of 23 and 65

Person

The Person class creates the master list of persons you want to contact. The class uses the builder pattern to create new objects. An `ArrayList` of `Person` objects is used for the examples that follow.

Person Properties

- A Person has the following properties:

```
9 public class Person {  
10    private String givenName;  
11    private String surName;  
12    private int age;  
13    private Gender gender;  
14    private String eMail;  
15    private String phone;  
16    private String address;  
17    private String city;  
18    private String state;  
19    private String code;
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

These fields are used as criteria in the search examples that follow.

Builder Pattern

- Allows object creation by using method chaining
 - Easier-to-read code
 - More flexible object creation
 - Object returns itself
 - A fluent approach
- Example

```
260     people.add(  
261         new Person.Builder()  
262             .givenName("Betty")  
263             .surName("Jones")  
264             .age(85)  
265             .gender(Gender.FEMALE)  
266             .email("betty.jones@example.com")  
267             .phoneNumber("211-33-1234")  
272             .build()  
273     );
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The builder pattern has been used in all our collection objects. The pattern uses method chaining to provide an easy way to create objects. Working with lambda expressions and streams is similar to the builder pattern.

Collection Iteration and Lambdas

- RoboCall06 Iterating with `forEach`

```
9 public class RoboCallTest06 {  
10  
11     public static void main(String[] args) {  
12  
13         List<Person> pl = Person.createShortList();  
14  
15         System.out.println("\n==== Print List ====");  
16         pl.forEach(p -> System.out.println(p));  
17  
18     }  
19 }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The `forEach` method has been added to all collections. This makes iteration much easier and provides a number of benefits. This example merely prints all the `Person` instances in the list.

The `Collection` interface extends the `Iterable` interface. The `Iterable` interface defines the `forEach` method.

RoboCallTest07: Stream and Filter

```
10 public class RoboCallTest07 {  
11  
12     public static void main(String[] args) {  
13  
14         List<Person> pl = Person.createShortList();  
15         RoboCall05 robo = new RoboCall05();  
16  
17         System.out.println("\n==== Calling all Drivers Lambda  
18         ===");
19         pl.stream()
20             .filter(p -> p.getAge() >= 23 && p.getAge() <= 65)
21             .forEach(p -> robo.roboCall(p));
22     }
23 }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Adding the `stream()` method to the statement opens up a whole host of new operations on collections. The `filter()` method, shown in the slide, is an example. It takes a `Predicate` as a parameter, and filters the result so that only collection elements that match the `Predicate` criteria are returned to `forEach`.

This is a big improvement on looping from before. First, the collection statement with a stream really describes what is happening (take this collection, filter out these elements, and return the results.) Second, the looping methods in `RoboCall05` are no longer needed. The selection of elements and their output are handled in one statement.

RobocallTest08: Stream and Filter Again

```
10 public class RoboCallTest08 {  
11  
12     public static void main(String[] args) {  
13  
14         List<Person> pl = Person.createShortList();  
15         RoboCall05 robo = new RoboCall05();  
16  
17         // Predicates  
18         Predicate<Person> allPilots =  
19             p -> p.getAge() >= 23 && p.getAge() <= 65;  
20  
21         System.out.println("\n==== Calling all Drivers Variable  
22         ===");  
22         pl.stream().filter(allPilots)  
23             .forEach(p -> robo.roboCall(p));  
24     }  
}
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

This once again shows that a lambda expression can be stored in a variable used or reused later. The resulting statements on lines 22 and 23 read very clearly.

SalesTxn Class

- Class used in examples and practices to follow
- Stores information about sales transactions
 - Seller and buyer
 - Product quantity and price
- Implemented with a Builder class
- Buyer class
 - Simple class to represent buyers and their volume discount level
- Helper enums
 - BuyerClass: Defines volume discount levels
 - State: Lists the states where transactions take place
 - TaxRate: Lists the sales tax rates for different states



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The SalesTxn class is an additional data set used in the course. Each object includes data about a sales transaction. Some of the data included for each transaction includes:

- The name of the sales person who sold the product and the buyer
- The transaction date
- The number of units sold
- The price per unit
- Sales tax rates
- Any discounts applied to the transaction

Java Streams

- Streams
 - `java.util.stream`
 - A sequence of elements on which various methods can be chained
- Method chaining
 - Multiple methods can be called in one statement
- Stream characteristics
 - They are immutable.
 - After the elements are consumed, they are no longer available from the stream.
 - A chain of operations can occur only once on a particular stream (a pipeline).
 - They can be serial (default) or parallel.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Streams are a new type of object added to Java SE 8. A stream is a sequence of elements supporting sequential and parallel aggregate operations. These operations can be called consecutively in a single statement. This feature is called “method chaining.”

Collections and streams, while being similar, have different goals. Collections are concerned with the efficient management of and access to their elements. By contrast, streams do not provide a means to directly access or manipulate their elements. Instead, streams are concerned with declaratively describing their source and the computational operations that will be performed in aggregate on that source.

The Filter Method

- The Stream class converts collection to a pipeline
 - Immutable data
 - Can only be used once and then tossed
- Filter method uses Predicate lambdas to select items.
- Syntax:

