

ORACLE®

Digital content includes:

- 170 practice exam questions
- Fully customizable test engine



OCP Java SE 8 Programmer II Exam Guide

(Exam 1Z0-809)

Complete Exam Preparation



ORACLE®
Certified Professional
Java SE 8 Programmer

Kathy Sierra, SCJP
Bert Bates, SCJP, OCA, OCP
Elisabeth Robson

Oracle
Press™

ORACLE®

Oracle Press™

OCP Java™ SE 8 Programmer II Exam Guide

(Exam 1Z0-809)

**Kathy Sierra
Bert Bates
Elisabeth Robson**

McGraw-Hill Education is an independent entity from Oracle Corporation and is not affiliated with Oracle Corporation in any manner. This publication and digital content may be used in assisting students to prepare for the OCP Java SE 8 Programmer II exam. Neither Oracle Corporation nor McGraw-Hill Education warrants that use of this publication and digital content will ensure passing the relevant exam. Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.



New York Chicago San Francisco
Athens London Madrid
Mexico City Milan New Delhi
Singapore Sydney Toronto

Copyright © 2018 by McGraw-Hill Education (Publisher). All rights reserved. Except as permitted under the United States Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher, with the exception that the program listings may be entered, stored, and executed in a computer system, but they may not be reproduced for publication.

ISBN: 978-1-26-011737-0
MHID: 1-26-011737-5

The material in this eBook also appears in the print version of this title: ISBN: 978-1-26-011738-7, MHID: 1-26-011738-3.

eBook conversion by codeMantra
Version 1.0

All trademarks are trademarks of their respective owners. Rather than put a trademark symbol after every occurrence of a trademarked name, we use names in an editorial fashion only, and to the benefit of the trademark owner, with no intention of infringement of the trademark. Where such designations appear in this book, they have been printed with initial caps.

McGraw-Hill Education eBooks are available at special quantity discounts to use as premiums and sales promotions or for use in corporate training programs. To contact a representative, please visit the Contact Us page at www.mhprofessional.com.

Oracle and Java are registered trademarks of Oracle Corporation and/or its affiliates. All other trademarks are the property of their respective owners, and McGraw-Hill Education makes no claim of ownership by the mention of products that contain these marks.

Screen displays of copyrighted Oracle software programs have been reproduced herein with the permission of Oracle Corporation and/or its affiliates.

Information has been obtained by Publisher from sources believed to be reliable. However, because of the possibility of human or mechanical error by our

sources, Publisher, or others, Publisher does not guarantee to the accuracy, adequacy, or completeness of any information included in this work and is not responsible for any errors or omissions or the results obtained from the use of such information.

Oracle Corporation does not make any representations or warranties as to the accuracy, adequacy, or completeness of any information contained in this Work, and is not responsible for any errors or omissions.

TERMS OF USE

This is a copyrighted work and McGraw-Hill Education and its licensors reserve all rights in and to the work. Use of this work is subject to these terms. Except as permitted under the Copyright Act of 1976 and the right to store and retrieve one copy of the work, you may not decompile, disassemble, reverse engineer, reproduce, modify, create derivative works based upon, transmit, distribute, disseminate, sell, publish or sublicense the work or any part of it without McGraw-Hill Education's prior consent. You may use the work for your own noncommercial and personal use; any other use of the work is strictly prohibited. Your right to use the work may be terminated if you fail to comply with these terms.

THE WORK IS PROVIDED "AS IS." McGRAW-HILL EDUCATION AND ITS LICENSORS MAKE NO GUARANTEES OR WARRANTIES AS TO THE ACCURACY, ADEQUACY OR COMPLETENESS OF OR RESULTS TO BE OBTAINED FROM USING THE WORK, INCLUDING ANY INFORMATION THAT CAN BE ACCESSED THROUGH THE WORK VIA HYPERLINK OR OTHERWISE, AND EXPRESSLY DISCLAIM ANY WARRANTY, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. McGraw-Hill Education and its licensors do not warrant or guarantee that the functions contained in the work will meet your requirements or that its operation will be uninterrupted or error free. Neither McGraw-Hill Education nor its licensors shall be liable to you or anyone else for any inaccuracy, error or omission, regardless of cause, in the work or for any damages resulting therefrom. McGraw-Hill Education has no responsibility for the content of any information accessed through the work. Under no circumstances shall McGraw-Hill Education and/or its licensors be liable for any indirect, incidental, special, punitive, consequential or similar damages that

result from the use of or inability to use the work, even if any of them has been advised of the possibility of such damages. This limitation of liability shall apply to any claim or cause whatsoever whether such claim or cause arises in contract, tort or otherwise.

ABOUT THE CONTRIBUTORS

About the Authors

Kathy Sierra was a lead developer for the SCJP exam for Java 5 and Java 6. Kathy worked as a Sun “master trainer,” and in 1997, founded JavaRanch.com, the world’s largest Java community website. Her bestselling Java books have won multiple *Software Development Magazine* awards, and she is a founding member of Oracle’s Java Champions program.

These days, Kathy is developing advanced training programs in a variety of domains (from horsemanship to computer programming), but the thread that ties all of her projects together is helping learners reduce cognitive load.

Bert Bates was a lead developer for many of Sun’s Java certification exams, including the SCJP for Java 5 and Java 6. Bert was also one of the lead developers for Oracle’s OCA 7 and OCP 7 exams and a contributor to the OCP 8 exam. He is a forum moderator on JavaRanch.com and has been developing software for more than 30 years (argh!). Bert is the co-author of several best-selling Java books, and he’s a founding member of Oracle’s Java Champions program. Now that the book is done, Bert plans to go whack a few tennis balls around and once again start riding his beautiful Icelandic horse, Eyrraros fra Gufudal-Fremri.

Elisabeth Robson has an MSc in Computer Science and was a software programmer and engineering manager at The Walt Disney Company for many years. Since 2012 she has been a freelance writer and instructor. She produces online training and has written four best-selling books, including *Head First Design Patterns* (O’Reilly).

About the Technical Review Team

This is the fifth edition of the book that we’ve cooked up. The first version we worked on was for Java 2. Then we updated the book for the SCJP 5, again for

the SCJP 6, then for the OCA 7 and OCP 7 exams, and now for the OCA 8 and OCP 8 exams. Every step of the way, we were unbelievably fortunate to have fantastic JavaRanch.com-centric technical review teams at our sides. Over the course of the last 15 years, we've been "evolving" the book more than rewriting it. Many sections from our original work on the Java 2 book are still intact. On the following pages, we'd like to acknowledge the members of the various technical review teams who have saved our bacon over the years.

About the Java 2 Technical Review Team

Johannes de Jong has been the leader of our technical review teams forever and ever. (He has more patience than any three people we know.) For the Java 2 book, he led our biggest team ever. Our sincere thanks go out to the following volunteers who were knowledgeable, diligent, patient, and picky, picky, picky!

Rob Ross, Nicholas Cheung, Jane Griscti, Ilja Preuss, Vincent Brabant, Kudret Serin, Bill Seipel, Jing Yi, Ginu Jacob George, Radiya, LuAnn Mazza, Anshu Mishra, Anandhi Navaneethakrishnan, Didier Varon, Mary McCartney, Harsha Pherwani, Abhishek Misra, and Suman Das.

About the SCJP 5 Technical Review Team



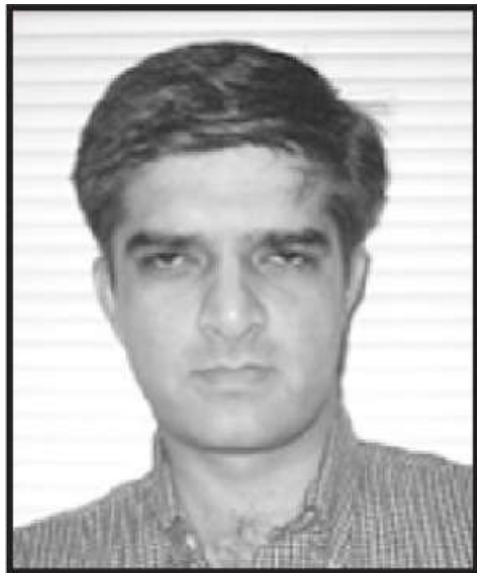
Andrew



Bill M.



Burk



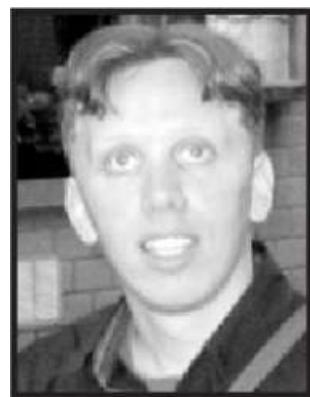
Devender



Gian



Jef



Jeoren



Jim



Johannes



Kristin



Marcelo



Marilyn



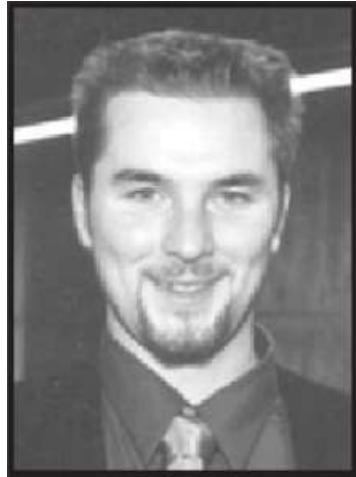
Mark



Mikalai



Seema



Valentin

We don't know who burned the most midnight oil, but we can (and did) count everybody's edits—so in order of most edits made, we proudly present our Superstars.

Our top honors go to **Kristin Stromberg**—every time you see a semicolon used correctly, tip your hat to Kristin. Next up is **Burk Hufnagel** who fixed more code than we care to admit. **Bill Mietelski** and **Gian Franco Casula** caught every kind of error we threw at them—awesome job, guys! **Devender Thareja** made sure we didn't use too much slang, and **Mark Spritzler** kept the humor coming. **Mikalai Zaikin** and **Seema Manivannan** made great catches every step of the way, and **Marilyn de Queiroz** and **Valentin Crettaz** both put in another stellar performance (saving our butts yet again).

Marcelo Ortega, **Jef Cumps** (another veteran), **Andrew Monkhouse**, and **Jeroen Sterken** rounded out our crew of Superstars—thanks to you all. **Jim Yingst** was a member of the Sun exam creation team, and he helped us write and review some of the twistier questions in the book (bwa-ha-ha-ha).

As always, every time you read a clean page, thank our reviewers, and if you do catch an error, it's most certainly because your authors messed up. And oh, one last thanks to **Johannes**. You rule, dude!

About the SCJP 6 Technical Review Team



Fred



Marc P.



Marc W.



Mikalai



Christophe

Since the upgrade to the Java 6 exam was like a small surgical strike we decided that the technical review team for this update to the book needed to be similarly fashioned. To that end, we hand-picked an elite crew of JavaRanch's top gurus to perform the review for the Java 6 exam.

Our endless gratitude goes to **Mikalai Zaikin**. Mikalai played a huge role in the Java 5 book, and he returned to help us out again for this Java 6 edition. We need to thank Volha, Anastasia, and Daria for letting us borrow Mikalai. His comments and edits helped us make huge improvements to the book. Thanks, Mikalai!

Marc Peabody gets special kudos for helping us out on a double header! In addition to helping us with Sun's new SCWCD exam, Marc pitched in with a great set of edits for this book—you saved our bacon this winter, Marc! (BTW, we didn't learn until late in the game that Marc, Bryan Basham, and Bert all share a passion for ultimate Frisbee!)

Like several of our reviewers, not only does **Fred Rosenberger** volunteer copious amounts of his time moderating at JavaRanch, he also found time to help us out with this book. Stacey and Olivia, you have our thanks for loaning us Fred for a while.

Marc Weber moderates at some of JavaRanch's busiest forums. Marc knows his stuff and uncovered some really sneaky problems that were buried in the book. While we really appreciate Marc's help, we need to warn you all to watch out—he's got a Phaser!

Finally, we send our thanks to **Christophe Verre**—if we can find him. It appears that Christophe performs his JavaRanch moderation duties from various locations around the globe, including France, Wales, and most recently Tokyo.

On more than one occasion Christophe protected us from our own lack of organization. Thanks for your patience, Christophe! It's important to know that these guys all donated their reviewer honorariums to JavaRanch! The JavaRanch community is in your debt.

The OCA 7 and OCP 7 Team

Contributing Authors



Tom



Jeanne

The OCA 7 exam is primarily a useful repackaging of some of the objectives from the SCJP 6 exam. On the other hand, the OCP 7 exam introduced a vast array of brand-new topics. We enlisted several talented Java gurus to help us cover some of the new topics on the OCP 7 exam. Thanks and kudos to **Tom McGinn** for his fantastic work in creating the massive JDBC chapter. Several reviewers told us that Tom did an amazing job channeling the informal tone we use throughout the book. Next, thanks to **Jeanne Boyarsky**. Jeanne was truly a renaissance woman on this project. She contributed to several OCP chapters; she wrote some questions for the master exams; she performed some project

management activities; and as if that wasn't enough, she was one of our most energetic technical reviewers. Jeanne, we can't thank you enough. Our thanks go to **Matt Heimer** for his excellent work on the concurrency chapter. A really tough topic, nicely handled! Finally, **Roel De Nijs** and **Roberto Perillo** made some nice contributions to the book *and* helped out on the technical review team —thanks, guys!

Technical Review Team



Roel



Mikalai



Vijitha



Roberto

Roel, what can we say? Your work as a technical reviewer is unparalleled. Roel caught so many technical errors, it made our heads spin. Between the printed book and all the material on the CD, we estimate that there are over 1,500 pages of “stuff” here. It’s huge! Roel grinded through page after page, never lost his focus, and made this book better in countless ways. Thank you, Roel!

In addition to her other contributions, **Jeanne** provided one of the most thorough technical reviews we received. (We think she enlisted her team of killer robots to help her!)

It seems like no K&B book would be complete without help from our old friend **Mikalai Zaikin**. Somehow, between earning 812 different Java certifications, being a husband and father (thanks to **Volha**, **Anastasia**, **Daria**, and **Ivan**), and being a “theoretical fisherman” [sic], Mikalai made substantial contributions to the quality of the book; we’re honored that you helped us again, Mikalai.

Next up, we’d like to thank **Vijitha Kumara**, JavaRanch moderator and tech reviewer extraordinaire. We had many reviewers help out during the long course of writing this book, but Vijitha was one of the few who stuck with us from [Chapter 1](#) all the way through the master exams and on to Chapter 15. Vijitha, thank you for your help and persistence!

Finally, thanks to the rest of our review team: **Roberto Perillo** (who also wrote some killer exam questions), **Jim Yingst** (was this your fourth time?), other repeat offenders: **Fred Rosenberger**, **Christophe Verre**, **Devaka Cooray**, **Marc Peabody**, and newcomer **Amit Ghorpade**—thanks, guys!

The OCP 8 Team

Approximately two-thirds of the OCP 8 exam’s objectives are the same as the OCP 7 exam, and about one-third are new topics focused on all of the amazing

new features introduced in Java 8. This time around, our entire review team was composed of veterans. In addition, it's about time that we give thanks to all of the folks on JavaRanch who took time to share errata with us. Because of our amazing xiireview team and the generosity of JavaRanchers (we know "CodeRanch"), this book was improved immensely.

Between the printed book and the two final mock exams, this book has well over 900 pages of material. One way of looking at this book is that every page is a series of factual claims. It's hard to estimate, but perhaps there are 20 such claims on every page. That means we're making about 18,000 factual claims in this book! This makes us authors feel better about saying that our various reviewers found hundreds of errors or areas in which our explanations could have been better. Hundreds out of 18,000 isn't too bad is it?

When we're in the mode of incorporating our reviewers' feedback, we often have to make tough choices. We know that our job isn't to restate the Java Language Spec. We have to make many decisions about how deeply to go into the topics on the exam. You should know that our reviewers constantly challenge us to go deeper. That's a good thing. If you feel as though we haven't covered a topic as deeply as you'd like or that we've oversimplified an explanation, the fault almost certainly lies with us authors, not with the reviewers!

With all of that said, it's time to thank the members of our amazing review team individually.

Technical Review Team



Mikalai



Campbell



Paweł



Frits



Roberto



Vijitha



Tim

Mikalai, wow, wow, wow! This is the fourth time (at least?) that Mikalai Zaikin has been one of our reviewers. Mikalai is a real expert, and he pushes us and makes us think. Mikalai is first and foremost a family man (hooray!), but he's also a geek, and—not satisfied with being only a Java expert—he also pursues other programming approaches as well. It wouldn't surprise us at all if he was into functional programming and other such wackiness. You all have a huge debt to pay to **Campbell**. Campbell Ritchie is a JavaRanch moderator and another true expert. Campbell is passionate about Java, and his edits really taught us a thing or two. Thanks for all your time Campbell! **Paweł Baczyński**, gave us a TON of good feedback. Paweł, send our thanks to your wife and kids; we appreciate their patience! Our next thanks go to veteran reviewer and JavaRanch moderator **Frits Walraven**. Frits is a published mock-exam-question creator (awesome), husband, father, and serial Java certificate holder. Get some sleep, Frits! Once again, we were honored to have **Roberto Perillo** on our review team. This is at least the third time Roberto has helped out. Given what a

thankless job this is, Roberto, we can't thank you enough. Roberto is a dad, hooray, and from what we hear he plays a mean guitar. With over 4000 JavaRanch posts to his credit, moderator **Vijitha Kumara** proved once again to be “in it for the long haul!” In addition to traveling and community service, Vijitha was with us right to the final mock exam. Vijitha, thanks for all your help! Last but not least, our thanks go to yet another JavaRanch moderator **Tim Cooke**. Rumor has it that Tim’s cat Polly (an Erlang aficionado!?), was at Tim’s side throughout the editing process. This might explain some of the attitude that came through in Tim’s edits. Tim focused his energies on editing the new FP-ish additions to the exam. Tim, thanks so much for all of your help!

For Jim, Joe, and Solveig

CONTENTS AT A GLANCE

- 1 Declarations, Access Control, and Enums
- 2 Object Orientation
- 3 Assertions and Java Exceptions
- 4 Dates, Times, Locales, and Resource Bundles
- 5 I/O and NIO
- 6 Generics and Collections
- 7 Inner Classes
- 8 Lambda Expressions and Functional Interfaces
- 9 Streams
- 10 Threads
- 11 Concurrency
- 12 JDBC
- A About the Online Content
- Index

CONTENTS

Acknowledgments

Preface

Introduction

1 Declarations, Access Control, and Enums

Java Class Design and Object Orientation: A Refresher

Define Classes and Interfaces (OCP Objectives 1.2, 2.1, and 2.2)

 Class Declarations and Modifiers

Exercise 1-1: Creating an Abstract Superclass and Concrete Subclass

Use Interfaces (OCP Objective 2.5)

 Declaring an Interface

 Declaring Interface Constants

 Declaring default Interface Methods

 Declaring static Interface Methods

Declare Class Members (OCP Objectives 1.2, 1.6, 2.1, and 2.2)

 Access Modifiers

 Nonaccess Member Modifiers

 Constructor Declarations

 Variable Declarations

Declare and Use enums (OCP Objective 2.4)

 Declaring enums

Certification Summary



 Two-Minute Drill

Q&A Self Test

 Self Test Answers

2 Object Orientation

Encapsulation (OCP Objective 1.1)
Inheritance and Polymorphism (OCP Objectives 1.2 and 1.3)
 The Evolution of Inheritance
 IS-A and HAS-A Relationships
Polymorphism (OCP Objective 1.3)
Overriding/Overloading (OCP Objectives 1.2, 1.3, and 2.5)
 Overridden Methods
 Overloaded Methods
Casting (OCP Objectives 1.2 and 1.3)
Implementing an Interface (OCP Objective 2.5)
 Java 8—Now with Multiple Inheritance!
Legal Return Types (OCP Objectives 1.2 and 1.3)
 Return Type Declarations
 Returning a Value
Constructors and Instantiation (OCP Objectives 1.2 and 1.3)
 Constructor Basics
 Constructor Chaining
 Rules for Constructors
 Determine Whether a Default Constructor Will Be Created
 Overloaded Constructors
Singleton Design Pattern (OCP Objective 1.5)
 What Is a Design Pattern?
 Problem
 Solution
 Benefits
Immutable Classes (OCP Objective 1.5)
Initialization Blocks (OCP Objective 1.6)
Statics (OCP Objective 1.6)
 Static Variables and Methods
Certification Summary
 Two-Minute Drill
Q&A Self Test
 Self Test Answers

3 Assertions and Java Exceptions

Working with the Assertion Mechanism (OCP Objective 6.5)

Assertions Overview
Using Assertions
Using Assertions Appropriately
Working with Exception Handling (OCP Objectives 6.1, 6.2, 6.3, and 6.4)
 Use the try Statement with multi-catch and finally Clauses
 AutoCloseable Resources with a try-with-resources Statement
Certification Summary
 Two-Minute Drill
Q&A Self Test
 Self Test Answers

4 Dates, Times, Locales, and Resource Bundles

Dates, Times, and Locales (OCP Objectives 7.1, 7.2, 7.3, and 12.1)
 Working with Dates and Times
 The java.time.* Classes for Dates and Times
Properties Files (OCP Objective 12.2)
Resource Bundles (OCP Objectives 12.1, 12.2, and 12.3)
 Java Resource Bundles
 Default Locale
 Choosing the Right Resource Bundle
Certification Summary
 Two-Minute Drill
Q&A Self Test
 Self Test Answers

5 I/O and NIO

File Navigation and I/O (OCP Objectives 8.1 and 8.2)
 Creating Files Using the File Class
 Using FileWriter and FileReader
 Using FileInputStream and FileOutputStream
 Combining I/O Classes
 Working with Files and Directories
 The java.io.Console Class
Files, Path, and Paths (OCP Objectives 9.1 and 9.2)
 Creating a Path
 Creating Files and Directories

Copying, Moving, and Deleting Files
Retrieving Information about a Path
Normalizing a Path
Resolving a Path
Relativizing a Path

File and Directory Attributes (OCP Objective 9.2)
Reading and Writing Attributes the Easy Way
Types of Attribute Interfaces
Working with BasicFileAttributes
Working with DosFileAttributes
Working with PosixFileAttributes
Reviewing Attributes

DirectoryStream (OCP Objectives 9.2 and 9.3)
FileVisitor
PathMatcher
WatchService

Serialization (Objective 8.2)
Working with ObjectOutputStream and ObjectInputStream
Object Graphs
Using writeObject and readObject
How Inheritance Affects Serialization
Serialization Is Not for Statics

Certification Summary

✓ Two-Minute Drill

Q&A Self Test
Self Test Answers

6 Generics and Collections

Override hashCode(), equals(), and toString() (OCP Objective 1.4)

The toString() Method
Overriding equals()
Overriding hashCode()

Collections Overview (OCP Objective 3.2)

So What Do You Do with a Collection?
Key Interfaces and Classes of the Collections Framework
List Interface

[Set Interface](#)

[Map Interface](#)

[Queue Interface](#)

[Using Collections \(OCP Objectives 2.6, 3.2, and 3.3\)](#)

[ArrayList Basics](#)

[Autoboxing with Collections](#)

[The Java 7 “Diamond” Syntax](#)

[Sorting Collections and Arrays](#)

[Navigating \(Searching\) TreeSets and TreeMaps](#)

[Other Navigation Methods](#)

[Backed Collections](#)

[Using the PriorityQueue Class and the Deque Interface](#)

[Method Overview for Arrays and Collections](#)

[Method Overview for List, Set, Map, and Queue](#)

[Generic Types \(OCP Objective 3.1\)](#)

[The Legacy Way to Do Collections](#)

[Generics and Legacy Code](#)

[Mixing Generic and Nongeneric Collections](#)

[Polymorphism and Generics](#)

[Generic Methods](#)

[Generic Declarations](#)

[Certification Summary](#)



[Two-Minute Drill](#)

[Q&A Self Test](#)

[Self Test Answers](#)

7 Inner Classes

[Nested Classes \(OCP Objective 2.3\)](#)

[Inner Classes](#)

[Coding a “Regular” Inner Class](#)

[Referencing the Inner or Outer Instance from Within the Inner Class](#)

[Method-Local Inner Classes](#)

[What a Method-Local Inner Object Can and Can’t Do](#)

[Anonymous Inner Classes](#)

[Plain-Old Anonymous Inner Classes, Flavor One](#)

[Plain-Old Anonymous Inner Classes, Flavor Two](#)

Argument-Defined Anonymous Inner Classes
Static Nested Classes
 Instantiating and Using Static Nested Classes
Lambda Expressions as Inner Classes (OCP Objective 2.6)
 Comparator Is a Functional Interface
Certification Summary
 Two-Minute Drill
Q&A Self Test
 Self Test Answers

8 Lambda Expressions and Functional Interfaces

Lambda Expression Syntax (OCP Objective 2.6)
 Passing Lambda Expressions to Methods
 Accessing Variables from Lambda Expressions
Functional Interfaces (OCP Objectives 3.5, 4.1, 4.2, 4.3, and 4.4)
 Built-in Functional Interfaces
 What Makes an Interface Functional?
 Categories of Functional Interfaces
Method References (OCP Objective 3.8)
 Kinds of Method References
Write Your Own Functional Interface
 Functional Interface Overview
Certification Summary
 Two-Minute Drill
Q&A Self Test
 Self Test Answers

9 Streams

What Is a Stream? (OCP Objective 3.4)
How to Create a Stream (OCP Objectives 3.5 and 9.3)
 Create a Stream from a Collection
 Build a Stream with Stream.of()
 Create a Stream from an Array
 Create a Stream from a File
 Primitive Value Streams
 Summary of Methods to Create Streams

Why Streams?

The Stream Pipeline (OCP Objective 3.6)

Streams Are Lazy

Operating on Streams (OCP Objectives 3.7 and 5.1)

Map-Filter-Reduce with average() and Optionals (OCP Objectives 5.3 and 5.4)

Reduce

Using reduce()

Associative Accumulations

map-filter-reduce Methods

Optionals (OCP Objective 5.3)

Searching and Sorting with Streams (OCP Objectives 5.2 and 5.5)

Searching to See Whether an Element Exists

Searching to Find and Return an Object

Sorting

Methods to Search and Sort Streams

Don't Modify the Source of a Stream

Collecting Values from Streams (OCP Objectives 3.8, 5.6, and 9.3)

Using collect() with Files.lines()

Exercise 9-1: Collecting Items in a List

Grouping and Partitioning

Summing and Averaging

Counting, joining, maxBy, and minBy

Stream Methods to Collect and Their Collectors

Streams of Streams (OCP Objective 5.7)

Generating Streams (OCP Objective 3.4)

Methods to Generate Streams

Caveat Time Again

A Taste of Parallel Streams

Certification Summary



Two-Minute Drill

Q&A Self Test

Self Test Answers

Exercise Answer

10 Threads

Defining, Instantiating, and Starting Threads (OCP Objective 10.1)

- Making a Thread
- Defining a Thread
- Instantiating a Thread
- Starting a Thread

Thread States and Transitions

- Thread States
- Preventing Thread Execution
- Sleeping

Exercise 10-1: Creating a Thread and Putting It to Sleep

Thread Priorities and `yield()`

Synchronizing Code, Thread Problems (OCP Objectives 10.2 and 10.3)

- Preventing the Account Overdraw
 - Synchronization and Locks
- Exercise 10-2:** Synchronizing a Block of Code
- Thread Deadlock
 - Thread Livelock
 - Thread Starvation
 - Race Conditions

Thread Interaction (OCP Objectives 10.2 and 10.3)

Using `notifyAll()` When Many Threads May Be Waiting

Certification Summary

 Two-Minute Drill

Q&A Self Test

Self Test Answers

Exercise Answers

I1 Concurrency

Concurrency with the `java.util.concurrent` Package

Apply Atomic Variables and Locks (OCP Objective 10.3)

- Atomic Variables
- Locks

Use `java.util.concurrent` Collections (OCP Objective 10.4)

- Copy-on-Write Collections
- Concurrent Collections
- Blocking Queues

Controlling Threads with CyclicBarrier
Use Executors and ThreadPools (OCP Objective 10.1)
 Identifying Parallel Tasks
 How Many Threads Can You Run?
 CPU-Intensive vs. I/O-Intensive Tasks
 Fighting for a Turn
 Decoupling Tasks from Threads

Use the Parallel Fork/Join Framework (OCP Objective 10.5)
 Divide and Conquer
 ForkJoinPool
 ForkJoinTask

Parallel Streams (OCP Objective 10.6)
 How to Make a Parallel Stream Pipeline
 Embarrassingly Parallel, Take Two (with Parallel Streams)
 A Parallel Stream Implementation of a RecursiveTask
 Reducing Parallel Streams with reduce()

Certification Summary

✓ Two-Minute Drill

Q&A Self Test
 Self Test Answers

12 JDBC

Starting Out: Introduction to Databases and JDBC
 Talking to a Database
 Bob's Books, Our Test Database

Core Interfaces of the JDBC API (OCP Objective 11.1)

Connect to a Database Using DriverManager (OCP Objective 11.2)
 The DriverManager Class
 The JDBC URL
 JDBC Driver Implementation Versions

Submit Queries and Read Results from the Database (OCP Objective 11.3)
 All of Bob's Customers
 Statements
 ResultSets
 When Things Go Wrong—Exceptions and Warnings

Certification Summary

-  [Two-Minute Drill](#)
- [Q&A](#) [Self Test](#)
- [Self Test Answers](#)

A About the Online Content

- [McGraw-Hill Professional Media Center Download](#)
- [Total Tester Online System Requirements](#)
- [Single User License Terms and Conditions](#)
- [Total Tester Online](#)
- [Technical Support](#)

Index

ACKNOWLEDGMENTS

Kathy and Bert (and in the cases of McGraw-Hill Education and JavaRanch, Elisabeth) would like to thank the following people:

- All the incredibly hard-working folks at McGraw-Hill Education: Tim Green (who's been putting up with us for 15 years now), LeeAnn Pickrell (and team), Lisa McClain, Jody McKenzie, and Jim Kussow. Thanks for all your help, and for being so responsive, patient, flexible, and professional, and the nicest group of people we could hope to work with.
- All of our friends at Kraftur (and our other horse-related friends) and most especially to Sherry; Steinar; Stina and the girls, Jec, Lucy, Cait, and Jennifer; Leslie and David; Annette and Bruce; Kacie; DJ; Gabrielle; and Mary. Thanks to Pedro and Ely, who can't believe it can take so long to finish a book.
- Some of the software professionals and friends who helped us in the early days: Tom Bender, Peter Loerincs, Craig Matthews, Leonard Coyne, Morgan Porter, and Mike Kavanaugh.
- Dave Gustafson and Marc Hedlund for their continued support, insights, and coaching.
- Our good friend at Oracle, Yvonne Prefontaine.
- The crew at Oracle who worked hard to build these exams: Tom McGinn, Matt Heimer, Mike Williams, Stuart Marks, and Mikalai Zaikin.
- Our old wonderful and talented certification team at Sun Educational Services, primarily the most persistent get-it-done person we know, Evelyn Cartagena.
- Our great friends and gurus, Simon Roberts, Bryan Basham, and Kathy Collina.
- Stu, Steve, Burt, and Eric for injecting some fun into the process.
- To Eden and Skyler, for being horrified that adults—out of school—

would study this hard for an exam.

- To the JavaRanch Trail Boss Paul Wheaton, for running the best Java community site on the Web, and to all the generous and patient JavaRanch moderators.
- To all the past and present Sun Ed Java instructors for helping to make learning Java a fun experience, including (to name only a few) Alan Petersen, Jean Tordella, Georgianna Meagher, Anthony Orapallo, Jacqueline Jones, James Cubeta, Teri Cubeta, Rob Weingruber, John Nyquist, Asok Perumainar, Steve Stelting, Kimberly Bobrow, Keith Ratliff, and the most caring and inspiring Java guy on the planet, Jari Paukku.
- Our furry and feathered friends Eyra, Kara, Draumur, Vafi, Boi, Niki, and Bokeh.
- Finally, to Elisabeth Robson (our amazing new co-author) and Eric Freeman, for your continued inspiration.

PREFACE

This book's primary objective is to help you prepare for and pass Oracle's OCP Java SE 8 Programmer II certification exam.

If you already have an SCJP 6 or OCP 7 certification and want to take an upgrade exam, all of the topics covered in the OCP 7 and SCJP 6 Upgrade exams are covered here as well.

This book follows closely both the breadth and the depth of the real exams. For instance, after reading this book, you probably won't emerge as an NIO.2 guru, but if you study the material and do well on the Self Tests, you'll have a basic understanding of NIO.2, and you'll do well on the exam. After completing this book, you should feel confident that you have thoroughly reviewed all of the objectives that Oracle has established for these exams.

In This Book

This book is organized to optimize your learning of the topics covered by the OCP 8 exam. Whenever possible, we've organized the chapters to parallel the Oracle objectives, but sometimes we'll mix up objectives or partially repeat them in order to present topics in an order better suited to learning the material.

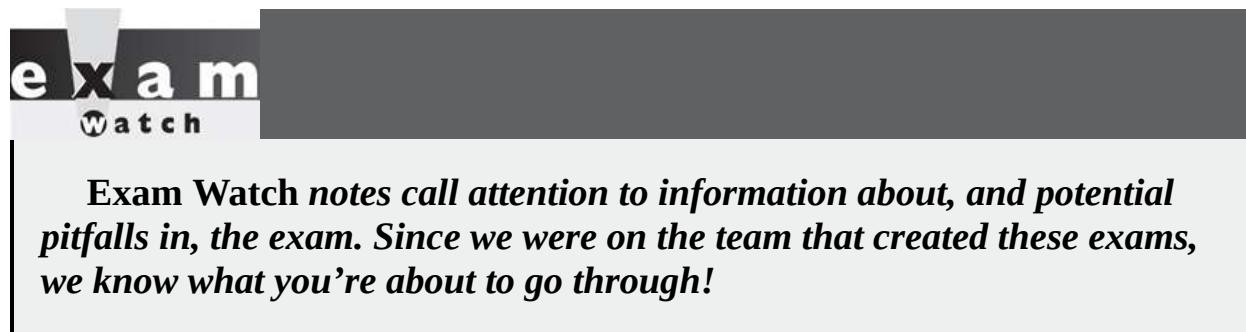
In the Chapters

We've created a set of chapter components that call your attention to important items, reinforce important points, and provide helpful exam-taking hints. Take a look at what you'll find in the chapters:

- Every chapter begins with the **Certification Objectives**—what you need to know in order to pass the section on the exam dealing with the chapter topic. The Certification Objective headings identify the objectives within the chapter, so you'll always know an objective when you see it!



- **On the Job** callouts discuss practical aspects of certification topics that might not occur on the exam, but that will be useful in the real world.



exam
watch

Exam Watch notes call attention to information about, and potential pitfalls in, the exam. Since we were on the team that created these exams, we know what you're about to go through!

- **Exercises** help you master skills that are likely to be an area of focus on the exam. Don't just read through the exercises; they are hands-on practice that you should be comfortable completing. Learning by doing is an effective way to increase your competency with a product.
- The **Certification Summary** is a succinct review of the chapter and a restatement of salient points regarding the exam.



- The **Two-Minute Drill** at the end of every chapter is a checklist of the main points of the chapter. It can be used for last-minute review.

Q&A

- The **Self Test** offers questions similar to those found on the certification exam, including multiple-choice and pseudo drag-and-drop questions. The answers to these questions, as well as explanations of the answers, can be found at the end of every chapter. By taking the Self Test after completing each chapter, you'll reinforce what you've learned from that chapter, while becoming familiar with the structure of the exam questions.

INTRODUCTION

Organization

This book is organized in such a way as to serve as an in-depth review for the OCP Java SE 8 Programmer II exam for both experienced Java professionals and those in the early stages of experience with Java technologies. Each chapter covers at least one major aspect of the exam, with an emphasis on the “why” as well as the “how to” of programming in the Java language.

Throughout this book and supplemental digital material, you’ll find support for three exams:

- OCP Java SE 8 Programmer II
- Upgrade to Java SE 8 Programmer from Java SE 7
- Upgrade to Java SE 8 Programmer from Java SE 6

What This Book Is Not

You will not find a beginner’s guide to learning Java in this book. All 900+ pages of this book are dedicated solely to helping you pass the exams. Since you cannot take this exam without another Java certification under your belt, in this book, we assume you have a working knowledge of everything covered in the OCA 8 exam. We do not, however, assume any level of prior knowledge of the individual topics covered. In other words, for any given topic (driven exclusively by the actual exam objectives), we start with the assumption that you are new to that topic. So we assume you’re new to the individual topics, but we assume that you are not new to Java.

We also do not pretend to be both preparing you for the exam and simultaneously making you a complete Java being. This is a certification exam study guide, and it’s very clear about its mission. That’s not to say that preparing for the exam won’t help you become a better Java programmer! On the contrary, even the most experienced Java developers often claim that having to prepare for the certification exam made them far more knowledgeable and well-rounded

programmers than they would have been without the exam-driven studying.

About the Digital Content

You'll receive access to online practice exam software with the equivalent of two 85-question exams for OCP Java SE 8 candidates. The digital content included with the book includes three chapters that complete the coverage necessary for the OCP 7 and SCJP 6 upgrade certifications. We've also included an Online Appendix, "Creating Streams from Files Methods," describing additional methods for processing files and directories.

Please see the Appendix for details on accessing the digital content and online practice exams.

Some Pointers

Once you've finished reading this book, set aside some time to do a thorough review. You might want to return to the book several times and make use of all the methods it offers for reviewing the material:

1. *Re-read all the Two-Minute Drills, or have someone quiz you.* You also can use the drills as a way to do a quick cram before the exam. You might want to make some flash cards out of 3×5 index cards that have the Two-Minute Drill material on them.
2. *Re-read all the Exam Watch notes.* Remember that these notes are written by authors who helped create the exam. They know what you should expect—and what you should be on the lookout for.
3. *Re-take the Self Tests.* Taking the tests right after you've read the chapter is a good idea because the questions help reinforce what you've just learned. However, it's an even better idea to go back later and do all the questions in the book in one sitting. Pretend that you're taking the live exam. (Whenever you take the Self Tests, mark your answers on a separate piece of paper. That way, you can run through the questions as many times as you need to until you feel comfortable with the material.)
4. *Complete the exercises.* The exercises are designed to cover exam topics, and there's no better way to get to know this material than by practicing. Be sure you understand why you are performing each step in each exercise. If there is something you are not clear on, re-read that section in

the chapter.

5. *Write lots of Java code.* We'll repeat this advice several times. When we wrote this book, we wrote hundreds of small Java programs to help us do our research. We have heard from hundreds of candidates who have passed the exam, and in almost every case, the candidates who scored extremely well on the exam wrote lots of code during their studies. Experiment with the code samples in the book, create horrendous lists of compiler errors—put away your IDE, crank up the command line, and write code!

A Note About the Certification Objectives

Some of the OCP 8 Certification Objectives are not exactly the most clearly written objectives. You may, as we sometimes did, find yourself squinting a little sideways at an objective when attempting to parse what exactly the objective is about and what the objective might be leaving unsaid. We've done our best to cover all the topics we *think* the objective writers meant to include. And, as a reminder, we do try to teach about 115 percent of what we think you'll need to know for the exam, just to cover all our (and your) bases. That said, there are some gray areas here and there. So look carefully at each objective, and if you think there's something we missed, go study that on your own (and do let us know!).

Introduction to the Material in the Book

The OCP 8 exam is considered one of the hardest in the IT industry, and we can tell you from experience that a large chunk of exam candidates goes in to the test unprepared. As programmers, we tend to learn only what we need to complete our current project, given the insane deadlines we're usually under.

But this exam attempts to prove your complete understanding of the Java language, not just the parts of it you've become familiar with in your work.

Experience alone will rarely get you through this exam with a passing mark, because even the things you think you know might work just a little differently than you imagined. It isn't enough to be able to get your code to work correctly; you must understand the core fundamentals in a deep way and with enough breadth to cover virtually anything that could crop up in the course of using the language.

Who Cares About Certification?

Employers do. Headhunters do. Programmers do. Passing this exam proves three important things to a current or prospective employer: you're smart; you know how to study and prepare for a challenging test; and, most of all, you know the Java language. If an employer has a choice between a candidate who has passed xxxvith the exam and one who hasn't, the employer knows that the certified programmer does not have to take time to learn the Java language.

But does it mean that you can actually develop software in Java? Not necessarily, but it's a good head start.

Taking the Programmer's Exam

In a perfect world, you would be assessed for your true knowledge of a subject, not simply how you respond to a series of test questions. But life isn't perfect, and it just isn't practical to evaluate everyone's knowledge on a one-to-one basis.

For the majority of its certifications, Oracle evaluates candidates using a computer-based testing service operated by Pearson VUE. To discourage simple memorization, Oracle exams present a potentially different set of questions to different candidates. In the development of the exam, hundreds of questions are compiled and refined using beta testers. From this large collection, questions are pulled together from each objective and assembled into many different versions of the exam.

Each Oracle exam has a specific number of questions, and the test's duration is designed to be generous. The time remaining is always displayed in the corner of the testing screen. If time expires during an exam, the test terminates, and incomplete answers are counted as incorrect.



Many experienced test-takers do not go back and change answers unless they have a good reason to do so. Only change an answer when you feel you may have misread or misinterpreted the question the first time. Nervousness may make you secondguess every answer and talk yourself out of a correct one.

Question Format

Oracle's Java exams pose questions in multiple-choice format.

Multiple-Choice Questions

In earlier versions of the exam, when you encountered a multiple-choice question, you were not told how many answers were correct, but with each version of the exam, the questions have become more difficult, so today, each multiple-choice question tells you how many answers to choose. The Self Test questions at the end of each chapter closely match the format, wording, and difficulty of the real exam questions, with two exceptions:

- Whenever we can, our questions will *not* tell you how many correct answers exist (we will say “Choose all that apply”). We do this to help you master the material. Some savvy test-takers can eliminate wrong answers when the number of correct answers is known. It’s also possible, if you know how many answers are correct, to choose the most plausible answers. Our job is to toughen you up for the real exam!
- The real exam typically numbers lines of code in a question. Sometimes we do not number lines of code—mostly so that we have the space to add comments at key places. On the real exam, when a code listing starts with line 1, it means that you’re looking at an entire source file. If a code listing starts at a line number greater than 1, that means you’re looking at a partial source file. When looking at a partial source file, assume that the code you can’t see is correct. (For instance, unless explicitly stated, you can assume that a partial source file will have the correct import and package statements.)