```
15      System.out.println("\n== CA Transations Lambda ==");  
16      tList.stream()  
17          .filter(t -> t.getState().equals("CA"))  
18          .forEach(SalesTxn::printSummary);
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The `filter` method takes a lambda expression as a parameter and filters the data based on the logical expression provided. This indicates that a `Predicate` is the target type of the filter. The elements that meet the filter criteria are passed to the `forEach` method, which does a `roboCall` on matching elements.

Method References

In some cases, the lambda expression merely calls a class method.

- .forEach(t -> t.printSummary())
- Alternatively, you can use a method reference
 - .forEach(SalesTxn::printSummary);
- You can use a method reference in the following situations:
 - Reference to a static method
 - ContainingClass::staticMethodName
 - Reference to an instance method
 - Reference to an instance method of an arbitrary object of a particular type (for example,
String::compareToIgnoreCase)
 - Reference to a constructor
 - ClassName::new



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In many situations, a method reference can be substituted for a lambda expression. If a lambda expression merely calls a method on that object, a Method Reference can be substituted.

Method Chaining

- Pipelines allow method chaining (like a builder).
- Methods include filter and many others.
- For example:

```
21      tList.stream()
22          .filter(t -> t.getState().equals("CA"))
23          .filter(t -> t.getBuyer().getName()
24              .equals("Acme Electronics"))
25          .forEach(SalesTxn::printSummary);
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Developers are not limited to only one method call. Multiple methods can be chained together in a single statement. This is called "method chaining." The statement structure looks similar to that of the builder pattern.

Analogy: If you think about it, these statements are very similar to SQL statements with `where` clauses. The syntax is different, but the idea is very similar.

Method Chaining

- You can use compound logical statements.
- You select what is best for the situation.

```
15     System.out.println("\n== CA Transactions for ACME ==");  
16     tList.stream()  
17         .filter(t -> t.getState().equals("CA") &&  
18             t.getBuyer().getName().equals("Acme Electronics"))  
19         .forEach(SalesTxn::printSummary);  
20  
21     tList.stream()  
22         .filter(t -> t.getState().equals("CA"))  
23         .filter(t -> t.getBuyer().getName()  
24             .equals("Acme Electronics"))  
25         .forEach(SalesTxn::printSummary);
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Sometimes method chaining can be an aesthetic choice over a compound logical expression.

Pipeline Defined

- A stream pipeline consists of:
 - A source
 - Zero or more intermediate operations
 - One terminal operation
- Examples
 - Source: A Collection (could be a file, a stream, and so on)
 - Intermediate: Filter, Map
 - Terminal: `forEach`



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In this lesson, streams have only been filtered and the results printed out. However, some stream methods like `map` can produce a new stream. Connecting these streams of data together in a single statement is called a stream pipeline.

A stream pipeline consists of a source (which might be an array, a collection, a generator function, an I/O channel, etc.), zero or more intermediate operations (which transform a stream into another stream, such as `filter(Predicate)`), and a terminal operation (which produces a result or side-effect, such as `count()` or `forEach(Consumer)`). Streams are lazy; computation on the source data is performed only when the terminal operation is initiated, and source elements are consumed only as needed.

Summary

After completing this lesson, you should be able to:

- Describe the Builder pattern
- Iterate through a collection by using lambda syntax
- Describe the Stream interface
- Filter a collection by using lambda expressions
- Call an existing method by using a method reference
- Chain multiple methods together
- Define pipelines in terms of lambdas and collections



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice Overview

- Practice 8-1: Update RoboCall to use Streams
- Practice 8-2: Mail Sales Executives using Method Chaining
- Practice 8-3: Mail Sales Employees over 50 using Method Chaining
- Practice 8-4: Mail Male Engineering Employees Under 65 Using Method Chaining



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Lambda Built-in Functional Interfaces

9

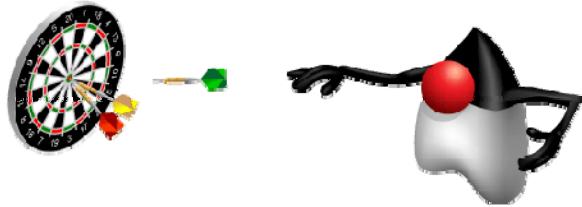
ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Objectives

After completing this lesson, you should be able to:

- List the built-in interfaces included in `java.util.function`
- Use primitive versions of base interfaces
- Use binary versions of base interfaces



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Built-in Functional Interfaces

- Lambda expressions rely on functional interfaces
 - Important to understand what an interface does
 - Concepts make using lambdas easier
- Focus on the purpose of main functional interfaces
- Become aware of many primitive variations
- Lambda expressions have properties like those of a variable
 - Use when needed
 - Can be stored and reused



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In Java 8, there are a lot of method signatures that refer to interfaces in `java.util.function`. Therefore, it is important to understand what these interfaces do and what variations on the basics exist. It makes writing lambda expressions a lot easier.

The `java.util.function` Package

- **Predicate:** An expression that returns a boolean
- **Consumer:** An expression that performs operations on an object passed as argument and has a void return type
- **Function:** Transforms a T to a U
- **Supplier:** Provides an instance of a T (such as a factory)
- **Primitive variations**
- **Binary variations**



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Predicate is not the only functional interface provided with Java. A number of standard interfaces are designed as a starter set for developers.

Example Assumptions

- The following two declarations are assumed for the examples that follow:

```
14     List<SalesTxn> tList = SalesTxn.createTxnList();  
15     SalesTxn first = tList.get(0);
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

One or both of the declarations pictured are assumed in the examples that follow.

Predicate

```
1 package java.util.function;  
2  
3 public interface Predicate<T> {  
4     public boolean test(T t);  
5 }  
6
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

A `Predicate` takes a generic class and returns a `boolean`. It has a single method, namely `test`.

Predicate: Example

```
16     Predicate<SalesTxn> massSales =
17         t -> t.getState().equals(State.MA);
18
19     System.out.println("\n== Sales - Stream");
20     tList.stream()
21         .filter(massSales)
22         .forEach(t -> t.printSummary());
23
24     System.out.println("\n== Sales - Method Call");
25     for(SalesTxn t:tList) {
26         if (massSales.test(t)){
27             t.printSummary();
28         }
29     }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In this example, a `SalesTxn` is tested to see if it was executed in the state of MA. The `filter` method takes a predicate as a parameter. In the second example (starting on line 24), notice that the predicate can call its `test` method with a `SalesTxn` as a parameter. This is a clear example of taking the generic value and returning a boolean.

Consumer

```
1 package java.util.function;  
2  
3 public interface Consumer<T> {  
4  
5     public void accept(T t);  
6  
7 }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

A `Consumer` takes a generic and returns nothing. It has a single method `accept`.

Consumer: Example

```
17 Consumer<SalesTxn> buyerConsumer = t ->
18     System.out.println("Id: " + t.getTxnId()
19             + " Buyer: " + t.getBuyer().getName());
20
21 System.out.println("== Buyers - Lambda");
22 tList.stream().forEach(buyerConsumer);
23
24 System.out.println("== First Buyer - Method");
25 buyerConsumer.accept(first);
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Note how the `Consumer` is defined and nothing is returned. The example takes a sales transaction and prints a couple values.

Two examples are provided in the slide. The first shows that the default parameter for `forEach` is `Consumer`. The second shows that once a lambda expression is stored, it can be executed on the specified type by using the `accept` method.

Function

```
1 package java.util.function;  
2  
3 public interface Function<T, R> {  
4  
5     public R apply(T t);  
6 }  
7
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

A `Function` takes one generic type and returns another. Notice that the input type comes first in the list and then the return type. So the `apply` method takes a `T` and returns an `R`.