When you find yourself stumped answering multiple-choice questions, use your scratch paper (or whiteboard) to write down the two or three answers you consider the strongest, then underline the answer you feel is most likely correct. Here is an example of what your scratch paper might look like when you've gone through the test once:

- ***21. B or C***

■ 33. A or C

This is extremely helpful when you mark the question and continue on. You can then return to the question and immediately pick up your thought process where you left off. Use this technique to avoid having to re-read and rethink questions. You will also need to use your scratch paper during complex, text-based scenario questions to create visual images to better understand the question. This technique is especially helpful if you are a visual learner.

Tips on Taking the Exam

The number of questions and passing percentages for every exam are subject to change. Always check with Oracle before taking the exam, at www.Oracle.com.

You are allowed to answer questions in any order, and you can go back and check your answers after you've gone through the test. There are no penalties for wrong answers, so it's better to at least attempt an answer than to not give one at all.

A good strategy for taking the exam is to go through once and answer all the questions that come to you quickly. You can then go back and do the others. Answering one question might jog your memory for how to answer a previous one.

Be very careful on the code examples. Check for syntax errors first: count curly braces, semicolons, and parentheses and then make sure there are as many left ones as right ones. Look for capitalization errors and other such syntax problems before trying to figure out what the code does.

Many of the questions on the exam will hinge on subtleties of syntax. You will need to have a thorough knowledge of the Java language in order to succeed.

This brings us to another issue that some candidates have reported. The testing center is supposed to provide you with sufficient writing implements so you can work problems out "on paper." In some cases, the centers have provided inadequate markers and dry-erase boards that are too small and cumbersome to use effectively. We recommend that you call ahead and verify that you will be supplied with a sufficiently large whiteboard, sufficiently fine-tipped markers, and a good eraser. What we'd really like to encourage is for everyone to complain to Oracle and Pearson VUE and have them provide actual pencils and at least several sheets of blank paper.

Tips on Studying for the Exam

First and foremost, give yourself plenty of time to study. Java is a complex programming language, and you can't expect to cram what you need to know into a single study session. It is a field best learned over time, by studying a subject and then applying your knowledge. Build yourself a study schedule and stick to it, but be reasonable about the pressure you put on yourself, especially if you're studying in addition to your regular duties at work.

One easy technique to use in studying for certification exams is the 15-minutes-per-day effort. Simply study for a minimum of 15 minutes every day. It is a small but significant commitment. If you have a day where you just can't focus, then give up at 15 minutes. If you have a day where it flows completely for you, study longer. As long as you have more of the "flow days," your chances of succeeding are excellent.

We strongly recommend you use flash cards when preparing for the programmer's exams. A flash card is simply a 3×5 or 4×6 index card with a question on the front and the answer on the back. You construct these cards yourself as you go through a chapter, capturing any topic you think might need more memorization or practice time. You can drill yourself with them by reading the question, thinking through the answer, and then turning the card over to see if you're correct. Or you can get another person to help you by holding up the card with the question facing you and then verifying your answer. Most of our students have found these to be tremendously helpful, especially because they're so portable that while you're in study mode, you can take them everywhere. Best not to use them while driving, though, except at red lights. We've taken ours everywhere—the doctor's office, restaurants, theaters, you name it.

Certification study groups are another excellent resource, and you won't find a larger or more willing community than on the JavaRanch.com Big Moose Saloon certification forums. If you have a question from this book, or any other mock exam question you may have stumbled upon, posting a question in a certification forum will get you an answer in nearly all cases within a day—usually, within a few hours. You'll find us (the authors) there several times a week, helping those just starting out on their exam preparation journey. (You won't actually think of it as anything as pleasant sounding as a "journey" by the time you're ready to take the exam.)

Finally, we recommend that you write a lot of little Java programs! During the course of writing this book, we wrote hundreds of small programs, and if you listen to what the most successful candidates say (you know, those guys who got

98 percent), they almost always report that they wrote a lot of code.

Scheduling Your Exam

You can purchase your exam voucher from Oracle or Pearson VUE. Visit Oracle.com (follow the training/certification links) or visit PearsonVue.com for exam scheduling details and locations of test centers.

Arriving at the Exam

As with any test, you'll be tempted to cram the night before. Resist that temptation. You should know the material by this point, and if you're groggy in the morning, you won't remember what you studied anyway. Get a good night's sleep.

Arrive early for your exam; it gives you time to relax and review key facts. Take the opportunity to review your notes. If you get burned out on studying, you can usually start your exam a few minutes early. We don't recommend arriving late. Your test could be cancelled, or you might not have enough time to complete the exam.

When you arrive at the testing center, you'll need to provide current, valid photo identification. Visit PearsonVue.com for details on the ID requirements. They just want to be sure that you don't send your brilliant Java guru next-door neighbor who you've paid to take the exam for you.

Aside from a brain full of facts, you don't need to bring anything else to the exam room. In fact, your brain is about all you're allowed to take into the exam!

All the tests are closed book, meaning you don't get to bring any reference materials with you. You're also not allowed to take any notes out of the exam room. The test administrator will provide you with a small marker board. If you're allowed to, we do recommend that you bring a water bottle or a juice bottle (call ahead for details of what's allowed). These exams are long and hard, and your brain functions much better when it's well hydrated. In terms of hydration, the ideal approach is to take frequent, small sips. You should also verify how many "bio-breaks" you'll be allowed to take during the exam!

Leave your pager and telephone in the car or turn them off. They only add stress to the situation, since they are not allowed in the exam room, and can sometimes still be heard if they ring outside of the room. Purses, books, and other materials must be left with the administrator before entering the exam.

Once in the testing room, you'll be briefed on the exam software. You might

be asked to complete a survey. The time you spend on the survey is *not* deducted from your actual test time—nor do you get more time if you fill out the survey quickly. Also, remember that the questions you get on the exam will *not* change depending on how you answer the survey questions. Once you’re done with the survey, the real clock starts ticking and the fun begins.

The testing software allows you to move forward and backward between questions. Most important, there is a Mark check box on the screen—this will prove to be a critical tool, as explained in the next section.

Test-Taking Techniques

Without a plan of attack, candidates can become overwhelmed by the exam or become sidetracked and run out of time. For the most part, if you are comfortable with the material, the allotted time is more than enough to complete the exam. The trick is to keep the time from slipping away during any one particular problem.

Your obvious goal is to answer the questions correctly and quickly, but other factors can distract you. Here are some tips for taking the exam more efficiently.

Size Up the Challenge

First, take a quick pass through all the questions in the exam. “Cherry-pick” the easy questions, answering them on the spot. Briefly read each question, noticing the type of question and the subject. As a guideline, try to spend less than 25 percent of your testing time in this pass.

This step lets you assess the scope and complexity of the exam, and it helps you determine how to pace your time. It also gives you an idea of where to find potential answers to some of the questions. Sometimes the wording of one question might lend clues or jog your thoughts for another question.

If you’re not entirely confident in your answer to a question, answer it anyway, but check the Mark box to flag it for later review. In the event that you run out of time, at least you’ve provided a “first guess” answer, rather than leaving it blank.

Second, go back through the entire test, using the insight you gained from the first go-through. For example, if the entire test looks difficult, you’ll know better than to spend more than a minute or two on each question. Create a pacing with small milestones—for example, “I need to answer 10 questions every 15 minutes.”

At this stage, it's probably a good idea to skip past the time-consuming questions, marking them for the next pass. Try to finish this phase before you're 50 to 60 percent through the testing time.

Third, go back through all the questions you marked for review, using the Review Marked button in the question review screen. This step includes taking a second look at all the questions you were unsure of in previous passes, as well as tackling the time-consuming ones you deferred until now. Chisel away at this group of questions until you've answered them all.

If you're more comfortable with a previously marked question, unmark the Review Marked button now. Otherwise, leave it marked. Work your way through the time-consuming questions now, especially those requiring manual calculations. Unmark them when you're satisfied with the answer.

By the end of this step, you've answered every question in the test, despite having reservations about some of your answers. If you run out of time in the next step, at least you won't lose points for lack of an answer. You're in great shape if you still have 10 to 20 percent of your time remaining.

Review Your Answers

Now you're cruising! You've answered all the questions, and you're ready to do a quality check. Take yet another pass (yes, one more) through the entire test, briefly re-reading each question and your answer.

Carefully look over the questions again to check for "trick" questions. Be particularly wary of those that include a choice of "Does not compile." Be alert for last-minute clues. You're pretty familiar with nearly every question at this point, and you may find a few clues that you missed before.

The Grand Finale

When you're confident with all your answers, finish the exam by submitting it for grading. After you finish your exam, you'll receive an e-mail from Oracle giving you a link to a page where your exam results will be available. As of this writing, you must ask for a hard copy certificate specifically or one will not be sent to you.

Retesting

If you don't pass the exam, don't be discouraged. Try to have a good attitude

about the experience, and get ready to try again. Consider yourself a little more educated. You'll know the format of the test a little better, and you'll have a good idea of the difficulty level of the questions you'll get next time around.

If you bounce back quickly, you'll probably remember several of the questions you might have missed. This will help you focus your study efforts in the right area.

Ultimately, remember that Oracle certifications are valuable because they're hard to get. After all, if anyone could get one, what value would it have? In the end, it takes a good attitude and a lot of studying, but you can do it!

OCP 8 Objectives Map

The following table for the OCP Java SE 8 Programmer II Exam describes the objectives and where you will find them in the book.

Oracle Certified Professional Java SE 8 Programmer II (Exam IZ0-804)

(Note: Some of the OCP objectives detailed here are similar to or duplicates of OCA 8 objectives. If you've read our *OCA 8 Java SE 8 Programmer I Exam Guide*, you will recognize some of the material covering those objectives, particularly material in [Chapters 1 and 2](#).)

Official Objective	Exam Guide Coverage
Java Class Design	
Implement encapsulation (1.1)	Chapter 1
Implement inheritance including visibility modifiers and composition (1.2)	Chapter 1
Implement polymorphism (1.3)	Chapter 1
Override the hashCode, equals, and toString methods from Object class (1.4)	Chapter 1
Create and use singleton classes and immutable classes (1.5)	Chapter 1
Develop code that uses the static keyword on initialize blocks, variables, methods, and classes (1.6)	Chapter 1
Advanced Java Class Design	
Develop code that uses abstract classes and methods (2.1)	Chapter 2
Develop code that uses the final keyword (2.2)	Chapter 2
Create inner classes including static inner class, local class, nested class, and anonymous inner class (2.3)	Chapter 2
Use enumerated types including methods, and constructors in an enum type (2.4)	Chapter 2
Develop code that declares, implements and/or extends interfaces and use the @Override annotation (2.5)	Chapter 2
Create and use Lambda expressions (2.6)	Chapters 6, 7, and 8
Generics and Collections	
Create and use a generic class (3.1)	Chapter 6
Create and use ArrayList, TreeSet, TreeMap, and ArrayDeque objects (3.2)	Chapter 6
Use java.util.Comparator and java.lang.Comparable interfaces (3.3)	Chapter 6
Collections Streams and Filters (3.4)	Chapter 6
Iterate using forEach methods of Streams and List (3.5)	Chapters 8 and 9
Describe Stream interface and Stream pipeline (3.6)	Chapter 9
Filter a collection by using lambda expressions (3.7)	Chapter 9
Use method references with Streams (3.8)	Chapter 9

Lambda Built-in Functional Interfaces	
Use the built-in interfaces included in the <code>java.util.function</code> package such as <code>Predicate</code> , <code>Consumer</code> , <code>Function</code> , and <code>Supplier</code> (4.1)	Chapter 8
Develop code that uses primitive versions of functional interfaces (4.2)	Chapter 8
Develop code that uses binary versions of functional interfaces (4.3)	Chapter 8
Develop code that uses the <code>UnaryOperator</code> interface (4.4)	Chapter 8
Java Stream API	
Develop code to extract data from an object using <code>peek()</code> and <code>map()</code> methods including primitive versions of the <code>map()</code> method (5.1)	Chapter 9
Search for data by using search methods of the Stream classes including <code>findFirst</code> , <code>findAny</code> , <code>anyMatch</code> , <code>allMatch</code> , <code>noneMatch</code> (5.2)	Chapter 9
Develop code that uses the <code>Optional</code> class (5.3)	Chapter 9
Develop code that uses Stream data methods and calculation methods (5.4)	Chapter 9
Sort a collection using Stream API (5.5)	Chapter 9
Save results to a collection using the <code>collect</code> method and group/partition data using the <code>Collectors</code> class (5.6)	Chapter 9
Use <code>flatMap()</code> methods in the Stream API (5.7)	Chapter 9
Exceptions and Assertions	
Use try-catch and throw statements (6.1)	Chapter 3
Use catch, multi-catch, and finally clauses (6.2)	Chapter 3
Use Autoclose resources with a try-with-resources statement (6.3)	Chapter 3
Create custom exceptions and Autocloseable resources (6.4)	Chapter 3
Test invariants by using assertions (6.5)	Chapter 3
Use Java SE 8 Date/Time API	
Create and manage date-based and time-based events including a combination of date and time into a single object using <code>LocalDate</code> , <code>LocalTime</code> , <code>LocalDateTime</code> , <code>Instant</code> , <code>Period</code> , and <code>Duration</code> (7.1)	Chapter 4
Work with dates and times across timezones and manage changes resulting from daylight savings including <code>Format</code> date and times values (7.2)	Chapter 4
Define and create and manage date-based and time-based events using <code>Instant</code> , <code>Period</code> , <code>Duration</code> , and <code>TemporalUnit</code> (7.3)	Chapter 4
Java I/O Fundamentals	
Read and write data from the console (8.1)	Chapter 5
Use <code>BufferedReader</code> , <code>BufferedWriter</code> , <code>File</code> , <code>FileReader</code> , <code>FileWriter</code> , <code>FileInputStream</code> , <code>FileOutputStream</code> , <code>ObjectOutputStream</code> , <code>ObjectInputStream</code> , and <code>PrintWriter</code> in the <code>java.io</code> package (8.2)	Chapter 5

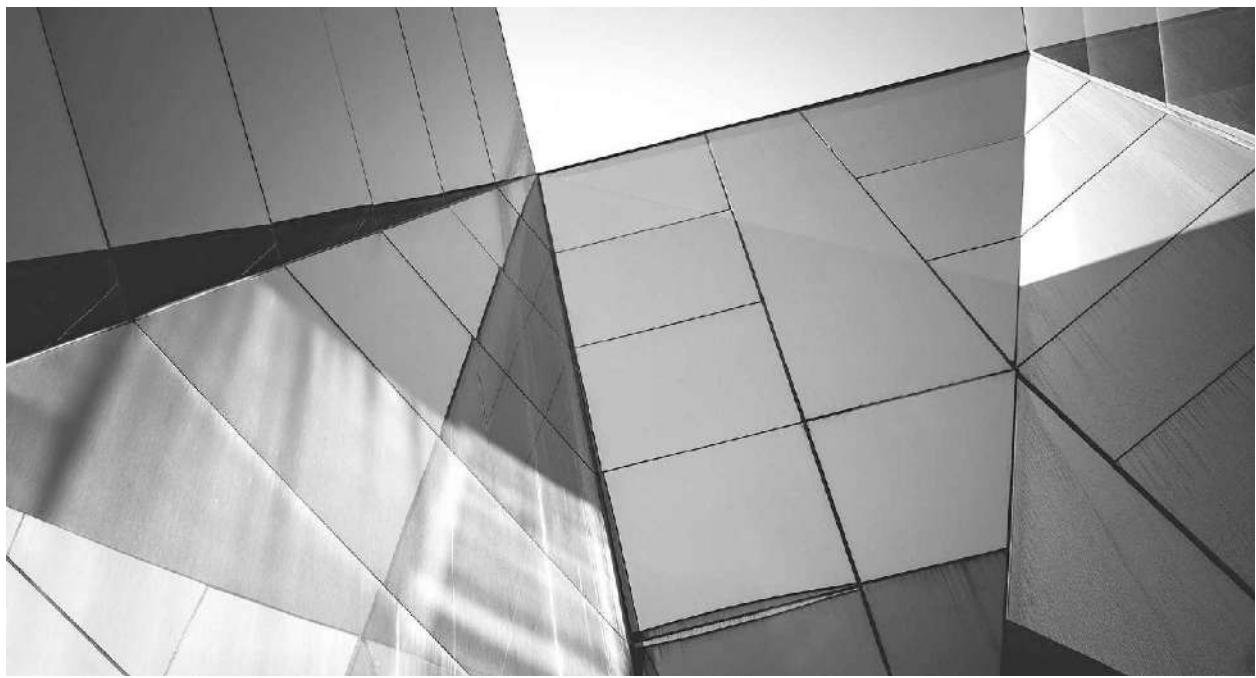
Java File I/O (NIO.2)	
Use Path interface to operate on file and directory paths (9.1)	Chapter 5
Use Files class to check, read, delete, copy, move, manage metadata of a file or directory (9.2)	Chapter 5
Use Stream API with NIO.2 (9.3)	Chapters 5, 9, and Online Appendix
Java Concurrency	
Create worker threads using Runnable, Callable and use an ExecutorService to concurrently execute tasks (10.1)	Chapters 10, 11
Identify potential threading problems among deadlock, starvation, livelock, and race conditions (10.2)	Chapter 10
Use synchronized keyword and java.util.concurrent.atomic package to control the order of thread execution (10.3)	Chapters 10, 11
Use java.util.concurrent collections and classes including CyclicBarrier and CopyOnWriteArrayList (10.4)	Chapter 11
Use parallel Fork/Join Framework (10.5)	Chapter 11
Use parallel Streams including reduction, decomposition, merging processes, pipelines and performance (10.6)	Chapter 11
Building Database Applications with JDBC	
Describe the interfaces that make up the core of the JDBC API including the Driver, Connection, Statement, and ResultSet interfaces and their relationships to provider implementations (11.1)	Chapter 12
Identify the components required to connect to a database using the DriverManager class including the JDBC URL (11.2)	Chapter 12
Submit queries and read results from the database including creating statements, returning result sets, iterating through the results, and properly closing result sets, statements, and connections (11.3)	Chapter 12
Localization	
Read and set the locale using the Locale object (12.1)	Chapter 4
Create and read a Properties file (12.2)	Chapter 4
Build a resource bundle for each locale and load a resource bundle in an application (12.3)	Chapter 4

Taking the Java SE 7 or Java 6 Upgrade Exam

For those of you who have your OCP 7 or SCJP 6 certification and want to take the upgrade exam to get your OCP 8 certification, this book contains everything you'll need to study. To be fair, there is a lot in this book that you won't need to study, but by comparing the upgrade objectives to the Table of Contents, you should be able to determine which chapters apply to you. There are a handful of objectives that do NOT overlap with the OCP 8 exam, and you can find coverage for those objectives in the included digital content. The digital content contains chapters that include coverage for

- Java 6, Objective 1.1, Develop code that uses String objects in the switch statement, binary literals, and numeric literals, including underscores in literals
- Java 7, Objective 6.2, Develop code that uses Java SE 8 I/O improvements, including Files.find(), Files.walk(), and lines() methods

Please refer to the Appendix for details on accessing the additional digital content.



1

Declarations, Access Control, and Enums

CERTIFICATION OBJECTIVES

- Declare Classes and Interfaces
 - Declare Class Members
 - Declare Constructors and Arrays
 - Create static Class Members
 - Use enums
-  Two-Minute Drill

Q&A Self Test

We assume that most of our readers have earned their OCA 8 Java certification and are reading this book in pursuit of the OCP 8 Java certification. If that's you, congratulations on earning your OCA 8! If you're NOT pursuing an Oracle Java certification, we'd like to think you might still find this book helpful. Oracle's certification exams are well regarded in the industry, and understanding the concepts in this book will make you a better Java programmer. But make no mistake, this book is REALLY focused on helping you pass the OCP 8.

Java Class Design and Object Orientation: A Refresher

If you compare the exam objectives of the OCA 8 exam and the OCP 8 exam, you'll notice some overlapping concepts. Specifically, many of the objectives in OCA 8 section 6 (Working with Methods and Encapsulation) and OCA 8 section

7 (Working with Inheritance) overlap heavily with the objectives in OCP 8 section 1 (Java Class Design) and OCP 8 section 2 (Advanced Java Class Design). This chapter is focused on most of those areas of conceptual overlap:

- 1.2 Inheritance, visibility, and composition (HAS-A)
- 1.6 The static keyword and init blocks
- 2.1 Abstract classes
- 2.2 The final keyword
- 2.4 Enums
- 2.5 Interfaces

You could consider this chapter a bit of a refresher from the OCA 8 exam, and if you feel really solid on the objectives listed above, you could consider skipping this chapter.

CERTIFICATION OBJECTIVE

Define Classes and Interfaces (OCP Objectives 1.2, 2.1, and 2.2)

- 1.2 *Implement inheritance including visibility modifiers and composition.*
- 2.1 *Develop code that uses abstract classes and methods.*
- 2.2 *Develop code that uses the final keyword.*

Class Declarations and Modifiers

The class declarations we'll discuss in this section are limited to top-level classes. In addition to top-level classes, Java provides for another category of class known as *nested classes* or *inner classes*. Inner classes will be covered in [Chapter 7](#). You're going to love learning about inner classes. No, really. Seriously.

The following code is a bare-bones class declaration:

```
class MyClass { }
```

This code compiles just fine, but you can also add modifiers before the class

declaration. In general, modifiers fall into two categories:

- Access modifiers (public, protected, private)
- Nonaccess modifiers (including strictfp, final, and abstract)

We'll look at access modifiers first, so you'll learn how to restrict or allow access to a class you create. Access control in Java is a little tricky because there are four access *controls* (levels of access) but only three access *modifiers*. The fourth access control level (called *default* or *package* access) is what you get when you don't use any of the three access modifiers. In other words, *every* class, method, constructor, and instance variable you declare has an access *control*, whether you explicitly type one or not. Although all four access *controls* (which means all three *modifiers*) work for most method and variable declarations, a class can be declared with only public or *default* access; the other two access control levels don't make sense for a class, as you'll see.



Java is a package-centric language; the developers assumed that for good organization and name scoping, you would put all your classes into packages. They were right, and you should. Imagine this nightmare: Three different programmers, in the same company but working on different parts of a project, write a class named Utilities. If those three Utilities classes have not been declared in any explicit package and are in the classpath, you won't have any way to tell the compiler or JVM which of the three you're trying to reference. Oracle recommends that developers use reverse domain names, appended with division and/or project names. For example, if your domain name is geeksanonymous.com and you're working on the client code for the TwelvePointOSteps program, you would name your package something like com.geeksanonymous.steps.client. That would essentially change the name of your class to com.geeksanonymous.steps.client.Utilities. You might still have name collisions within your company if you don't come up with your own naming schemes, but you're guaranteed not to collide with classes developed outside your company (assuming they follow Oracle's naming convention, and if they don't, well, Really Bad Things could happen).

Class Access

What does it mean to access a class? When we say code from one class (class A) has access to another class (class B), it means class A can do one of three things:

- Create an *instance* of class B.
- *Extend* class B (in other words, become a subclass of class B).
- Access certain methods and variables within class B, depending on the access control of those methods and variables.

In effect, access means *visibility*. If class A can't *see* class B, the access level of the methods and variables within class B won't matter; class A won't have any way to access those methods and variables.

Default Access A class with default access has *no* modifier preceding it in the declaration! It's the access control you get when you don't type a modifier in the class declaration. Think of *default* access as *package-level* access, because a class with default access can be seen only by classes within the same package. For example, if class A and class B are in different packages, and class A has default access, class B won't be able to create an instance of class A or even declare a variable or return type of class A. In fact, class B has to pretend that class A doesn't even exist, or the compiler will complain. Look at the following source file:

```
package cert;  
class Beverage { }
```

Now look at the second source file:

```
package exam.stuff;  
import cert.Beverage;  
class Tea extends Beverage { }
```

As you can see, the superclass (`Beverage`) is in a different package from the subclass (`Tea`). The `import` statement at the top of the `Tea` file is trying (fingers crossed) to import the `Beverage` class. The `Beverage` file compiles fine, but when we try to compile the `Tea` file, we get *something like* this:

Can't access class cert.Beverage. Class or interface must be public, in same package, or an accessible member class.

```
import cert.Beverage;
```

Note: For various reasons, the error messages we show throughout this book might not match the error messages you get. Don't worry, the real point is to understand when you're apt to get an error of some sort.

Tea won't compile because its superclass, Beverage, has default access and is in a different package. You can do one of two things to make this work. You could put both classes in the same package, or you could declare Beverage as public, as the next section describes.

When you see a question with complex logic, be sure to look at the access modifiers first. That way, if you spot an access violation (for example, a class in package A trying to access a default class in package B), you'll know the code won't compile so you don't have to bother working through the logic. It's not as if you don't have anything better to do with your time while taking the exam. Just choose the "Compilation fails" answer and zoom on to the next question.

Public Access

A class declaration with the `public` keyword gives all classes from all packages access to the `public` class. In other words, *all* classes in the Java Universe (JU) have access to a `public` class. Don't forget, though, that if a `public` class you're trying to use is in a different package from the class you're writing, you'll still need to import the `public` class or use the fully qualified name.

In the example from the preceding section, we may not want to place the subclass in the same package as the superclass. To make the code work, we need to add the keyword `public` in front of the superclass (`Beverage`) declaration, as follows:

```
package cert;  
public class Beverage { }
```

This changes the `Beverage` class so it will be visible to all classes in all packages. The class can now be instantiated from all other classes, and any class is now free to subclass (extend from) it—unless, that is, the class is also marked with the nonaccess modifier `final`. Read on.

Other (Nonaccess) Class Modifiers

You can modify a class declaration using the keyword `final`, `abstract`, or `strictfp`. These modifiers are in addition to whatever access control is on the class, so you could, for example, declare a class as both `public` and `final`. But you can't always mix nonaccess modifiers. You're free to use `strictfp` in combination with `final`, for example, but you must never, ever, ever mark a class as both `final` and `abstract`. You'll see why in the next two sections.

You won't need to know how `strictfp` works, so we're focusing only on modifying a class as `final` or `abstract`. For the exam, you need to know only that `strictfp` is a keyword and can be used to modify a class or a method, but never a variable. Marking a class as `strictfp` means that any method code in the class will conform strictly to the IEEE 754 standard rules for floating points. Without that modifier, floating points used in the methods might behave in a platform-dependent way. If you don't declare a class as `strictfp`, you can still get `strictfp` behavior on a method-by-method basis by declaring a method as `strictfp`. If you don't know the IEEE 754 standard, now's not the time to learn it. You have, as they say, bigger fish to fry.

Final Classes

When used in a class declaration, the `final` keyword means the class can't be subclassed. In other words, no other class can ever extend (inherit from) a `final` class, and any attempts to do so will result in a compiler error.

So why would you ever mark a class `final`? After all, doesn't that violate the whole OO notion of inheritance? You should make a `final` class only if you need an absolute guarantee that none of the methods in that class will ever be overridden. If you're deeply dependent on the implementations of certain methods, then using `final` gives you the security that nobody can change the implementation out from under you.

You'll notice many classes in the Java core libraries are `final`. For example, the `String` class cannot be subclassed. Imagine the havoc if you couldn't guarantee how a `String` object would work on any given system your application is running on! If programmers were free to extend the `String` class (and thus substitute their new `String` subclass instances where `java.lang.String` instances are expected), civilization—as we know it—could collapse. So use `final` for safety, but only when you're certain that your `final` class has, indeed, said all that ever needs to be said in its methods. Marking a class `final` means, in essence, your class can't ever be improved upon, or even

specialized, by another programmer.

There's a benefit to having nonfinal classes in this scenario: Imagine that you find a problem with a method in a class you're using, but you don't have the source code. So you can't modify the source to improve the method, but you can extend the class and override the method in your new subclass and substitute the subclass everywhere the original superclass is expected. If the class is `final`, though, you're stuck.

Let's modify our `Beverage` example by placing the keyword `final` in the declaration:

```
package cert;
public final class Beverage {
    public void importantMethod() { }
}
```

Now let's try to compile the `Tea` subclass:

```
package exam.stuff;
import cert.Beverage;
class Tea extends Beverage { }
```

We get an error—something like this:

```
Can't subclass final classes: class
cert.Beverage class Tea extends Beverage{
1 error
```

In practice, you'll almost never make a final class. A final class obliterates a key benefit of OO—extensibility. Unless you have a serious safety or security issue, assume that someday another programmer will need to extend your class. If you don't, the next programmer forced to maintain your code will hunt you down and <insert really scary thing>.

Abstract Classes An abstract class can never be instantiated. Its sole purpose,

mission in life, *raison d'être*, is to be extended (subclassed). (Note, however, that you can compile and execute an abstract class, as long as you don't try to make an instance of it.) Why make a class if you can't make objects out of it? Because the class might be just too, well, *abstract*. For example, imagine you have a class `Car` that has generic methods common to all vehicles. But you don't want anyone actually creating a generic abstract `Car` object. How would they initialize its state? What color would it be? How many seats? Horsepower? All-wheel drive? Or more importantly, how would it behave? In other words, how would the methods be implemented?

No, you need programmers to instantiate actual car types such as `BMWBoxster` and `SubaruOutback`. We'll bet the Boxster owner will tell you his car does things the Subaru can do "only in its dreams." Take a look at the following abstract class:

```
abstract class Car {  
    private double price;  
    private String model;  
    private String year;  
    public abstract void goFast();  
    public abstract void goUpHill();  
    public abstract void impressNeighbors();  
    // Additional, important, and serious code goes here  
}
```

The preceding code will compile fine. However, if you try to instantiate a `Car` in another body of code, you'll get a compiler error, something like this:

```
AnotherClass.java:7: class Car is an abstract  
class. It can't be instantiated.
```

```
    Car x = new Car();  
1 error
```

Notice that the methods marked abstract end in a semicolon rather than curly braces.

Look for questions with a method declaration that ends with a semicolon, rather than curly braces. If the method is in a class—as opposed to an interface—then both the method and the class must be marked abstract. You might get a question that asks how you could fix a code sample that includes a method ending in a semicolon, but without an `abstract` modifier on the class or method. In that case, you could either mark the method and class `abstract` or change the semicolon to code (like a curly brace pair). Remember if you change a method from `abstract` to non`abstract`, don’t forget to change the semicolon at the end of the method declaration into a curly brace pair!

We’ll look at `abstract` methods in more detail later in this objective, but always remember that if even a single method is `abstract`, the whole class must be declared `abstract`. One `abstract` method spoils the whole bunch. You can, however, put non`abstract` methods in an `abstract` class. For example, you might have methods with implementations that shouldn’t change from `Car` type to `Car` type, such as `getColor()` or `setPrice()`. By putting non`abstract` methods in an `abstract` class, you give all concrete subclasses (*concrete* just means *not abstract*) inherited method implementations. The good news there is that concrete subclasses get to inherit functionality and need to implement only the methods that define subclass-specific behavior.

(By the way, if you think we misused *raison d’être* earlier, don’t send an e-mail. We’d like to see you work it into a programmer certification book.)

Coding with `abstract` class types (including interfaces, discussed later in this chapter) lets you take advantage of *polymorphism* and gives you the greatest degree of flexibility and extensibility. You’ll learn more about polymorphism in [Chapter 2](#).

You can’t mark a class as both `abstract` and `final`. They have nearly opposite meanings. An `abstract` class must be subclassed, whereas a `final` class must not be subclassed. If you see this combination of `abstract` and `final` modifiers used for a class or method declaration, the code will not compile.

EXERCISE 1-1

Creating an Abstract Superclass and Concrete Subclass

The following exercise will test your knowledge of public, default, final, and abstract classes. Create an abstract superclass named `Fruit` and a concrete subclass named `Apple`. The superclass should belong to a package called `food`, and the subclass can belong to the default package (meaning it isn't put into a package explicitly). Make the superclass `public` and give the subclass `default` access.

1. Create the superclass as follows:

```
package food;  
public abstract class Fruit{ /* any code you want */}
```

2. Create the subclass in a separate file as follows:

```
import food.Fruit;  
class Apple extends Fruit{ /* any code you want */}
```

3. Create a directory called `food` off the directory in your classpath setting.
 4. Attempt to compile the two files. If you want to use the `Apple` class, make sure you place the `Fruit.class` file in the `food` subdirectory.
-

CERTIFICATION OBJECTIVE

Use Interfaces (OCP Objective 2.5)

2.5 *Develop code that declares, implements, and/or extends interfaces and use the @Override annotation.*

Declaring an Interface

In general, when you create an interface, you're defining a contract for *what* a class can do, without saying anything about *how* the class will do it.

Note: As of Java 8, you can now also describe the *how*, but you usually won't. Until we get to Java 8's new interface-related features—default** and **static** methods—we will discuss interfaces from a traditional perspective, which is, again, defining a contract for *what* a class can do.**

An interface is a contract. You could write an interface `Bounceable`, for example, that says in effect, "This is the `Bounceable` interface. Any concrete

class type that implements this interface must agree to write the code for the `bounce()` and `setBounceFactor()` methods.“

By defining an interface for `Bounceable`, any class that wants to be treated as a `Bounceable` thing can simply implement the `Bounceable` interface and provide code for the interface’s two methods.

Interfaces can be implemented by any class and from any inheritance tree. This lets you take radically different classes and give them a common characteristic. For example, you might want both a `Ball` and a `Tire` to have bounce behavior, but `Ball` and `Tire` don’t share any inheritance relationship; `Ball` extends `Toy` whereas `Tire` extends only `java.lang.Object`. But by making both `Ball` and `Tire` implement `Bounceable`, you’re saying that `Ball` and `Tire` can be treated as “Things that can bounce,” which in Java translates to “Things on which you can invoke the `bounce()` and `setBounceFactor()` methods.” [Figure 1-1](#) illustrates the relationship between interfaces and classes.

FIGURE 1-1

The relationship between interfaces and classes

```
interface Bounceable
```

```
void bounce( );
void setBounceFactor(int bf);
```

What you declare.

```
interface Bounceable
```

```
public abstract void bounce( );
public abstract void setBounceFactor(int bf);
```

What the compiler sees.

```
Class Tire implements Bounceable
public void bounce( ){...}
public void setBounceFactor(int bf){ }
```

What the implementing class must do.

(All interface methods must be implemented and must be marked public.)

Think of an interface as a 100-percent abstract class. Like an abstract class, an interface defines abstract methods that take the following form:

```
abstract void bounce(); // Ends with a semicolon rather than  
// curly braces
```

But although an abstract class can define both abstract and nonabstract methods, an interface *generally* has only abstract methods. Another way interfaces differ from abstract classes is that interfaces have very little flexibility in how the methods and variables defined in the interface are declared. These rules are strict:

- All interface methods are implicitly `public`. Unless declared as `default` or `static`, they are also implicitly abstract. In other words, you do not need to actually type the `public` or `abstract` modifiers in the method declaration, but the method is still always `public` and `abstract`.
- All variables defined in an interface must be `public`, `static`, and `final` —in other words, interfaces can declare only constants, not instance variables.
- Interface methods cannot be marked `final`, `strictfp`, or `native`. (More on these modifiers later in the chapter.)
- An interface can *extend* one or more other interfaces.
- An interface cannot extend anything but another interface.
- An interface cannot implement another interface or class.
- An interface must be declared with the keyword `interface`.
- Interface types can be used polymorphically (see [Chapter 2](#) for more details).

The following is a legal interface declaration:

```
public abstract interface Rollable { }
```

Typing in the `abstract` modifier is considered redundant; interfaces are implicitly abstract whether you type `abstract` or not. You just need to know that both of these declarations are legal and functionally identical:

```
public abstract interface Rollable { }  
public interface Rollable { }
```

The `public` modifier is required if you want the interface to have `public` rather than default access.

We've looked at the interface declaration, but now we'll look closely at the methods within an interface:

```
public interface Bounceable {  
    public abstract void bounce();  
    public abstract void setBounceFactor(int bf);  
}
```

Typing in the `public` and `abstract` modifiers on the methods is redundant, though, because all interface methods are implicitly `public` and `abstract`. Given that rule, you can see the following code is exactly equivalent to the preceding interface:

```
public interface Bounceable {  
    void bounce(); // No modifiers  
    void setBounceFactor(int bf); // No modifiers  
}
```

You must remember that all interface methods not declared `default` or `static` are `public` and `abstract` regardless of what you see in the interface definition.

Look for interface methods declared with any combination of `public`, `abstract`, or no modifiers. For example, the following five method declarations, if declared within their own interfaces, are legal and identical!

```
void bounce();  
public void bounce();  
abstract void bounce();  
public abstract void bounce();  
abstract public void bounce();
```

The following interface method declarations won't compile:

```
final void bounce();      // final and abstract can never be used
                         // together, and abstract is implied
private void bounce();    // interface methods are always public
protected void bounce();  // (same as above)
```

Declaring Interface Constants

You're allowed to put constants in an interface. By doing so, you guarantee that any class implementing the interface will have access to the same constant. By placing the constants right in the interface, any class that implements the interface has direct access to the constants, just as if the class had inherited them.

You need to remember one key rule for interface constants. They must always be

```
public static final
```

So that sounds simple, right? After all, interface constants are no different from any other publicly accessible constants, so they obviously must be declared `public`, `static`, and `final`. But before you breeze past the rest of this discussion, think about the implications: **Because interface constants are defined in an interface, they don't have to be *declared* as `public`, `static`, or `final`. They must be `public`, `static`, and `final`, but you don't actually have to declare them that way.** Just as interface methods are always public and abstract whether you say so in the code or not, any variable defined in an interface must be—and implicitly is—a public constant. See if you can spot the problem with the following code (assume two separate files):

```
interface Foo {  
    int BAR = 42;  
    void go();  
}  
  
class Zap implements Foo {  
    public void go() {  
        BAR = 27;  
    }  
}
```

You can't change the value of a constant! Once the value has been assigned, the value can never be modified. The assignment happens in the interface itself (where the constant is declared), so the implementing class can access it and use it, but as a read-only value. So the `BAR = 27` assignment will not compile.



Look for interface definitions that define constants, but without explicitly using the required modifiers. For example, the following are all identical:

```
public int x = 1;           // Looks non-static and non-final,  
                           // but isn't!  
  
int x = 1;                 // Looks default, non-final,  
                           // non-static, but isn't!  
  
static int x = 1;           // Doesn't show final or public  
  
final int x = 1;            // Doesn't show static or public  
  
public static int x = 1;     // Doesn't show final  
  
public final int x = 1;      // Doesn't show static  
  
static final int x = 1;       // Doesn't show public  
  
public static final int x = 1; // what you get implicitly
```

Any combination of the required (but implicit) modifiers is legal, as is using no modifiers at all! On the exam, you can expect to see questions you won't be able to answer correctly unless you know, for example, that an interface variable is final and can never be given a value by the implementing (or any other) class.

Declaring default Interface Methods

As of Java 8, interfaces can include inheritable* methods with concrete implementations. (*The strict definition of “inheritance” has gotten a little fuzzy with Java 8; we’ll talk more about inheritance in the next chapter.) These concrete methods are called **default** methods. Later in the book (mostly in [Chapter 2](#)), we’ll talk a lot about the various OO-related rules that are impacted because of **default** methods. For now, we’ll just cover the simple declaration rules:

- **default** methods are declared by using the **default** keyword. The **default** keyword can be used only with interface method signatures, not class method signatures.
- **default** methods are **public** by definition, and the **public** modifier is optional.
- **default** methods **cannot** be marked as **private**, **protected**, **static**,

`final`, or `abstract`.

- `default` methods must have a concrete method body.

Here are some examples of legal and illegal `default` methods:

```
interface TestDefault {  
    default int m1(){ return 1; }          // legal  
    public default void m2(){ ; }          // legal  
    // static default void m3(){ ; }        // illegal: default cannot be marked static  
    // default void m4();                  // illegal: default must have a method body  
}
```

Declaring static Interface Methods

As of Java 8, interfaces can include `static` methods with concrete implementations. As with interface `default` methods, there are OO implications that we'll discuss later in the chapter.

For now, we'll focus on the basics of declaring and using `static` interface methods:

- `static` interface methods are declared by using the `static` keyword.
- `static` interface methods are `public`, by default, and the `public` modifier is optional.
- `static` interface methods cannot be marked as `private`, `protected`, `final`, or `abstract`.
- `static` interface methods must have a concrete method body.
- When invoking a `static` interface method, the method's type (interface name) **MUST** be included in the invocation.

Here are some examples of legal and illegal `static` interface methods and their use:

```
interface StaticIface {
    static int m1(){ return 42; }      // legal
    public static void m2(){ ; }       // legal
    // final static void m3(){ ; }     // illegal: final not allowed
    // abstract static void m4(){ ; }   // illegal: abstract not allowed
    // static void m5();              // illegal: needs a method body
}

public class TestSIF implements StaticIface {
    public static void main(String[] args) {
        System.out.println(StaticIface.m1()); // legal: m1()'s type
                                            // must be included
        new TestSIF().go();
        // System.out.println(m1()); // illegal: reference to interface
                                    // is required
    }
    void go() {
        System.out.println(StaticIface.m1()); // also legal from an instance
    }
}
```

which produces this output:

```
42
42
```

Later, we'll return to our discussion of default methods and static methods for interfaces.

CERTIFICATION OBJECTIVE

Declare Class Members (OCP Objectives 1.2, 1.6, 2.1, and 2.2)

1.2 *Implement inheritance including visibility modifiers and composition.*

1.6 *Develop code that uses static keyword on initialize blocks, variables, methods, and classes.*

2.1 *Develop code that uses abstract classes and methods.*

2.2 *Develop code that uses the final keyword.*

We've looked at what it means to use a modifier in a class declaration, and now we'll look at what it means to modify a method or variable declaration.

Methods and instance (nonlocal) variables are collectively known as *members*. You can modify a member with both access and nonaccess modifiers, and you have more modifiers to choose from (and combine) than when you're declaring a class.

Access Modifiers

Because method and variable members are usually given access control in exactly the same way, we'll cover both in this section.