Function: Example

```
17     Function<SalesTxn, String> buyerFunction =
18         t -> t.getBuyer().getName();
19
20     System.out.println("\n== First Buyer");
21     System.out.println(buyerFunction.apply(first));
22 }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The example takes a SalesTxn and returns a String. The Function interface is used frequently in the update Collection APIs.

Supplier

```
1 package java.util.function;  
2  
3 public interface Supplier<T> {  
4  
5     public T get();  
6 }  
7
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The `Supplier` returns a generic type and takes no parameters.

Supplier: Example

```
15     List<SalesTxn> tList = SalesTxn.createTxnList();
16     Supplier<SalesTxn> txnSupplier =
17         () -> new SalesTxn.Builder()
18             .txnId(101)
19             .salesPerson("John Adams")
20             .buyer(Buyer.getBuyerMap().get("PriceCo"))
21             .product("Widget")
22             .paymentType("Cash")
23             .unitPrice(20)
24     //... Lines ommited
25         .build();
26
27
28
29     tList.add(txnSupplier.get());
30
31     System.out.println("\n== TList");
32
33     tList.stream().forEach(SalesTxn::printSummary);
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In the example, the `Supplier` creates a new `SalesTxn`. On line 31, notice that calling `get` generates a `SalesTxn` from the lambda that was defined earlier.

Primitive Interface

- Primitive versions of all main interfaces
 - Will see these a lot in method calls
- Return a primitive
 - Example: `ToDoubleFunction`
- Consume a primitive
 - Example: `DoubleFunction`
- Why have these?
 - Avoids auto-boxing and unboxing



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

If you look at the API docs, there are a number of primitive interfaces that mirror the main types: `Predicate`, `Consumer`, `Function`, `Supplier`. These are provided to avoid the negative performance consequences of auto-boxing and unboxing.

Return a Primitive Type

```
1 package java.util.function;  
2  
3 public interface ToDoubleFunction<T> {  
4  
5     public double applyAsDouble(T t);  
6 }  
7
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The ToDoubleFunction interface takes a generic type and returns a double.

Return a Primitive Type: Example

```
18     ToDoubleFunction<SalesTxn> discountFunction =
19         t -> t.getTransactionTotal()
20             * t.getDiscountRate();
21
22     System.out.println("\n== Discount");
23     System.out.println(
24         discountFunction.applyAsDouble(first));
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

This example calculates a value from a transaction and returns a `double`. Notice that the method name changes a little, but this is still a `Function`. Pass in one type and return something else, in this case a `double`.

Process a Primitive Type

```
1 package java.util.function;  
2  
3 public interface DoubleFunction<R> {  
4  
5     public R apply(double value);  
6 }  
7
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Notice that a `DoubleFunction` specifies only one generic type, but a `Function` takes two. The `apply` method takes a `double` and returns the generic type. So the `double`, in this case, is the input and the generic type is the output.

Process Primitive Type: Example

```
9     A06DoubleFunction test = new A06DoubleFunction();
10
11    DoubleFunction<String> calc =
12        t -> String.valueOf(t * 3);
13
14    String result = calc.apply(20);
15    System.out.println("New value is: " + result);
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The example computes a value and then returns the result as a `String`. The value is passed in on line 14.

Binary Types

```
1 package java.util.function;  
2  
3 public interface BiPredicate<T, U> {  
4  
5     public boolean test(T t, U u);  
6 }  
7
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The binary version of the standard interfaces allows two generic types as input. In this example, the `BiPredicate` takes two parameters and returns a `boolean`.

Binary Type: Example

```
14     List<SalesTxn> tList = SalesTxn.createTxnList();
15     SalesTxn first = tList.get(0);
16     String testState = "CA";
17
18     BiPredicate<SalesTxn, String> stateBiPred =
19         (t, s) -> t.getState().getStr().equals(s);
20
21     System.out.println("\n== First is CA?");
22     System.out.println(
23         stateBiPred.test(first, testState));
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

This example takes a `SalesTxn` and a `String` to do a comparison and return a result. The `test` method merely takes two parameters instead of one.

Unary Operator

```
1 package java.util.function;  
2  
3 public interface UnaryOperator<T> extends  
Function<T,T> {  
4     @Override  
5     public T apply(T t);  
6 }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The `UnaryOperator` takes a class as input and returns an object of the same class.