Whereas a *class* can use just two of the four access control levels (default or public), members can use all four:

- public
- protected
- default
- private

Default protection is what you get when you don't type an access modifier in the member declaration. The default and protected access control types have almost identical behavior, except for one difference that we will mention later.

Note: As of Java 8, the word default can ALSO be used to declare certain methods in interfaces. When used in an interface's method declaration, default has a different meaning than what we are describing in this section of this chapter.

It's crucial that you know access control inside and outside for the exam. There will be quite a few questions in which access control plays a role. Some

questions test several concepts of access control at the same time, so not knowing one small part of access control could mean you blow an entire question.

What does it mean for code in one class to have access to a member of another class? For now, ignore any differences between methods and variables. If class A has access to a member of class B, it means that class B's member is visible to class A. When a class does not have access to another member, the compiler will slap you for trying to access something that you're not even supposed to know exists!

You need to understand two different access issues:

- Whether method code in one class can *access* a member of another class
- Whether a subclass can *inherit* a member of its superclass

The first type of access occurs when a method in one class tries to access a method or a variable of another class, using the dot operator (.) to invoke a method or retrieve a variable. For example:

```
class Zoo {  
    public String coolMethod() {  
        return "Wow baby";  
    }  
}  
class Moo {  
    public void useAZoo() {  
        Zoo z = new Zoo();  
        // If the preceding line compiles Moo has access  
        // to the Zoo class  
        // But... does it have access to the coolMethod()  
        System.out.println("A Zoo says, " + z.coolMethod());  
        // The preceding line works because Moo can access the  
        // public method  
    }  
}
```

The second type of access revolves around which, if any, members of a superclass a subclass can access through inheritance. We're not looking at whether the subclass can, say, invoke a method on an instance of the superclass (which would just be an example of the first type of access). Instead, we're looking at whether the subclass *inherits* a member of its superclass. Remember, if a subclass *inherits* a member, it's exactly as if the subclass actually declared the member itself. In other words, if a subclass *inherits* a member, the subclass *has* the member. Here's an example:

```

class Zoo {
    public String coolMethod() {
        return "Wow baby";
    }
}

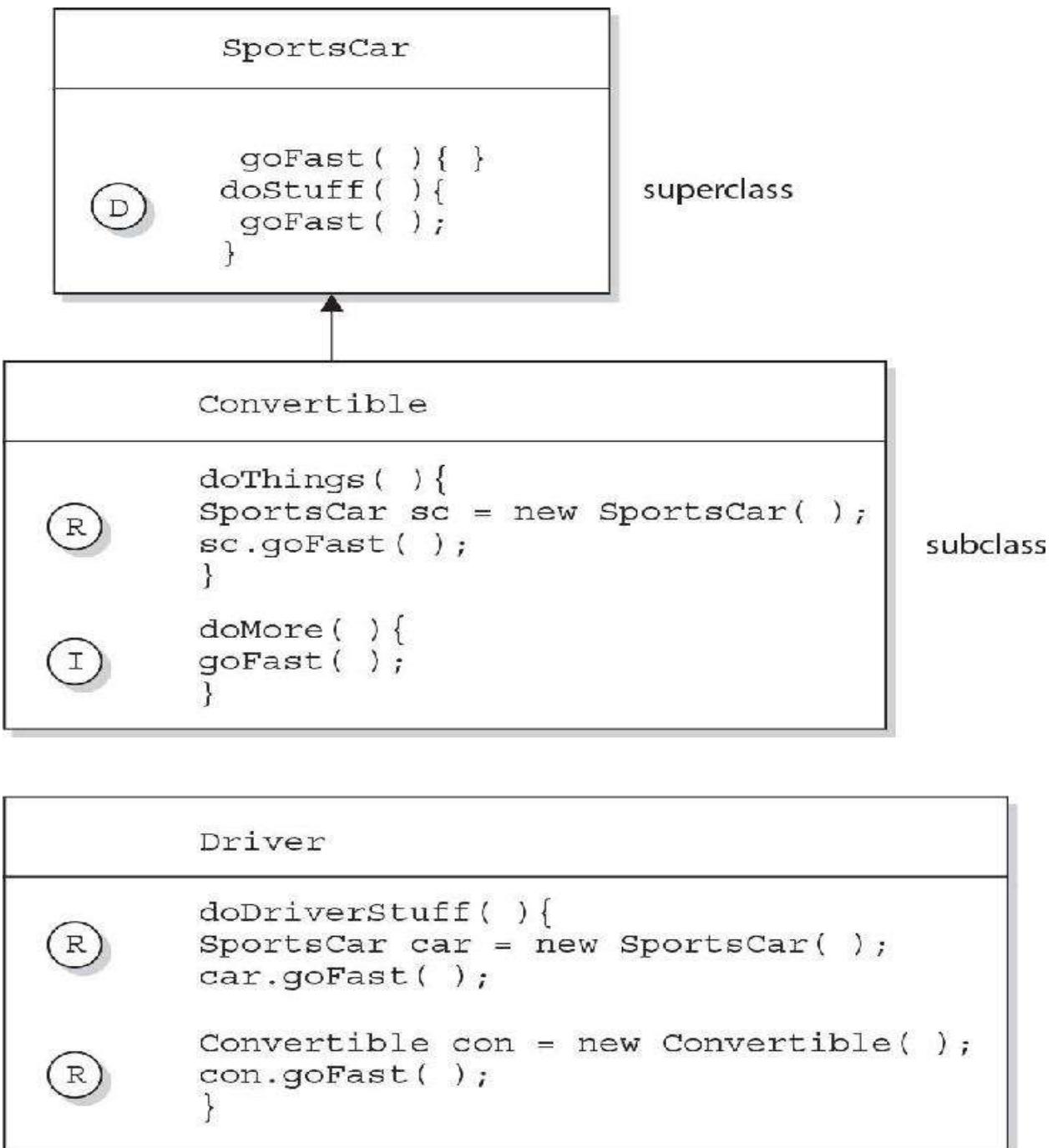
class Moo extends Zoo {
    public void useMyCoolMethod() {
        // Does an instance of Moo inherit the coolMethod()?
        System.out.println("Moo says, " + this.coolMethod());
        // The preceding line works because Moo can inherit the
        // public method
        // Can an instance of Moo invoke coolMethod() on an
        // instance of Zoo?
        Zoo z = new Zoo();
        System.out.println("Zoo says, " + z.coolMethod());
        // coolMethod() is public, so Moo can invoke it on a Zoo
        // reference
    }
}

```

[Figure 1-2](#) compares a class inheriting a member of another class and accessing a member of another class using a reference of an instance of that class.

FIGURE 1-2

Comparison of inheritance vs. dot operator for member access



Three ways to access a method:

- (D) Invoking a method declared in the same class
- (R) Invoking a method using a reference of the class
- (I) Invoking an inherited method

Much of access control (both types) centers on whether the two classes involved are in the same or different packages. Don't forget, though, that if class A *itself* can't be accessed by class B, then no members within class A can be accessed by class B.

You need to know the effect of different combinations of class and member access (such as a default class with a public variable). To figure this out, first look at the access level of the class. If the class itself will not be visible to another class, then none of the members will be visible either, even if the member is declared `public`. Once you've confirmed that the class is visible, then it makes sense to look at access levels on individual members.

Public Members

When a method or variable member is declared `public`, it means all other classes, regardless of the package they belong to, can access the member (assuming the class itself is visible).

Look at the following source file:

```
package book;
import cert.*; // Import all classes in the cert package
class Goo {
    public static void main(String[] args) {
        Sludge o = new Sludge();
        o.testIt();
    }
}
```

Now look at the second file:

```
package cert;
public class Sludge {
    public void testIt() { System.out.println("sludge"); }
}
```

As you can see, Goo and Sludge are in different packages. However, Goo can invoke the method in Sludge without problems because both the Sludge class and its testIt() method are marked public.

For a subclass, if a member of its superclass is declared public, the subclass inherits that member regardless of whether both classes are in the same package:

```
package cert;
public class Roo {
    public String doRooThings() {
        // imagine the fun code that goes here
        return "fun";
    }
}
```

The Roo class declares the doRooThings() member as public. So if we make a subclass of Roo, any code in that Roo subclass can call its own inherited doRooThings() method.

Notice in the following code that the doRooThings() method is invoked without having to preface it with a reference:

```
package notcert;    // Not the package Roo is in
import cert.Roo;
class Cloo extends Roo {
    public void testCloo() {
        System.out.println(doRooThings());
    }
}
```

Remember, if you see a method invoked (or a variable accessed) without the dot operator (.), it means the method or variable belongs to the class where you see that code. It also means that the method or variable is implicitly being accessed using the this reference. So in the preceding code, the call to

`doRooThings()` in the `Cloo` class could also have been written as `this.doRooThings()`. The reference `this` always refers to the currently executing object—in other words, the object running the code where you see the `this` reference. Because the `this` reference is implicit, you don’t need to preface your member access code with it, but it won’t hurt. Some programmers include it to make the code easier to read for new (or non) Java programmers.

Besides being able to invoke the `doRooThings()` method on itself, code from some other class can call `doRooThings()` on a `Cloo` instance, as in the following:

```
package notcert;
class Toon {
    public static void main(String[] args) {
        Cloo c = new Cloo();
        System.out.println(c.doRooThings()); // No problem; method
                                            // is public
    }
}
```

Private Members

Members marked `private` can’t be accessed by code in any class other than the class in which the `private` member was declared. Let’s make a small change to the `Roo` class from an earlier example:

```
package cert;
public class Roo {
    private String doRooThings() {
        // imagine the fun code that goes here, but only the Roo
        // class knows
        return "fun";
    }
}
```

The `doRooThings()` method is now `private`, so no other class can use it. If we try to invoke the method from any other class, we'll run into trouble:

```
package notcert;
import cert.Roo;
class UseARoo {
    public void testIt() {
        Roo r = new Roo(); //So far so good; class Roo is public
        System.out.println(r.doRooThings()); // Compiler error!
    }
}
```

If we try to compile `UseARoo`, we get a compiler error, something like this:

```
cannot find symbol
symbol  : method doRooThings()
```

It's as if the method `doRooThings()` doesn't exist, and as far as any code outside of the `Roo` class is concerned, this is true. A `private` member is invisible to any code outside the member's own class.

What about a subclass that tries to inherit a `private` member of its

superclass? When a member is declared `private`, a subclass can't inherit it. For the exam, you need to recognize that a subclass can't see, use, or even think about the `private` members of its superclass. You can, however, declare a matching method in the subclass. But regardless of how it looks, ***it is not an overriding method!*** It is simply a method that happens to have the same name as a `private` method (which you're not supposed to know about) in the superclass. The rules of overriding do not apply, so you can make this newly declared-but-just-happens-to-match method declare new exceptions, or change the return type, or do anything else you want it to do.

```
package cert;
public class Roo {
    private String doRooThings() { // do fun, secret stuff
        return "fun";
    }
}
```

The `doRooThings()` method is now off limits to all subclasses, even those in the same package as the superclass:

```
package cert;                                // Cloo and Roo are in the same package
class Cloo extends Roo {                      // Still OK, superclass Roo is public
    public void testCloo() {
        System.out.println(doRooThings()); // Compiler error!
    }
}
```

If we try to compile the subclass `Cloo`, the compiler is delighted to spit out an error, something like this:

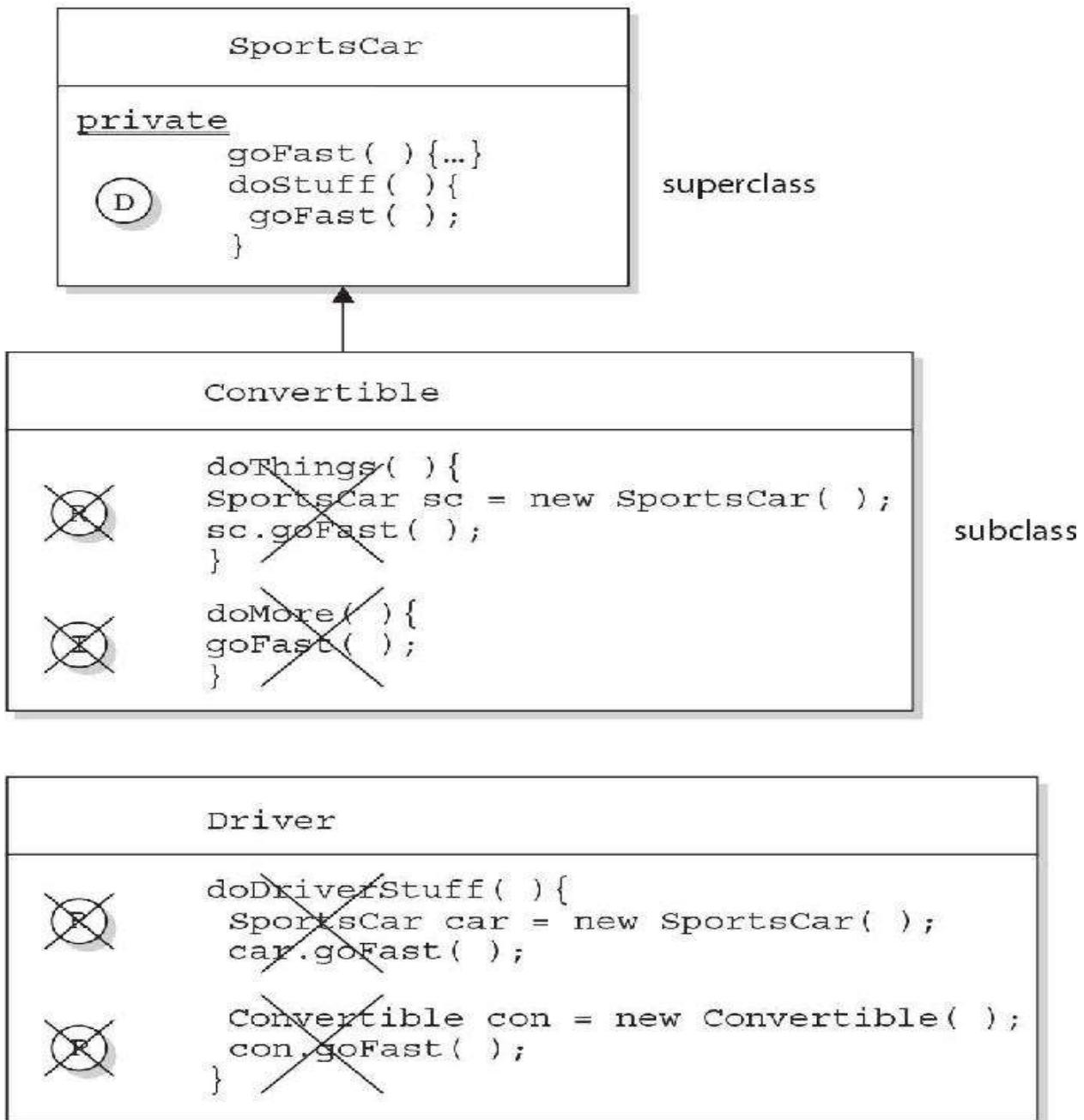
```
%javac Cloo.java  
Cloo.java:4: Undefined method: doRooThings()  
    System.out.println(doRooThings());  
1 error
```

Can a `private` method be overridden by a subclass? That's an interesting question, but the answer is no. Because the subclass, as we've seen, cannot inherit a `private` method, it, therefore, cannot override the method—overriding depends on inheritance. We'll cover the implications of this in more detail a little later in this chapter, but for now, just remember that a method marked `private` cannot be overridden. [Figure 1-3](#) illustrates the effects of the `public` and `private` modifiers on classes from the same or different packages.

FIGURE 1-3

Effects of public and private access

The effect of private access control



Three ways to access a method:

- D** Invoking a method declared in the same class
- R** Invoking a method using a reference of the class
- I** Invoking an inherited method

Protected and Default Members

Note: Just a reminder, in the next several sections, when we use the word “default,” we’re talking about access control. We’re NOT talking about the new kind of Java 8 interface method that can be declared default.

The protected and default access control levels are almost identical, but with one critical difference: a *default* member may be accessed only if the class accessing the member belongs to the same package, whereas a *protected* member can be accessed (through inheritance) by a subclass ***even if the subclass is in a different package***. Take a look at the following two classes:

```
package certification;
public class OtherClass {
    void testIt() {    // No modifier means method has default
                      // access
        System.out.println("OtherClass");
    }
}
```

In another source code file, you have the following:

```
package somethingElse;
import certification.OtherClass;
class AccessClass {
    static public void main(String[] args) {
        OtherClass o = new OtherClass();
        o.testIt();
    }
}
```

As you can see, the `testIt()` method in the first file has *default* (think

package-level) access. Notice also that class `OtherClass` is in a different package from the `AccessClass`. Will `AccessClass` be able to use the method `testIt()`? Will it cause a compiler error? Will Daniel ever marry Francesca? Stay tuned.

```
No method matching testIt() found in class
certification.OtherClass.    o.testIt();
```

From the preceding results, you can see that `AccessClass` can't use the `OtherClass` method `testIt()` because `testIt()` has default access and `AccessClass` is not in the same package as `OtherClass`. So `AccessClass` can't see it, the compiler complains, and we have no idea who Daniel and Francesca are.

Default and protected behavior differ only when we talk about subclasses. If the `protected` keyword is used to define a member, any subclass of the class declaring the member can access it *through inheritance*. It doesn't matter if the superclass and subclass are in different packages; the `protected` superclass member is still visible to the subclass (although visible only in a very specific way, as you'll see a little later). This is in contrast to the default behavior, which doesn't allow a subclass to access a superclass member unless the subclass is in the same package as the superclass.

Whereas default access doesn't extend any special consideration to subclasses (you're either in the package or you're not), the `protected` modifier respects the parent-child relationship, even when the child class moves away (and joins a new package). So when you think of *default* access, think *package* restriction. No exceptions. But when you think `protected`, think *package + kids*. A class with a `protected` member is marking that member as having package-level access for all classes, but with a special exception for subclasses outside the package.

But what does it mean for a subclass-outside-the-package to have access to a superclass (parent) member? It means the subclass inherits the member. It does not, however, mean the subclass-outside-the-package can access the member using a reference to an instance of the superclass. In other words, `protected` = inheritance. `Protected` does not mean the subclass can treat the `protected` superclass member as though it were public. So if the subclass-outside-the-package gets a reference to the superclass (by, for example, creating an instance of the superclass somewhere in the subclass's code), the subclass cannot use the dot operator on the superclass reference to access the `protected` member. To a

subclass-outside-the-package, a protected member might as well be default (or even private), when the subclass is using a reference to the superclass. **The subclass can see the protected member only through inheritance.**

Are you confused? Hang in there and it will all become clearer with the next batch of code examples.

Protected Details

Let's take a look at a protected instance variable (remember, an instance variable is a member) of a superclass.

```
package certification;
public class Parent {
    protected int x = 9; // protected access
}
```

The preceding code declares the variable `x` as protected. This makes the variable *accessible* to all other classes *inside* the certification package, as well as *inheritable* by any subclasses *outside* the package.

Now let's create a subclass in a different package and attempt to use the variable `x` (that the subclass inherits):

```
package other;                                // Different package
import certification.Parent;
class Child extends Parent {
    public void testIt() {
        System.out.println("x is " + x); // No problem; Child
                                         // inherits x
    }
}
```

The preceding code compiles fine. Notice, though, that the `Child` class is accessing the protected variable through inheritance. Remember, any time we

talk about a subclass having access to a superclass member, we could be talking about the subclass inheriting the member, not simply accessing the member through a reference to an instance of the superclass (the way any other nonsubclass would access it). Watch what happens if the subclass `Child` (outside the superclass's package) tries to access a `protected` variable using a `Parent` class reference:

```
package other;
import certification.Parent;
class Child extends Parent {
    public void testIt() {
        System.out.println("x is " + x);           // No problem; Child
                                                    // inherits x
        Parent p = new Parent();                  // Can we access x using
                                                    // the p reference?
        System.out.println("x in parent is " + p.x); // Compiler error!
    }
}
```

The compiler is more than happy to show us the problem:

```
%javac -d . other/Child.java
other/Child.java:9: x has protected access in certification.Parent
System.out.println("x in parent is " + p.x);
^
1 error
```

So far, we've established that a `protected` member has essentially package-level or default access to all classes except for subclasses. We've seen that subclasses outside the package can inherit a `protected` member. Finally, we've seen that subclasses outside the package can't use a superclass reference to access a `protected` member. ***For a subclass outside the package, the protected***

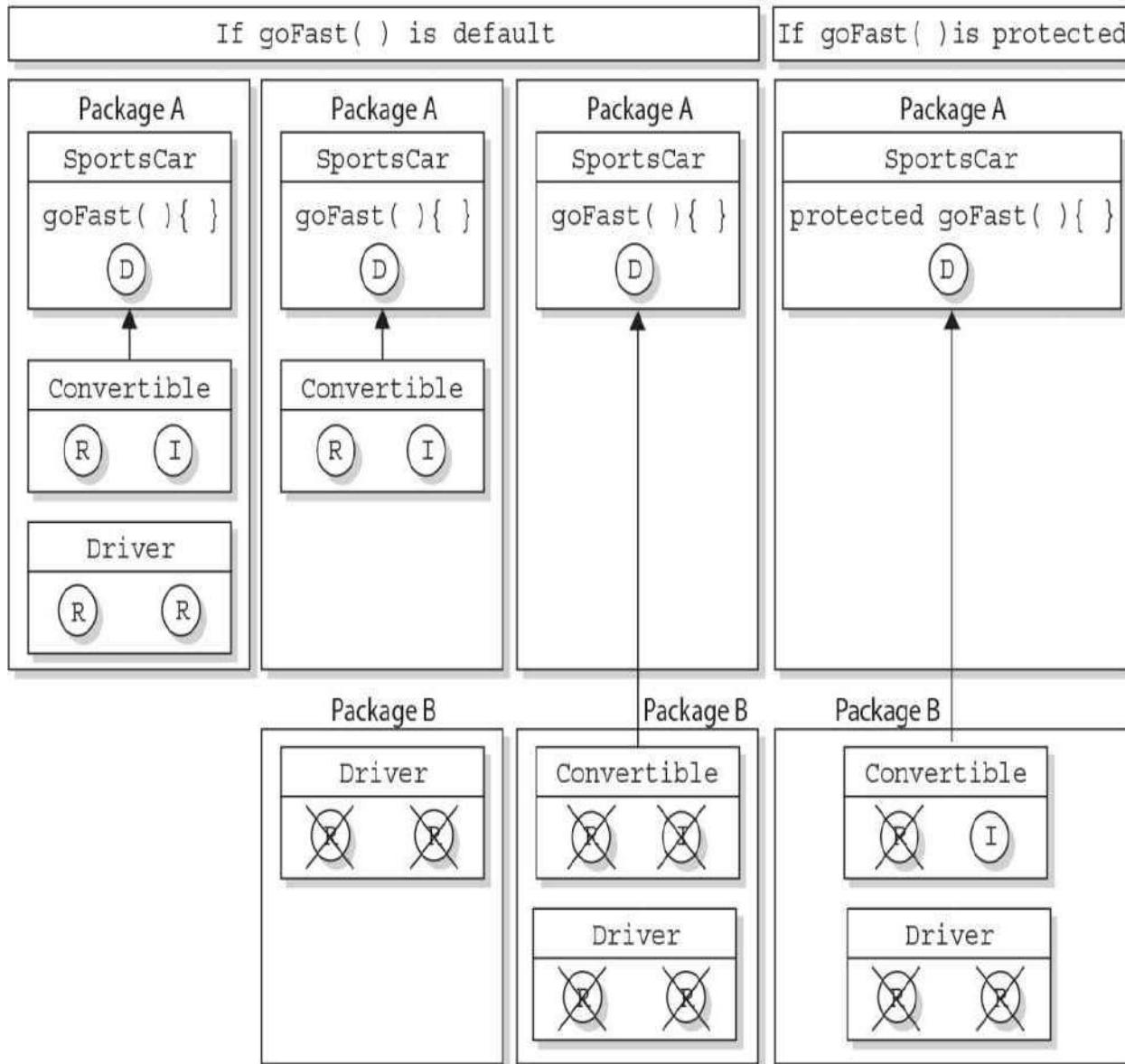
member can be accessed only through inheritance.

But there's still one more issue we haven't looked at: What does a protected member look like to other classes trying to use the subclass-outside-the-package to get to the subclass's inherited protected superclass member? For example, using our previous Parent/Child classes, what happens if some other class—Neighbor, say—in the same package as the Child (subclass) has a reference to a Child instance and wants to access the member variable `x`? In other words, how does that protected member behave once the subclass has inherited it? Does it maintain its protected status, such that classes in the Child's package can see it?

No! Once the subclass-outside-the-package inherits the protected member, that member (as inherited by the subclass) becomes private to any code outside the subclass, with the exception of subclasses of the subclass. So if class Neighbor instantiates a Child object, then even if class Neighbor is in the same package as class Child, class Neighbor won't have access to the Child's inherited (but protected) variable `x`. [Figure 1-4](#) illustrates the effect of protected access on classes and subclasses in the same or different packages.

FIGURE 1-4

Effects of protected access



Key:

<div style="border: 1px solid black; padding: 5px; margin-bottom: 10px;"> D goFast(){ } doStuff(){ } goFast(); </div> <p>Where goFast is <i>Declared</i> in the same class.</p>	<div style="border: 1px solid black; padding: 5px; margin-bottom: 10px;"> R doThings(){ SportsCar sc = new SportsCar(); sc.goFast(); } </div> <p>Invoking goFast() using a <i>Reference</i> to the class in which goFast() was declared.</p>	<div style="border: 1px solid black; padding: 5px; margin-bottom: 10px;"> I doMore(){ goFast(); } </div> <p>Invoking the goFast() method <i>Inherited</i> from a superclass.</p>
--	---	--

Whew! That wraps up `protected`, the most misunderstood modifier in Java. Again, it's used only in very special cases, but you can count on it showing up on the exam. Now that we've covered the `protected` modifier, we'll switch to default member access, a piece of cake compared to `protected`.

Default Details

Let's start with the default behavior of a member in a superclass. We'll modify the `Parent`'s member `x` to make it default:

```
package certification;
public class Parent {
    int x = 9; // No access modifier, means default
                // (package) access
}
```

Notice we didn't place an access modifier in front of the variable `x`. Remember, if you don't type an access modifier before a class or member declaration, the access control is default, which means package level. We'll now attempt to access the default member from the `Child` class that we saw earlier.

When we try to compile the `Child.java` file, we get an error something like this:

```
Child.java:4: Undefined variable: x
        System.out.println("x is " + x);
1 error
```

The compiler gives an error as when a member is declared as `private`. The subclass `Child` (in a different package from the superclass `Parent`) can't see or use the default superclass member `x`! Now, what about default access for two classes in the same package?

```
package certification;
public class Parent{
    int x = 9; // default access
}
```

And in the second class you have the following:

```
package certification;
class Child extends Parent{
    static public void main(String[] args) {
        Child sc = new Child();
        sc.testIt();
    }
    public void testIt() {
        System.out.println("Variable x is " + x); // No problem;
    }
}
```

The preceding source file compiles fine, and the class `Child` runs and displays the value of `x`. Just remember that default members are visible to subclasses only if those subclasses are in the same package as the superclass.

Local Variables and Access Modifiers

Can access modifiers be applied to local variables? NO!

There is never a case where an access modifier can be applied to a local variable, so watch out for code like the following:

```

class Foo {
    void doStuff() {
        private int x = 7;
        this.doMore(x);
    }
}

```

You can be certain that any local variable declared with an access modifier will not compile. In fact, there is only one modifier that can ever be applied to local variables—`final`.

That about does it for our discussion on member access modifiers. [Table 1-1](#) shows all the combinations of access and visibility; you really should spend some time with it. Next, we’re going to dig into the other (nonaccess) modifiers that you can apply to member declarations.

TABLE 1-1 Determining Access to Class Members

Visibility	Public	Protected	Default	Private
From the same class	Yes	Yes	Yes	Yes
From any class in the same package	Yes	Yes	Yes	No
From a subclass in the same package	Yes	Yes	Yes	No
From a subclass outside the same package	Yes	<i>Yes, through inheritance</i>	No	No
From any nonsubclass class outside the package	Yes	No	No	No

Nonaccess Member Modifiers

We've discussed member access, which refers to whether code from one class can invoke a method (or access an instance variable) from another class. That still leaves a boatload of other modifiers you can use on member declarations. Two you're already familiar with—`final` and `abstract`—because we applied them to class declarations earlier in this chapter. But we still have to take a quick look at `transient` and `synchronized`, and then a long look at the Big One, `static`, much later in the chapter.

We'll look first at modifiers applied to methods, followed by a look at modifiers applied to instance variables. We'll wrap up this section with a look at how `static` works when applied to variables and methods.

Final Methods

The `final` keyword prevents a method from being overridden in a subclass and is often used to enforce the API functionality of a method. For example, the `Thread` class has a method called `isAlive()` that checks whether a thread is still active. If you extend the `Thread` class, though, there is really no way that you can correctly implement this method yourself (it uses native code, for one thing), so the designers have made it `final`. Just as you can't subclass the `String` class (because we need to be able to trust in the behavior of a `String` object), you can't override many of the methods in the core class libraries. This can't-be-overridden restriction provides for safety and security, but you should use it with great caution. Preventing a subclass from overriding a method stifles many of the benefits of OO, including extensibility through polymorphism. A typical `final` method declaration looks like this:

```
class SuperClass{  
    public final void showSample() {  
        System.out.println("One thing.");  
    }  
}
```

It's legal to extend `SuperClass`, since the `class` isn't marked `final`, but we can't override the `final method` `showSample()`, as the following code attempts to do:

```
class SubClass extends SuperClass {  
    public void showSample() { // Try to override the final  
        // superclass method  
        System.out.println("Another thing.");  
    }  
}
```

Attempting to compile the preceding code gives us something like this:

```
%javac FinalTest.java  
FinalTest.java:5: The method void showSample() declared in class  
SubClass cannot override the final method of the same signature  
declared in class SuperClass.  
Final methods cannot be overridden.  
    public void showSample() { }  
1 error
```

Final Arguments

Method arguments are the variable declarations that appear in between the parentheses in a method declaration. A typical method declaration with multiple arguments looks like this:

```
public Record getRecord(int fileNumber, int recNumber) {}
```

Method arguments are essentially the same as local variables. In the preceding example, the variables `fileNumber` and `recNumber` will both follow all the rules applied to local variables. This means they can also have the modifier `final`:

```
public Record getRecord(int fileNumber, final int recNumber) {}
```

In this example, the variable `recNumber` is declared as `final`, which, of course,

means it can't be modified within the method. In this case, "modified" means reassigning a new value to the variable. In other words, a `final` parameter must keep the same value as the argument had when it was passed into the method. In the case of reference variables, what this means is that you might be able to change the values in the object the `final` reference variable refers to, but you CANNOT force the `final` reference variable to refer to a different object.

Abstract Methods

An abstract method is a method that's been *declared* (as `abstract`) but not *implemented*. In other words, the method contains no functional code. And if you recall from the earlier section "Abstract Classes," an abstract method declaration doesn't even have curly braces, but instead closes with a semicolon. In other words, *it has no method body*. You mark a method `abstract` when you want to force subclasses to provide the implementation. For example, if you write an abstract class `Car` with a method `goUpHill()`, you might want to force each subtype of `Car` to define its own `goUpHill()` behavior, specific to that particular type of car.

```
public abstract void showSample();
```

Notice that the abstract method ends with a semicolon instead of curly braces. **It is illegal to have even a single abstract method in a class that is not explicitly declared abstract!** Look at the following illegal class:

```
public class IllegalClass {  
    public abstract void doIt();  
}
```

The preceding class will produce the following error if you try to compile it:

```
IllegalClass.java:1: class IllegalClass must be declared  
abstract.
```

```
It does not define void doIt() from class IllegalClass.
```

```
public class IllegalClass{
```

```
1 error
```

You can, however, have an abstract class with no abstract methods. The following example will compile fine:

```
public abstract class LegalClass {  
    void goodMethod() {  
        // lots of real implementation code here  
    }  
}
```

In the preceding example, `goodMethod()` is not abstract. Three different clues tell you it's not an abstract method:

- The method is not marked abstract.
- The method declaration includes curly braces, as opposed to ending in a semicolon. In other words, the method has a method body.
- The method **might** provide actual implementation code inside the curly braces.

Any class that extends an abstract class must implement all abstract methods of the superclass, unless the subclass is *also* abstract. The rule is this:
The first concrete subclass of an abstract class must implement *all* abstract methods of the superclass.

Concrete just means nonabstract, so if you have an abstract class extending another abstract class, the abstract subclass doesn't need to provide implementations for the inherited abstract methods. Sooner or later, though, somebody's going to make a nonabstract subclass (in other words, a class that can be instantiated), and that subclass will have to implement all the abstract methods from up the inheritance tree. The following example demonstrates an inheritance tree with two abstract classes and one concrete class:

```

public abstract class Vehicle {
    private String type;
    public abstract void goUpHill(); // Abstract method
    public String getType() {           // Non-abstract method
        return type;
    }
}

public abstract class Car extends Vehicle {
    public abstract void goUpHill(); // Still abstract
    public void doCarThings() {
        // special car code goes here
    }
}

public class Mini extends Car {
    public void goUpHill() {
        // Mini-specific going uphill code
    }
}

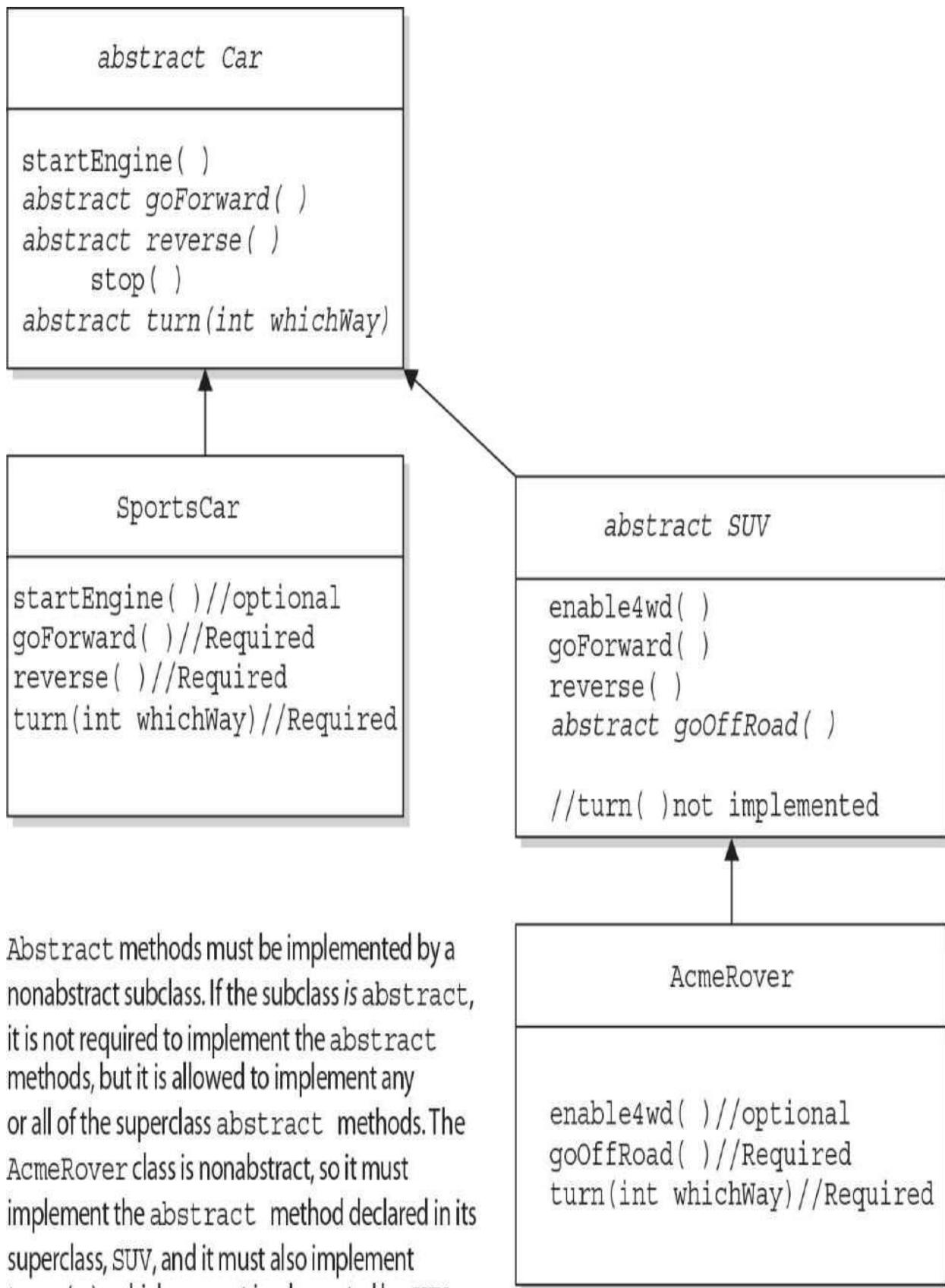
```

So how many methods does class `Mini` have? Three. It inherits both the `getType()` and `doCarThings()` methods because they're `public` and concrete (nonabstract). But because `goUpHill()` is abstract in the superclass `Vehicle` and is never implemented in the `Car` class (so it remains abstract), it means class `Mini`—as the first concrete class below `Vehicle`—must implement the `goUpHill()` method. In other words, class `Mini` can't pass the buck (of abstract method implementation) to the next class down the inheritance tree, but class `Car` can, because `Car`, like `Vehicle`, is abstract. [Figure 1-5](#) illustrates the effects

of the abstract modifier on concrete and abstract subclasses.

FIGURE 1-5

The effects of the abstract modifier on concrete and abstract subclasses



Look for concrete classes that don't provide method implementations for abstract methods of the superclass. The following code won't compile:

```
public abstract class A {  
    abstract void foo();  
}  
class B extends A {  
    void foo(int i) {}  
}
```

Class B won't compile because it doesn't implement the inherited `abstract` method `foo()`. Although the `foo(int i)` method in class B might appear to be an implementation of the superclass's `abstract` method, it is simply an overloaded method (a method using the same identifier, but different arguments), so it doesn't fulfill the requirements for implementing the superclass's `abstract` method. We'll look at the differences between overloading and overriding in detail in [Chapter 2](#).

A method can never, ever, ever be marked as both `abstract` and `final`, or both `abstract` and `private`. Think about it—`abstract` methods must be implemented (which essentially means overridden by a subclass), whereas `final` and `private` methods cannot ever be overridden by a subclass. Or to phrase it another way, an `abstract` designation means the superclass doesn't know anything about how the subclasses should behave in that method, whereas a `final` designation means the superclass knows everything about how all subclasses (however far down the inheritance tree they may be) should behave in that method. The `abstract` and `final` modifiers are virtually opposites. Because `private` methods cannot even be seen by a subclass (let alone inherited), they, too, cannot be overridden, so they, too, cannot be marked `abstract`.

Finally, you need to know that—for top-level classes—the `abstract` modifier can never be combined with the `static` modifier. We'll cover `static` methods 36 later in this objective, but for now just remember that the following would be illegal:

```
abstract static void doStuff();
```

And it would give you an error that should be familiar by now:

```
MyClass.java:2: illegal combination of modifiers: abstract and static  
    abstract static void doStuff();
```

Synchronized Methods

The `synchronized` keyword indicates that a method can be accessed by only one thread at a time. In [Chapter 10](#), we'll study the `synchronized` keyword extensively, but for now...all we're concerned with is knowing that the `synchronized` modifier can be applied only to methods—not variables, not classes, just methods. A typical `synchronized` declaration looks like this:

```
public synchronized Record retrieveUserInfo(int id) { }
```

You should also know that the `synchronized` modifier can be matched with any of the four access control levels (which means it can be paired with any of the three access modifier keywords).

Methods with Variable Argument Lists (var-args)

Java allows you to create methods that can take a variable number of arguments. Depending on where you look, you might hear this capability referred to as “variable-length argument lists,” “variable arguments,” “var-args,” “varargs,” or our personal favorite (from the department of obfuscation), “variable arity parameters.” They’re all the same thing, and we’ll use the term “var-args” from here on out.

As a bit of background, we’d like to clarify how we’re going to use the terms “argument” and “parameter” throughout this book.

- **arguments** The things you specify between the parentheses when you’re *invoking* a method:

```
public synchronized Record retrieveUserInfo(int id) { }
```

- **parameters** The things in the *method’s signature* that indicate what the method must receive when it’s invoked:

```
doStuff("a", 2); // invoking doStuff, so "a" & 2 are  
// arguments
```

Let’s review the declaration rules for var-args:

- **Var-arg type** When you declare a var-arg parameter, you must specify the type of the argument(s) this parameter of your method can receive. (This can be a primitive type or an object type.)
- **Basic syntax** To declare a method using a var-arg parameter, you follow the type with an ellipsis (...), a space (preferred but optional), and then the name of the array that will hold the parameters received.
- **Other parameters** It's legal to have other parameters in a method that uses a var-arg.
- **Var-arg limits** **The var-arg must be the last parameter in the method's signature, and you can have only one var-arg in a method.**

Let's look at some legal and illegal var-arg declarations:

Legal:

```
void doStuff(int... x) { }           // expects from 0 to many ints
                                    // as parameters
void doStuff2(char c, int... x) { }   // expects first a char,
                                    // then 0 to many ints
void doStuff3(Animal...animal) { }    // 0 to many Animal objects
                                    // (no space before the argument is legal)
```

Illegal:

```
void doStuff4(int x...) { }         // bad syntax
void doStuff5(int... x, char... y) { } // too many var-args
void doStuff6(String... s, byte b) { } // var-arg must be last
```

Constructor Declarations

In Java, objects are constructed. Every time you make a new object, at least one constructor is invoked. Every class has a constructor, although if you don't create one explicitly, the compiler will build one for you. There are tons of rules concerning constructors, and we're saving our detailed discussion for [Chapter 2](#). For now, let's focus on the basic declaration rules. Here's a simple example:

```
class Foo {  
    protected Foo() { }          // this is Foo's constructor  
    protected void Foo() { }     // this is a badly named, but legal, method  
}
```

The first thing to notice is that constructors look an awful lot like methods. A key difference is that a constructor can't ever, ever, ever, have a return type... ever! Constructor declarations can, however, have all of the normal access modifiers, and they can take arguments (including var-args), just like methods. The other BIG RULE to understand about constructors is that they must have the same name as the class in which they are declared. Constructors can't be marked `static` (they are, after all, associated with object instantiation), and they can't be marked `final` or `abstract` (because they can't be overridden). Here are some legal and illegal constructor declarations:

```

class Foo2 {
    // legal constructors
    Foo2() { }
    private Foo2(byte b) { }
    Foo2(int x) { }
    Foo2(int x, int... y) { }

    // illegal constructors
    void Foo2() { }           // it's a method, not a constructor
    Foo() { }                 // not a method or a constructor
    Foo2(short s);           // looks like an abstract method
    static Foo2(float f) { }  // can't be static
    final Foo2(long x) { }   // can't be final
    abstract Foo2(char c) { } // can't be abstract
    Foo2(int... x, int t) { } // bad var-arg syntax
}

```

Variable Declarations

There are two types of variables in Java:

- **Primitives** A primitive can be one of eight types: `char`, `boolean`, `byte`, `short`, `int`, `long`, `double`, or `float`. Once a primitive has been declared, its primitive type can never change, although in most cases its value can change.
- **Reference variables** A reference variable is used to refer to (or access) an object. A reference variable is declared to be of a specific type, and that type can never be changed. A reference variable can be used to refer to any object of the declared type or of a *subtype* of the declared type (a compatible type). We'll talk a lot more about using a reference variable to refer to a subtype in [Chapter 2](#), when we discuss polymorphism.