UnaryOperator: Example

- If you need to pass in something and return the same type, use the UnaryOperator interface.

```
17     UnaryOperator<String> unaryStr =
18         s -> s.toUpperCase();
19
20     System.out.println("== Upper Buyer");
21     System.out.println(
22         unaryStr.apply(first.getBuyer().getName()));
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The `UnaryOperator` interface takes a generic type and returns that same type. This example takes a `String` and returns the `String` in uppercase.

Wildcard Generics Review

- Wildcards for generics are used extensively.
- ? super T
 - This class and any of its super types
- ? extends T
 - This class and any of its subtypes



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

When using the built-in functional interfaces, generic wildcard statements are used frequently. The two most common wildcards you will see are listed in the slide.

Summary

After completing this lesson, you should be able to:

- List the built-in interfaces included in `java.util.function`
- Use primitive versions of base interfaces
- Use binary versions of base interfaces



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice Overview

- Practice 9-1: Create Consumer Lambda Expression
- Practice 9-2: Create a Function Lambda Expression
- Practice 9-3: Create a Supplier Lambda Expression
- Practice 9-4: Create a BiPredicate Lambda Expression



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

10

Lambda Operations

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Objectives

After completing this lesson, you should be able to:

- Extract data from an object by using map
- Describe the types of stream operations
- Describe the Optional class
- Describe lazy processing
- Sort a stream
- Save results to a collection by using the `collect` method
- Group and partition data by using the `Collectors` class



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Streams API

- Streams
 - `java.util.stream`
 - A sequence of elements on which various methods can be chained
- The Stream class converts collection to a pipeline.
 - Immutable data
 - Can only be used once
 - Method chaining
- Java API doc *is your friend*
- Classes
 - `DoubleStream`, `IntStream`, `LongStream`



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

A stream pipeline consists of a source, zero or more intermediate operations (which transform a stream into another stream), and a terminal operation that ends the use of a stream.

To perform a computation, stream operations are composed into a stream pipeline.

A stream pipeline consists of:

- A source: An array, a collection, a generator function, an I/O channel
- Zero or more intermediate operations, which transform a stream into another stream
 - `filter(Predicate)`
 - `map(Function)`
 - `mapToInt(IntFunction)`
 - `mapToLong(ToLongFunction)`
 - `mapToDouble(DoubleFunction)`
 - `distinct()`
 - `sorted()`
 - `skip(Long)`
 - `limit(Long)`
- A terminal operation, which produces a result or side effect
 - `count()`
 - `reduce(BinaryOperator)`
 - `collect(Collector)`
 - `forEach(Consumer)`

Streams may be lazy. Computation on the source data is performed only when the terminal operation is initiated, and source elements are consumed only as needed.

Types of Operations

- Intermediate
 - filter() map() peek()
- Terminal
 - forEach() count() sum() average() min()
max() collect()
- Terminal short-circuit
 - findFirst() findAny() anyMatch()
allMatch() noneMatch()



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The above is a list of stream methods by their operation type.

Extracting Data with Map