Declaring Primitives and Primitive Ranges

Primitive variables can be declared as class variables (statics), instance variables, method parameters, or local variables. You can declare one or more primitives, of the same primitive type, in a single line. Here are a few examples of primitive variable declarations:

```
byte b;  
boolean myBooleanPrimitive;  
int x, y, z; // declare three int primitives
```



On previous versions of the exam, you needed to know how to calculate ranges for all the Java primitives. For the current exam, you can skip some of that detail, but it's still important to understand that for the integer types the sequence from small to big is byte, short, int, and long, and that doubles are bigger than floats.

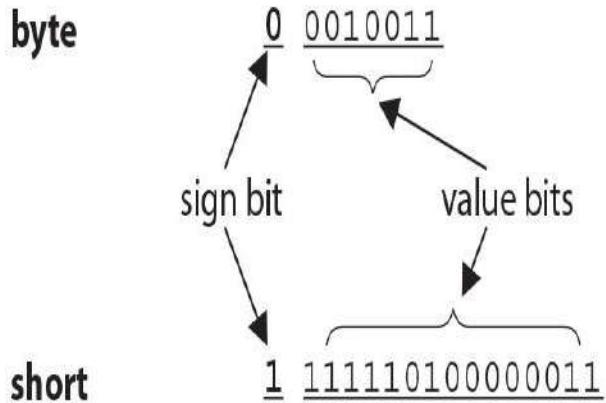
You will also need to know that the number types (both integer and floatingpoint types) are all signed and how that affects their ranges.

First, let's review the concepts.

All six number types in Java are made up of a certain number of 8-bit bytes and are *signed*, meaning they can be negative or positive. The leftmost bit (the most significant digit) is used to represent the sign, where a 1 means negative and 0 means positive, as shown in [Figure 1-6](#). The rest of the bits represent the value, using two's complement notation.

FIGURE 1-6

The sign bit for a byte



sign bit: 0 = positive
1 = negative

value bits:

byte: 7 bits can represent 2^7 or 128 different values:
0 thru 127 -or- -128 thru -1

short: 15 bits can represent 2^{15} or 32768 values:
0 thru 32767 -or- -32768 thru -1

Table 1-2 shows the primitive types with their sizes and ranges. Figure 1-6 shows that with a byte, for example, there are 256 possible numbers (or 2^8). Half of these are negative, and half – 1 are positive. The positive range is one less than the negative range because the number 0 is stored as a positive binary number. We use the formula $-2^{(\text{bits}-1)}$ to calculate the negative range, and we use $2^{(\text{bits}-1)} - 1$ for the positive range. Again, if you know the first two columns of this table, you'll be in good shape for the exam.

TABLE 1-2 Ranges of Numeric Primitives

Type	Bits	Bytes	Minimum Range	Maximum Range
byte	8	1	-2^7	$2^7 - 1$
short	16	2	-2^{15}	$2^{15} - 1$
int	32	4	-2^{31}	$2^{31} - 1$
long	64	8	-2^{63}	$2^{63} - 1$
float	32	4	n/a	n/a
double	64	8	n/a	n/a

Determining the range for floating-point numbers is complicated, but luckily you don't need to know these for the exam (although you are expected to know that a double holds 64 bits and a float 32).

There is not a range of boolean values; a boolean can be only true or false. If someone asks you for the bit depth of a boolean, look them straight in the eye and say, "That's virtual-machine dependent." They'll be impressed.

The char type (a character) contains a single, 16-bit Unicode character. Although the extended ASCII set known as ISO Latin-1 needs only 8 bits (256 different characters), a larger range is needed to represent characters found in languages other than English. Unicode characters are actually represented by unsigned 16-bit integers, which means 2^{16} possible values, ranging from 0 to 65535 ($2^{16} - 1$). Remember from the OCA 8 exam that because a char is really an integer type, it can be assigned to any number type large enough to hold 65535 (which means anything larger than a short; although both chars and shorts are 16-bit types, remember that a short uses 1 bit to represent the sign, so fewer positive numbers are acceptable in a short).

Declaring Reference Variables

Reference variables can be declared as static variables, instance variables, method parameters, or local variables. You can declare one or more reference

variables, of the same type, in a single line:

```
Object o;  
Dog myNewDogReferenceVariable;  
String s1, s2, s3; // declare three String vars.
```

Instance Variables

Instance variables are defined inside the class, but outside of any method, and are initialized only when the class is instantiated. Instance variables are the fields that belong to each unique object. For example, the following code defines fields (instance variables) for the name, title, and manager for employee objects:

```
class Employee {  
    // define fields (instance variables) for employee instances  
    private String name;  
    private String title;  
    private String manager;  
    // other code goes here including access methods for private  
    // fields  
}
```

The preceding Employee class says that each employee instance will know its own name, title, and manager. In other words, each instance can have its own unique values for those three fields. For the exam, you need to know that instance variables

- Can use any of the four access *levels* (which means they can be marked with any of the three access *modifiers*)
- Can be marked final
- Can be marked transient
- Cannot be marked abstract
- Cannot be marked synchronized

- Cannot be marked `strictfp`
- Cannot be marked `native`
- Cannot be marked `static` because then they'd become class variables

[Figure 1-7](#) compares the way in which modifiers can be applied to methods versus variables.

FIGURE 1-7

Comparison of modifiers on variables vs. methods

Local Variables	Variables (nonlocal)	Methods
<code>final</code>	<code>final</code> <code>public</code> <code>protected</code> <code>private</code> <code>static</code> <code>transient</code> <code>volatile</code>	<code>final</code> <code>public</code> <code>protected</code> <code>private</code> <code>static</code> <code>abstract</code> <code>synchronized</code> <code>strictfp</code> <code>native</code>

Local (Automatic/Stack/Method) Variables

A local variable is a variable declared within a method. That means the variable is not just initialized within the method, but also declared within the method. Just as the local variable starts its life inside the method, it's also destroyed when the method has completed. Local variables are always on the stack, not the heap. Although the value of the variable might be passed into, say, another method that then stores the value in an instance variable, the variable itself lives only within the scope of the method.

Just don't forget that while the local variable is on the stack, if the variable is an object reference, the object itself will still be created on the heap. There is no such thing as a stack object, only a stack variable. You'll often hear programmers use the phrase "local object," but what they really mean is "locally declared reference variable." So if you hear programmers use that expression, you'll know that they're just too lazy to phrase it in a technically precise way. You can tell them we said that—unless they know where we live.

Local variable declarations can't use most of the modifiers that can be applied to instance variables, such as `public` (or the other access modifiers), `transient`, `volatile`, `abstract`, or `static`, but as you saw earlier, local variables can be marked `final`. Remember, before a local variable can be *used*, it must be *initialized* with a value. For instance:

```
class TestServer {  
    public void logIn() {  
        int count = 10;  
    }  
}
```

Typically, you'll initialize a local variable in the same line in which you declare it, although you might still need to reassign it later in the method. The key is to remember that a local variable must be initialized before you try to use it. The compiler will reject any code that tries to use a local variable that hasn't been assigned a value because—unlike instance variables—local variables don't get default values.

A local variable can't be referenced in any code outside the method in which it's declared. In the preceding code example, it would be impossible to refer to the variable `count` anywhere else in the class except within the scope of the method `logIn()`. Again, that's not to say that the value of `count` can't be passed

out of the method to take on a new life. But the variable holding that value, count, can't be accessed once the method is complete, as the following illegal code demonstrates:

```
class TestServer {  
    public void logIn() {  
        int count = 10;  
    }  
    public void doSomething(int i) {  
        count = i; // Won't compile! Can't access count outside  
                   // method logIn()  
    }  
}
```

It is possible to declare a local variable with the same name as an instance variable. It's known as *shadowing*, as the following code demonstrates:

```
class TestServer {  
    int count = 9;          // Declare an instance variable named count  
    public void logIn() {  
        int count = 10;      // Declare a local variable named count  
        System.out.println("local variable count is " + count);  
    }  
    public void count() {  
        System.out.println("instance variable count is " + count);  
    }  
    public static void main(String[] args) {  
        new TestServer().logIn();  
        new TestServer().count();  
    }  
}
```

The preceding code produces the following output:

```
local variable count is 10  
instance variable count is 9
```

Why on Earth (or the planet of your choice) would you want to do that? Normally, you won't. But one of the more common reasons is to name a parameter with the same name as the instance variable to which the parameter will be assigned.

The following (wrong) code is trying to set an instance variable's value using a parameter:

```
class Foo {  
    int size = 27;  
    public void setSize(int size) {  
        size = size; // ??? which size equals which size???  
    }  
}
```

So you've decided that—for overall readability—you want to give the parameter the same name as the instance variable its value is destined for, but how do you resolve the naming collision? Use the keyword `this`. The keyword `this` always, always, always refers to the object currently running. The following code shows `this` in action:

```
class Foo {  
    int size = 27;  
    public void setSize(int size) {  
        this.size = size; // this.size means the current object's  
                         // instance variable, size. The size  
                         // on the right is the parameter  
    }  
}
```

Array Declarations

In Java, arrays are objects that store multiple variables of the same type or variables that are all subclasses of the same type. Arrays can hold either primitives or object references, but an array itself will always be an object on the heap, even if the array is declared to hold primitive elements. In other words, there is no such thing as a primitive array, but you can make an array of primitives.

For the exam, you need to know three things:

- **How to make an array reference variable (declare)**
- **How to make an array object (construct)**
- **How to populate the array with elements (initialize)**



Arrays are efficient, but many times you'll want to use one of the Collection types from `java.util` (including `HashMap`, `ArrayList`, and `TreeSet`). Collection classes offer more flexible ways to access an object (for insertion, deletion, reading, and so on) and, unlike arrays, can expand or contract dynamically as you add or remove elements. There are Collection types for a wide range of needs. Do you need a fast sort? A group of objects with no duplicates? A way to access a name-value pair? Java provides a wide variety of Collection types to address these situations, and [Chapter 6](#) discusses Collections in more detail.

Arrays are declared by stating the type of elements the array will hold (an object or a primitive), followed by square brackets to either side of the identifier.
Declaring an Array of Primitives:

```
int[] key;           // Square brackets before name (recommended)
int key[];          // Square brackets after name (legal but less
                    // readable)
```

Declaring an Array of Object References:

```
Thread[] threads; // Recommended  
Thread threads [] ; // Legal but less readable
```



When declaring an array reference, you should always put the array brackets immediately after the declared type, rather than after the identifier (variable name). That way, anyone reading the code can easily tell that, for example, `key` is a reference to an `int` array object, not an `int` primitive.

We can also declare multidimensional arrays, which are, in fact, arrays of arrays. This can be done in the following manner:

```
String[][][] occupantName;  
String[] managerName [] ;
```

The first example is a three-dimensional array (an array of arrays of arrays), and the second is a two-dimensional array. Notice in the second example, we have one square bracket before the variable name and one after. This is perfectly legal to the compiler, proving once again that just because it's legal doesn't mean it's right.



It is never legal to include the size of the array in your declaration. Yes, we know you can do that in some other languages, which is why you might see a question or two that include code similar to the following:

```
int[5] scores;
```

The preceding code won't compile. Remember, the JVM doesn't allocate space until you actually instantiate the array object. That's when size matters.

Final Variables

Declaring a variable with the `final` keyword makes it impossible to reassign that variable once it has been initialized with an explicit value (notice we said “explicit” rather than “default”). For primitives, this means that once the variable is assigned a value, the value can’t be altered. For example, if you assign 10 to the `int` variable `x`, then `x` is going to stay 10, forever. So that’s straightforward for primitives, but what does it mean to have a `final` object reference variable? A reference variable marked `final` can never be reassigned to refer to a different object. The data within the object can be modified, but the reference variable cannot be changed. In other words, a `final` reference still allows you to modify the state of the object it refers to, but you can’t modify the reference variable to make it refer to a different object. Burn this in: there are no `final` objects, only `final` references.

We’ve now covered how the `final` modifier can be applied to classes, methods, and variables. [Figure 1-8](#) highlights the key points and differences of the various applications of `final`.

FIGURE 1-8

Effect of `final` on variables, methods, and classes

**final
class**

```
final class Foo
```

**final class
cannot be
subclassed**

```
class Bar extends Foo
```

**final
method**

```
class Baz
```

```
final void go( )
```

**final method
cannot be
overridden by
a subclass**

```
class Bat extends Baz
```

```
final void go( )
```

**final
variable**

```
class Roo
```

```
final int size = 42;
```

```
void changeSize( ){  
    size = 16;  
}
```

**final variable cannot be
assigned a new value once
the initial method is made
(the initial assignment of a
value must happen before
the constructor completes).**

Transient Variables

If you mark an instance variable as `transient`, you're telling the JVM to skip (ignore) this variable when you attempt to serialize the object containing it. Serialization is one of Java's coolest features; it lets you save (sometimes called

“flatten”) an object by writing its state (in other words, the value of its instance variables) to a special type of I/O stream. With serialization, you can save an object to a file or even ship it over a wire for reinflating (deserializing) at the other end in another JVM. In [Chapter 5](#), we’ll do a deep dive into serialization.

Static Variables and Methods

Note: The discussion of static in this section DOES NOT include the new static interface method discussed earlier in this chapter. Don’t you just love how the Java 8 folks reused important Java terms?

The static modifier is used to create variables and methods that will exist independently of any instances created for the class. All static members exist before you ever make a new instance of a class, and there will be only one copy of a static member regardless of the number of instances of that class. In other words, all instances of a given class share the same value for any given static variable. We’ll cover static members in great detail in the next chapter.

Things you can mark as static:

- Methods
- Variables
- A class nested within another class, but not within a method
- Initialization blocks

Things you can’t mark as static:

- Constructors (makes no sense; a constructor is used only to create instances)
- Classes (unless they are nested)
- Interfaces (unless they are nested)
- Method local inner classes (not on the OCA 8 exam)
- Inner class methods and instance variables (not on the OCA 8 exam)
- Local variables

CERTIFICATION OBJECTIVE

Declare and Use enums (OCP Objective 2.4)

2.4 *Use enumerated types including methods, and constructors in an enum type.*

Declaring enums

Java lets you restrict a variable to having one of only a few predefined values—in other words, one value from an enumerated list. (The items in the enumerated list are called, surprisingly, `enums`.)

Using `enums` can help reduce the bugs in your code. For instance, imagine you’re creating a commercial-coffee-establishment application, and in your coffee shop application, you might want to restrict your `CoffeeSize` selections to `BIG`, `HUGE`, and `OVERWHELMING`. If you let an order for a `LARGE` or a `GRANDE` slip in, it might cause an error. `enums` to the rescue. With the following simple declaration, you can guarantee that the compiler will stop you from assigning anything to a `CoffeeSize` except `BIG`, `HUGE`, or `OVERWHELMING`:

```
enum CoffeeSize { BIG, HUGE, OVERWHELMING };
```

From then on, the only way to get a `CoffeeSize` will be with a statement something like this:

```
CoffeeSize cs = CoffeeSize.BIG;
```

It’s not required that `enum` constants be in all caps, but, borrowing from the Oracle code convention that constants are named in caps, it’s a good idea.

The basic components of an `enum` are its constants (that is, `BIG`, `HUGE`, and `OVERWHELMING`), although in a minute you’ll see that there can be a lot more to an `enum`. `enums` can be declared as their own separate class or as a class member; however, they must not be declared within a method!

Here’s an example declaring an `enum` *outside* a class:

```
enum CoffeeSize { BIG, HUGE, OVERWHELMING } // this cannot be
                                              // private or protected
class Coffee {
    CoffeeSize size;
}
public class CoffeeTest1 {
    public static void main(String[] args) {
```

```

        Coffee drink = new Coffee();
        drink.size = CoffeeSize.BIG;           // enum outside class
    }
}

```

The preceding code can be part of a single file (or, in general, enum classes can exist in their own file like `CoffeeSize.java`). But remember, in this case the file must be named `CoffeeTest1.java` because that's the name of the public class in the file. The key point to remember is that an enum that isn't enclosed in a class can be declared with only the `public` or `default` modifier, just like a non-inner class. Here's an example of declaring an enum *inside* a class:

```

class Coffee2 {
    enum CoffeeSize {BIG, HUGE, OVERWHELMING }
    CoffeeSize size;
}
public class CoffeeTest2 {
    public static void main(String[] args) {
        Coffee2 drink = new Coffee2();
        drink.size = Coffee2.CoffeeSize.BIG;      // enclosing class
                                                // name required
    }
}

```

The key points to take away from these examples are that enums can be declared as their own class or enclosed in another class, and that the syntax for accessing an enum's members depends on where the enum was declared.

The following is NOT legal:

```

public class CoffeeTest1 {
    public static void main(String[] args) {
        enum CoffeeSize { BIG, HUGE, OVERWHELMING } // WRONG! Cannot
                                                    // declare enums
                                                    // in methods
        Coffee drink = new Coffee();
        drink.size = CoffeeSize.BIG;
    }
}

```

To make it more confusing for you, the Java language designers made it optional to put a semicolon at the end of the enum declaration (when no other declarations for this enum follow):

```

public class CoffeeTest1 {
    enum CoffeeSize { BIG, HUGE, OVERWHELMING }; // <-semicolon
                                                // is optional here
    public static void main(String[] args) {
        Coffee drink = new Coffee();
        drink.size = CoffeeSize.BIG;
    }
}

```

So what gets created when you make an `enum`? The most important thing to remember is that `enums` are not `Strings` or `ints`! Each of the enumerated `CoffeeSize` values is actually an instance of `CoffeeSize`. In other words, `BIG` is of type `CoffeeSize`. Think of an `enum` as a kind of class that looks something (but not exactly) like this:

```

// conceptual example of how you can think
// about enums
class CoffeeSize {
    public static final CoffeeSize BIG =
                    new CoffeeSize("BIG", 0);
    public static final CoffeeSize HUGE =
                    new CoffeeSize("HUGE", 1);
    public static final CoffeeSize OVERWHELMING =
                    new CoffeeSize("OVERWHELMING", 2);

    CoffeeSize(String enumName, int index) {
        // stuff here
    }
    public static void main(String[] args) {
        System.out.println(CoffeeSize.BIG);
    }
}

```

Notice how each of the enumerated values, `BIG`, `HUGE`, and `OVERWHELMING`, is an instance of type `CoffeeSize`. They're represented as `static` and `final`, which, in the Java world, is thought of as a constant. Also notice that each `enum` value knows its index or position—in other words, the order in which `enum` values are declared matters. You can think of the `CoffeeSize` `enums` as existing in an array of type `CoffeeSize`, and as you'll see in a later chapter, you can iterate through the values of an `enum` by invoking the `values()` method on any `enum` type.

Declaring Constructors, Methods, and Variables in an enum

Because an enum really is a special kind of class, you can do more than just list the enumerated constant values. You can add constructors, instance variables, methods, and something really strange known as a *constant specific class body*. To understand why you might need more in your enum, think about this scenario: Imagine you want to know the actual size, in ounces, that map to each of the three `CoffeeSize` constants. For example, you want to know that `BIG` is 8 ounces, `HUGE` is 10 ounces, and `OVERWHELMING` is a whopping 16 ounces.

You could make a lookup table using some other data structure, but that would be a poor design and hard to maintain. The simplest way is to treat your enum values (`BIG`, `HUGE`, and `OVERWHELMING`) as objects, each of which can have its own instance variables. Then you can assign those values at the time the enums are initialized by passing a value to the enum constructor. This takes a little explaining, but first look at the following code:

```

enum CoffeeSize {
    // 8, 10 & 16 are passed to the constructor
    BIG(8), HUGE(10), OVERWHELMING(16);
    CoffeeSize(int ounces) {      // constructor
        this.ounces = ounces;
    }

    private int ounces;          // an instance variable
    public int getOunces() {
        return ounces;
    }
}

class Coffee {
    CoffeeSize size;           // each instance of Coffee has an enum

    public static void main(String[] args) {
        Coffee drink1 = new Coffee();
        drink1.size = CoffeeSize.BIG;

        Coffee drink2 = new Coffee();
        drink2.size = CoffeeSize.OVERWHELMING;

        System.out.println(drink1.size.getOunces()); // prints 8
        for(CoffeeSize cs: CoffeeSize.values())
            System.out.println(cs + " " + cs.getOunces());
    }
}

```

which produces:

```

8
BIG 8
HUGE 10
OVERWHELMING 16

```

Note: Every enum has a static method, `values()`, that returns an array of the enum's values in the order they're declared.

Note: Every enum has a static method, `values()`, that returns an array of the enum's values in the order they're declared.

The key points to remember about enum constructors are

- You can NEVER invoke an enum constructor directly. The enum constructor is invoked automatically with the arguments you define after

the constant value. For example, `BIG(8)` invokes the `CoffeeSize` constructor that takes an `int`, passing the `int` literal 8 to the constructor. (Behind the scenes, of course, you can imagine that `BIG` is also passed to the constructor, but we don't have to know—or care—about the details.)

- You can define more than one argument to the constructor, and you can overload the `enum` constructors, just as you can overload a normal class constructor. We discuss constructors in much more detail in [Chapter 2](#). To initialize a `CoffeeSize` with both the number of ounces and, say, a lid type, you'd pass two arguments to the constructor as `BIG(8, "A")`, which means you have a constructor in `CoffeeSize` that takes both an `int` and a `String`.

And, finally, you can define something really strange in an `enum` that looks like an anonymous inner class. It's known as a *constant specific class body*, and you use it when you need a particular constant to override a method defined in the `enum`.

Imagine this scenario: you want `enums` to have two methods—one for ounces and one for lid code (a `String`). Now imagine that most coffee sizes use the same lid code, "B", but the `OVERWHELMING` size uses type "A". You can define a `getLidCode()` method in the `CoffeeSize` `enum` that returns "B", but then you need a way to override it for `OVERWHELMING`. You don't want to do some hard-to-maintain `if/then` code in the `getLidCode()` method, so the best approach might be to somehow have the `OVERWHELMING` constant override the `getLidCode()` method.

This looks strange, but you need to understand the basic declaration rules:

```
enum CoffeeSize {  
    BIG(8),  
    HUGE(10),  
    OVERWHELMING(16) { // start a code block that defines  
                        // the "body" for this constant  
        public String getLidCode() { // override the method  
            // defined in CoffeeSize  
            return "A";  
        }  
    }; // the semicolon is REQUIRED when more code follows
```

```

    CoffeeSize(int ounces) {
        this.ounces = ounces;
    }

    private int ounces;

    public int getOunces() {
        return ounces;
    }
    public String getLidCode() {          // this method is overridden
                                         // by the OVERWHELMING constant
        return "B";                      // the default value we want to
                                         // return for CoffeeSize constants
    }
}

```

CERTIFICATION SUMMARY

After absorbing the material in this chapter, you should be familiar with some of the nuances of the Java language. You may also be experiencing confusion around why you ever wanted to take this exam in the first place. That's normal at this point. If you hear yourself asking, "What was I thinking?" just lie down until it passes. We would like to tell you that it gets easier...that this was the toughest chapter and it's all downhill from here.

Let's briefly review what you'll need to know for the exam:

You now have a good understanding of access control as it relates to classes, methods, and variables. You've looked at how access modifiers (`public`, `protected`, and `private`) define the access control of a class or member.

You learned that abstract classes can contain both abstract and nonabstract methods, but that if even a single method is marked abstract, the class must be marked abstract. Don't forget that a concrete (nonabstract) subclass of an abstract class must provide implementations for all the abstract methods of the superclass, but that an abstract class does not have to implement the abstract methods from its superclass. An abstract subclass can "pass the buck" to the first concrete subclass.

We covered interface implementation. Remember that interfaces can extend another interface (even multiple interfaces) and that any class that implements an interface must implement all methods from all the interfaces in the inheritance

tree of the interface the class is implementing.

You've also looked at the other modifiers, including `static`, `final`, `abstract`, `synchronized`, and so on. You've learned how some modifiers can never be combined in a declaration, such as mixing `abstract` with either `final` or `private`.

Keep in mind that there are no `final` objects in Java, unless you go out of your way to develop your class to create "immutable objects," which is a design approach we'll discuss in [Chapter 2](#). A reference variable marked `final` can never be changed, but the object it refers to can be modified. You've seen that `final` applied to methods means a subclass can't override them, and when applied to a class, the `final` class can't be subclassed.

Methods can be declared with a var-arg parameter (which can take from zero to many arguments of the declared type), but that you can have only one var-arg per method, and it must be the method's last parameter.

Remember that although the values of nonfinal variables can change, a reference variable's type can never change.

You also learned that arrays are objects that contain many variables of the same type. Arrays can also contain other arrays.

Remember what you've learned about `static` variables and methods, especially that `static` members are per-class as opposed to per-instance. Don't forget that a `static` method can't directly access an instance variable from the class it's in because it doesn't have an explicit reference to any particular instance of the class.

Finally, we covered enums. An enum is a safe and flexible way to implement constants. Because they are a special kind of class, enums can be declared very simply, or they can be quite complex—including such attributes as methods, variables, constructors, and a special type of inner class called a constant specific class body.

Before you hurl yourself at the practice test, spend some time with the following optimistically named "Two-Minute Drill." Come back to this particular drill often, as you work through this book and especially when you're doing that last-minute cramming. Because—and here's the advice you wished your mother had given you before you left for college—it's not what you know, it's when you know it.

For the exam, knowing what you can't do with the Java language is just as important as knowing what you can do. Give the sample questions a try! They're very similar to the difficulty and structure of the real exam questions and should be an eye opener for how difficult the exam can be. Don't worry if you get a lot

of them wrong. If you find a topic that you are weak in, spend more time reviewing and studying. Many programmers need two or three serious passes through a chapter (or an individual objective) before they can answer the questions confidently.



TWO-MINUTE DRILL

Remember that in this chapter, when we talk about classes, we're referring to non-inner classes, in other words, *top-level* classes.

Class Access Modifiers (OCP Objective 1.2)

- There are three access modifiers: `public`, `protected`, and `private`.
- There are four access levels: `public`, `protected`, `default`, and `private`.
- Classes can have only `public` or `default` access.
- A class with `default` access can be seen only by classes within the same package.
- A class with `public` access can be seen by all classes from all packages.
- Class visibility revolves around whether code in one class can
 - Create an instance of another class
 - Extend (or subclass) another class
 - Access methods and variables of another class

Class Modifiers (Nonaccess) (OCP Objectives 1.2, 2.1, and 2.2)

- Classes can also be modified with `final`, `abstract`, or `strictfp`.
- A class cannot be both `final` and `abstract`.
- A `final` class cannot be subclassed.
- An `abstract` class cannot be instantiated.
- A single `abstract` method in a class means the whole class must be `abstract`.
- An `abstract` class can have both `abstract` and non`abstract` methods.
- The first concrete class to extend an `abstract` class must implement all of its `abstract` methods.

Interface Implementation (OCP Objectives 1.2, 2.1, 2.2, and 2.5)

- ❑ Usually, interfaces are contracts for what a class can do, but they say nothing about the way in which the class must do it.
- ❑ Interfaces can be implemented by any class from any inheritance tree.
- ❑ Usually, an interface is like a 100-percent abstract class and is implicitly abstract whether or not you type the `abstract` modifier in the declaration.
- ❑ Usually interfaces have only `abstract` methods.
- ❑ Interface methods are, by default, `public` and usually `abstract`—explicit declaration of these modifiers is optional.
- ❑ Interfaces can have constants, which are always implicitly `public`, `static`, and `final`.
- ❑ Interface constant declarations of `public`, `static`, and `final` are optional in any combination.
- ❑ As of Java 8, interfaces can have concrete methods declared as either `default` or `static`.

Note: This section uses some concepts that we HAVE NOT yet covered. Don't panic: once you've read through the entire book, this section will make sense as a reference.

- ❑ A legal nonabstract implementing class has the following properties:
 - ❑ It provides concrete implementations for the interface's methods.
 - ❑ It must follow all legal override rules for the methods it implements.
 - ❑ It must not declare any new checked exceptions for an implementation method.
 - ❑ It must not declare any checked exceptions that are broader than the exceptions declared in the interface method.
 - ❑ It may declare runtime exceptions on any interface method implementation regardless of the interface declaration.
 - ❑ It must maintain the exact signature (allowing for covariant returns) and return type of the methods it implements (but does not have to declare the exceptions of the interface).
- ❑ A class implementing an interface can itself be `abstract`.
- ❑ An `abstract`-implementing class does not have to implement the interface methods (but the first concrete subclass must).

- A class can extend only one class (no multiple inheritance), but it can implement many interfaces.
- Interfaces can extend one or more other interfaces.
- Interfaces cannot extend a class or implement a class or interface.
- When taking the exam, verify that interface and class declarations are legal before verifying other code logic.

Member Access Modifiers (OCP Objective 1.2)

- Methods and instance (nonlocal) variables are known as “members.”
- Members can use all four access levels: public, protected, default, and private.
- Member access comes in two forms:
 - Code in one class can access a member of another class.
 - A subclass can inherit a member of its superclass.
- If a class cannot be accessed, its members cannot be accessed.
- Determine class visibility before determining member visibility.
- public members can be accessed by all other classes, even in other packages.
- If a superclass member is public, the subclass inherits it—regardless of package.
- Members accessed without the dot operator (.) must belong to the same class.
- this. always refers to the currently executing object.
- this.aMethod() is the same as just invoking aMethod().
- private members can be accessed only by code in the same class.
- private members are not visible to subclasses, so private members cannot be inherited.
- Default and protected members differ only when subclasses are involved:
 - Default members can be accessed only by classes in the same package.
 - protected members can be accessed by other classes in the same package, plus subclasses regardless of package.

- `protected` = package + kids (kids meaning subclasses).
- For subclasses outside the package, the `protected` member can be accessed only through inheritance; a subclass outside the package cannot access a `protected` member by using a reference to a superclass instance. (In other words, inheritance is the only mechanism for a subclass outside the package to access a `protected` member of its superclass.)
- A `protected` member inherited by a subclass from another package is not accessible to any other class in the subclass package, except for the subclass's own subclasses.

Local Variables (OCP Objective 2.2)

- Local (method, automatic, or stack) variable declarations cannot have access modifiers.
- `final` is the only modifier available to local variables.
- Local variables don't get default values, so they must be initialized before use.

Other Modifiers—Members (OCP Objectives 2.1 and 2.2)

- `final` methods cannot be overridden in a subclass.
- `abstract` methods are declared with a signature, a return type, and an optional `throws` clause, but they are not implemented.
- `abstract` methods end in a semicolon—no curly braces.
- Three ways to spot a nonabstract method:
 - The method is not marked `abstract`.
 - The method has curly braces.
 - The method **MIGHT** have code between the curly braces.
- The first nonabstract (concrete) class to extend an abstract class must implement all of the abstract class's abstract methods.
- The `synchronized` modifier applies only to methods and code blocks.
- `synchronized` methods can have any access control and can also be marked `final`.
- `abstract` methods must be implemented by a subclass, so they must be inheritable. For that reason

- abstract methods cannot be private.
- abstract methods cannot be final.

Methods with var-args (OCP Objective 1.2)

- Methods can declare a parameter that accepts from zero to many arguments, a so-called var-arg method.
- A var-arg parameter is declared with the syntax `type... name`, for instance: `doStuff(int... x) { }`.
- A var-arg method can have only one var-arg parameter.
- In methods with normal parameters and a var-arg, the var-arg must come last.

Constructors (OCP Objectives 1.2 and 2.4)

- Constructors must have the same name as the class
- Constructors can have arguments, but they cannot have a return type.
- Constructors can use any access modifier (even `private!`).

Variable Declarations (OCP Objectives 2.1 and 2.2)

- Instance variables can
 - Have any access control
 - Be marked `final` or `transient`
- Instance variables can't be `abstract` or `synchronized`.
- It is legal to declare a local variable with the same name as an instance variable; this is called "shadowing."
- `final` variables have the following properties:
 - `final` variables cannot be reassigned once assigned a value.
 - `final` reference variables cannot refer to a different object once the object has been assigned to the `final` variable.
 - `final` variables must be initialized before the constructor completes.
- There is no such thing as a `final` object. An object reference marked `final` does NOT mean the object itself can't change.
- The `transient` modifier applies only to instance variables.
- The `volatile` modifier applies only to instance variables.

Array Declarations (OCP Objective 1.2)

- Arrays can hold primitives or objects, but the array itself is always an object.
- When you declare an array, the brackets can be to the left or to the right of the variable name.
- It is never legal to include the size of an array in the declaration.
- An array of objects can hold any object that passes the IS-A (or `instanceof`) test for the declared type of the array. For example, if `Horse` extends `Animal`, then a `Horse` object can go into an `Animal` array.

Static Variables and Methods (OCP Objective 1.6)

- They are not tied to any particular instance of a class.
- No class instances are needed in order to use `static` members of the class or interface.
- There is only one copy of a `static` variable/class and all instances share it.
- `static` methods do not have direct access to nonstatic members.

enums (OCP Objective 2.4)

- An `enum` specifies a list of constant values assigned to a type.
- An `enum` is NOT a `String` or an `int`; an `enum` constant's type is the `enum` type. For example, `SUMMER` and `FALL` are of the `enum` type `Season`.
- An `enum` can be declared outside or inside a class, but NOT in a method.
- An `enum` declared outside a class must NOT be marked `static`, `final`, `abstract`, `protected`, or `private`.
- `enums` can contain constructors, methods, variables, and constant specific class bodies.
- `enum` constants can send arguments to the `enum` constructor, using the syntax `BIG(8)`, where the `int` literal 8 is passed to the `enum` constructor.
- `enum` constructors can have arguments and can be overloaded.
- `enum` constructors can NEVER be invoked directly in code. They are always called automatically when an `enum` is initialized.
- The semicolon at the end of an `enum` declaration is optional. These are legal:

```
enum Foo { ONE, TWO, THREE}  
enum Foo { ONE, TWO, THREE};
```

- MyEnum.values() returns an array of MyEnum's values.

Q SELF TEST

The following questions will help measure your understanding of the material presented in this chapter. Read all the choices carefully, as there may be more than one correct answer. Choose all correct answers for each question. Stay focused.

If you have a rough time with these at first, don't beat yourself up. Be positive. Repeat nice affirmations to yourself: "I am smart enough to understand enums," and "OK, so that other guy knows enums better than I do, but I bet he can't <insert something you are good at> like me."

- 1.** Which are true? (Choose all that apply.)
 - A.** "X extends Y" is correct if and only if X is a class and Y is an interface
 - B.** "X extends Y" is correct if and only if X is an interface and Y is a class
 - C.** "X extends Y" is correct if X and Y are either both classes or both interfaces
 - D.** "X extends Y" is correct for all combinations of X and Y being classes and/or interfaces

- 2.** Given:

```
class Rocket {  
    private void blastOff() { System.out.print("bang "); }  
}  
public class Shuttle extends Rocket {  
    public static void main(String[] args) {  
        new Shuttle().go();  
    }  
    void go() {  
        blastOff();  
        // Rocket.blastOff(); // line A  
    }  
    private void blastOff() { System.out.print("sh-bang "); }  
}
```

- Which are true? (Choose all that apply.)

- A.** As the code stands, the output is bang
- B.** As the code stands, the output is sh-bang
- C.** As the code stands, compilation fails
- D.** If line A is uncommented, the output is bang bang
- E.** If line A is uncommented, the output is sh-bang bang
- F.** If line A is uncommented, compilation fails

3. Given:

```
1. enum Animals {  
2.     DOG("woof"), CAT("meow"), FISH("burble");  
3.     String sound;  
4.     Animals(String s) { sound = s; }  
5. }  
6. class TestEnum {  
7.     static Animals a;  
8.     public static void main(String[] args) {  
9.         System.out.println(a.DOG.sound + " " + a.FISH.sound);  
10.    }  
11. }
```

What is the result?

- A.** woof burble
- B.** Multiple compilation errors
- C.** Compilation fails due to an error on line 2
- D.** Compilation fails due to an error on line 3
- E.** Compilation fails due to an error on line 4
- F.** Compilation fails due to an error on line 9

4. Given two files:

```

1. package pkgA;
2. public class Foo {
3.     int a = 5;
4.     protected int b = 6;
5.     public int c = 7;
6. }

3. package pkgB;
4. import pkgA.*;
5. public class Baz {
6.     public static void main(String[] args) {
7.         Foo f = new Foo();
8.         System.out.print(" " + f.a);
9.         System.out.print(" " + f.b);
10.        System.out.println(" " + f.c);
11.    }
12. }

```

What is the result? (Choose all that apply.)

- A.** 5 6 7
- B.** 5 followed by an exception
- C.** Compilation fails with an error on line 7
- D.** Compilation fails with an error on line 8
- E.** Compilation fails with an error on line 9
- F.** Compilation fails with an error on line 10

5. Given:

```

1. public class Electronic implements Device
   { public void doIt() { } }
2.
3. abstract class Phone1 extends Electronic { }
4.
5. abstract class Phone2 extends Electronic
   { public void doIt(int x) { } }
6.
7. class Phone3 extends Electronic implements Device
   { public void doStuff() { } }
8.
9. interface Device { public void doIt(); }

```

What is the result? (Choose all that apply.)

- A.** Compilation succeeds
- B.** Compilation fails with an error on line 1

- C. Compilation fails with an error on line 3
- D. Compilation fails with an error on line 5
- E. Compilation fails with an error on line 7
- F. Compilation fails with an error on line 9

6. Given:

```
3. public class TestDays {  
4.     public enum Days { MON, TUE, WED };  
5.     public static void main(String[] args) {  
6.         for(Days d : Days.values() )  
7.             ;  
8.         Days [] d2 = Days.values();  
9.         System.out.println(d2[2]);  
10.    }  
11. }
```

What is the result? (Choose all that apply.)

- A. TUE
- B. WED
- C. The output is unpredictable
- D. Compilation fails due to an error on line 4
- E. Compilation fails due to an error on line 6
- F. Compilation fails due to an error on line 8
- G. Compilation fails due to an error on line 9

7. Given:

```
4. public class Frodo extends Hobbit {  
5.     public static void main(String[] args) {  
6.         int myGold = 7;  
7.         System.out.println(countGold(myGold, 6));  
8.     }  
9. }  
10. class Hobbit {  
11.     int countGold(int x, int y) { return x + y; }  
12. }
```

What is the result?

- A. 13
- B. Compilation fails due to multiple errors
- C. Compilation fails due to an error on line 6

- D.** Compilation fails due to an error on line 7
- E.** Compilation fails due to an error on line 11

8. Given:

```
interface Gadget {  
    void doStuff();  
}  
abstract class Electronic {  
    void getPower() { System.out.print("plug in "); }  
}  
public class Tablet extends Electronic implements Gadget {  
    void doStuff() { System.out.print("show book "); }  
    public static void main(String[] args) {  
        new Tablet().getPower();  
        new Tablet().doStuff();  
    }  
}
```

Which are true? (Choose all that apply.)

- A.** The class Tablet will NOT compile
- B.** The interface Gadget will NOT compile
- C.** The output will be plug in show book
- D.** The abstract class Electronic will NOT compile
- E.** The class Tablet CANNOT both extend and implement

9. Given:

```
interface MyInterface {  
    // insert code here  
}
```

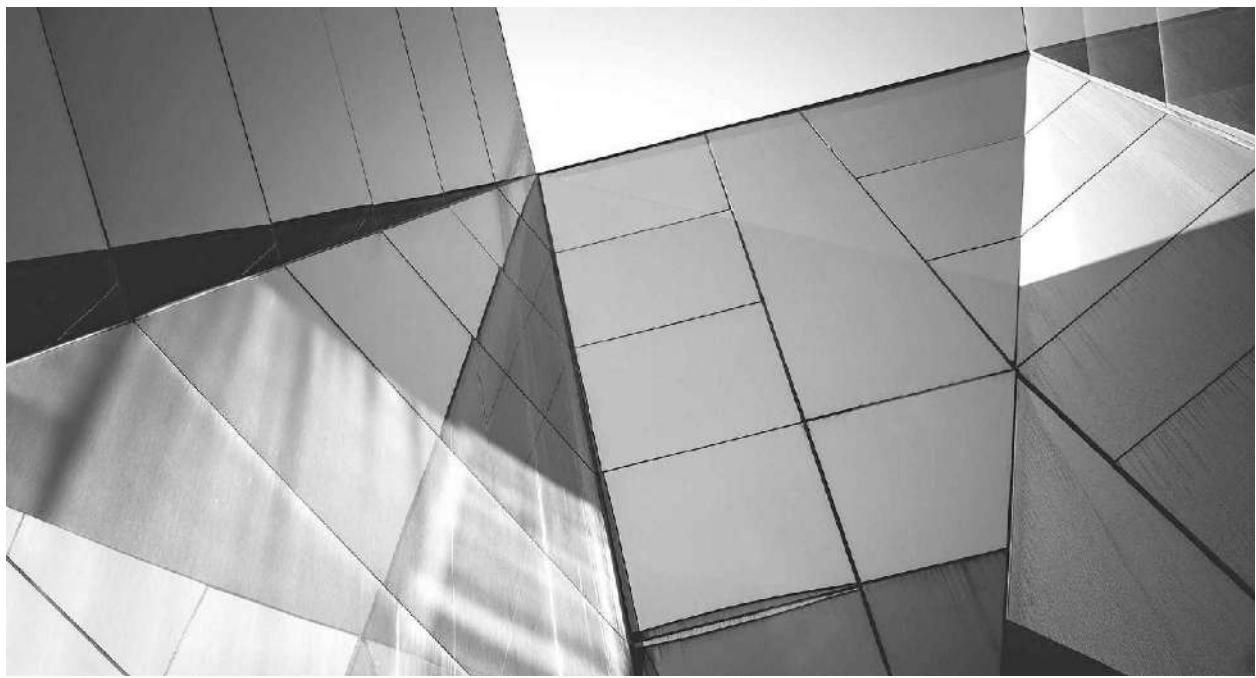
Which lines of code—inserted independently at `insert code here`—will compile? (Choose all that apply.)

- A.** `public static m1() {}`
- B.** `default void m2() {}`
- C.** `abstract int m3();`
- D.** `final short m4() {return 5;}`
- E.** `default long m5();`
- F.** `static void m6() {}`

A SELF TEST ANSWERS

1. **C** is correct.
 - ☒ **A** is incorrect because classes implement interfaces, they don't extend them. **B** is incorrect because interfaces only "inherit from" other interfaces. **D** is incorrect based on the preceding rules. (OCP Objectives 1.2 and 2.5)
2. **B** and **F** are correct. Since `Rocket.blastOff()` is `private`, it can't be overridden, and it is invisible to class `Shuttle`.
 - ☒ **A, C, D, and E** are incorrect based on the above. (OCP Objective 1.2 and 1.3)
3. **A** is correct; enums can have constructors and variables.
 - ☒ **B, C, D, E, and F** are incorrect; these lines all use correct syntax. (OCP Objective 2.4)
4. **D** and **E** are correct. Variable `a` has default access, so it cannot be accessed from outside the package. Variable `b` has protected access in `pkgA`.
 - ☒ **A, B, C, and F** are incorrect based on the above information. (OCP Objective 1.2)
5. **A** is correct; all of these are legal declarations.
 - ☒ **B, C, D, E, and F** are incorrect based on the above information. (OCP Objectives 1.2, 2.1, and 2.5)
6. **B** is correct. Every enum comes with a `static values()` method that returns an array of the enum's values in the order in which they are declared in the enum.
 - ☒ **A, C, D, E, F, and G** are incorrect based on the above information. (OCP Objectives 1.6 and 2.4)
7. **D** is correct. The `countGold()` method cannot be invoked from a static context.
 - ☒ **A, B, C, and E** are incorrect based on the above information. (OCP Objective 1.6)
8. **A** is correct. By default, an interface's methods are `public` so the `Tablet.doStuff` method must be `public`, too. The rest of the code is valid.

- B, C, D, and E** are incorrect based on the above. (OCP Objectives 1.2 and 2.5)
9. **B, C, and F** are correct. As of Java 8, interfaces can have default and static methods.
- A, D, and E** are incorrect. **A** has no return type; **D** cannot have a method body; and **E** needs a method body. (OCP Objective 2.5)



2

Object Orientation

CERTIFICATION OBJECTIVES

- Implement Encapsulation
 - Implement Inheritance
 - Use IS-A and HAS-A Relationships
 - Use Polymorphism
 - Use Overriding and Overloading
 - Use the @Override Annotation
 - Understand Casting
 - Use Interfaces
 - Understand and Use Return Types
 - Develop Constructors
 - Use the Singleton Pattern
 - Develop Immutable Classes
 - Use static Members
-  Two-Minute Drill

Q&A Self Test

This chapter will prepare you for many of the object-oriented objectives and questions you'll encounter on the exam. As with [Chapter 1](#), most of this chapter is a refresher of some of the topics you learned while studying for the OCA 8 exam. Apart from the discussions of the @Override annotation, the singleton pattern, and immutable classes, if you feel you mastered the OCA 8 section 6 objectives (Working with Methods and Encapsulation) and the section 7 objectives (Working with Inheritance) while studying for the OCA 8, you

might be able to skip this chapter. If you’re not sure, try your hand at the Self Test at the end of the chapter.

CERTIFICATION OBJECTIVE

Encapsulation (OCP Objective 1.1)

1.1 *Implement encapsulation.*

Imagine you wrote the code for a class and another dozen programmers from your company all wrote programs that used your class. Now imagine that later on, you didn’t like the way the class behaved, because some of its instance variables were being set (by the other programmers from within their code) to values you hadn’t anticipated. *Their* code brought out errors in *your* code. (Relax, this is just hypothetical.) Well, it is a Java program, so you should be able to ship out a newer version of the class, which they could replace in their programs without changing any of their own code.

This scenario highlights two of the promises/benefits of an object-oriented (OO) language: flexibility and maintainability. But those benefits don’t come automatically. You have to do something. You have to write your classes and code in a way that supports flexibility and maintainability. So what if Java supports OO? It can’t design your code for you. For example, imagine you made your class with public instance variables, and those other programmers were setting the instance variables directly, as the following code demonstrates:

```
public class BadOO {  
    public int size;  
    public int weight;  
    ...
```

```

}
public class ExploitBadOO {
    public static void main (String [] args) {
        BadOO b = new BadOO();
        b.size = -5; // Legal but bad!!
    }
}

```

And now you're in trouble. How are you going to change the class in a way that lets you handle the issues that come up when somebody changes the `size` variable to a value that causes problems? Your only choice is to go back in and write method code for adjusting `size` (a `setSize(int a)` method, for example) and then insulate the `size` variable with, say, a private access modifier. But as soon as you make that change to your code, you break everyone else's!

The ability to make changes in your implementation code without breaking the code of others who use your code is a key benefit of encapsulation. You want to hide implementation details behind a public programming interface. By *interface*, we mean the set of accessible methods your code makes available for other code to call—in other words, your code’s API. By hiding implementation details, you can rework your method code (perhaps also altering the way variables are used by your class) without forcing a change in the code that calls your changed method.

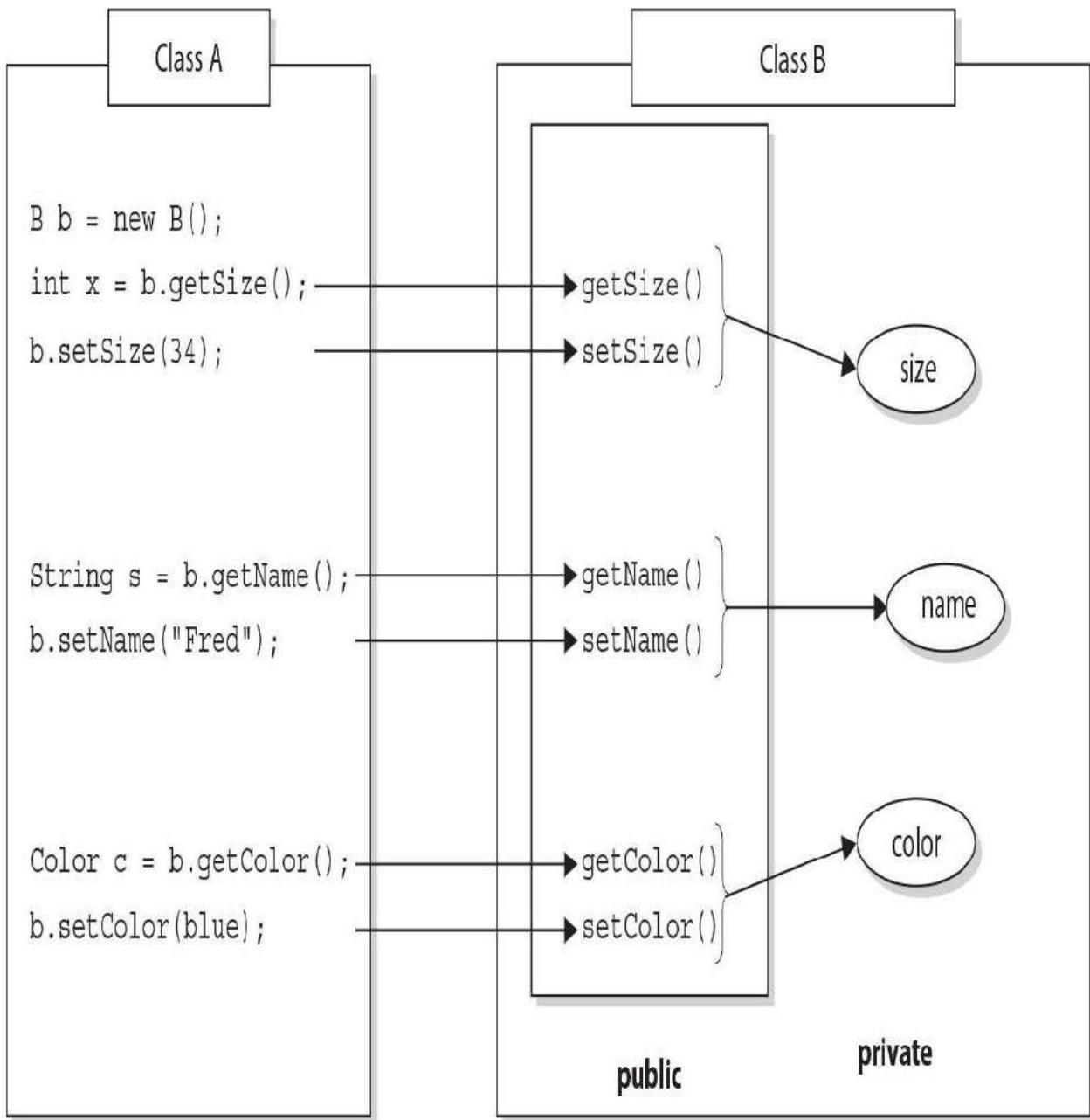
If you want maintainability, flexibility, and extensibility (and, of course, you do), your design must include encapsulation. How do you do that?

- Keep instance variables hidden (with an access modifier, often `private`).
- Make public accessor methods, and force calling code to use those methods rather than directly accessing the instance variable. These so-called accessor methods allow users of your class to **set** a variable’s value or **get** a variable’s value.
- For these accessor methods, use the most common naming convention of `set<SomeProperty>` and `get<SomeProperty>`.

[Figure 2-1](#) illustrates the idea that encapsulation forces callers of our code to go through methods rather than accessing variables directly.

FIGURE 2-1

The nature of encapsulation



Class A cannot access Class B instance variable data without going through getter and setter methods. Data is marked private; only the accessor methods are public.

We call the access methods *getters* and *setters*, although some prefer the fancier terms *accessors* and *mutators*. (Personally, we don't like the word "mutate.") Regardless of what you call them, they're methods that other programmers must go through in order to access your instance variables. They look simple, and you've probably been using them forever:

```
public class Box {  
    // hide the instance variable; only an instance  
    // of Box can access it  
    private int size;  
    // Provide public getters and setters  
    public int getSize() {  
        return size;  
    }  
  
    public void setSize(int newSize) {  
        size = newSize;  
    }  
}
```

Wait a minute. How useful is the previous code? It doesn't even do any validation or processing. What benefit can there be from having getters and setters that add no functionality? The point is, you can change your mind later and add more code to your methods without breaking your API. Even if today you don't think you really need validation or processing of the data, good OO design dictates that you plan for the future. To be safe, force calling code to go through your methods rather than going directly to instance variables. *Always*. Then you're free to rework your method implementations later, without risking the wrath of those dozen programmers who know where you live.

Look out for code that appears to be asking about the behavior of a method, when the problem is actually a lack of encapsulation. Look at the following example, and see if you can figure out what's going on:

```
class Foo {  
    public int left = 9;  
    public int right = 3;  
    public void setLeft(int leftNum) {  
        left = leftNum;  
        right = leftNum/3;  
    }  
    // lots of complex test code here  
}
```

Now consider this question: Is the value of right always going to be one-third the value of left? It looks like it will, until you realize that users of the Foo class don't need to use the setLeft() method! They can simply go straight to the instance variables and change them to any arbitrary int value.

CERTIFICATION OBJECTIVE

Inheritance and Polymorphism (OCP Objectives 1.2 and 1.3)

1.2 *Implement inheritance including visibility modifiers and composition.*

1.3 *Implement polymorphism.*

Inheritance is everywhere in Java. It's safe to say that it's almost (almost?)

impossible to write even the tiniest Java program without using inheritance. To explore this topic, we're going to use the `instanceof` operator. This code:

```
class Test {  
    public static void main(String [] args) {  
        Test t1 = new Test();  
        Test t2 = new Test();  
        if (!t1.equals(t2))  
            System.out.println("they're not equal");  
        if (t1 instanceof Object)  
            System.out.println("t1's an Object");  
    }  
}
```

produces this output:

```
they're not equal  
t1's an Object
```

Where did that `equals` method come from? The reference variable `t1` is of type `Test`, and there's no `equals` method in the `Test` class. Or is there? The second `if` test asks whether `t1` is an instance of class `Object`, and because it *is* (more on that soon), the `if` test succeeds.

Hold on...how can `t1` be an instance of type `Object`, when we just said it was of type `Test`? I'm sure you're way ahead of us here, but it turns out that every class in Java is a subclass of class `Object` (except, of course, class `Object` itself). In other words, every class you'll ever use or ever write will inherit from class `Object`. You'll always have an `equals` method, `notify` and `wait` methods, and others available to use. Whenever you create a class, you automatically inherit all of class `Object`'s methods.

Why? Let's look at that `equals` method, for instance. Java's creators correctly assumed that it would be common for Java programmers to want to compare instances of their classes to check for equality. If class `Object` didn't have an

`equals` method, you'd have to write one yourself—you and every other Java programmer. That one `equals` method has been inherited billions of times. (To be fair, `equals` has also been *overridden* billions of times, but we're getting ahead of ourselves.)

The Evolution of Inheritance

Until Java 8, when the topic of inheritance was discussed, it usually revolved around subclasses inheriting methods from their superclasses. While this simplification was never perfectly correct, it became less correct with the new features available in Java 8. As [Table 2-1](#) shows, it's now possible to inherit concrete methods from interfaces. This is a big change. For the rest of the chapter, when we talk about inheritance generally, we will tend to use the terms "subtypes" and "supertypes" to acknowledge that both classes and interfaces need to be accounted for. We will tend to use the terms "subclass" and "superclass" when we're discussing a specific example that's under discussion. Inheritance is a key aspect of most of the topics we'll be discussing in this chapter, so be prepared for LOTS of discussion about the interactions between supertypes and subtypes!

TABLE 2-1 Inheritable Elements of Classes and Interfaces

Elements of Types	Classes	Interfaces
Instance variables	Yes	Not applicable
Static variables	Yes	Only constants
Abstract methods	Yes	Yes
Instance methods	Yes	Java 8, default methods
Static methods	Yes	Java 8, inherited no, accessible yes
Constructors	No	Not applicable
Initialization blocks	No	Not applicable

As you study [Table 2-1](#), you'll notice that, as of Java 8, interfaces can contain two types of concrete methods: `static` and `default`. We'll discuss these important additions later in this chapter.

[Table 2-1](#) summarizes the elements of classes and interfaces relative to inheritance.

For the exam, you'll need to know that you can create inheritance relationships in Java by *extending* a class or by implementing an interface. It's also important to understand that the two most common reasons to use inheritance are

- To promote code reuse
- To use polymorphism

Let's start with reuse. A common design approach is to create a fairly generic version of a class with the intention of creating more specialized subclasses that inherit from it. For example:

```
class GameShape {  
    public void displayShape() {  
        System.out.println("displaying shape");  
    }  
    // more code  
}  
  
class PlayerPiece extends GameShape {  
    public void movePiece() {  
        System.out.println("moving game piece");  
    }  
    // more code  
}  
  
public class TestShapes {  
    public static void main (String[] args) {  
        PlayerPiece shape = new PlayerPiece();  
        shape.displayShape();  
        shape.movePiece();  
    }  
}
```

outputs:

```
displaying shape  
moving game piece
```

Notice that the `PlayerPiece` class inherits the generic `displayShape()` method from the less-specialized class `GameShape` and also adds its own method, `movePiece()`. Code reuse through inheritance means that methods with generic functionality—such as `displayShape()`, which could apply to a wide range of different kinds of shapes in a game—don’t have to be reimplemented. That means all specialized subclasses of `GameShape` are guaranteed to have the capabilities of the more general superclass. You don’t want to have to rewrite the `displayShape()` code in each of your specialized components of an online game.

But you knew that. You’ve experienced the pain of duplicate code when you make a change in one place and have to track down all the other places where that same (or very similar) code exists.

The second (and related) use of inheritance is to allow your classes to be accessed polymorphically—a capability provided by interfaces as well, but we’ll get to that in a minute. Let’s say that you have a `GameLauncher` class that wants to loop through a list of different kinds of `GameShape` objects and invoke `displayShape()` on each of them. At the time you write this class, you don’t know every possible kind of `GameShape` subclass that anyone else will ever write. And you sure don’t want to have to redo *your* code just because somebody decided to build a dice shape six months later.

The beautiful thing about polymorphism (“many forms”) is that you can treat any *subclass* of `GameShape` as a `GameShape`. In other words, you can write code in your `GameLauncher` class that says, “I don’t care what kind of object you are as long as you inherit from (extend) `GameShape`. And as far as I’m concerned, if you extend `GameShape`, then you’ve definitely got a `displayShape()` method, so I know I can call it.”

Imagine we now have two specialized subclasses that extend the more generic `GameShape` class, `PlayerPiece` and `TilePiece`:

```

class GameShape {
    public void displayShape() {
        System.out.println("displaying shape");
    }
    // more code
}

class PlayerPiece extends GameShape {
    public void movePiece() {
        System.out.println("moving game piece");
    }
    // more code
}

class TilePiece extends GameShape {
    public void getAdjacent() {
        System.out.println("getting adjacent tiles");
    }
    // more code
}

```

Now imagine a test class has a method with a declared argument type of `GameShape`, which means it can take any kind of `GameShape`. In other words, any subclass of `GameShape` can be passed to a method with an argument of type `GameShape`. This code:

```
public class TestShapes {
    public static void main (String[] args) {
        PlayerPiece player = new PlayerPiece();
        TilePiece tile = new TilePiece();
        doShapes(player);
        doShapes(tile);
    }

    public static void doShapes(GameShape shape) {
        shape.displayShape();
    }
}
```

outputs:

```
displaying shape
displaying shape
```

The key point is that the `doShapes()` method is declared with a `GameShape` argument but can be passed any subtype (in this example, a subclass) of `GameShape`. The method can then invoke any method of `GameShape`, without any concern for the actual runtime class type of the object passed to the method. There are implications, though. The `doShapes()` method knows only that the objects are a type of `GameShape` since that's how the parameter is declared. And using a reference variable declared as type `GameShape`—regardless of whether the variable is a method parameter, local variable, or instance variable—means that *only* the methods of `GameShape` can be invoked on it. The methods you can call on a reference are totally dependent on the *declared* type of the variable, no matter what the actual object is, that the reference is referring to. That means you can't use a `GameShape` variable to call, say, the `getAdjacent()` method even if

the object passed in *is* of type `TilePiece`. (We'll see this again when we look at interfaces.)

IS-A and HAS-A Relationships

Note: As of early 2018, the OCP 8 exam doesn't mention IS-A and HAS-A relationships explicitly, but inheritance and polymorphism are all about IS-A relationships, and composition is another way of saying "HAS-A."

IS-A

In OO, the concept of IS-A is based on inheritance (or interface implementation). IS-A is a way of saying, "This thing is a type of that thing." For example, a Mustang is a type of Horse, so in OO terms we can say, "Mustang IS-A Horse." Subaru IS-A Car. Broccoli IS-A Vegetable (not a very fun one, but it still counts). You express the IS-A relationship in Java through the keywords `extends` (for *class* inheritance) and `implements` (for *interface* implementation).

```
public class Car {  
    // Cool Car code goes here  
}
```

```
public class Subaru extends Car {  
    // Important Subaru-specific stuff goes here  
    // Don't forget Subaru inherits accessible Car members which  
    // can include both methods and variables.  
}
```

A Car is a type of Vehicle, so the inheritance tree might start from the `Vehicle` class as follows:

```
public class Vehicle { ... }  
public class Car extends Vehicle { ... }  
public class Subaru extends Car { ... }
```

In OO terms, you can say the following:

- Vehicle is a superclass of Car.
- Car is a subclass of Vehicle.
- Car is a superclass of Subaru.
- Subaru is a subclass of Vehicle.
- Car inherits from Vehicle.
- Subaru inherits from both Vehicle and Car.
- Subaru is derived from Car.
- Car is derived from Vehicle.
- Subaru is derived from Vehicle.
- Subaru is a subtype of both Vehicle and Car.

Returning to our IS-A relationship, the following statements are true:

- "Car extends Vehicle" means "Car IS-A Vehicle."
- "Subaru extends Car" means "Subaru IS-A Car."

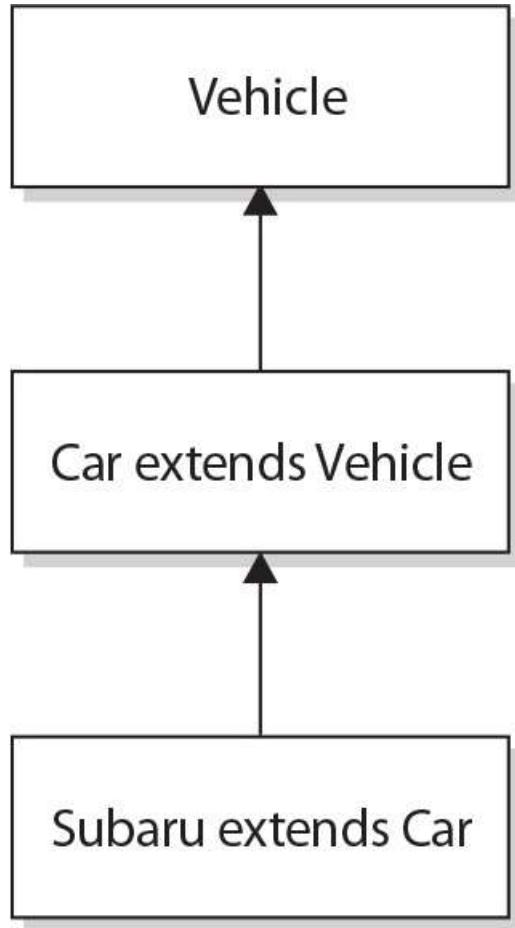
And we can also say:

“Subaru IS-A Vehicle”

because a class is said to be “a type of” anything further up in its inheritance tree. If the expression (`Foo instanceof Bar`) is true, then class `Foo` IS-A `Bar`, even if `Foo` doesn’t directly extend `Bar`, but instead extends some other class that is a subclass of `Bar`. [Figure 2-2](#) illustrates the inheritance tree for `Vehicle`, `Car`, and `Subaru`. The arrows move from the subclass to the superclass. In other words, a class’s arrow points toward the class from which it extends.

FIGURE 2-2

Inheritance tree for `Vehicle`, `Car`, `Subaru`



HAS-A

HAS-A relationships are based on use, rather than inheritance. In other words, class A HAS-A B if code in class A has a reference to an instance of class B. For example, you can say the following:

A Horse IS-A Animal. A Horse HAS-A Halter.

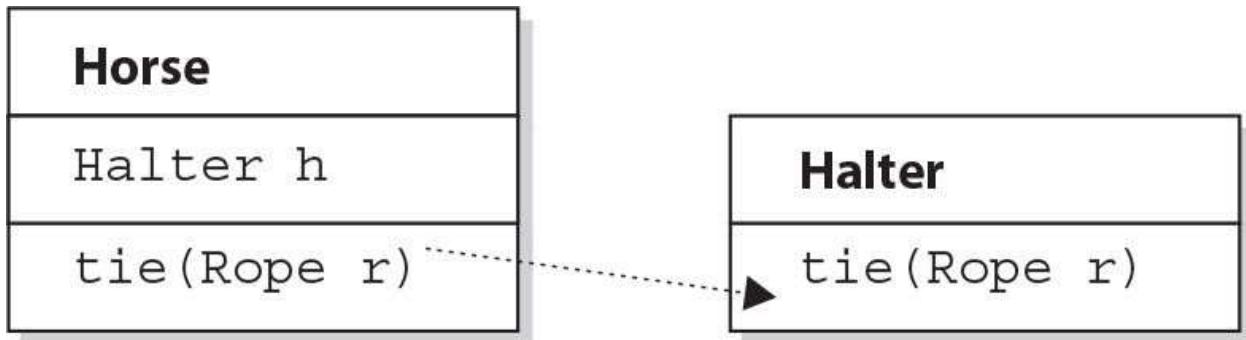
The code might look like this:

```
public class Animal { }
public class Horse extends Animal {
    private Halter myHalter;
}
```

In this code, the `Horse` class has an instance variable of type `Halter` (a halter is a piece of gear you might have if you have a horse), so you can say that a “`Horse HAS-A Halter`.” In other words, `Horse` has a reference to a `Halter`. `Horse` code can use that `Halter` reference to invoke methods on the `Halter` and get `Halter` behavior without having `Halter`-related code (methods) in the `Horse` class itself. [Figure 2-3](#) illustrates the HAS-A relationship between `Horse` and `Halter`.

FIGURE 2-3

HAS-A relationship between `Horse` and `Halter`



Horse class has a Halter, because Horse declares an instance variable of type Halter.
When code invokes `tie()` on a Horse instance, the Horse invokes `tie()` on the Horse object's Halter instance variable.

HAS-A relationships allow you to design classes that follow good OO practices by not having monolithic classes that do a gazillion different things. Classes (and their resulting objects) should be specialists. As our friend Andrew says, “Specialized classes can actually help reduce bugs.” The more specialized the class, the more likely it is that you can reuse the class in other applications. If you put all the `Halter`-related code directly into the `Horse` class, you’ll end up duplicating code in the `Cow` class, `UnpaidIntern` class, and any other class that might need `Halter` behavior. By keeping the `Halter` code in a separate specialized `Halter` class, you have the chance to reuse the `Halter` class in multiple applications.

Users of the `Horse` class (that is, code that calls methods on a `Horse` instance)

think that the `Horse` class has `Halter` behavior. The `Horse` class might have a `tie(LeadRope rope)` method, for example. Users of the `Horse` class should never have to know that when they invoke the `tie()` method, the `Horse` object turns around and delegates the call to its `Halter` class by invoking `myHalter.tie(rope)`. The scenario just described might look like this:

```
public class Horse extends Animal {  
    private Halter myHalter = new Halter();  
    public void tie(LeadRope rope) {  
        myHalter.tie(rope); // Delegate tie behavior to the  
                           // Halter object  
    }  
}  
  
public class Halter {  
    public void tie(LeadRope aRope) {  
        // Do the actual tie work here  
    }  
}
```

FROM THE CLASSROOM

Object-Oriented Design

IS-A and HAS-A relationships and encapsulation are just the tip of the iceberg when it comes to OO design. Many books and graduate theses have been dedicated to this topic. The reason for the emphasis on proper design is simple: money. The cost to deliver a software application has been estimated to be as much as ten times more expensive for poorly designed programs.

Even the best OO designers (often called “architects”) make mistakes. It

is difficult to visualize the relationships between hundreds, or even thousands, of classes. When mistakes are discovered during the implementation (code writing) phase of a project, the amount of code that must be rewritten can sometimes mean programming teams have to start over from scratch.

The software industry has evolved to aid the designer. Visual object modeling languages, such as the Unified Modeling Language (UML), allow designers to design and easily modify classes without having to write code first because OO components are represented graphically. This allows designers to create a map of the class relationships and helps them recognize errors before coding begins. Another innovation in OO design is design patterns. Designers noticed that many OO designs were applied consistently from project to project and that it was useful to apply the same designs because it reduced the potential to introduce new design errors. OO designers then started to share these designs with each other. Now there are many catalogs of these design patterns both on the Internet and in book form.

Although passing the Java certification exam does not require you to understand OO design this thoroughly, hopefully this background information will help you better appreciate why the test writers chose to (tacitly) include encapsulation and IS-A and HAS-A relationships on the exam.

—Jonathan Meeks,
Sun Certified Java Programmer

In OO, we don't want callers to worry about which class or object is actually doing the real work. To make that happen, the `Horse` class hides implementation details from `Horse` users. `Horse` users ask the `Horse` object to do things (in this case, tie itself up), and the `Horse` will either do it or, as in this example, ask something else (like perhaps an inherited `Animal` class method) to do it. To the caller, though, it always appears that the `Horse` object takes care of itself. Users of a `Horse` should not even need to know that there is such a thing as a `Halter` class.

CERTIFICATION OBJECTIVE

Polymorphism (OCP Objective 1.3)

1.3 Implement polymorphism.

Remember, any Java object that can pass more than one IS-A test can be considered polymorphic. Other than objects of type `Object`, *all* Java objects are polymorphic in that they pass the IS-A test for their own type and for class `Object`.

Remember, too, that the only way to access an object is through a reference variable. There are a few key things you should know about references:

- A reference variable can be of only one type, and once declared, that type can never be changed (although the object it references can change).
- A reference is a variable, so it can be reassigned to other objects (unless the reference is declared `final`).
- A reference variable's type determines the methods that can be invoked on the object the variable is referencing.
- A reference variable can refer to any object of the same type as the declared reference, or—this is the big one—**it can refer to any subtype of the declared type!**
- A reference variable can be declared as a class type or an interface type. If the variable is declared as an interface type, it can reference any object of any class that *implements* the interface.

Earlier we created a `GameShape` class that was extended by two other classes, `PlayerPiece` and `TilePiece`. Now imagine you want to animate some of the shapes on the gameboard. But not *all* shapes are able to be animated, so what do you do with class inheritance?

Could we create a class with an `animate()` method and have only *some* of the `GameShape` subclasses inherit from that class? If we can, then we could have `PlayerPiece`, for example, extend *both* the `GameShape` class and `Animatable` class, whereas the `TilePiece` would extend only `GameShape`. But no, this won't work! Java supports only single class inheritance! That means a class can have only one immediate superclass. In other words, if `PlayerPiece` is a class, there is no way to say something like this:

```
class PlayerPiece extends GameShape, Animatable { // NO!
    // more code
}
```

A *class* cannot *extend* more than one class: that means one parent per class. A class *can* have multiple ancestors, however, because class B could extend class A, and class C could extend class B, and so on. So any given class might have multiple classes up its inheritance tree, but that's not the same as saying a class directly extends two classes.



Some languages (such as C++) allow a class to extend more than one other class. This capability is known as “multiple inheritance.” The reason that Java’s creators chose not to allow multiple class inheritance is that it can become quite messy. In a nutshell, the problem is that if a class extended two other classes, and both superclasses had, say, a *doStuff()* method, which version of *doStuff()* would the subclass inherit? This issue can lead to a scenario sometimes called the “Deadly Diamond of Death,” because of the shape of the class diagram that can be created in a multiple inheritance design. The diamond is formed when classes B and C both extend A and both B and C inherit a method from A. If class D extends both B and C, and both B and C have overridden the method in A, class D has, in theory, inherited two different implementations of the same method. Drawn as a class diagram, the shape of the four classes looks like a diamond.



To reiterate, as of Java 8, interfaces can have concrete methods (marked default or static methods). This allows for a form of multiple inheritance, which we'll discuss later in the chapter.

So if that doesn’t work, what else could you do? You could simply put the

`animate()` code in `GameShape`, and then disable the method in classes that can't be animated. But that's a bad design choice for many reasons—it's more error-prone; it makes the `GameShape` class less cohesive; and it means the `GameShape` API “advertises” that all shapes can be animated when, in fact, that's not true since only some of the `GameShape` subclasses will be able to run the `animate()` method successfully.

So what *else* could you do? You already know the answer—create an `Animatable` *interface* and have only the `GameShape` subclasses that can be animated implement that interface. Here's the interface:

```
public interface Animatable {  
    public void animate();  
}
```

And here's the modified `PlayerPiece` class that implements the interface:

```
class PlayerPiece extends GameShape implements Animatable {  
    public void movePiece() {  
        System.out.println("moving game piece");  
    }  
    public void animate() {  
        System.out.println("animating...");  
    }  
    // more code  
}
```

So now we have a `PlayerPiece` that passes the IS-A test for both the `GameShape` class and the `Animatable` interface. That means a `PlayerPiece` can be treated polymorphically as one of four things at any given time, depending on the declared type of the reference variable:

- An `Object` (since any object inherits from `Object`)

- A GameShape (since PlayerPiece extends GameShape)
- A PlayerPiece (since that's what it really is)
- An Animatable (since PlayerPiece implements Animatable)

The following are all legal declarations. Look closely:

```
PlayerPiece player = new PlayerPiece();
Object o = player;
GameShape shape = player;
Animatable mover = player;
```

There's only one object here—an instance of type PlayerPiece—but there are four different types of reference variables, all referring to that one object on the heap. Pop quiz: Which of the preceding reference variables can invoke the `displayShape()` method? Hint: Only two of the four declarations can be used to invoke the `displayShape()` method.

Remember that method invocations allowed by the compiler are based solely on the declared type of the reference, regardless of the object type. So looking at the four reference types again—`Object`, `GameShape`, `PlayerPiece`, and `Animatable`—which of these four types know about the `displayShape()` method?

You guessed it—both the `GameShape` class and the `PlayerPiece` class are known (by the compiler) to have a `displayShape()` method, so either of those reference types can be used to invoke `displayShape()`. Remember that to the compiler, a `PlayerPiece` IS-A `GameShape`, so the compiler says, “I see that the declared type is `PlayerPiece`, and since `PlayerPiece` extends `GameShape`, that means `PlayerPiece` inherited the `displayShape()` method. Therefore, `PlayerPiece` can be used to invoke the `displayShape()` method.”

Which methods can be invoked when the `PlayerPiece` object is being referred to using a reference declared as type `Animatable`? Only the `animate()` method. Of course, the cool thing here is that any class from any inheritance tree can also implement `Animatable`, so that means if you have a method with an argument declared as type `Animatable`, you can pass in `PlayerPiece` objects, `SpinningLogo` objects, and anything else that's an instance of a class that implements `Animatable`. And you can use that parameter (of type `Animatable`) to invoke the `animate()` method but not the `displayShape()` method (which it

might not even have), or anything other than what is known to the compiler based on the reference type. The compiler always knows, though, that you can invoke the methods of class `Object` on any object, so those are safe to call regardless of the reference—class or interface—used to refer to the object.

We've left out one big part of all this, which is that even though the compiler only knows about the declared reference type, the Java Virtual Machine (JVM) at runtime knows what the object really is. And that means even if the `PlayerPiece` object's `displayShape()` method is called using a `GameShape` reference variable, if the `PlayerPiece` overrides the `displayShape()` method, the JVM will invoke the `PlayerPiece` version! The JVM looks at the real object at the other end of the reference, "sees" that it has overridden the method of the declared reference variable type, and invokes the method of the object's actual class. But there is one other thing to keep in mind:

Polymorphic method invocations apply only to *instance methods*. You can always refer to an object with a more general reference variable type (a superclass or interface), but at runtime, the ONLY things that are dynamically selected based on the actual *object* (rather than the *reference type*) are instance methods. Not static methods. Not variables. Only overridden instance methods are dynamically invoked based on the real object's type.

Because this definition depends on a clear understanding of overriding and the distinction between static methods and instance methods, we'll cover those later in the chapter.

CERTIFICATION OBJECTIVE

Overriding/Overloading (OCP Objectives 1.2, 1.3, and 2.5)

- 1.2 *Implement inheritance including visibility modifiers and composition.*
- 1.3 *Implement polymorphism.*
- 2.5 *Develop code that declares, implements and/or extends interfaces and use the @Override annotation.*

The exam will use overridden and overloaded methods on many, many questions. These two concepts are often confused (perhaps because they have

similar names?), but each has its own unique and complex set of rules. It's important to get really clear about which "over" uses which rules!

Overridden Methods

Any time a type inherits a method from a supertype, you have the opportunity to override the method (unless, as you learned earlier, the method is marked `final`). The key benefit of overriding is the ability to define behavior that's specific to a particular subtype. The following example demonstrates a `Horse` subclass of `Animal` overriding the `Animal` version of the `eat()` method:

```
public class Animal {  
    public void eat() {  
        System.out.println("Generic Animal Eating Generically");  
    }  
}  
  
class Horse extends Animal {  
    public void eat() {  
        System.out.println("Horse eating hay, oats, "  
                           + "and horse treats");  
    }  
}
```

For abstract methods, you inherit from a supertype; you have no choice: You *must implement* the method in the subtype ***unless the subtype is also abstract***. Abstract methods must be *implemented* by the first concrete subclass, but this is a lot like saying the concrete subclass *overrides* the abstract methods of the supertype(s). So you could think of abstract methods as methods you're forced to override—eventually.

The `Animal` class creator might have decided that for the purposes of polymorphism, all `Animal` subtypes should have an `eat()` method defined in a unique way. Polymorphically, when an `Animal` reference refers not to an `Animal` instance but to an `Animal` subclass instance, the caller should be able to invoke

`eat()` on the `Animal` reference; however, the actual runtime object (say, a `Horse` instance) will run its own specific `eat()` method. Marking the `eat()` method abstract is the `Animal` programmer's way of saying to all subclass developers, "It doesn't make any sense for your new subtype to use a generic `eat()` method, so you have to come up with your *own* `eat()` method implementation!" A (nonabstract) example of using polymorphism looks like this:

```
public class TestAnimals {
    public static void main (String [] args) {
        Animal a = new Animal();
        Animal b = new Horse(); // Animal ref, but a Horse object
        a.eat(); // Runs the Animal version of eat()
        b.eat(); // Runs the Horse version of eat()
    }
}
class Animal {
    public void eat() {
        System.out.println("Generic Animal Eating Generically");
    }
}
class Horse extends Animal {
    public void eat() {
        System.out.println("Horse eating hay, oats, "
                           + "and horse treats");
    }
    public void buck() { }
}
```

In the preceding code, the test class uses an `Animal` reference to invoke a method on a `Horse` object. Remember, the compiler will allow only methods in class `Animal` to be invoked when using a reference to an `Animal`. The following would not be legal given the preceding code:

```
Animal c = new Horse();
c.buck(); // Can't invoke buck();
// Animal class doesn't have that method
```

To reiterate, the compiler looks only at the reference type, not the instance type. Polymorphism lets you use a more abstract supertype (including an interface) reference to one of its subtypes (including interface implementers).

The overriding method cannot have a more restrictive access modifier than the method being overridden (for example, you can't override a method marked `public` and make it `protected`). Think about it: If the `Animal` class advertises a `public eat()` method and someone has an `Animal` reference (in other words, a reference declared as type `Animal`), that someone will assume it's safe to call `eat()` on the `Animal` reference regardless of the actual instance that the `Animal` reference is referring to. If a subtype were allowed to sneak in and change the access modifier on the overriding method, then suddenly at runtime—when the JVM invokes the true object's (`Horse`) version of the method rather than the reference type's (`Animal`) version—the program would die a horrible death. (Not to mention the emotional distress for the one who was betrayed by the rogue subtype.)

Let's modify the polymorphic example you saw earlier in this section:

```

public class TestAnimals {
    public static void main (String [] args) {
        Animal a = new Animal();
        Animal b = new Horse(); // Animal ref, but a Horse object
        a.eat(); // Runs the Animal version of eat()
        b.eat(); // Runs the Horse version of eat()
    }
}
class Animal {
    public void eat() {
        System.out.println("Generic Animal Eating Generically");
    }
}
class Horse extends Animal {
    private void eat() { // whoa! - it's private!
        System.out.println("Horse eating hay, oats, "
                           + "and horse treats");
    }
}

```

If this code compiled (which it doesn't), the following would fail at runtime:

```

Animal b = new Horse(); // Animal ref, but a Horse
// object, so far so good
b.eat(); // Meltdown at runtime!

```

The variable `b` is of type `Animal`, which has a `public eat()` method. But remember that at runtime, Java uses virtual method invocation to dynamically

select the actual version of the method that will run, based on the actual instance. An `Animal` reference can always refer to a `Horse` instance because `Horse` IS-A(n) `Animal`. What makes that supertype reference to a subtype instance possible is that the subtype is guaranteed to be able to do everything the supertype can do. Whether the `Horse` instance overrides the inherited methods of `Animal` or simply inherits them, anyone with an `Animal` reference to a `Horse` instance is free to call all accessible `Animal` methods. For that reason, an overriding method must fulfill the contract of the superclass.

Note: In [Chapter 3](#) we will explore exception handling in detail. Once you've studied [Chapter 3](#), you'll appreciate this single handy list of overriding rules. The rules for overriding a method are as follows:

- The argument list must exactly match that of the overridden method. If they don't match, you can end up with an overloaded method you didn't intend.
- The return type must be the same as, or a subtype of, the return type declared in the original overridden method in the superclass. (More on this in a few pages when we discuss covariant returns.)
- The access level can't be more restrictive than that of the overridden method.
- The access level CAN be less restrictive than that of the overridden method.
- Instance methods can be overridden only if they are inherited by the subtype. A subtype within the same package as the instance's supertype can override any supertype method that is not marked `private` or `final`. A subtype in a different package can override only those nonfinal methods marked `public` or `protected` (since `protected` methods are inherited by the subtype).
- The overriding method CAN throw any unchecked (runtime) exception, regardless of whether the overridden method declares the exception.
- The overriding method must NOT throw checked exceptions that are new or broader than those declared by the overridden method. For example, a method that declares a `FileNotFoundException` cannot be overridden by a method that declares a `SQLException`, `Exception`, or any other non-runtime exception unless it's a subclass of `FileNotFoundException`.
- The overriding method can throw narrower or fewer exceptions. Just because an overridden method "takes risks" doesn't mean that the

overriding subtype's exception takes the same risks. Bottom line: an overriding method doesn't have to declare any exceptions that it will never throw, regardless of what the overridden method declares.