```
map(Function<? super T,? extends R> mapper)
```

- A map takes one Function as an argument.
 - A Function takes one generic and returns something else.
- Primitive versions of map
 - mapToInt () mapToLong () mapToDouble ()



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The map method is typically used to extract data from a field and perform a calculation or operation. The results of the mapping operation are returned as a stream.

Taking a Peek

`peek(Consumer<? super T> action)`

- The peek method performs the operation specified by the lambda expression and returns the elements to the stream.
- Great for printing intermediate results



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The `peek` method of the `Stream` class allows you to look at element data in the stream. After `peek` is called, all elements in the current stream are returned to the stream and are available to the next stream in the pipeline.

Caution: With the `peek` method, you can change element data in the stream. Any changes will be made to the underlying collection. However, this would not be a best practice as the data would not be accessed in a thread-safe manner. Manipulating the data in this way is strongly discouraged.

Search Methods: Overview

- `findFirst()`
 - Returns the first element that meets the specified criteria
- `allMatch()`
 - Returns `true` if all the elements meet the criteria
- `noneMatch()`
 - Returns `true` if none of the elements meet the criteria
- All of the above are short-circuit terminal operations.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The `findFirst` method of the `Stream` class finds the first element in the stream specified by the filters in the pipeline. The `findFirst` method is a terminal short-circuit operation. This means intermediate operations are performed in a lazy manner, resulting in more efficient processing of the data in the stream. A terminal operation ends the processing of a pipeline.

The `allMatch` method returns whether all elements of this stream match the provided predicate. The method may not evaluate the predicate on all elements if not necessary for determining the result. If the stream is empty, `true` is returned and the predicate is not evaluated.

The `noneMatch` method returns whether no elements of this stream match the provided predicate. May not evaluate the predicate on all elements if not necessary for determining the result. If the stream is empty, `true` is returned and the predicate is not evaluated.

Search Methods

- Nondeterministic search methods
 - Used for nondeterministic cases. In effect, situations where parallel is more effective.
 - Results may vary between invocations.
- `findAny()`
 - Returns the first element found that meets the specified criteria
 - Results may vary when performed in parallel.
- `anyMatch()`
 - Returns true if any elements meet the criteria
 - Results may vary when performed in parallel.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The `findAny` method returns an `Optional<T>` describing some element of the stream, or an empty `Optional<T>` if the stream is empty. The behavior of this operation is explicitly nondeterministic; it is free to select any element in the stream. This is to allow for maximal performance in parallel operations; the cost is that multiple invocations on the same source may not return the same result. (If a stable result is desired, use `findFirst()` instead.) This is a short-circuiting terminal operation.

The `anyMatch` method returns whether any elements of this stream match the provided predicate. The method may not evaluate the predicate on all elements if it is not necessary for determining the result. If the stream is empty, `false` is returned and the predicate is not evaluated. This is a short-circuiting terminal operation.

Optional Class

- `Optional<T>`
 - A container object that may or may not contain a non-null value
 - If a value is present, `isPresent()` returns true.
 - `get()` returns the value.
 - Found in `java.util`.
- **Optional primitives**
 - `OptionalDouble` `OptionalInt` `OptionalLong`



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

An `Optional<T>` is a container object that may or may not contain a non-null value. If a value is present, `isPresent()` returns `true` and `get()` returns the value. There are a number of additional methods that can be used with this class. See the API documentation for more details.

Lazy Operations

- Lazy operations:
 - Can be optimized
 - Perform only required operations

```
== First CO Bonus ==  
Stream start  
Executives  
CO Executives  
    Bonus paid: $7,200.00  
Stream start  
Executives  
CO Executives  
    Bonus paid: $6,600.00  
Stream start  
Executives  
CO Executives  
    Bonus paid: $8,400.00
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The slide shows two lists of operations on a list of Employees. The list on the right must go through all the employee elements as it uses the `forEach` terminal operation. The list on the left uses the `findFirst` method and thus can use lazy operations. As soon as the first element is found, the iteration stops.

Stream Data Methods

`count()`

- Returns the count of elements in this stream

`max(Comparator<? super T> comparator)`

- Returns the maximum element of this stream according to the provided Comparator

`min(Comparator<? super T> comparator)`

- Returns the minimum element of this stream according to the provided Comparator



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The `count` method returns the number of elements in the current stream. This is a terminal operation.

The `max` method returns the highest matching value given a `Comparator` to rank elements. The `max` method is a terminal operation.

The `min` method returns the lowest matching value given a `Comparator` to rank elements. The `min` method is a terminal operation.

Performing Calculations

`average()`

- Returns an optional describing the arithmetic mean of elements of this stream
- Returns an empty optional if this stream is empty
- Type returned depends on primitive class.

`sum()`

- Returns the sum of elements in this stream
- Methods are found in primitive streams:
 - DoubleStream, IntStream, LongStream



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The `average` method returns the average of a list of values passed from a stream. The `average` method is a terminal operation.

The `sum` method calculates a sum based on the stream passed to it. Notice that the `mapToDouble` method is called before the stream is passed to `sum`. If you look at the `Stream` class, no `sum` method is included. Instead, a `sum` method is included in the primitive version of the `Stream` class, `IntStream`, `DoubleStream`, and `LongStream`. The `sum` method is a terminal operation.