- You cannot override a method marked `final`.
- You cannot override a method marked `static`. We'll look at an example in a few pages when we discuss `static` methods in more detail.
- If a method can't be inherited, you cannot override it. Remember that overriding implies that you're reimplementing a method you inherited! For example, the following code is not legal, and even if you added an `eat()` method to `Horse`, it wouldn't be an override of `Animal`'s `eat()` method.

```
public class TestAnimals {  
    public static void main (String [] args) {  
        Horse h = new Horse();  
        h.eat(); // Not legal because Horse didn't inherit eat()  
    }  
}  
class Animal {  
    private void eat() {  
        System.out.println("Generic Animal Eating Generically");  
    }  
}  
class Horse extends Animal { }
```

Invoking a Supertype Version of an Overridden Method

Often, you'll want to take advantage of some of the code in the supertype version of a method, yet still override it to provide some additional specific behavior. It's like saying, "Run the supertype version of the method, and then come back down here and finish with my subtype additional method code." (Note that there's no requirement that the supertype version run before the

subtype code.) It's easy to do in code using the keyword `super` as follows:

```
public class Animal {  
    public void eat() {}  
    public void printYourself() {  
        // Useful printing code goes here  
    }  
}  
  
class Horse extends Animal {  
    public void printYourself() {  
        // Take advantage of Animal code, then add some more  
        super.printYourself(); // Invoke the superclass  
        // (Animal) code  
        // Then do Horse-specific  
        // print work here  
    }  
}
```

In a similar way, you can access an interface's overridden method with the syntax:

```
InterfaceX.super.doStuff();
```

Note: Using `super` to invoke an overridden method applies only to instance methods. (Remember that `static` methods can't be overridden.) And you can use `super` only to access a method in a type's supertype, not the supertype of the supertype—that is, you **cannot** say `super.super.doStuff()` and you **cannot** say `InterfaceX.super.super.doStuff()`.

If a method is overridden but you use a polymorphic (supertype) reference to refer to the subtype object with the overriding method, the compiler assumes you're calling the supertype version of the method. If the supertype version declares a checked exception, but the overriding subtype method does not, the compiler still thinks you are calling a method that declares an exception. Let's look at an example:

```
class Animal {  
    public void eat() throws Exception {  
        // throws an Exception  
    }  
}  
class Dog2 extends Animal {  
    public void eat() { /* no Exceptions */}  
    public static void main(String [] args) {  
        Animal a = new Dog2();  
        Dog2 d = new Dog2();  
        d.eat();           // ok  
        a.eat();           // compiler error -  
                           // unreported exception  
    }  
}
```

This code will not compile because of the exception declared on the Animal eat() method. This happens even though, at runtime, the eat() method used would be the Dog version, which does not declare the exception.

Examples of Illegal Method Overrides

Let's take a look at overriding the `eat()` method of `Animal`:

```
public class Animal {  
    public void eat() { }  
}
```

[Table 2-2](#) lists examples of illegal overrides of the `Animal eat()` method, given the preceding version of the `Animal` class.

TABLE 2-2 Examples of Illegal Overrides

Illegal Override Code	Problem with the Code
<code>private void eat() { }</code>	Access modifier more restrictive
<code>public void eat() throws IOException { }</code>	Declares a checked exception not defined by superclass version
<code>public void eat(String food) { }</code>	A legal overload, not an override, because the argument list changed
<code>public String eat() { }</code>	Not an override because of the return type, and not an overload either because there's no change in the argument list

Using `@Override`

Java 5 introduced the `@override` annotation, which you can use to help catch errors with overriding or implementing at compile time. If you intend to override a method in a superclass or implement a method in an interface, you can annotate that method with `@override`, like this:

```
public class Animal {
    public void eat() {
        System.out.println("Generic Animal Eating Generically");
    }
}

class Horse extends Animal {
    @Override // ask the compiler for verification
    public void eat() {
        System.out.println("Horse eating hay, oats, "
            + "and horse treats");
    }
}
```

Now, if you make a mistake in how you override the `eat()` method, the compiler will warn you. For instance, let's say you accidentally add a parameter to the `eat()` method:

```
public void eat(int j) {
    System.out.println("Horse eating hay, oats, "
        + "and horse treats");
}
```

Without the `@Override`, the Java compiler is fine with this code, but what you've done is *overload* the `eat()` method, not *override* it. If you then call the `eat()` method on a `Horse`, without passing in an `int`, you'll get the `Animal`'s `eat()` method, not the `Horse`'s `eat()` method, as you intended.

Add the `@Override`, however, and, you'll see a compiler error something like:

```
Animal.java:7: error: method does not override or implement a method from a supertype
@Override
^
```

This also works if you, let's say, misspell the name of the method you want to override:

```
public class Animal {
    public void eat() {
        System.out.println("Generic Animal Eating Generically");
    }
    public void walk() { } // method you intend to override
}
class Horse extends Animal {
    @Override
    public void eat() {
        System.out.println("Horse eating hay, oats, "
            + "and horse treats");
    }
    @Override
    public void walkie() { } // you meant to override, but spelled it wrong
}
```

Again, you'll see a compiler error indicating that something is wrong.

`@Override` is not required when overriding or implementing, but it can help prevent mistakes, and it also makes it easier for other programmers reading your code to know what you intended.

Overloaded Methods

Overloaded methods let you reuse the same method name in a class, but with different arguments (and, optionally, a different return type). Overloading a

method often means you're being a little nicer to those who call your methods because your code takes on the burden of coping with different argument types rather than forcing the caller to do conversions prior to invoking your method. The rules aren't too complex:

- Overloaded methods **MUST** change the argument list.
- Overloaded methods **CAN** change the return type.
- Overloaded methods **CAN** change the access modifier.
- Overloaded methods **CAN** declare new or broader checked exceptions.
- A method can be overloaded in the *same* type or in a *subtype*. In other words, if class A defines a `doStuff(int i)` method, then subclass B could define a `doStuff(String s)` method without overriding the superclass version that takes an `int`. So two methods with the same name but in different types can still be considered overloaded if the subtype inherits one version of the method and then declares another overloaded version in its type definition.



Less experienced Java developers are often confused about the subtle differences between overloaded and overridden methods. Be careful to recognize when a method is overloaded rather than overridden. You might see a method that appears to be violating a rule for overriding, but that is actually a legal overload, as follows:

```
public class Foo {  
    public void doStuff(int y, String s) {}  
    public void moreThings(int x) {}  
}  
class Bar extends Foo {  
    public void doStuff(int y, long s) throws IOException {}  
}
```

It's tempting to see the IOException as the problem because the overridden doStuff() method doesn't declare an exception and IOException is checked by the compiler. But the doStuff() method is not overridden! Subclass Bar overloads the doStuff() method by varying the argument list, so all the code, including the IOException, is fine.

Legal Overloads

Let's look at a method we want to overload:

```
public void changeSize(int size, String name, float pattern) { }
```

The following methods are legal overloads of the changeSize() method:

```
.c void changeSize(int size, String name) { }
ite int changeSize(int size, float pattern) { }
.c void changeSize(float pattern, String name)
throws IOException { }
```

Invoking Overloaded Methods

When a method is invoked, more than one method of the same name might exist for the object type you're invoking a method on. For example, the Horse class might have three methods with the same name but with different argument lists, which means the method is overloaded.

Decide which of the matching methods to invoke based on the arguments. If you invoke the method with a `String` argument, the overloaded version that takes a `String` is called. If you invoke a method of the same name but pass it a `float`, the overloaded version that takes a `float` will run. If you invoke the method of the same name but pass it a `Foo` object and there isn't an overloaded version that takes a `Foo`, then the compiler will complain that it can't find a match. The following are examples of invoking overloaded methods:

```

class Adder {
    public int addThem(int x, int y) {
        return x + y;
    }

    // Overload the addThem method to add doubles instead of ints
    public double addThem(double x, double y) {
        return x + y;
    }
}

// From another class, invoke the addThem() method
public class TestAdder {
    public static void main (String [] args) {
        Adder a = new Adder();
        int b = 27;
        int c = 3;
        int result = a.addThem(b,c);           // Which addThem is invoked?
        double doubleResult = a.addThem(22.5,9.3); // Which addThem?
    }
}

```

In this `TestAdder` code, the first call to `a.addThem(b, c)` passes two `ints` to the method, so the first version of `addThem()`—the overloaded version that takes two `int` arguments—is called. The second call to `a.addThem(22.5, 9.3)` passes two `doubles` to the method, so the second version of `addThem()`—the overloaded version that takes two `double` arguments—is called.

Invoking overloaded methods that take object references rather than primitives is a little more interesting. Say you have an overloaded method such that one version takes an `Animal` and one takes a `Horse` (subclass of `Animal`). If you pass a `Horse` object in the method invocation, you'll invoke the overloaded

version that takes a Horse. Or so it looks at first glance:

```
class Animal { }
class Horse extends Animal { }
class UseAnimals {
    public void doStuff(Animal a) {
        System.out.println("In the Animal version");
    }
    public void doStuff(Horse h) {
        System.out.println("In the Horse version");
    }
    public static void main (String [] args) {
        UseAnimals ua = new UseAnimals();
        Animal animalObj = new Animal();
        Horse horseObj = new Horse();
        ua.doStuff(animalObj);
        ua.doStuff(horseObj);
    }
}
```

The output is what you expect:

```
In the Animal version
In the Horse version
```

But what if you use an Animal reference to a Horse object?

```
Animal animalRefToHorse = new Horse();
ua.doStuff(animalRefToHorse);
```

Which of the overloaded versions is invoked? You might want to answer, “The one that takes a Horse since it’s a Horse object at runtime that’s being passed to the method.” But that’s not how it works. The preceding code would actually print this:

```
in the Animal version
```

Even though the actual object at runtime is a Horse and not an Animal, the choice of which overloaded method to call (in other words, the signature of the method) is NOT dynamically decided at runtime.

Just remember—the *reference* type (not the object type) determines which overloaded method is invoked!

To summarize, which overridden version of the method to call (in other words, from which class in the inheritance tree) is decided at *runtime* based on *object* type, but which overloaded version of the method to call is based on the *reference* type of the argument passed at *compile* time.

If you invoke a method passing it an Animal reference to a Horse object, the compiler knows only about the Animal, so it chooses the overloaded version of the method that takes an Animal. It does not matter that, at runtime, a Horse is actually being passed.



Can main() be overloaded?

```
class DuoMain {  
    public static void main(String[] args) {  
        main(1);  
    }  
    static void main(int i) {  
        System.out.println("overloaded main");  
    }  
}
```

Absolutely! But the only `main()` with JVM superpowers is the one with the signature you've seen about 100 times already in this book.

Polymorphism in Overloaded and Overridden Methods How does polymorphism work with overloaded methods? From what we just looked at, it doesn't appear that polymorphism matters when a method is overloaded. If you pass an `Animal` reference, the overloaded method that takes an `Animal` will be invoked, even if the actual object passed is a `Horse`. Once the `Horse` masquerading as `Animal` gets in to the method, however, the `Horse` object is still a `Horse` despite being passed into a method expecting an `Animal`. So it's true that polymorphism doesn't determine which overloaded version is called; polymorphism does come into play when the decision is about which overridden version of a method is called. But sometimes a method is both overloaded and overridden. Imagine that the `Animal` and `Horse` classes look like this:

```
public class Animal {
    public void eat() {
        System.out.println("Generic Animal Eating Generically");
    }
}
public class Horse extends Animal {
    public void eat() {
        System.out.println("Horse eating hay ");
    }
    public void eat(String s) {
        System.out.println("Horse eating " + s);
    }
}
```

Notice that the `Horse` class has both overloaded and overridden the `eat()` method. [Table 2-3](#) shows which version of the three `eat()` methods will run

depending on how they are invoked.

TABLE 2-3 Examples of Legal and Illegal Overrides

Method Invocation Code	Result
Animal a = new Animal(); a.eat();	Generic Animal Eating Generically
Horse h = new Horse(); h.eat();	Horse eating hay
Animal ah = new Horse(); ah.eat();	Horse eating hay Polymorphism works—the actual object type (<i>Horse</i>), not the reference type (<i>Animal</i>), is used to determine which <i>eat ()</i> is called.
Horse he = new Horse(); he.eat ("Apples");	Horse eating Apples The overloaded <i>eat (String s)</i> method is invoked.
Animal a2 = new Animal(); a2.eat ("treats");	Compiler error! Compiler sees that the <i>Animal</i> class doesn't have an <i>eat ()</i> method that takes a <i>String</i> .
Animal ah2 = new Horse(); ah2.eat ("Carrots");	Compiler error! Compiler still looks only at the reference and sees that <i>Animal</i> doesn't have an <i>eat ()</i> method that takes a <i>String</i> . Compiler doesn't care that the actual object might be a <i>Horse</i> at runtime.

Don't be fooled by a method that's overloaded but not overridden by a subclass. It's perfectly legal to do the following:

```
public class Foo {  
    void doStuff() { }  
}  
class Bar extends Foo {  
    void doStuff(String s) { }  
}
```

The Bar class has two doStuff() methods: the no-arg version it inherits from Foo (and does not override) and the overloaded doStuff(String s) defined in the Bar class. Code with a reference to a Foo can invoke only the no-arg version, but code with a reference to a Bar can invoke either of the overloaded versions.

Table 2-4 summarizes the difference between overloaded and overridden methods.

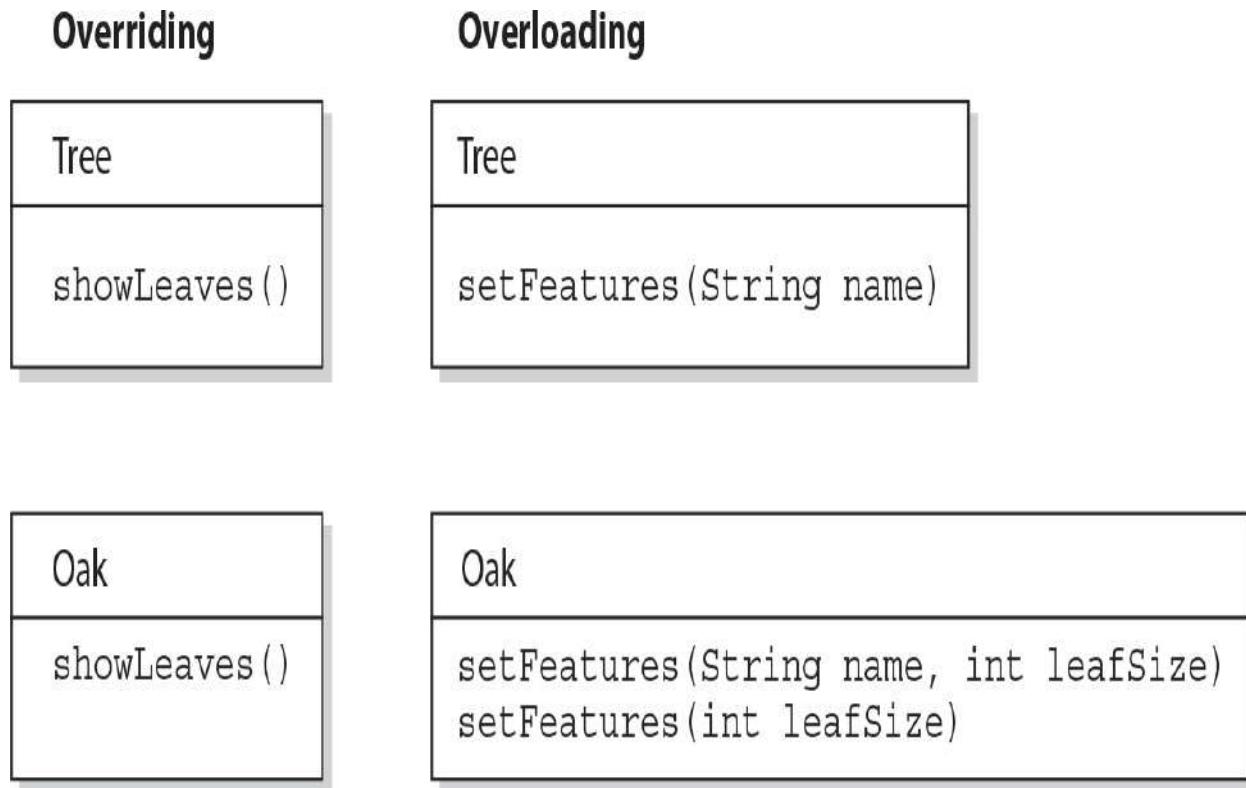
TABLE 2-4 Differences Between Overloaded and Overridden Methods

Overloaded Method		Overridden Method
Argument(s)	Must change.	Must not change.
Return type	Can change.	Can't change except for covariant returns. (Covered later this chapter.)
Exceptions	Can change.	Can reduce or eliminate. Must not throw new or broader checked exceptions.
Access	Can change.	Must not make more restrictive (can be less restrictive).
Invocation	<p><i>Reference</i> type determines which overloaded version (based on declared argument types) is selected. Happens at <i>compile</i> time. The actual <i>method</i> that's invoked is still a virtual method invocation that happens at runtime, but the compiler will already know the <i>signature</i> of the method to be invoked. So at runtime, the argument match will already have been nailed down, just not the <i>class</i> in which the method lives.</p>	<p><i>Object</i> type (in other words, <i>the type of the actual instance on the heap</i>) determines which method is selected. Happens at <i>runtime</i>.</p>

We'll cover constructor overloading later in the chapter, where we'll also cover the other constructor-related topics that are on the exam. [Figure 2-4](#) illustrates the way overloaded and overridden methods appear in class relationships.

FIGURE 2-4

Overloaded and overridden methods in class relationships



CERTIFICATION OBJECTIVE

Casting (OCP Objectives 1.2 and 1.3)

1.2 *Implement inheritance including visibility modifiers and composition.*

1.3 *Implement polymorphism.*

You've seen how it's both possible and common to use general reference variable types to refer to more specific object types. It's at the heart of polymorphism. For example, this line of code should be second nature by now:

```
Animal animal = new Dog();
```

But what happens when you want to use that `animal` reference variable to invoke a method that only class `Dog` has? You know it's referring to a `Dog`, and you want to do a `Dog`-specific thing? In the following code, we've got an array of

Animals, and whenever we find a Dog in the array, we want to do a special Dog thing. Let's agree for now that all this code is okay, except we're not sure about the line of code that invokes the playDead method:

```
class Animal {  
    void makeNoise() {System.out.println("generic noise"); }  
}  
class Dog extends Animal {  
    void makeNoise() {System.out.println("bark"); }  
    void playDead() { System.out.println("roll over"); }  
}  
  
class CastTest2 {  
    public static void main(String [] args) {  
        Animal [] a = {new Animal(), new Dog(), new Animal() };  
        for(Animal animal : a) {  
            animal.makeNoise();  
            if(animal instanceof Dog) {  
                animal.playDead();          // try to do a Dog behavior?  
            }  
        }  
    }  
}
```

When we try to compile this code, the compiler says something like this:

```
cannot find symbol
```

The compiler is saying, “Hey, class Animal doesn't have a playDead() method.”

Let's modify the `if` code block:

```
if(animal instanceof Dog) {  
    Dog d = (Dog) animal;      // casting the ref. var.  
    d.playDead();  
}
```

The new and improved code block contains a cast, which in this case is sometimes called a *downcast*, because we're casting down the inheritance tree to a more specific class. Now the compiler is happy. Before we try to invoke `playDead`, we cast the `animal` variable to type `Dog`. What we're saying to the compiler is, "We know it's really referring to a `Dog` object, so it's okay to make a new `Dog` reference variable to refer to that object." In this case we're safe, because before we ever try the cast, we do an `instanceof` test to make sure.

It's important to know that the compiler is forced to trust us when we do a downcast, even when we screw up:

```
class Animal {}  
class Dog extends Animal {}  
class DogTest {  
    public static void main(String [] args) {  
        Animal animal = new Animal();  
        Dog d = (Dog) animal;          // compiles but fails later  
    }  
}
```

It can be maddening! This code compiles! But when we try to run it, we'll get an exception, something like this:

```
java.lang.ClassCastException
```

Why can't we trust the compiler to help us out here? Can't it see that `animal`

is of type `Animal`? All the compiler can do is verify that the two types are in the same inheritance tree, so depending on whatever code might have come before the downcast, it's possible that `animal` is of type `Dog`. The compiler must allow things that might possibly work at runtime. However, if the compiler knows with certainty that the cast could not possibly work, compilation will fail. The following replacement code block will NOT compile:

```
Animal animal = new Animal();
Dog d = (Dog) animal;
String s = (String) animal;    // animal can't EVER be a String
```

In this case, you'll get an error something like this:

```
inconvertible types:Animal cannot be converted to a String
```

Unlike downcasting, *upcasting* (casting *up* the inheritance tree to a more general type) works implicitly (that is, you don't have to type in the cast) because when you upcast you're implicitly restricting the number of methods you can invoke, as opposed to *downcasting*, which implies that later on you might want to invoke a more *specific* method. Here's an example:

```
class Animal { }
class Dog extends Animal { }

class DogTest {
    public static void main(String [] args) {
        Dog d = new Dog();
        Animal a1 = d;           // upcast ok with no explicit cast
        Animal a2 = (Animal) d;  // upcast ok with an explicit cast
    }
}
```

Both of the previous upcasts will compile and run without exception because a Dog IS-A(n) Animal, which means that anything an Animal can do, a Dog can do. A Dog can do more, of course, but the point is that anyone with an Animal reference can safely call Animal methods on a Dog instance. The Animal methods may have been overridden in the Dog class, but all we care about now is that a Dog can always do at least everything an Animal can do. The compiler and JVM know it, too, so the implicit upcast is always legal for assigning an object of a subtype to a reference of one of its supertype classes (or interfaces). If Dog implements Pet and Pet defines beFriendly(), then a Dog can be implicitly cast to a Pet, but the only Dog method you can invoke then is beFriendly(), which Dog was forced to implement because Dog implements the Pet interface.

One more thing...if Dog implements Pet, then, if Beagle extends Dog but Beagle does not declare that it implements Pet, Beagle is still a Pet! Beagle is a Pet simply because it extends Dog, and Dog's already taken care of the Pet parts for itself and for all its children. The Beagle class can always override any method it inherits from Dog, including methods that Dog implemented to fulfill its interface contract.

And just one more thing...if Beagle does declare that it implements Pet, just so that others looking at the Beagle class API can easily see that Beagle IS-A Pet without having to look at Beagle's superclasses, Beagle still doesn't need to implement the beFriendly() method if the Dog class (Beagle's superclass) has already taken care of that. In other words, if Beagle IS-A Dog and Dog IS-A Pet, then Beagle IS-A Pet and has already met its Pet obligations for implementing the beFriendly() method since it inherits the beFriendly() method. The compiler is smart enough to say, "I know Beagle already IS a Dog, but it's okay to make it more obvious by adding a cast."

So don't be fooled by code that shows a concrete class that declares it implements an interface but doesn't implement the *methods* of the interface. Before you can tell whether the code is legal, you must know what the supertypes of this implementing class have declared. If any supertype in its inheritance tree has already provided concrete (that is, nonabstract) method implementations, then regardless of whether the supertype declares that it implements the interface, the subclass is under no obligation to reimplement (override) those methods.



The exam creators will tell you that they're forced to jam tons of code into little spaces "because of the exam engine." Although that's partially true, they also like to obfuscate. The following code

```
Animal a = new Dog();  
Dog d = (Dog) a;  
d.doDogStuff();
```

can be replaced with this easy-to-read bit of fun:

```
Animal a = new Dog();  
(Dog) a).doDogStuff();
```

In this case the compiler needs all those parentheses; otherwise, it thinks it's been handed an incomplete statement.

CERTIFICATION OBJECTIVE

Implementing an Interface (OCP Objective 2.5)

2.5 *Develop code that declares, implements and/or extends interfaces and use the @Override annotation (sic*).*

* This objective tests for two different concepts: the use of interfaces in general and the use of the @Override annotation. As we mentioned in the introduction, some of the official objectives aren't very well worded. When in doubt about how broadly the exam covers a topic, we suggest that you assume a broader scope, rather than a narrower scope.

When you implement an interface, you're agreeing to adhere to the contract defined in the interface. That means you're agreeing to provide legal implementations for every abstract method defined in the interface, and that anyone who knows what the interface methods look like (not how they're implemented, but how they can be called and what they return) can rest assured that they can invoke those methods on an instance of your implementing class.

For example, if you create a class that implements the `Runnable` interface (so your code can be executed by a specific thread), you must provide the `public void run()` method. Otherwise, the poor thread could be told to go execute your `Runnable` object's code and—surprise, surprise—the thread then discovers the object has no `run()` method! (At which point, the thread would blow up and the JVM would crash in a spectacular yet horrible explosion.) Thankfully, Java prevents this meltdown from occurring by running a compiler check on any class that claims to implement an interface. If the class says it's implementing an interface, it darn well better have an implementation for each abstract method in the interface (with a few exceptions that we'll look at in a moment).

Assuming an interface `Bounceable`, with two methods, `bounce()` and `setBounceFactor()`, the following class will compile:

```
public class Ball implements Bounceable { // Keyword
                                         // 'implements'
    public void bounce() { }
    public void setBounceFactor(int bf) { }
}
```

Okay, we know what you're thinking: "This has got to be the worst implementation class in the history of implementation classes." It compiles, though. And it runs. The interface contract guarantees that a class will have the method (in other words, others can call the method subject to access control), but it never guaranteed a good implementation—or even any actual implementation code in the body of the method. (Keep in mind, though, that if the interface declares that a method is NOT `void`, your class's implementation code has to include a return statement.) The compiler will never say, "Um, excuse me, but did you really mean to put nothing between those curly braces? HELLO. This is a method after all, so shouldn't it do something?"

Implementation classes must adhere to the same rules for method implementation as a class extending an abstract class. To be a legal implementation class, a nonabstract implementation class must do the following:

- Provide concrete (nonabstract) implementations for all abstract methods from the declared interface.
- Follow all the rules for legal overrides, such as the following:

- Declare no checked exceptions on implementation methods other than those declared by the interface method, or subclasses of those declared by the interface method.
- Maintain the signature of the interface method, and maintain the same return type (or a subtype). (But it does not have to declare the exceptions declared in the interface method declaration.)



Implementation classes are NOT required to implement an interface's static or default methods. We'll discuss this in more depth later in the chapter.

But wait, there's more! An implementation class can itself be abstract! For example, the following is legal for a class Ball implementing Bounceable:

```
abstract class Ball implements Bounceable { }
```

Notice anything missing? We never provided the implementation methods. And that's okay. If the implementation class is abstract, it can simply pass the buck to its first concrete subclass. For example, if class BeachBall extends Ball and BeachBall is not abstract, then BeachBall has to provide an implementation for all the abstract methods from Bounceable:

```
class BeachBall extends Ball {  
    // Even though we don't say it in the class declaration above,  
    // BeachBall implements Bounceable, since BeachBall's abstract  
    // superclass (Ball) implements Bounceable
```

```
public void bounce() {  
    // interesting BeachBall-specific bounce code  
  
}  
public void setBounceFactor(int bf) {  
    // clever BeachBall-specific code for setting  
    // a bounce factor  
  
}  
// if class Ball defined any abstract methods,  
// they'll have to be  
// implemented here as well.  
}
```

Look for classes that claim to implement an interface but don't provide the correct method implementations. Unless the implementing class is abstract, the implementing class must provide implementations for all abstract methods defined in the interface.

You need to know two more rules, and then we can put this topic to sleep (or put you to sleep; we always get those two confused):

1. A class can implement more than one interface. It's perfectly legal to say, for example, the following:

```
public class Ball implements Bounceable, Serializable,  
Runnable { ... }
```

You can extend only one class, but you can implement many interfaces (which, as of Java 8, means a form of multiple inheritance, which we'll discuss shortly). In other words, subclassing defines who and what you are, whereas implementing defines a role you can play or a hat you can wear, despite how different you might be from some other class implementing the same interface (but from a different inheritance tree). For example, a Person extends HumanBeing (although for some, that's

debatable). But a Person may also implement Programmer, Snowboarder, Employee, Parent, or PersonCrazyEnoughToTakeThisExam.

2. An interface can itself extend another interface. The following code is perfectly legal:

```
public interface Bounceable extends Moveable { } // ok!
```

What does that mean? The first concrete (nonabstract) implementation class of Bounceable must implement all the abstract methods of Bounceable, plus all the abstract methods of Moveable! The subinterface, as we call it, simply adds more requirements to the contract of the superinterface. You'll see this concept applied in many areas of Java, especially Java EE, where you'll often have to build your own interface that extends one of the Java EE interfaces.

Hold on, though, because here's where it gets strange. An interface can extend more than one interface! Think about that for a moment. You know that when we're talking about classes, the following is illegal:

```
public class Programmer extends Employee, Geek { } // Illegal!
```

As we mentioned earlier, a class is not allowed to extend multiple classes in Java. An interface, however, is free to extend multiple interfaces:

```
interface Bounceable extends Moveable, Spherical { // ok!
    void bounce();
    void setBounceFactor(int bf);
}

interface Moveable {
    void moveIt();
}

interface Spherical {
    void doSphericalThing();
}
```

In the next example, Ball is required to implement Bounceable, plus all

abstract methods from the interfaces that `Bounceable` extends (including any interfaces those interfaces extend, and so on, until you reach the top of the stack —or is it the bottom of the stack?). So `Ball` would need to look like the following:

```
class Ball implements Bounceable {  
  
    public void bounce() { }          // Implement Bounceable's methods  
    public void setBounceFactor(int bf) { }  
  
    public void moveIt() { }          // Implement Moveable's method  
  
    public void doSphericalThing() { } // Implement Spherical  
}  

```

If class `Ball` fails to implement any of the abstract methods from `Bounceable`, `Moveable`, or `Spherical`, the compiler will jump up and down wildly, red in the face, until it does. Unless, that is, class `Ball` is marked `abstract`. In that case, `Ball` could choose to implement some, all, or none of the abstract methods from any of the interfaces, thus leaving the rest of the implementations to a concrete subclass of `Ball`, as follows:

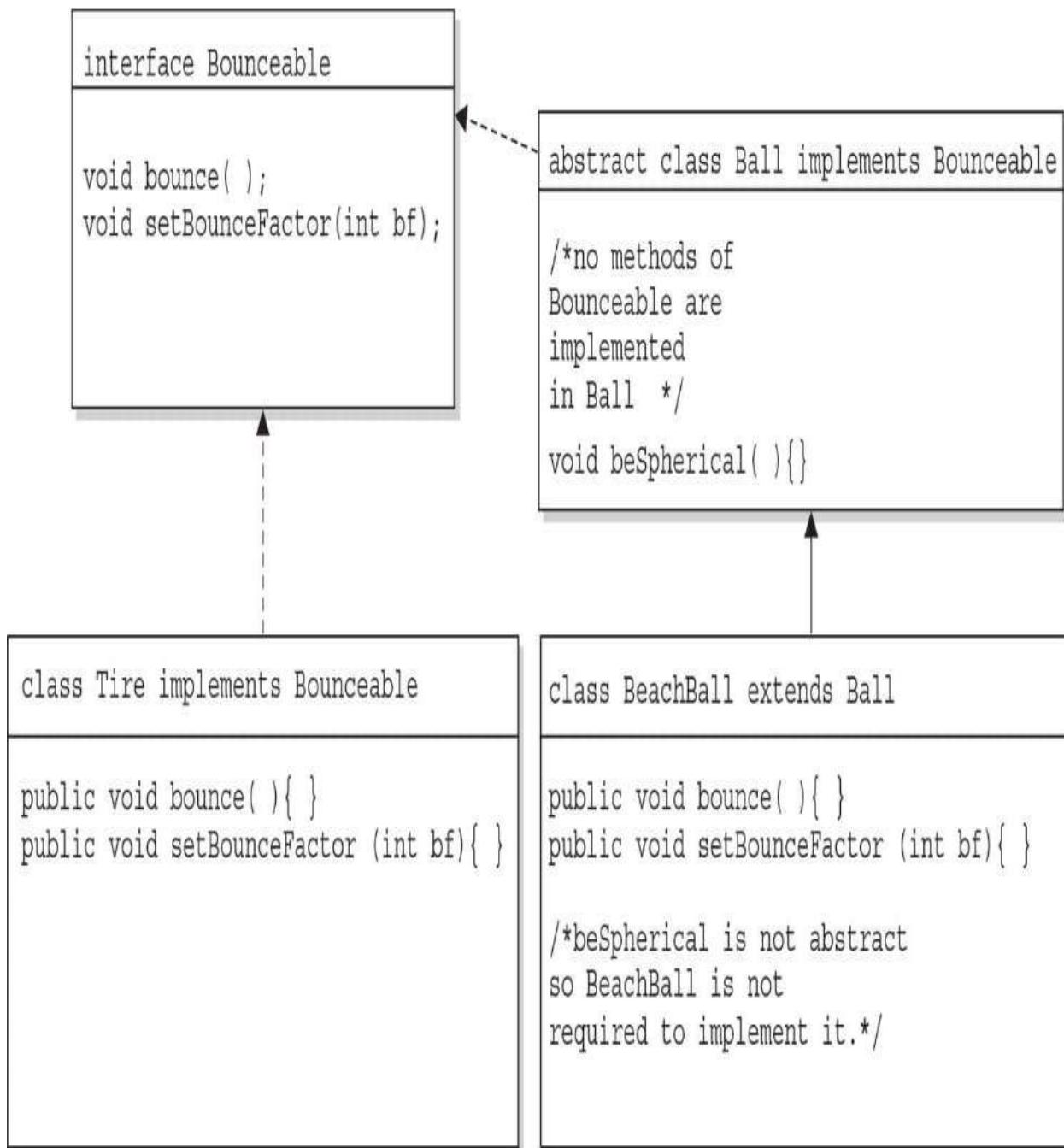
```
abstract class Ball implements Bounceable {  
    public void bounce() { ... }    // Define bounce behavior  
    public void setBounceFactor(int bf) { ... }  
    // Don't implement the rest; leave it for a subclass  
}
```

```
class SoccerBall extends Ball { // class SoccerBall must
                                // implement the interface
                                // methods that Ball didn't
    public void moveIt() { ... }
    public void doSphericalThing() { ... }
    // SoccerBall can choose to override the Bounceable methods
    // implemented by Ball
    public void bounce() { ... }
}
```

[Figure 2-5](#) compares concrete and abstract examples of extends and implements, for both classes and interfaces.

FIGURE 2-5

Comparing concrete and abstract examples of extends and implements



Because **BeachBall** is the first concrete class to implement **Bounceable**, it must provide implementations for all methods of **Bounceable**, except those defined in the abstract class **Ball**. Because **Ball** did not provide implementations of **Bounceable** methods, **BeachBall** was required to implement all of them.



Look for illegal uses of extends and implements. The following shows examples of legal and illegal class and interface declarations:

```

class Foo { }                                // OK
class Bar implements Foo { }                 // No! Can't implement a class
interface Baz { }                           // OK
interface Fi { }                            // OK
interface Fee implements Baz { }           // No! an interface can't
                                            // implement an interface
interface Zee implements Foo { }           // No! an interface can't
                                            // implement a class
interface Zoo extends Foo { }              // No! an interface can't
                                            // extend a class
interface Boo extends Fi { }               // OK. An interface can extend
                                            // an interface
class Toon extends Foo, Button { }          // No! a class can't extend
                                            // multiple classes
class Zoom implements Fi, Baz { }           // OK. A class can implement
                                            // multiple interfaces
interface Vroom extends Fi, Baz { }         // OK. An interface can extend
                                            // multiple interfaces
class Yow extends Foo implements Fi { }      // OK. A class can do both
                                            // (extends must be 1st)
class Yow extends Foo implements Fi, Baz { } // OK. A class can do all three
                                            // (extends must be 1st)

```

Burn these into your memory, and watch for abuses in the questions you get on the exam. Regardless of what the question appears to be testing, the real problem might be the class or interface declaration. Before you get caught up in, say, tracing a complex threading flow, check to see if the code will even compile. (Just that tip alone may be worth your putting us in

(your will!) (You'll be impressed by the effort the exam developers put into distracting you from the real problem.) (How did people manage to write anything before parentheses were invented?)

Java 8—Now with Multiple Inheritance!

It might have already occurred to you that since interfaces can now have concrete methods and classes can implement multiple interfaces, the specter of multiple inheritance and the Deadly Diamond of Death can rear its ugly head! Well, you're partly correct. A class CAN implement interfaces with duplicate, concrete method signatures! But the good news is that the compiler's got your back, and if you DO want to implement both interfaces, you'll have to provide an overriding method in your class. Let's look at the following code:

```
interface I1 {
    default int doStuff() { return 1; }
}
interface I2 {
    default int doStuff() { return 2; }
}
public class MultiInt implements I1, I2 { // needs to override doStuff
    public static void main(String[] args) {
        new MultiInt().go();
    }
    void go() {
        System.out.println(doStuff());
    }
    // public int doStuff() {
    //     return 3;
    // }
}
```

As the code stands, it WILL NOT COMPILE because it's not clear which version of `doStuff()` should be used. In order to make the code compile, you need to override `doStuff()` in the class. Uncommenting the class's `doStuff()` method would allow the code to compile and, when run, produce the output:

3

CERTIFICATION OBJECTIVE

Legal Return Types (OCP Objectives 1.2 and 1.3)

1.2 *Implement inheritance including visibility modifiers and composition.*

1.3 *Implement polymorphism.*

This section covers two aspects of return types: what you can declare as a return type and what you can actually return as a value. What you can and cannot declare is pretty straightforward, but it all depends on whether you're overriding an inherited method or simply declaring a new method (which includes overloaded methods). We'll take just a quick look at the difference between return type rules for overloaded and overriding methods, because we've already covered that in this chapter. We'll cover a small bit of new ground, though, when we look at 113polymorphic return types and the rules for what is and is not legal to actually return.

Return Type Declarations

This section looks at what you're allowed to declare as a return type, which depends primarily on whether you are overriding, overloading, or declaring a new method.

Return Types on Overloaded Methods

Remember that method overloading is not much more than name reuse. The overloaded method is a completely different method from any other method of the same name. So if you inherit a method but overload it in a subtype, you're not subject to the restrictions of overriding, which means you can declare any return type you like. What you can't do is change *only* the return type. To overload a method, remember, you must change the argument list. The following code shows an overloaded method:

```
public class Foo{
    void go() { }
}
public class Bar extends Foo {
    String go(int x) {
        return null;
    }
}
```

Notice that the `Bar` version of the method uses a different return type. That's perfectly fine. As long as you've changed the argument list, you're overloading the method, so the return type doesn't have to match that of the supertype version. What you're NOT allowed to do is this:

```
public class Foo{
    void go() { }
}
public class Bar extends Foo {
    String go() { // Not legal! Can't change only the return type
        return null;
    }
}
```

Overriding and Return Types and Covariant Returns

When a subtype wants to change the method implementation of an inherited method (an override), the subtype must define a method that matches the inherited version exactly. Or, since Java 5, you're allowed to change the return type in the overriding method as long as the new return type is a *subtype* of the declared return type of the overridden (superclass) method.

Let's look at a covariant return in action:

```
class Alpha {  
    Alpha doStuff(char c) {  
        return new Alpha();  
    }  
}  
  
class Beta extends Alpha {  
    Beta doStuff(char c) {      // legal override since Java 1.5  
        return new Beta();  
    }  
}
```

Since Java 5, this code compiles. If you were to attempt to compile this code with a 1.4 compiler or with the source flag as follows,

```
javac -source 1.4 Beta.java
```

you would get a compiler error like this:

```
attempting to use incompatible return type
```

Other rules apply to overriding, including those for access modifiers and declared exceptions, but those rules aren't relevant to the return type discussion.



For the exam, be sure you know that overloaded methods can change the return type, but overriding methods can do so only within the bounds of covariant returns. Just that knowledge alone will help you through a wide range of exam questions.

Returning a Value

You have to remember only six rules for returning a value:

1. You can return `null` in a method with an object reference return type.

```
public Button doStuff() {  
    return null;  
}
```

2. An array is a perfectly legal return type.

```
public String[] go() {  
    return new String[] {"Fred", "Barney", "Wilma"};  
}
```

3. In a method with a primitive return type, you can return any value or variable that can be implicitly converted to the declared return type.

```
public int foo() {  
    char c = 'c';  
    return c; // char is compatible with int  
}
```

4. In a method with a primitive return type, you can return any value or variable that can be explicitly cast to the declared return type.

```
public int foo() {  
    float f = 32.5f;  
    return (int) f;  
}
```

5. You must *not* return anything from a method with a void return type.

```
public void bar() {  
    return "this is it"; // Not legal!!  
}
```

(Although you can say `return;`)

6. In a method with an object reference return type, you can return any object type that can be implicitly cast to the declared return type.

```
public Animal getAnimal() {  
    return new Horse(); // Assume Horse extends Animal  
}  
  
public Object getObject() {  
    int[] nums = {1,2,3};  
    return nums; // Return an int array, which is still an object  
}  
  
public interface Chewable {}  
public class Gum implements Chewable {}  
  
public class TestChewable {  
    // Method with an interface return type  
    public Chewable getChewable() {  
        return new Gum(); // Return interface implementer  
    }  
}
```



Watch for methods that declare an abstract class or interface return type, and know that any object that passes the IS-A test (in other words, would test true using the instanceof operator) can be returned from that method. For example:

```
public abstract class Animal { }
public class Bear extends Animal { }
public class Test {
    public Animal go() {
        return new Bear(); // OK, Bear "is-a" Animal
    }
}
```

This code will compile, and the return value is a subtype.

CERTIFICATION OBJECTIVE

Constructors and Instantiation (OCP Objectives 1.2 and 1.3)

- 1.2 *Implement inheritance including visibility modifiers and composition.*
- 1.3 *Implement polymorphism.*

Objects are constructed. You CANNOT make a new object without invoking a constructor. In fact, you can't make a new object without invoking not just the constructor of the object's actual class type, but also the constructor of each of its superclasses! Constructors are the code that runs whenever you use the keyword `new`. (Okay, to be a bit more accurate, there can also be initialization blocks that run when you say `new`, and we're going to cover initialization blocks and their static initialization counterparts after we discuss constructors.) We've got plenty to talk about here—we'll look at how constructors are coded, who codes them, and how they work at runtime. So grab your hardhat and a hammer, and let's do some object building.

Constructor Basics

Every class, *including abstract classes*, MUST have a constructor. Burn that into your brain. But just because a class must have a constructor doesn't mean the

programmer has to type it. A constructor looks like this:

```
class Foo {  
    Foo() { } // The constructor for the Foo class  
}
```

Notice what's missing? There's no return type! Two key points to remember about constructors are that they have no return type and their names must exactly match the class name. Typically, constructors are used to initialize the instance variable state, as follows:

```
class Foo {  
    int size;  
    String name;  
    Foo(String name, int size) {  
        this.name = name;  
        this.size = size;  
    }  
}
```

In the preceding code example, the `Foo` class does not have a no-arg constructor. That means the following will fail to compile,

```
Foo f = new Foo(); // Won't compile, no matching constructor
```

but the following will compile:

```
Foo f = new Foo("Fred", 43); // No problem. Arguments match  
                            // the Foo constructor.
```

So it's very common (and desirable) for a class to have a no-arg constructor, regardless of how many other overloaded constructors are in the class (yes,

constructors can be overloaded). You can't always make that work for your classes; occasionally you have a class where it makes no sense to create an instance without supplying information to the constructor. A `java.awt.Color` object, for example, can't be created by calling a no-arg constructor. That would be like saying to the JVM, "Make me a new Color object, and I really don't care what color it is...you pick." Do you seriously want the JVM making your style decisions?

Constructor Chaining

We know that constructors are invoked at runtime when you say `new` on some class type as follows:

```
Horse h = new Horse();
```

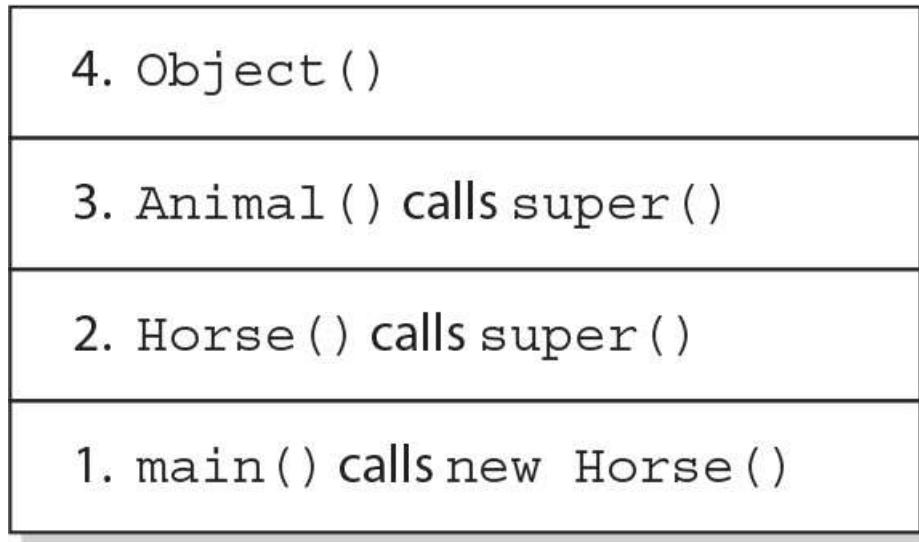
But what *really* happens when you say `new Horse()`? (Assume `Horse` extends `Animal` and `Animal` extends `Object`.)

1. The `Horse` constructor is invoked. Every constructor invokes the constructor of its superclass with an (implicit) call to `super()`, unless the constructor invokes an overloaded constructor of the same class (more on that in a minute).
2. The `Animal` constructor is invoked (`Animal` is the superclass of `Horse`).
3. The `Object` constructor is invoked (`Object` is the ultimate superclass of all classes, so class `Animal` extends `Object` even though you don't actually type "extends `Object`" into the `Animal` class declaration; it's implicit.) At this point we're on the top of the stack.
4. If class `Object` had any instance variables, then they would be given their explicit values. By *explicit* values, we mean values that are assigned at the time the variables are declared, such as `int x = 27`, where 27 is the explicit value (as opposed to the default value) of the instance variable.
5. The `Object` constructor completes.
6. The `Animal` instance variables are given their explicit values (if any).
7. The `Animal` constructor completes.
8. The `Horse` instance variables are given their explicit values (if any).
9. The `Horse` constructor completes.

[Figure 2-6](#) shows how constructors work on the call stack.

FIGURE 2-6

Constructors on the call stack



Rules for Constructors

The following list summarizes the rules you'll need to know for the exam (and to understand the rest of this section). You MUST remember these, so be sure to study them more than once.

- Constructors can use any access modifier, including `private`. (A `private` constructor means only code within the class itself can instantiate an object of that type, so if the `private` constructor class wants to allow an instance of the class to be used, the class must provide a static method or variable that allows access to an instance created from within the class.)
- The constructor name must match the name of the class.
- Constructors must not have a return type.
- It's legal (but stupid) to have a method with the same name as the class, but that doesn't make it a constructor. If you see a return type, it's a method rather than a constructor. In fact, you could have both a method and a constructor with the same name—the name of the class—in the same class, and that's not a problem for Java. Be careful not to mistake a method for a constructor—be sure to look for a return type.

- If you don't type a constructor into your class code, a default constructor will be automatically generated by the compiler.
- The default constructor is **ALWAYS** a no-arg constructor.
- If you want a no-arg constructor and you've typed any other constructor(s) into your class code, the compiler won't provide the no-arg constructor (or any other constructor) for you. In other words, if you've typed in a constructor with arguments, you won't have a no-arg constructor unless you typed it in yourself!
- Every constructor has, as its first statement, either a call to an overloaded constructor (`this()`) or a call to the superclass constructor (`super()`), although remember this call can be inserted by the compiler.
- If you do type in a constructor (as opposed to relying on the compiler-generated default constructor) and you do not type in the call to `super()` or a call to `this()`, the compiler will insert a no-arg call to `super()` for you as the very first statement in the constructor.
- A call to `super()` can either be a no-arg call or can include arguments passed to the super constructor.
- A no-arg constructor is not necessarily the default (that is, compiler-supplied) constructor, although the default constructor is always a no-arg constructor. The default constructor is the one the compiler provides! Although the default constructor is always a no-arg constructor, you're free to put in your own no-arg constructor.
- You cannot make a call to an instance method or access an instance variable until after the super constructor runs.
- Only static variables and methods can be accessed as part of the call to `super()` or `this()`. (Example: `super(Animal.NAME)` is OK, because `NAME` is declared as a static variable.)
- Abstract classes have constructors, and those constructors are always called when a concrete subclass is instantiated.
- Interfaces do not have constructors. Interfaces are not part of an object's inheritance tree.
- The only way a constructor can be invoked is from within another constructor. In other words, you can't write code that actually calls a constructor as follows:

```
class Horse {  
    Horse() { } // constructor  
    void doStuff() {  
        Horse(); // calling the constructor - illegal!  
    }  
}
```

Determine Whether a Default Constructor Will Be Created

The following example shows a Horse class with two constructors:

```
class Horse {  
    Horse() { }  
    Horse(String name) { }  
}
```

Will the compiler put in a default constructor for this class? No!

How about for the following variation of the class?

```
class Horse {  
    Horse(String name) { }  
}
```

Now will the compiler insert a default constructor? No!

What about this class?

```
class Horse { }
```

Now we're talking. The compiler will generate a default constructor for this class because the class doesn't have any constructors defined.

Okay, what about this class?

```
class Horse {  
    void Horse() { }  
}
```

It might look like the compiler won't create a constructor, since one is already in the `Horse` class. Or is it? Take another look at the preceding `Horse` class.

What's wrong with the `Horse()` constructor? It isn't a constructor at all! It's simply a method that happens to have the same name as the class. Remember, the return type is a dead giveaway that we're looking at a method, not a constructor.

How do you know for sure whether a default constructor will be created?
Because you didn't write any constructors in your class.

How do you know what the default constructor will look like? Because...

- The default constructor has the same access modifier as the class.
- The default constructor has no arguments.
- The default constructor includes a no-arg call to the super constructor (`super()`).

[Table 2-5](#) shows what the compiler will (or won't) generate for your class.

TABLE 2-5 Compiler-Generated Constructor Code

Class Code (What You Type)	Compiler-Generated Constructor Code (in Bold)
class Foo { }	class Foo { Foo() { super() ; } }
class Foo { Foo() { } }	class Foo { Foo() { super() ; } }
public class Foo { }	public class Foo { public Foo() { super() ; } }
class Foo { Foo(String s) { } }	class Foo { Foo(String s) { super() ; } }
class Foo { Foo(String s) { super() ; } }	<i>Nothing; compiler doesn't need to insert anything.</i>
class Foo { void Foo() { } }	class Foo { void Foo() { } Foo() { super() ; } }
	(void Foo() is a method, not a constructor.)

What happens if the super constructor has arguments? Constructors can have arguments just as methods can, and if you try to invoke a method that takes, say, an `int`, but you don't pass anything to the method, the compiler will complain as follows:

```
class Bar {  
    void takeInt(int x) {}  
}  
  
class UseBar {  
    public static void main (String [] args) {  
        Bar b = new Bar();  
        b.takeInt(); // Try to invoke a no-arg takeInt() method  
    }  
}
```

The compiler will complain that you can't invoke `takeInt()` without passing an `int`. Of course, the compiler enjoys the occasional riddle, so the message it spits out on some versions of the JVM (your mileage may vary) is less than obvious:

```
UseBar.java:7: takeInt(int) in Bar cannot be applied to ()  
    b.takeInt();  
    ^
```

But you get the idea. The bottom line is that there must be a match for the method. And by match, we mean the argument types must be able to accept the values or variables you're passing and in the order you're passing them. Which brings us back to constructors (and here you were thinking we'd never get there), which work exactly the same way.

So if your super constructor (that is, the constructor of your immediate superclass/parent) has arguments, you must type in the call to `super()`,

supplying the appropriate arguments. Crucial point: if your superclass does not have a no-arg constructor, you must type a constructor in your class (the subclass) because you need a place to put in the call to `super()` with the appropriate arguments.

The following is an example of the problem:

```
class Animal {  
    Animal(String name) {}  
}  
  
class Horse extends Animal {  
    Horse() {  
        super(); // Problem!  
    }  
}
```

And once again the compiler treats us with something like the stunning lucidity below:

```
Horse.java:7: cannot resolve symbol  
symbol : constructor Animal ()  
location: class Animal  
super(); // Problem!  
^
```

If you're lucky (and it's a full moon), *your* compiler might be a little more explicit. But again, the problem is that there just isn't a match for what we're trying to invoke with `super()`—an `Animal` constructor with no arguments.

Another way to put this is that if your superclass does *not* have a no-arg constructor, then in your subclass you will not be able to use the default constructor supplied by the compiler. It's that simple. Because the compiler can *only* put in a call to a no-arg `super()`, you won't even be able to compile

something like this:

```
class Clothing {  
    Clothing(String s) {}  
}  
class TShirt extends Clothing {}
```

Trying to compile this code gives us exactly the same error we got when we put a constructor in the subclass with a call to the no-arg version of super():

```
Clothing.java:4: cannot resolve symbol  
symbol  : constructor Clothing ()  
location: class Clothing  
class TShirt extends Clothing {}  
^
```

In fact, the preceding `Clothing` and `TShirt` code is implicitly the same as the following code, where we've supplied a constructor for `TShirt` that's identical to the default constructor supplied by the compiler:

```
class Clothing {
    Clothing(String s) { }
}

class TShirt extends Clothing {
    // Constructor identical to compiler-supplied
    // default constructor

    TShirt() {
        super(); // Won't work!
    } // tries to invoke a no-arg Clothing constructor
    // but there isn't one
}
```

One last point on the whole default constructor thing (and it's probably very obvious, but we have to say it or we'll feel guilty for years), **constructors are never inherited**. They aren't methods. They can't be overridden (because they aren't methods, and only instance methods can be overridden). So the type of constructor(s) your superclass has in no way determines the type of default constructor you'll get. Some folks mistakenly believe that the default constructor somehow matches the super constructor, either by the arguments the default constructor will have (remember, the default constructor is always a no-arg) or by the arguments used in the compiler-supplied call to `super()`.

So although constructors can't be overridden, they can—and often are—overloaded (which you've already seen).

Overloaded Constructors

Overloading a constructor means typing in multiple versions of the constructor, each having a different argument list, like the following examples:

```
class Foo {  
    Foo() {}  
    Foo(String s) {}  
}
```

The preceding `Foo` class has two overloaded constructors: one that takes a string and one with no arguments. Because there's no code in the no-arg version, it's actually identical to the default constructor the compiler supplies—but remember, since there's already a constructor in this class (the one that takes a string), the compiler won't supply a default constructor. If you want a no-arg constructor to overload the with-args version you already have, you're going to have to type it yourself, just as in the `Foo` example.

Overloading a constructor is typically used to provide alternative ways for clients to instantiate objects of your class. For example, if a client knows the animal name, they can pass that to an `Animal` constructor that takes a string. But if they don't know the name, the client can call the no-arg constructor and that constructor can supply a default name. Here's what it looks like:

```
1. public class Animal {  
2.     String name;  
3.     Animal(String name) {  
4.         this.name = name;  
5.     }  
6.  
7.     Animal() {  
8.         this(makeRandomName());  
9.     }  
10.  
11.    static String makeRandomName() {  
12.        int x = (int) (Math.random() * 5);  
13.        String name = new String[] {"Fluffy", "Fido",  
14.                                "Rover", "Spike",  
15.                                "Gigi"} [x];  
16.  
17.        return name;  
18.    }  
19.  
20.    public static void main (String [] args) {  
21.        Animal a = new Animal();  
22.        System.out.println(a.name);  
23.        Animal b = new Animal("Zeus");  
24.        System.out.println(b.name);  
25.    }  
26. }
```

Running the code four times produces output something like this:

```
% java Animal  
Gigi  
Zeus
```

```
% java Animal  
Fluffy  
Zeus
```

```
% java Animal  
Rover  
Zeus
```

```
% java Animal  
Fluffy  
Zeus
```

There's a lot going on in the preceding code. [Figure 2-7](#) shows the call stack for constructor invocations when a constructor is overloaded. Take a look at the call stack, and then let's walk through the code straight from the top.

FIGURE 2-7

Overloaded constructors on the call stack

4. Object ()
3. Animal (String s) calls super ()
2. Animal () calls this (randomlyChosenNameString)
1. main () calls new Animal ()

- **Line 2** Declare a `String` instance variable name.
- **Lines 3–5** Constructor that takes a `String` and assigns it to an instance variable name.
- **Line 7** Here's where it gets fun. Assume every animal needs a name, but the client (calling code) might not always know what the name should be, so the `Animal1` class will assign a random name. The no-arg constructor generates a name by invoking the `makeRandomName()` method.
- **Line 8** The no-arg constructor invokes its own overloaded constructor that takes a `String`, in effect calling it the same way it would be called if client code were doing a new to instantiate an object, passing it a `String` for the name. The overloaded invocation uses the keyword `this`, but uses it as though it were a method named `this()`. So line 8 is simply calling the constructor on line 3, passing it a randomly selected `String` rather than a client-code chosen name.
- **Line 11** Notice that the `makeRandomName()` method is marked `static!` That's because you cannot invoke an instance (in other words, nonstatic) method (or access an instance variable) until after the super constructor has run. And because the super constructor will be invoked from the constructor on line 3, rather than from the one on line 7, line 8 can use only a static method to generate the name. If we wanted all animals not specifically named by the caller to have the same default name, say, "Fred," then line 8 could have read `this("Fred")`; rather than calling a method that returns a string with the randomly chosen name.
- **Line 12** This doesn't have anything to do with constructors, but since we're all here to learn, it generates a random integer between 0 and 4.
- **Line 13** Weird syntax, we know. We're creating a new `String` object (just a single `String` instance), but we want the string to be selected randomly from a list. Except we don't have the list, so we need to make it. So in that one line of code we
 1. Declare a `String` variable name.
 2. Create a `String` array (anonymously—we don't assign the array itself to a variable).
 3. Retrieve the string at index `[x]` (`x` being the random number generated on line 12) of the newly created `String` array.

4. Assign the string retrieved from the array to the declared instance variable name. (Throwing in unusual syntax, especially for code wholly unrelated to the real question, is in the spirit of the exam. Don't be startled! Okay, be startled, but then just say to yourself, "Whoa!" and get on with it.)

- **Line 18** We're invoking the no-arg version of the constructor (causing a random name from the list to be passed to the other constructor).
- **Line 20** We're invoking the overloaded constructor that takes a string representing the name.

The key point to get from this code example is in line 8. Rather than calling `super()`, we're calling `this()`, and `this()` always means a call to another constructor in the same class. Okay, fine, but what happens after the call to `this()`? Sooner or later the `super()` constructor gets called, right? Yes, indeed. A call to `this()` just means you're delaying the inevitable. Some constructor, somewhere, must make the call to `super()`.

Key Rule: The first line in a constructor must be a call to `super()` or a call to `this()`.

No exceptions. If you have neither of those calls in your constructor, the compiler will insert the no-arg call to `super()`. In other words, if constructor `A()` has a call to `this()`, the compiler knows that constructor `A()` will not be the one to invoke `super()`.

The preceding rule means a constructor can never have both a call to `super()` and a call to `this()`. Because each of those calls must be the first statement in a constructor, you can't legally use both in the same constructor. That also means the compiler will not put a call to `super()` in any constructor that has a call to `this()`.

Thought question: What do you think will happen if you try to compile the following code?

```
class A {  
    A() {  
        this("foo");  
    }  
    A(String s) {  
        this();  
    }  
}
```

Java 8 compilers should catch the problem in this code and report an error, something like:

Error:
recursive constructor invocation

Older compilers may not actually catch the problem. They might assume you know what you're doing. Can you spot the flaw? Given that a super constructor must always be called, where would the call to `super()` go? Remember, the compiler won't put in a default constructor if you've already got one or more constructors in your class. And when the compiler doesn't put in a default constructor, it still inserts a call to `super()` in any constructor that doesn't explicitly have a call to the super constructor—unless, that is, the constructor already has a call to `this()`. So in the preceding code, where can `super()` go? The only two constructors in the class both have calls to `this()`, and, in fact, you'll get exactly what you'd get if you typed the following method code:

```
public void go() {  
    doStuff();  
}  
  
public void doStuff() {  
    go();  
}
```

Now can you see the problem? Of course you can. The stack explodes! It gets higher and higher and higher until it just bursts open and method code goes spilling out, oozing from the JVM right onto the floor. Two overloaded constructors both calling `this()` are two constructors calling each other—over and over and over, resulting in this:

```
% java A  
Exception in thread "main" java.lang.StackOverflowError
```

The benefit of having overloaded constructors is that you offer flexible ways to instantiate objects from your class. The benefit of having one constructor invoke another overloaded constructor is to avoid code duplication. In the Animal example, there wasn't any code other than setting the name, but imagine if after line 4 there was still more work to be done in the constructor. By putting all the other constructor work in just one constructor, and then having the other constructors invoke it, you don't have to write and maintain multiple versions of that other important constructor code. Basically, each of the other not-the-real-one overloaded constructors will call another overloaded constructor, passing it whatever data it needs (data the client code didn't supply).

Constructors and instantiation become even more exciting (just when you thought it was safe) when you get to inner classes, but we know you can stand to have only so much fun in one chapter, and besides, you don't have to deal with inner classes until [Chapter 7](#).

CERTIFICATION OBJECTIVE

Singleton Design Pattern (OCP Objective 1.5)

1.5 Create and use singleton classes and immutable classes.

We just finished discussing how constructors play a role whenever new objects are created. In this section, we'll discuss the singleton design pattern that allows us to ensure we only have one instance of a class within an application. (Note: These days not everyone is a fan of the singleton pattern, but that's a discussion for another time.) *Singleton* is called a creational design pattern because it deals with creating objects. But wait, what's this "design pattern"?

What Is a Design Pattern?

Wikipedia currently defines a design pattern as "a general reusable solution to a commonly occurring problem within a given context." What does that mean? Programmers often encounter the same problem repeatedly. Rather than have everyone come up with their own solution to common programming issues, we use a "best practice"-type solution that has been documented and proven to work. The word "general" is important. We can't just copy and paste a design pattern into our code. It's just an idea. We should write an implementation for it and put that in our code.

Using a design pattern has a few advantages. We get to use a solution that is known to work. The tradeoffs, if any, are well documented, so we don't stumble over problems that have already been solved. Design patterns also serve as communication aids. Your boss can say, "We will use a singleton," and that one word is enough to tell you what is expected.

When books or web pages document patterns, they do so using a consistent format. We pay homage to this universal format by including sections for the "Problem," "Solution," and "Benefits" of the singleton pattern. The "Problem" section explains why we need the pattern—what problem we are trying to solve. The "Solution" section explains how to implement the pattern. The "Benefits" section reviews why we need the pattern and how it has helped us solve the problem. Some of the benefits are hinted at in the "Problem" section. Others are additional benefits that come from the pattern.



The OCP 8 exam covers only one pattern; this is just to get your feet wet. Whole books have been written on the topic of design patterns. Head First Design Patterns (O'Reilly Media, 2004) covers more patterns. And the most famous book on patterns—Design Patterns: Elements of Reusable Object-Oriented Software (Addison-Wesley Professional, 1994), also known as “Gang of Four”—covers 23 design patterns. You may notice that these books are more than 10 years old. That's because the classic patterns haven't changed.

Problem

Let's suppose we're putting on a show. We're going to perform the show once, and we only have a few seats in the theater.

```
import java.util.*;  
  
public class Show {  
    private Set<String> availableSeats;
```

```

public Show() {
    availableSeats = new HashSet<String>();
    availableSeats.add("1A");
    availableSeats.add("1B");
}
public boolean bookSeat(String seat) {
    return availableSeats.remove(seat);
}
public static void main(String[] args) {
    ticketAgentBooks("1A");
    ticketAgentBooks("1A");
}
private static void ticketAgentBooks(String seat) {
    Show show = new Show();           // a new Show gets created
                                      // each time we call the method
    System.out.println(show.bookSeat(seat));
}
}

```

This code prints out `true` twice. That's a problem. We just put two people in the same seat. Why? We created a new `Show` object every time we needed it. Even though we want to use the same theater and seats, `Show` deals with a new set of seats each time, which means we've double-booked seats.

Solution

There are a few ways to implement the singleton pattern. Here's the simplest:

```
import java.util.*;  
  
public class Show {  
    private static final Show INSTANCE // store one instance  
        = new Show(); // (this is the singleton)  
    private Set<String> availableSeats;  
  
    public static Show getInstance() { // callers can get to  
        return INSTANCE; // the instance  
    }  
    private Show() { // callers can't create  
        // directly anymore.  
        // Must use getInstance()  
        availableSeats = new HashSet<String>();  
        availableSeats.add("1A");  
        availableSeats.add("1B");  
    }  
    public boolean bookSeat(String seat) {  
        return availableSeats.remove(seat);  
    }  
}
```

```

public static void main(String[] args) {
    ticketAgentBooks("1A");
    ticketAgentBooks("1A");
}
private static void ticketAgentBooks(String seat) {
    Show show = Show.getInstance();
    System.out.println(show.bookSeat(seat));
}
}

```

Now the code prints `true` and `false`. Much better! We are no longer going to have two people sitting in the same seat. The bolded bits in the code call attention to the implementation of the singleton pattern.

The key parts of the singleton pattern are

- A private static variable to store the single instance called the singleton. This variable is usually final to keep developers from accidentally changing it.
- A public static method for callers to get a reference to the instance.
- A private constructor so no callers can instantiate the object directly.

Remember, the code doesn't create a new `Show` each time, but merely returns the singleton instance of `Show` each time `getInstance()` is called.

To understand this a little better, let's consider what happens if we change parts of the code.

If the constructor weren't private, we wouldn't have a singleton. Callers would be free to ignore `getInstance()` and instantiate their own instances, which would leave us with multiple instances in the program and defeat the purpose entirely.

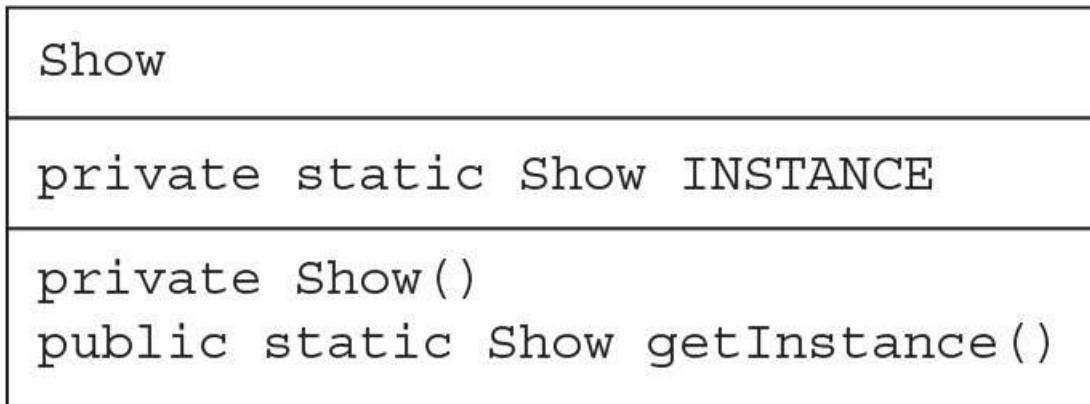
If `getInstance()` weren't public, we would still have a singleton. However, it wouldn't be as useful because only methods in the same package would be able to use the singleton.

If `getInstance()` weren't static, we'd have a bigger problem. Callers

couldn't instantiate the class directly, which means they wouldn't be able to call `getInstance()` at all.

If `INSTANCE` weren't static and final, we could have multiple instances at different points in time. These keywords signal that we assign the field once and it stays that way for the life of the program.

When talking about design patterns, it is also common to communicate the pattern in diagram form. The singleton pattern diagram looks like this:



The diagrams use a format called Unified Modeling Language (UML). The diagrams in this book use some aspects of UML, such as a box with three sections representing each class. Actual UML uses more notation, such as showing public versus private visibility. You can think of this as faux-UML.

As long as the method in the diagram keeps the same signature, we can change our logic to other implementations of the singleton pattern. One “feature” of the above implementation is that it creates the `Show` object before we need it. This is called *eager initialization*, which is good if the object isn’t expensive to create or we know it will be needed every time the program runs. Sometimes, however, we want to create the object only on the first use. This is called *lazy initialization*.

```
private static Show INSTANCE;
private Set<String> availableSeats;
public static Show getInstance() {
    if (INSTANCE == null) {
        INSTANCE = new Show();
    }
    return INSTANCE;
}
```

In this case, `INSTANCE` isn't set to be a `Show` until the first time `getInstance()` is called. Walking through what happens, the first time `getInstance()` is called, Java sees `INSTANCE` is still null and creates the singleton. The second time `getInstance()` is called, Java sees `INSTANCE` has already been set and simply returns it. In this example, `INSTANCE` isn't final because that would prevent the code from compiling.



The singleton code here assumes you are only running one thread at a time. It is NOT thread-safe. What if this were a web site and two users managed to be booking a seat at the exact same time? If `getInstance()` were running at the exact same time, both of them could see that `INSTANCE` was null and create a new `Show` at the same time. There are a few ways to solve this. One is to add `synchronized` to the `getInstance()` method. This works, but comes with a small performance hit. We're getting way beyond the scope of the exam, but you can Google “double checked locked pattern” for more information.

You might have noticed that the code for `getInstance()` can get a bit complicated. Java 5 gave us a much shorter way of creating a singleton:

```
public enum ShowEnum {                                // this is an enum
    INSTANCE;                                     // instead of a class

    private Set<String> availableSeats;
    private ShowEnum() {
        availableSeats = new HashSet<String>();
        availableSeats.add("1A");
        availableSeats.add("1B");
    }
    public boolean bookSeat(String seat) {
        return availableSeats.remove(seat);
    }
    public static void main(String[] args) {
        ticketAgentBooks("1A");
        ticketAgentBooks("1A");
    }
}

private static void ticketAgentBooks(String seat) {
    ShowEnum show = ShowEnum.INSTANCE;    // we don't even
                                         // need a method to
                                         // get the instance
    System.out.println(show.bookSeat(seat));
}
```

Short and sweet. By definition, there is only one instance of an enum constant. You are probably wondering why we've had this whole discussion of

the singleton pattern when you can write it so easily. The main reason is that enums were introduced with Java 5, and there is a ton of older code out there that you need to be able to understand. Another reason is that sometimes you still need the older versions of the pattern.

Benefits

Benefits of the singleton pattern include the following:

- The primary benefit is that there is only one instance of the object in the program. When an object's instance variables are keeping track of information that is used across the program, it's useful. For example, consider a web site visitor counter. You only want one count that is shared.
- Another benefit is performance. Some objects are expensive to create; for example, maybe we need to make a database call to look up the state for the object.

CERTIFICATION OBJECTIVE

Immutable Classes (OCP Objective 1.5)

1.5 *Create and use singleton classes and immutable classes.*

Immutable classes aren't a new idea, but they're new to the OCP exam. Because they are inherently thread-safe, immutable classes are particularly useful in applications that include concurrency and/or parallelism. In this world of big data and fast data, the need for concurrency and parallelism is on the rise, so adding immutable classes to the exam is a timely choice.

You're already familiar with so-called "immutable classes" because String is a great example of one! One way of looking at the String class is that you might want to create an object that has a string value AND you want to allow others to have access to your string's value, but you don't want to let anyone (including you) change the object's value. That's exactly what a String object allows.

You can create your own "immutable classes" whenever you have a design goal similar to the one described above—in other words, whenever you want to create a class whose objects' values are immutable. In addition, usually when

you want a type that creates only immutable objects, you won't want people to be able to extend your class and undo your immutability goal, so immutable classes should be marked `final`. Again, this is just like the `String` class (which is also a `final` class).

Other than the immutability of values and their “final-ness,” there is nothing unique about these classes. You can design them to be instantiated via constructors or by using a factory method. They can have behavior like the `String` class has. The only difference is that—by definition—the class’s object’s variables can’t be changed. Let’s look at what steps are necessary to develop an immutable class.

Imagine your team is developing an audio synthesizer app for Android. An important part of the synth software is the ADSR envelope. (ADSR stands for attack, decay, sustain, and release.) A given ADSR envelope describes a unique sound. You want your users to be able to create, save, and share the custom ADSR envelope objects they’ve created, but your users don’t want anyone to be able to alter their unique sonic creations. Below is the code for our ADSR immutable class. (Note: For the sake of brevity, we omitted the code for the sustain and release values, but it will be the same as the code we show for attack and decay.)

```
final class ADSR {                                // final class, can't be
    extended
    private final StringBuilder name;
    private final int attack;
```

```

private final int decay;
// sustain and release code omitted (for brevity)

public ADSR(StringBuilder n,           // public constructor
            int a, int d) {
    this.name = n;
    this.attack = a;
    this.decay = d;
}

// Notice there are NO SETTERS !

public StringBuilder getName() {      // return a new object
    // not the original
    StringBuilder nameCopy = new StringBuilder(name);
    return nameCopy;
}
public ADSR getADSR() { return this; } // return an immutable object
}

```

The code above allows the following:

- You can build your own immutable ADSR objects with secret values.
- You can share your ADSR objects.
- Your users can read the name field of your objects, but they can't mutate your objects.

Does this all make sense? Does the code look bulletproof? There is a problem with the way we implemented the ADSR class! Let's look at a test class:

```
public class TestADSR {  
    public static void main(String[] args) {  
        StringBuilder sb = new StringBuilder("a1 ");  
        ADSR a1 = new ADSR(sb, 5, 7);  
        ADSR a2 = a1.getADSR();  
        System.out.println(a1.getName());  
        sb.append("alter the name ");  
        System.out.println(a1.getName());  
    }  
}
```

which produces the output:

```
a1  
a1 alter the name
```

Oops! Our constructor is using the same `StringBuilder` object as the one used by the method that invoked the constructor. (Note: If you’re thinking “Don’t use a `StringBuilder`, use a `String`” to give yourself extra points. We used a `StringBuilder` to demonstrate how to deal with instance variables that are of a mutable, nonprimitive type.) So we gotta fix that! Here’s the fixed version of the `ADSR` class:

```

// immutable class - version 2
final class ADSR {                                // final class, can't be extended
    private final StringBuilder name;
    private final int attack;
    private final int decay;
    // sustain and release code omitted (for brevity)

    public ADSR(StringBuilder n,                  // public constructor
                int a, int d) {
        name = new StringBuilder(n);           // make a new object for the name
    }

    this.attack = a;
    this.decay = d;
}

public StringBuilder getName() {                   // return a new object
    // not the original
    StringBuilder nameCopy = new StringBuilder(name);
    if(nameCopy != name)
        System.out.println("different objects");
    return nameCopy;
}

public ADSR getADSR() { return this; }
}

public class TestADSR {
    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder("a1 ");
        ADSR a1 = new ADSR(sb, 5, 7);
        ADSR a2 = a1.getADSR();
        System.out.println(a1.getName());
        sb.append("alter the name ");
        System.out.println(a1.getName());
    }
}

```

which produces:

```

a1
a1

```

Much better! Here's a list of things to do to create an immutable class:

1. Mark the class final so that it cannot be extended.
2. Mark its variables private and final.
3. If the constructor takes any mutable objects as arguments, make new copies of those objects in the constructor.
4. Do NOT provide any setter methods!
5. If any of the getter methods return a mutable object reference, make a copy of the actual object, and return a reference to the copy.

Notice that points 3 and 5 both discuss making copies of objects. This idea is an aspect of what is sometimes referred to as “defensive copying.” If some of this section reminds you of ideas in the “Encapsulation” section earlier in this chapter, congratulations! An immutable class is—by definition—extremely well encapsulated.

In the real world there are some more advanced ideas associated with immutable classes, but if you understand this section, you'll be fine for the exam.



In this section we've often put quotes around the phrase “immutable class” because it's a bit of a misnomer. More accurately, what we're talking about here are classes whose objects are immutable.

CERTIFICATION OBJECTIVE

Initialization Blocks (OCP Objective 1.6)

1.6 Develop code that uses static keyword (sic) on initialize blocks, variables, methods, and classes.

We've talked about two places in a class where you can put code that

performs operations: methods and constructors. There is also a third place in a Java program where operations can be performed: initialization blocks. Static initialization blocks run when the class is first loaded, and instance initialization blocks run whenever an instance is created (a bit similar to a constructor). Let's look at an example:

```
class SmallInit {  
    static int x;  
    int y;  
  
    static { x = 7; }          // static init block  
    { y = 8; }                // instance init block  
}
```

As you can see, the syntax for initialization blocks is pretty terse. They don't have names, they can't take arguments, and they don't return anything. A *static* initialization block runs *once* when the class is first loaded. An *instance* initialization block runs once *every time a new instance is created*. Remember when we talked about the order in which constructor code executed? Instance init block code runs right after the call to `super()` in a constructor and before any of the other code in the constructor—in other words, after all super constructors have run.

You can have many initialization blocks in a class. It is important to note that unlike methods or constructors, *the order in which initialization blocks appear in a class matters*. When it's time for initialization blocks to run, if a class has more than one, they will run in the order in which they appear in the class file—in other words, from the top down. Based on the rules we just discussed, can you determine the output of the following program?

```
class Init {  
    Init(int x) { System.out.println("1-arg const"); }  
    Init() { System.out.println("no-arg const"); }  
    static { System.out.println("1st static init"); }  
    { System.out.println("1st instance init"); }  
    { System.out.println("2nd instance init"); }  
    static { System.out.println("2nd static init"); }  
  
    public static void main(String [] args) {  
        new Init();  
        new Init(7);  
    }  
}
```

To figure this out, remember these rules:

- `init` blocks execute in the order in which they appear.
- Static `init` blocks run once, when the class is first loaded.
- Instance `init` blocks run every time a class instance is created.
- Instance `init` blocks run after the constructor's call to `super()`.

With those rules in mind, the following output should make sense:

```
1st static init  
2nd static init  
1st instance init  
2nd instance init  
no-arg const  
1st instance init  
2nd instance init  
1-arg const
```

As you can see, the instance init blocks each ran twice. Instance init blocks are often used as a place to put code that all the constructors in a class should share. That way, the code doesn't have to be duplicated across constructors.

Finally, if you make a mistake in your static init block, the JVM can throw an `ExceptionInInitializerError`. Let's look at an example:

```
class InitError {  
    static int [] x = new int[4];  
    static { x[4] = 5; }           // bad array index!  
    public static void main(String [] args) { }  
}
```

It produces something like this:

```
Exception in thread "main" java.lang.ExceptionInInitializerError  
Caused by: java.lang.ArrayIndexOutOfBoundsException: 4  
        at InitError.<clinit>(InitError.java:3)
```



By convention, `init` blocks usually appear near the top of the class file, somewhere around the constructors. However, this is the OCP exam we're talking about. Don't be surprised if you find an `init` block tucked in between a couple of methods, looking for all the world like a compiler error waiting to happen!

CERTIFICATION OBJECTIVE

Statics (OCP Objective 1.6)

1.6 *Develop code that uses static keyword (sic) on initialize blocks, variables, methods, and classes.*

Static Variables and Methods

The `static` modifier has such a profound impact on the behavior of a method or variable that we're treating it as a concept entirely separate from the other modifiers. To understand the way a `static` member works, we'll look first at a reason for using one. Imagine you've got a utility class or interface with a method that always runs the same way; its sole function is to return, say, a random number. It wouldn't matter which instance of the class performed the method—it would always behave exactly the same way. In other words, the method's behavior has no dependency on the state (instance variable values) of an object. So why, then, do you need an object when the method will never be instance-specific? Why not just ask the type itself to run the method?

Let's imagine another scenario: Suppose you want to keep a running count of all instances instantiated from a particular class. Where do you actually keep that variable? It won't work to keep it as an instance variable within the class whose instances you're tracking, because the count will just be initialized back to a default value with each new instance. The answer to both the utility-method-always-runs-the-same scenario and the keep-a-running-total-of-instances scenario is to use the `static` modifier. Variables and methods marked `static` belong to the type, rather than to any particular instance. In fact, for classes, you can use a `static` method or variable without having any instances of that class at all. You need only have the type available to be able to invoke a `static` method

or access a static variable. static variables, too, can be accessed without having an instance of a class. But if there are instances, a static variable of a class will be shared by all instances of that class; there is only one copy.

The following code declares and uses a static counter variable:

```
class Frog {  
    static int frogCount = 0; // Declare and initialize  
                            // static variable  
    public Frog() {  
        frogCount += 1;          // Modify the value in the constructor  
    }  
    public static void main (String [] args) {  
        new Frog();  
        new Frog();  
        new Frog();  
        System.out.println("Frog count is now " + frogCount);  
    }  
}
```

In the preceding code, the static frogCount variable is set to zero when the Frog class is first loaded by the JVM, before any Frog instances are created! (By the way, you don't actually need to initialize a static variable to zero; static variables get the same default values instance variables get, but it's good practice to initialize them anyway.) Whenever a Frog instance is created, the Frog constructor runs and increments the static frogCount variable. When this code executes, three Frog instances are created in main(), and the result is

```
Frog count is now 3
```

Now imagine what would happen if frogCount were an instance variable (in other words, nonstatic):

```
class Frog {  
    int frogCount = 0; // Declare and initialize  
                      // instance variable  
    public Frog() {  
        frogCount += 1; // Modify the value in the constructor  
    }  
    public static void main (String [] args) {  
        new Frog();  
        new Frog();  
        new Frog();  
        System.out.println("Frog count is now " + frogCount);  
    }  
}
```

When this code executes, it should still create three `Frog` instances in `main()`, but the result is...a compiler error! We can't get this code to compile, let alone run.

```
Frog.java:11: nonstatic variable frogCount cannot be referenced  
from a static context  
    System.out.println("Frog count is " + frogCount);  
                                         ^  
1 error
```

The JVM doesn't know which `Frog` object's `frogCount` you're trying to access. The problem is that `main()` is itself a `static` method and thus isn't running against any particular instance of the class; instead it's running on the class itself. A `static` method can't access a nonstatic (instance) variable because there is no instance! That's not to say there aren't instances of the class alive on the heap, but rather that even if there are, the `static` method doesn't know

anything about them. The same applies to instance methods; a static method can't directly invoke a nonstatic method. Think static = class, nonstatic = instance. Making the method called by the JVM (`main()`) a static method means the JVM doesn't have to create an instance of your class just to start running code.

e x a m Watch

One of the mistakes most often made by new Java programmers is attempting to access an instance variable (which means nonstatic variable) from the static `main()` method (which doesn't know anything about any instances, so it can't access the variable). The following code is an example of illegal access of a nonstatic variable from a static method:

```
class Foo {  
    int x = 3;  
    public static void main (String [] args) {  
        System.out.println("x is " + x);  
    }  
}
```

Understand that this code will never compile, because you can't access a nonstatic (instance) variable from a static method. Just think of the compiler saying, "Hey, I have no idea which Foo object's x variable you're trying to print!" Remember, it's the class running the `main()` method, not an instance of the class.

Of course, the tricky part for the exam is that the question won't look as obvious as the preceding code. The problem you're being tested for—accessing a nonstatic variable from a static method—will be buried in code that might appear to be testing something else. For example, the preceding code would be more likely to appear as

```
class Foo {  
    int x = 3;  
    float y = 4.3f;  
    public static void main (String [] args) {  
        for (int z = x; z < ++x; z--, y = y + z)  
            // complicated looping and branching code  
    }  
}
```

So while you're trying to follow the logic, the real issue is that x and y can't be used within main() because x and y are instance, not static, variables! The same applies for accessing nonstatic methods from a static method. The rule is that a static method of a class can't access a nonstatic (instance) method or variable of its own class.

Accessing Static Methods and Variables

Since you don't need to have an instance in order to invoke a static method or access a static variable, how do you invoke or use a static member? What's the syntax? We know that with a regular old instance method, you use the dot operator on a reference to an instance:

```
class Frog {  
    int frogSize = 0;  
    public int getFrogSize() {  
        return frogSize;  
    }  
    public Frog(int s) {  
        frogSize = s;  
    }  
    public static void main (String [] args) {  
        Frog f = new Frog(25);  
        System.out.println(f.getFrogSize()); // Access instance  
                                         // method using f  
    }  
}
```

In the preceding code, we instantiate a `Frog`, assign it to the reference variable `f`, and then use that `f` reference to invoke a method on the `Frog` instance we just created. In other words, the `getFrogSize()` method is being invoked on a specific `Frog` object on the heap.