Sorting

`sorted()`

- Returns a stream consisting of the elements sorted according to natural order

`sorted(Comparator<? super T> comparator)`

- Returns a stream consisting of the elements sorted according to the Comparator



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The `sorted` method can be used to sort stream elements based on their natural order. This is an intermediate operation.

Comparator Updates

```
comparing(Function<? super T,? extends U> keyExtractor)
```

- Allows you to specify any field to sort on based on a method reference or lambda
- Primitive versions of the Function also supported

```
thenComparing(Comparator<? super T> other)
```

- Specify additional fields for sorting.

```
reversed()
```

- Reverse the sort order by appending to the method chain.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The sorted method can also take a Comparator as a parameter. Combined with the comparing method, the Comparator class provides a great deal of flexibility when sorting a stream.

The thenComparing method can be added to the comparing method to do a multilevel sort on the elements in the stream. The thenComparing method takes a Comparator as a parameter just like the comparing method.

The reversed method can be appended to a pipeline, thus reversing the sort order of the elements in the stream.

Saving Data from a Stream

```
collect(Collector<? super T,A,R> collector)
```

- Allows you to save the result of a stream to a new data structure
- Relies on the `Collectors` class
- Examples
 - `stream().collect(Collectors.toList());`
 - `stream().collect(Collectors.toMap());`



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The `collect` method allows you to save the results of all the filtering, mapping, and sorting that takes place in a pipeline. Notice how the `collect` method is called. It takes a `Collectors` class as a parameter. The `Collectors` class provides a number of ways to return the elements left in a pipeline.

Collectors Class

- `averagingDouble(ToDoubleFunction<? super T> mapper)`
 - Produces the arithmetic mean of a double-valued function applied to the input elements
- `groupingBy(Function<? super T, ? extends K> classifier)`
 - A "group by" operation on input elements of type T, grouping elements according to a classification function, and returning the results in a map
- `joining()`
 - Concatenates the input elements into a String, in encounter order
- `partitioningBy(Predicate<? super T> predicate)`
 - Partitions the input elements according to a Predicate

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The `groupingBy` method of the `Collectors` class allows you to generate a `Map` based on the elements contained in a stream. The keys are based off of a selected field in a class. Matching objects are placed into an `ArrayList` that becomes the value for the key.

The `joining` method of the `Collectors` class allows you to join together elements returned from a stream.

The `partitioningBy` method offers an interesting way to create a `Map`. The method takes a `Predicate` as an argument and creates a `Map` with two boolean keys. One key is `true` and includes all the elements that met the true criteria of the `Predicate`. The other key, `false`, contains all the elements that resulted in `false` values as determined by the `Predicate`.

Quick Streams with Stream.of

- The Stream.of method allows you to easily create a stream.

```
11 public static void main(String[] args) {  
12  
13     Stream.of("Monday", "Tuesday", "Wednesday", "Thursday")  
14         .filter(s -> s.startsWith("T"))  
15         .forEach(s -> System.out.println("Matching Days: " +  
16             s));  
16 }
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The Stream.of method can be used to create a stream out of an array of elements. The elements can be listed as shown in the slide or from the results of method calls.

Flatten Data with flatMap

- Use the flatMap method to flatten data in a stream.

```
17     Path file = new File("tempest.txt").toPath();
18
19     try{
20
21         long matches = Files.lines(file)
22             .flatMap(line -> Stream.of(line.split(" ")))
23             .filter(word -> word.contains("my"))
24             .peek(s -> System.out.println("Match: " + s))
25             .count();
26
27     System.out.println("# of Matches: " + matches);
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The flatMap method can be used to convert data into a stream. The example splits line data into words.

Summary

After completing this lesson, you should be able to:

- Extract data from an object using map
- Describe the types of stream operations
- Describe the Optional class
- Describe lazy processing
- Sort a stream
- Save results to a collection by using the `collect` method
- Group and partition data by using the `Collectors` class



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice Overview

- Practice 10-1: Using Map and Peek
- Practice 10-2: FindFirst and Lazy Operations
- Practice 10-3: Analyze Transactions with Stream Methods
- Practice 10-4: Perform Calculations with Primitive Streams
- Practice 10-5: Sort Transactions with Comparator
- Practice 10-6: Collect Results with Streams
- Practice 10-7: Join Data with Streams
- Practice 10-8: Group Data with Streams



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.