But this approach (using a reference to an object) isn't appropriate for accessing a `static` method because there might not be any instances of the class at all! So the way we access a `static` method (or `static` variable) is to use the dot operator on the type name, as opposed to using it on a reference to an instance, as follows:

```
class Frog {  
    private static int frogCount = 0; // static variable  
    static int getCount() {           // static getter method  
        return frogCount;  
    }  
    public Frog() {  
        frogCount += 1;             // Modify the value in the constructor  
    }  
}  
  
class TestFrog {  
    public static void main (String [] args) {  
        new Frog();  
        new Frog();  
        new Frog();  
        System.out.println("from static " + Frog.getCount()); // static context  
        new Frog();
```

```
new TestFrog().go();
Frog f = new Frog();
System.out.println("use ref var " + f.getCount());           // use reference var
}
void go() {
    System.out.println("from instance " + Frog.getCount()); // instance context
}
}
```

which produces the output:

```
from static 3
from instance 4
use ref var 5
```

But just to make it really confusing, the Java language also allows you to use an object reference variable to access a **static** member. Did you catch the last line of `main()`? It included this invocation:

```
f.getCount(); // Access a static using an instance variable
```

In the preceding code, we instantiate a `Frog`, assign the new `Frog` object to the reference variable `f`, and then use the `f` reference to invoke a **static** method! But even though we are using a specific `Frog` instance to access the **static** method, the rules haven't changed. This is merely a syntax trick to let you use an object reference variable (but not the object it refers to) to get to a **static** method or variable, but the **static** member is still unaware of the particular instance used to invoke the **static** member. In the `Frog` example, the compiler knows that the reference variable `f` is of type `Frog`, and so the `Frog` class **static** method is run with no awareness or concern for the `Frog` instance at the other end of the `f` reference. In other words, the compiler cares only that reference variable `f` is declared as type `Frog`.

Invoking **static** methods from interfaces is almost the same as invoking **static**

methods from classes, except the “instance variable syntax trick” just discussed works only for static methods in classes. The following code demonstrates how interface static methods can and cannot be invoked:

```
interface FrogBoilable {
    static int getCtoF(int cTemp) {                      // interface static method
        return (cTemp * 9 / 5) + 32;
    }
    default String hop() { return "hopping"; } // interface default method
}

public class DontBoilFrogs implements FrogBoilable {
    public static void main(String[] args) {
        new DontBoilFrogs().go();
    }

    void go() {
        System.out.println(hop());                         // 1 - ok for default method
        // System.out.println(getCtoF(100));                // 2 - cannot find symbol

        System.out.println(
            FrogBoilable.getCtoF(100));                  // 3 - ok for static method
        DontBoilFrogs dbf = new DontBoilFrogs();
        // System.out.println(dbf.getCtoF(100));            // 4 - cannot find symbol
    }
}
```

Let's review the code:

- **Line 1** is a legal invocation of an interface's `default` method.
- **Line 2** is an **illegal** attempt to invoke an interface's `static` method.

- **Line 3** is THE legal way to invoke an interface's static method.
- **Line 4** is another **illegal** attempt to invoke an interface's static method.

[Figure 2-8](#) illustrates the effects of the static modifier on methods and variables.

FIGURE 2-8

The effects of static on methods and variables

```
class Foo

int size = 42;
static void doMore( ) {
    int x = size;
```

static method cannot
access an instance
(nonstatic) variable

```
class Bar

void go() {}
static void doMore( ) {
    go( );
```

static method cannot
access a nonstatic
method

```
class Baz

static int count;
static void woo( ){ }
static void doMore( ) {
    woo( );
    int x = count;
}
```

static method
can access a static
method or variable

Finally, remember that *static methods can't be overridden!* This doesn't mean they can't be redefined in a subclass, but redefining and overriding aren't the same thing. Let's look at an example of a redefined (remember, not overridden) static method:

```
class Animal {  
    static void doStuff() {  
        System.out.print("a ");  
    }  
}  
  
class Dog extends Animal {  
    static void doStuff() { // it's a redefinition,  
        System.out.print("d "); // not an override  
    }  
}  
  
public static void main(String [] args) {  
    Animal [] a = {new Animal(), new Dog(), new Animal()};  
    for(int x = 0; x < a.length; x++) {  
        a[x].doStuff(); // invoke the static method  
    }  
    Dog.doStuff(); // invoke using the class name  
}
```

Running this code produces this output:

a a a d

Remember, the syntax `a[x].doStuff()` is just a shortcut (the syntax trick)—the compiler is going to substitute something like `Animal.doStuff()` instead. Notice also that you can invoke a static method by using the class name.

We also didn't use the enhanced `for` loop here, even though we could have. Expect to see a mix of both Java 1.4 and Java 5–8 coding styles and practices on the exam.

CERTIFICATION SUMMARY

We started the chapter by discussing the importance of encapsulation in good OO design, and then we talked about how good encapsulation is implemented: with private instance variables and public getters and setters.

Next, we covered the importance of inheritance, so that you can grasp overriding, overloading, polymorphism, reference casting, return types, and constructors.

We covered IS-A and HAS-A. IS-A is implemented using inheritance, and HAS-A is implemented by using instance variables that refer to other objects.

Polymorphism was next. Although a reference variable's type can't be changed, it can be used to refer to an object whose type is a subtype of its own. We learned how to determine what methods are invocable for a given reference variable.

We looked at the difference between overridden and overloaded methods, learning that an overridden method occurs when a subtype inherits a method from a supertype and then reimplements the method to add more specialized behavior. We learned that, at runtime, the JVM will invoke the subtype version on an instance of a subtype and the supertype version on an instance of the supertype. Abstract methods must be “overridden” (technically, abstract methods must be implemented, as opposed to overridden, since there really isn't anything to override).

We saw that overriding methods must declare the same argument list and return type or they can return a subtype of the declared return type of the supertype's overridden method), and that the access modifier can't be more restrictive. The overriding method also can't throw any new or broader checked exceptions that weren't declared in the overridden method. You also learned that the overridden method can be invoked using the syntax `super.doSomething();`.

Overloaded methods let you reuse the same method name in a class, but with different arguments (and, optionally, a different return type). Whereas overriding

methods must not change the argument list, overloaded methods must. But unlike overriding methods, overloaded methods are free to vary the return type, access modifier, and declared exceptions any way they like.

We ended our discussions of overriding by looking at the `@Override` annotation. You can use this annotation to tell the compiler that you intend for the method that follows the annotation to either be an override of a superclass method or an implementation of an interface method. The compiler will tell you if you're doing it wrong.

We learned the mechanics of casting (mostly downcasting) reference variables and when it's necessary to do so.

Implementing interfaces came next. An interface describes a *contract* that the implementing class must follow. The rules for implementing an interface are similar to those for extending an abstract class. As of Java 8, interfaces can have concrete methods, which are labeled `default` or `static`. Also, remember that a class can implement more than one interface and that interfaces can extend another interface.

We also looked at method return types and saw that you can declare any return type you like (assuming you have access to a class for an object reference return type), unless you're overriding a method. Barring a covariant return, an overriding method must have the same return type as the overridden method of the superclass. We saw that, although overriding methods must not change the return type, overloaded methods can (as long as they also change the argument list).

Finally, you learned that it is legal to return any value or variable that can be implicitly converted to the declared return type. So, for example, a `short` can be returned when the return type is declared as an `int`. And (assuming `Horse` extends `Animal`), a `Horse` reference can be returned when the return type is declared an `Animal`.

We covered constructors in detail, learning that if you don't provide a constructor for your class, the compiler will insert one. The compiler-generated constructor is called the default constructor, and it is always a no-arg constructor with a no-arg call to `super()`. The default constructor will never be generated if even a single constructor exists in your class (regardless of the arguments of that constructor); so if you need more than one constructor in your class and you want a no-arg constructor, you'll have to write it yourself. We also saw that constructors are not inherited and that you can be confused by a method that has the same name as the class (which is legal). The return type is the giveaway that a method is not a constructor because constructors do not have return types.

We saw how all the constructors in an object's inheritance tree will always be invoked when the object is instantiated using `new`. We also saw that constructors can be overloaded, which means defining constructors with different argument lists. A constructor can invoke another constructor of the same class using the keyword `this()`, as though the constructor were a method named `this()`. We saw that every constructor must have either `this()` or `super()` as the first statement (although the compiler can insert it for you).

After constructors, we did a quick introduction to design patterns, focusing on a common pattern, the singleton. You use the singleton pattern whenever you want to make sure that only one instance of a class can ever be created.

After singleton, we introduced immutable classes. The phrase "immutable class" is a bit misleading; you should think of these as thread-safe classes whose objects are made immutable through the generous use of the `private` and `final` keywords and by not providing any setter methods.

Next, we discussed the two kinds of initialization blocks and how and when their code runs.

We looked at `static` methods and variables. `static` members are tied to the class or interface, not an instance, so there is only one copy of any `static` member. A common mistake is to attempt to reference an instance variable from a `static` method. Use the respective class or interface name with the dot operator to access `static` members.

And, once again, you learned that the exam includes tricky questions designed largely to test your ability to recognize just how tricky the questions can be.



TWO-MINUTE DRILL

Here are some of the key points from each certification objective in this chapter.

Encapsulation, IS-A, HAS-A* (OCP Objectives 1.1 and 1.2)

- Encapsulation helps hide implementation behind an interface (or API).
- Encapsulated code has two features:
 - Instance variables are kept protected (usually with the `private` modifier).
 - Getter and setter methods provide access to instance variables.

- IS-A refers to inheritance or implementation.
- IS-A is expressed with the keyword `extends` or `implements`.
- IS-A, “inherits from,” and “is a subtype of” are all equivalent expressions.
- HAS-A means an instance of one class “has a” reference to an instance of another class or another instance of the same class. *HAS-A is NOT on the exam, but it’s good to know.

Inheritance (OCP Objective 1.2)

- Inheritance allows a type to be a subtype of a supertype and thereby inherit public and protected variables and methods of the supertype.
- Inheritance is a key concept that underlies IS-A, polymorphism, overriding, overloading, and casting.
- All classes (except class `Object`) are subclasses of type `Object`, and therefore they inherit `Object`’s methods.

Polymorphism (OCP Objective 1.3)

- Polymorphism means “many forms.”
- A reference variable is always of a single, unchangeable type, but it can refer to a subtype object.
- A single object can be referred to by reference variables of many different types—as long as they are the same type or a supertype of the object.
- The reference variable’s type (not the object’s type) determines which methods can be called!
- Polymorphic method invocations apply only to overridden *instance* methods.

Overriding and Overloading (OCP Objectives 1.2 and 1.3)

- Methods can be overridden or overloaded; constructors can be overloaded but not overridden.
- With respect to the method it overrides, the overriding method
 - Must have the same argument list
 - Must have the same return type or a subclass (known as a covariant

return)

- Must not have a more restrictive access modifier
- May have a less restrictive access modifier
- Must not throw new or broader checked exceptions
- May throw fewer or narrower checked exceptions, or any unchecked exception
- `final` methods cannot be overridden.
- Only inherited methods may be overridden, and remember that private methods are not inherited.
- A subclass uses `super.overriddenMethodName()` to call the superclass version of an overridden method.
- A subclass uses `MyInterface.super.overriddenMethodName()` to call the super interface version on an overridden method.
- Overloading means reusing a method name but with different arguments.
- Overloaded methods
 - Must have different argument lists
 - May have different return types, if argument lists are also different
 - May have different access modifiers
 - May throw different exceptions
- Methods from a supertype can be overloaded in a subtype.
- Polymorphism applies to overriding, not to overloading.
- Object type (not the reference variable's type) determines which overridden method is used at runtime.
- Reference type determines which overloaded method will be used at compile time.

@Override Annotation (OCP Objective 2.5)

- The `@Override` annotation can be used to ask the compiler to verify that you've properly overridden a method.
- The `@Override` annotation can be used to ask the compiler to verify that you've properly implemented an interface method.

Reference Variable Casting (OCP Objectives 1.2 and 1.3)

- There are two types of reference variable casting: downcasting and upcasting.
 - Downcasting** If you have a reference variable that refers to a subtype object, you can assign it to a reference variable of the subtype. You must make an explicit cast to do this, and the result is that you can access the subtype's members with this new reference variable.
 - Upcasting** You can assign a reference variable to a supertype reference variable explicitly or implicitly. This is an inherently safe operation because the assignment restricts the access capabilities of the new variable.

Implementing an Interface (OCP Objective 2.5)

- When you implement an interface, you are fulfilling its contract.
- You implement an interface by properly and concretely implementing all the abstract methods defined by the interface.
- A single class can implement many interfaces.

Return Types (OCP Objectives 1.2 and 1.3)

- Overloaded methods can change return types; overridden methods cannot, except in the case of covariant returns.
- Object reference return types can accept `null` as a return value.
- An array is a legal return type, both to declare and return as a value.
- For methods with primitive return types, any value that can be implicitly converted to the return type can be returned.
- Nothing can be returned from a `void`, but you can return nothing. You're allowed to simply say `return` in any method with a `void` return type to bust out of a method early. But you can't return nothing from a method with a non-`void` return type.
- Methods with an object reference return type can return a subtype.
- Methods with an interface return type can return any implementer.

Constructors and Instantiation (OCP Objectives 1.2 and 1.3)

- A constructor is always invoked when a new object is created.

- When a new object is created, a constructor for each superclass in the object's inheritance tree will be invoked.
- Every class, even an abstract class, has at least one constructor.
- Constructors must have the same name as the class.
- Constructors don't have a return type. If you see code with a return type, it's a method with the same name as the class; it's not a constructor.
- Typical constructor execution occurs as follows:
 - The constructor calls its superclass constructor, which calls its superclass constructor, and so on, all the way up to the `Object` constructor.
 - The `Object` constructor executes and then returns to the calling constructor, which runs to completion and then returns to its calling constructor, and so on, back down to the completion of the constructor of the actual instance being created.
- Constructors can use any access modifier (even `private!`).
- The compiler will create a default constructor if you don't create any constructors in your class.
- The default constructor is a no-arg constructor with a no-arg call to `super()`.
- The first statement of every constructor must be a call either to `this()` (an overloaded constructor) or to `super()`.
- The compiler will add a call to `super()` unless you have already put in a call to `this()` or `super()`.
- Instance members are accessible only after the super constructor runs.
- Abstract classes have constructors that are called when a concrete subclass is instantiated.
- Interfaces do not have constructors.
- If your superclass does not have a no-arg constructor, you must create a constructor and insert a call to `super()` with arguments matching those of the superclass constructor.
- Constructors are never inherited; thus, they cannot be overridden.
- A constructor can be directly invoked only by another constructor (using a call to `super()` or `this()`).
- Regarding issues with calls to `this()`:

- They may appear only as the first statement in a constructor.
- The argument list determines which overloaded constructor is called.
- Constructors can call constructors, and so on, but sooner or later one of them better call super() or the stack will explode.
- Calls to this() and super() cannot be in the same constructor. You can have one or the other, but never both.

Singleton Design Pattern (OCP Objective 1.5)

- A design pattern is “a general reusable solution to a commonly occurring problem within a given context.”
- Having only one instance of the object allows a program to share its state.
- This pattern might improve performance by not repeating the same work.
- This pattern often stores a single instance as a static variable.
- We can instantiate right away (eager) or when needed (lazy).

Immutable Classes (OCP Objective 1.5)

- Since they are thread-safe, they are great for concurrent and/or parallel applications.
- They should be built using the following guidelines:
 - Mark the class final
 - Mark the variables private and final
 - Do NOT provide setter methods
 - Whenever references to mutable types are sent or received, a defensive copy should be used.
- Unlike singletons, many instances of an immutable class can be made.

Initialization Blocks (OCP Objective 1.6)

- Use static init blocks—static { /* code here */ }—for code you want to have run once, when the class is first loaded. Multiple blocks run from the top down.
- Use normal init blocks—{ /* code here */ }—for code you want to have run for every new instance, right after all the super constructors have run. Again, multiple blocks run from the top of the class down.

Statics (OCP Objective 1.6)

- Use static methods to implement behaviors that are not affected by the state of any instances.
- Use static variables to hold data that is class specific as opposed to instance specific—there will be only one copy of a static variable.
- All static members belong to the class, not to any instance.
- A static method can't access an instance variable directly.
- Use the dot operator to access static members, but remember that using a reference variable with the dot operator is really a syntax trick, and the compiler will substitute the class name for the reference variable; for instance:
`d.doStuff();`
becomes
`Dog.doStuff();`
- To invoke an interface's static method, use `MyInterface.doStuff()` syntax.
- static methods can't be overridden, but they can be redefined.

Q SELF TEST

1. Given:

```
public abstract interface Frobinate { public void  
    twiddle(String s); }
```

Which is a correct class? (Choose all that apply.)

- A. public abstract class Frob implements Frobnicate {
 public abstract void twiddle(String s) { }
}
- B. public abstract class Frob implements Frobnicate { }
- C. public class Frob extends Frobnicate {
 public void twiddle(Integer i) { }
}
- D. public class Frob implements Frobnicate {
 public void twiddle(Integer i) { }
}
- E. public class Frob implements Frobnicate {
 public void twiddle(String i) { }
 public void twiddle(Integer s) { }
}

2. Given:

```
class Top {  
    public Top(String s) { System.out.print("B"); }  
}  
public class Bottom2 extends Top {  
    public Bottom2(String s) { System.out.print("D"); }  
    public static void main(String [] args) {  
        new Bottom2("C");  
        System.out.println(" ");  
    }  
}
```

What is the result?

- A. BD
- B. DB
- C. BDC

D. DBC

E. Compilation fails

3. Given:

```
class Clidder {  
    private final void flipper() { System.out.println("Clidder"); }  
}  
  
public class Clidlet extends Clidder {  
    public final void flipper() { System.out.println("Clidlet"); }  
    public static void main(String [] args) {  
        new Clidlet().flipper();  
    }  
}
```

What is the result?

A. Clidlet

B. Clidder

C. Clidder
 Clidlet

D. Clidlet
 Clidder

E. Compilation fails

Special Note: The next question crudely simulates a “drag-and-drop” style of question that you probably will NOT encounter on the exam, but just in case, we’ve left a few drag-and-drop questions in the book.

4. Using the **fragments** below, complete the following **code** so it compiles.
Note that you may not have to fill in all of the slots.

Code:

```
class AgedP {  
    _____ _____ _____ _____ _____  
    public AgedP(int x) {  
        _____ _____ _____ _____ _____  
    } }  
    public class Kinder extends AgedP {  
        _____ _____ _____ _____ _____  
        public Kinder(int x) {  
            _____ _____ _____ _____ _____ ();  
        } }
```

Fragments: Use the following fragments zero or more times:

AgedP	super	this	
()	{	}
;			

5. Given:

```
class Bird {  
    { System.out.print("b1 "); }  
    public Bird() { System.out.print("b2 "); }  
}  
class Raptor extends Bird {  
    static { System.out.print("r1 "); }  
    public Raptor() { System.out.print("r2 "); }  
    { System.out.print("r3 "); }  
    static { System.out.print("r4 "); }  
}  
class Hawk extends Raptor {  
    public static void main(String[] args) {  
        System.out.print("pre ");  
        new Hawk();  
        System.out.println("hawk ");  
    }  
}
```

What is the result?

- A. pre b1 b2 r3 r2 hawk
- B. pre b2 b1 r2 r3 hawk
- C. pre b2 b1 r2 r3 hawk r1 r4
- D. r1 r4 pre b1 b2 r3 r2 hawk
- E. r1 r4 pre b2 b1 r2 r3 hawk
- F. pre r1 r4 b1 b2 r3 r2 hawk
- G. pre r1 r4 b2 b1 r2 r3 hawk
- H. The order of output cannot be predicted
- I. Compilation fails

Note: You'll probably never see this many choices on the real exam!

6. Given the following:

```
1. class X { void do1() { } }
```

```
2. class Y extends X { void do2() { } }
```

```
3.
```

```
4. class Chrome {
```

```
5.     public static void main(String [] args) {
```

```
6.         X x1 = new X();
```

```
7.         X x2 = new Y();
```

```
8.         Y y1 = new Y();
```

```
9.         // insert code here
```

```
10.    }
```

Which of the following, inserted at line 9, will compile? (Choose all that apply.)

- A. x2.do2();
- B. (Y)x2.do2();
- C. ((Y)x2).do2();
- D. None of the above statements will compile

7. Given:

```
public class Locomotive {  
    Locomotive() { main("hi"); }  
  
    public static void main(String[] args) {  
        System.out.print("2 ");  
    }  
    public static void main(String args) {  
        System.out.print("3 " + args);  
    }  
}
```

What is the result? (Choose all that apply.)

- A. 2 will be included in the output
- B. 3 will be included in the output
- C. hi will be included in the output
- D. Compilation fails
- E. An exception is thrown at runtime

8. Given:

```
3. class Dog {  
4.     public void bark() { System.out.print("woof "); }  
5. }  
6. class Hound extends Dog {  
7.     public void sniff() { System.out.print("sniff "); }  
8.     public void bark() { System.out.print("howl "); }  
9. }  
10. public class DogShow {  
11.     public static void main(String[] args) { new DogShow().go(); }  
12.     void go() {  
13.         new Hound().bark();  
14.         ((Dog) new Hound()).bark();  
15.         ((Dog) new Hound()).sniff();  
16.     }  
17. }
```

What is the result? (Choose all that apply.)

- A. howl howl sniff
- B. howl woof sniff
- C. howl howl followed by an exception
- D. howl woof followed by an exception
- E. Compilation fails with an error at line 14
- F. Compilation fails with an error at line 15

9. Given:

```
3. public class Redwood extends Tree {  
4.     public static void main(String[] args) {  
5.         new Redwood().go();  
6.     }  
7.     void go() {  
8.         go2(new Tree(), new Redwood());  
9.         go2((Redwood) new Tree(), new Redwood());  
10.    }  
11.    void go2(Tree t1, Redwood r1) {  
12.        Redwood r2 = (Redwood)t1;  
13.        Tree t2 = (Tree)r1;  
14.    }  
15. }  
16. class Tree { }
```

What is the result? (Choose all that apply.)

- A. An exception is thrown at runtime
- B. The code compiles and runs with no output
- C. Compilation fails with an error at line 8
- D. Compilation fails with an error at line 9
- E. Compilation fails with an error at line 12
- F. Compilation fails with an error at line 13

10. Given:

```
3. public class Tenor extends Singer {  
4.     public static String sing() { return "fa"; }  
5.     public static void main(String[] args) {  
6.         Tenor t = new Tenor();  
7.         Singer s = new Tenor();  
8.         System.out.println(t.sing() + " " + s.sing());  
9.     }  
10. }  
11. class Singer { public static String sing() { return "la"; } }
```

What is the result?

- A. fa fa
- B. fa la
- C. la la
- D. Compilation fails
- E. An exception is thrown at runtime

11. Given:

```
3. class Alpha {  
4.     static String s = " ";  
5.     protected Alpha() { s += "alpha "; }  
6. }  
7. class SubAlpha extends Alpha {  
8.     private SubAlpha() { s += "sub "; }  
9. }  
10. public class SubSubAlpha extends Alpha {  
11.     private SubSubAlpha() { s += "subsub "; }  
12.     public static void main(String[] args) {  
13.         new SubSubAlpha();  
14.         System.out.println(s);  
15.     }  
16. }
```

What is the result?

- A. subsub
- B. sub subsub
- C. alpha subsub
- D. alpha sub subsub
- E. Compilation fails
- F. An exception is thrown at runtime

12. Given:

```
3. class Building {  
4.     Building() { System.out.print("b "); }  
5.     Building(String name) {  
6.         this(); System.out.print("bn " + name);  
7.     }  
8. }  
9. public class House extends Building {  
10.    House() { System.out.print("h "); }  
11.    House(String name) {  
12.        this(); System.out.print("hn " + name);  
13.    }  
14.    public static void main(String[] args) { new House("x"); }  
15. }
```

What is the result?

- A. h hn x
- B. hn x h
- C. b h hn x
- D. b hn x h
- E. bn x h hn x
- F. b bn x h hn x
- G. bn x b h hn x
- H. Compilation fails

13. Given:

```
3. class Mammal {  
4.     String name = "furry ";  
5.     String makeNoise() { return "generic noise"; }  
6. }  
7. class Zebra extends Mammal {  
8.     String name = "stripes ";  
9.     String makeNoise() { return "bray"; }  
10. }  
11. public class ZooKeeper {  
12.     public static void main(String[] args) { new ZooKeeper().go(); }  
13.     void go() {  
14.         Mammal m = new Zebra();  
15.         System.out.println(m.name + m.makeNoise());  
16.     }  
17. }
```

What is the result?

- A. **furry bray**
- B. **stripes bray**
- C. **furry generic noise**
- D. **stripes generic noise**
- E. Compilation fails
- F. An exception is thrown at runtime

14. Given:

```
1. interface FrogBoilable {  
2.     static int getCToF(int cTemp) {  
3.         return (cTemp * 9 / 5) + 32;  
4.     }  
5.     default String hop() { return "hopping "; }  
6. }  
7. public class DontBoilFrogs implements FrogBoilable {  
8.     public static void main(String[] args) {  
9.         new DontBoilFrogs().go();  
10.    }  
11.    void go() {  
12.        System.out.print(hop());  
13.        System.out.println(getCToF(100));  
14.        System.out.println(FrogBoilable.getCToF(100));  
15.        DontBoilFrogs dbf = new DontBoilFrogs();  
16.        System.out.println(dbf.getCToF(100));  
17.    }  
18. }
```

What is the result? (Choose all that apply.)

- A. hopping 212
- B. Compilation fails due to an error on line 2
- C. Compilation fails due to an error on line 5
- D. Compilation fails due to an error on line 12
- E. Compilation fails due to an error on line 13
- F. Compilation fails due to an error on line 14
- G. Compilation fails due to an error on line 16

15. Given:

```
interface I1 {
    default int doStuff() { return 1; }
}
interface I2 {
    default int doStuff() { return 2; }
}
public class MultiInt implements I1, I2 {
    public static void main(String[] args) {
        new MultiInt().go();
    }
    void go() {
        System.out.println(doStuff());
    }
    int doStuff() {
        return 3;
    }
}
```

What is the result?

- A. 1
- B. 2
- C. 3
- D. The output is unpredictable
- E. Compilation fails
- F. An exception is thrown at runtime

16. Given:

```
interface MyInterface {  
    default int doStuff() {  
        return 42;  
    }  
}  
public class IfaceTest implements MyInterface {  
    public static void main(String[] args) {  
        new IfaceTest().go();  
    }  
    void go() {  
        // INSERT CODE HERE  
    }  
    public int doStuff() {  
        return 43;  
    }  
}
```

Which line(s) of code, inserted independently at // INSERT CODE HERE, will allow the code to compile? (Choose all that apply.)

- A. System.out.println("class: " + doStuff());
- B. System.out.println("iface: " + super.doStuff());
- C. System.out.println("iface: " + MyInterface.super.doStuff());
- D. System.out.println("iface: " + MyInterface.doStuff());
- E. System.out.println("iface: " + super.MyInterface.doStuff());
- F. None of the lines, A–E, will allow the code to compile

17. Given:

```
interface i1 {
    void doStuff(int x);
}

class Patton {
    void stuff(String s) {
        System.out.println("stuff ");
    }
}

public class override extends Patton implements i1 {
    public static void main(String[] args) {
        new override().doStuff(1);
        new override().stuff("x");
    }

    @Override
    void doStuff(int x) {
        System.out.print("doStuff ");
    }

    @Override
    void stuff(String s) {
        System.out.println("my stuff ");
    }
}
```

What is the result? (Choose all that apply.)

- A. Stuff stuff
- B. doStuff stuff
- C. doStuff my stuff
- D. An exception is thrown at runtime

- E. Compilation fails due to an @Override-related error
 - F. Compilation fails due to an error other than an @Override-related error
- 18.** Which statements about singletons are true? (Choose all that apply.)
- A. The singleton pattern ensures that no two objects of the same class will have duplicate state.
 - B. A class that properly implements a singleton must have a constructor marked either private or protected.
 - C. Typically, a singleton's public-facing API has only one method.
 - D. The singleton pattern is considered a creational design pattern.
 - E. A properly designed singleton must declare at least one enum.
- 19.** Which statements about immutable classes are true? (Choose all that apply.)
- A. They should be marked `final` so that they cannot be subclassed.
 - B. Their fields should allow updates only via setter methods.
 - C. Their objects must be instantiated via factory methods.
 - D. None of their fields can be of mutable types.
 - E. Reference variables sent in as instantiation arguments must be dealt with defensively.
 - F. Reference variables returned in getters must be dealt with defensively.
 - G. Properly designed immutable classes are well encapsulated.

A SELF TEST ANSWERS

- 1.** **B** and **E** are correct. **B** is correct because an abstract class need not implement any or all of an interface's methods. **E** is correct because the class implements the interface method and additionally overloads the `twiddle()` method.
 - A**, **C**, and **D** are incorrect. **A** is incorrect because abstract methods have no body. **C** is incorrect because classes implement interfaces; they don't extend them. **D** is incorrect because overloading a method is not implementing it. (OCP Objectives 1.2, 1.3, and 2.5)
- 2.** **E** is correct. The implied `super()` call in `Bottom2`'s constructor cannot be satisfied because there is no no-arg constructor in `Top`. A default no-arg

constructor is generated by the compiler only if the class has no constructor defined explicitly.

- ☒ **A, B, C, and D** are incorrect based on the above. (OCP Objectives 1.2 and 1.3)
- 3. ☑ **A** is correct. Although a `final` method cannot be overridden, in this case, the method is private and, therefore, hidden. The effect is that a new, accessible method `flipper` is created. Therefore, no polymorphism occurs in this example; the method invoked is simply that of the child class; and no error occurs.
- ☒ **B, C, D, and E** are incorrect based on the preceding. (OCP Objectives 1.3 and 2.2)

Special Note: This next question crudely simulates a style of question known as “drag-and-drop.” Up through the SCJP 6 exam, drag-and-drop questions were included. As of early 2018, Oracle DOES NOT include any drag-and-drop questions on its Java exams, but just in case Oracle’s policy changes, we left a few in the book.

4. Here is the answer:

```
class AgedP {  
    AgedP() {}  
    public AgedP(int x) {  
    }  
}  
public class Kinder extends AgedP {  
    public Kinder(int x) {  
        super();  
    }  
}
```

As there is no droppable tile for the variable `x` and the parentheses (in the `Kinder` constructor) are already in place and empty, there is no way to construct a call to the superclass constructor that takes an argument. Therefore, the only remaining possibility is to create a call to the no-arg

superclass constructor. This is done as `super()`. The line cannot be left blank, as the parentheses are already in place. Further, since the superclass constructor called is the no-arg version, this constructor must be created. It will not be created by the compiler because another constructor is already present. (OCP Objectives 1.2 and 1.3)

5. **D** is correct. Static `init` blocks are executed at class loading time; instance `init` blocks run right after the call to `super()` in a constructor. When multiple `init` blocks of a single type occur in a class, they run in order, from the top down.
 - A, B, C, E, F, G, H, and I** are incorrect based on the above. Note: You'll probably never see this many choices on the real exam! (OCP Objective 2.6)
6. **C** is correct. Before you can invoke `Y`'s `do2` method, you have to cast `x2` to be of type `Y`.
 - A, B, and D** are incorrect based on the preceding. **B** looks like a proper cast, but without the second set of parentheses, the compiler thinks it's an incomplete statement. (OCP Objectives 1.2 and 1.3)
7. **A** is correct. It's legal to overload `main()`. Because no instances of `Locomotive` are created, the constructor does not run and the overloaded version of `main()` does not run.
 - B, C, D, and E** are incorrect based on the preceding. (OCP Objectives 1.2 and 1.3)
8. **F** is correct. Class `Dog` doesn't have a `sniff` method.
 - A, B, C, D, and E** are incorrect based on the above information. (OCP Objectives 1.2 and 1.3)
9. **A** is correct. A `ClassCastException` will be thrown when the code attempts to downcast a `Tree` to a `Redwood`.
 - B, C, D, E, and F** are incorrect based on the above information. (OCP Objectives 1.2 and 1.3)
10. **B** is correct. The code is correct, but polymorphism doesn't apply to static methods.
 - A, C, D, and E** are incorrect based on the above information. (OCP Objectives 1.3 and 1.6)

- 11.** **C** is correct. Watch out, because SubSubAlpha extends Alpha! Because the code doesn't attempt to make a SubAlpha, the private constructor in SubAlpha is okay.
- A, B, D, E, and F** are incorrect based on the above information. (OCP Objectives 1.2 and 1.3)
- 12.** **C** is correct. Remember that constructors call their superclass constructors, which execute first, and that constructors can be overloaded.
- A, B, D, E, F, G, and H** are incorrect based on the above information. (OCP Objectives 1.2 and 1.3)
- 13.** **A** is correct. Polymorphism is only for instance methods, not instance variables.
- B, C, D, E, and F** are incorrect based on the above information. (OCP Objective 1.3)
- 14.** **E** and **G** are correct. Neither of these lines of code uses the correct syntax to invoke an interface's static method.
- A, B, C, D, and F** are incorrect based on the above information. (OCP Objective 2.5)
- 15.** **E** is correct. This is kind of a trick question; the implementing method must be marked `public`. If it was, all the other code is legal, and the output would be 3. If you understood all the multiple inheritance rules and just missed the access modifier, give yourself half credit.
- A, B, C, D, and F** are incorrect based on the above information. (OCP Objective 2.5)
- 16.** **A** and **C** are correct. **A** uses correct syntax to invoke the class's method, and **C** uses the correct syntax to invoke the interface's overloaded `default` method.
- B, D, E, and F** are incorrect based on the above information. (OCP Objective 2.5)
- 17.** **F** is correct. The `@Override` methods are properly overridden, but interface methods are `public`, so the compiler will complain about weaker access privileges. This question could be called a "misdirection" question. On the surface, it appears to be about `@Override`, but you might get bitten by another problem. We agree that these are tricky, but you will encounter

this sort of tricky question on the real exam.

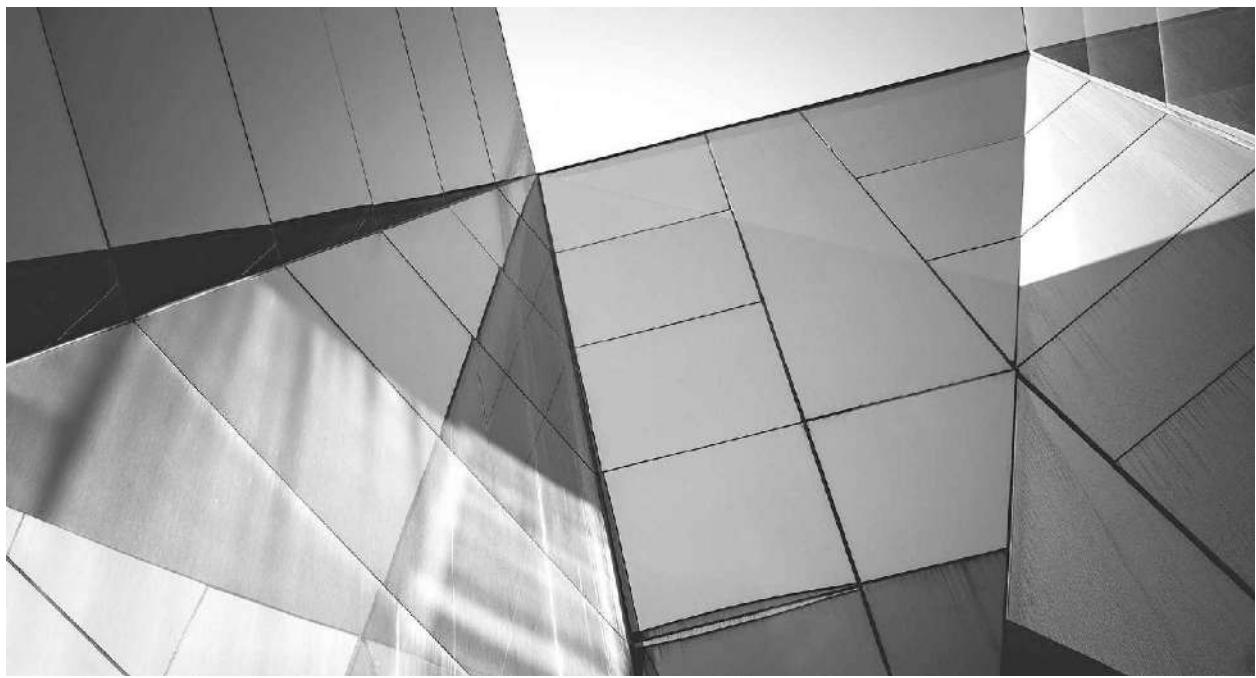
A, B, C, D, and E are incorrect based on the above information. (OCP Objective 2.5)

18. **D** is a correct statement about singletons.

A, B, C, and E are incorrect. **A** is simply incorrect. **B** is almost right; a singleton's constructor can only be marked private. **C** is simply incorrect; a singleton usually has many methods. **E** is incorrect because, although you can implement a singleton using an enum, using an enum is not required. (OCP Objective 1.5)

19. **A, E, F, and G** are correct statements about immutable classes.

B is incorrect because immutable classes cannot have any setter methods. **C** is incorrect because immutable classes can also use constructors. **D** is incorrect because fields can be of mutable types, but they must be dealt with defensively. (OCP Objective 1.5)



3

Assertions and Java Exceptions

CERTIFICATION OBJECTIVES

- Use try-catch and throw Statements
 - Use catch, Multi-catch, and finally Clauses
 - Use Autoclose Resources with a try-with-resources Statement
 - Create Custom Exceptions and Auto-closeable Resources
 - Test Invariants by Using Assertions
-  Two-Minute Drill

Q&A Self Test

The assertion mechanism gives you a way to do testing and debugging checks on conditions you expect to smoke out while developing, when you don't want the runtime overhead associated with exception handling.

When you do need to use exception handling, you can take advantage of two features added to exception handling in Java 7. First, multi-catch gives you a way of dealing with two or more exception types at once, and second, try-with-resources lets you close your resources very easily.

CERTIFICATION OBJECTIVE

Working with the Assertion Mechanism (OCP Objective 6.5)

6.5 Test invariants by using assertions.

You know you're not supposed to make assumptions, but you can't help it when you're writing code. You put them in comments:

```
if (x > 2) {  
    // do something  
} else if (x < 2) {  
    // do something  
} else {  
    // x must be 2  
    // do something else  
}
```

You write print statements with them:

```
while (true) {  
    if (x > 2) {  
        break;  
    }  
    System.out.print("If we got here " +  
                    "something went horribly wrong");  
}
```

Assertions let you test your assumptions during development, without the expense (in both your time and program overhead) of writing exception handlers, for exceptions that you assume will never happen once the program is out of development and fully deployed.

For the OCP 8 exam, you're expected to know the basics of how assertions work, including how to enable them, how to use them, and how *not* to use them.

Assertions Overview

Suppose you assume that a number passed into a method (say, `methodA()`) will never be negative. While testing and debugging, you want to validate your assumption, but you don't want to have to strip out `print` statements, runtime exception handlers, or `if/else` tests when you're done with development. But leaving any of those in is, at the least, a performance hit. Assertions to the rescue! Check out the following code:

```
private void methodA(int num) {  
    if (num >= 0) {  
        useNum(num + x);  
    } else { // num < 0 (this should never happen!)  
        System.out.println("Yikes! num is a negative number! " + num);  
    }  
}
```

Because you're so certain of your assumption, you don't want to take the time (or program performance hit) to write exception-handling code. And at runtime, you don't want the `if/else` either because if you do reach the `else` condition, it means your earlier logic (whatever was running prior to this method being called) is flawed.

Assertions let you test your assumptions during development, but the assertion code basically evaporates when the program is deployed, leaving behind no overhead or debugging code to track down and remove. Let's rewrite `methodA()` to validate that the argument was not negative:

```
private void methodA(int num) {  
    assert (num>=0); // throws an AssertionError  
                    // if this test isn't true  
    useNum(num + x);  
}
```

Not only do assertions make code stay cleaner and tighter, but also, because assertions are inactive unless specifically “turned on” (enabled), the code will run as though it were written like this:

```
private void methodA(int num) {  
    useNum(num + x); // we've tested this;  
                      // we now know we're good here  
}
```

Assertions work quite simply. You always assert that something is true. If it is, no problem. Code keeps running. But if your assertion turns out to be wrong (`false`), then a stop-the-world `AssertionError` is thrown (which you should never, ever handle!) right then and there, so you can fix whatever logic flaw led to the problem.

Assertions come in two flavors: *really simple* and *simple*, as follows:

Really simple:

```
private void doStuff() {  
    assert (y > x);  
    // more code assuming y is greater than x  
}
```

Simple:

```
private void doStuff() {  
    assert (y > x): "y is " + y + " x is " + x;  
    // more code assuming y is greater than x  
}
```

The difference between the two is that the simple version adds a second expression separated from the first (boolean expression) by a colon—this expression’s string value is added to the stack trace. Both versions throw an

immediate `AssertionError`, but the simple version gives you a little more debugging help, whereas the really simple version tells you only that your assumption was false.



Assertions are typically enabled when an application is being tested and debugged, but disabled when the application is deployed. The assertions are still in the code, although ignored by the JVM, so if you do have a deployed application that starts misbehaving, you can always choose to enable assertions in the field for additional testing.

Assertion Expression Rules

Assertions can have either one or two expressions, depending on whether you're using the "simple" or the "really simple." The first expression (we'll call it `expression1`) must always result in a boolean value! Follow the same rules you use for `if` and `while` tests. The whole point is to assert `aTest`, which means you're asserting that `aTest` is `true`. If it is `true`, no problem. If it's not `true`, however, then your assumption was wrong and you get an `AssertionError`.

The second expression (`expression2`), used only with the simple version of an `assert` statement, can be anything that results in a value. Remember, the second expression is used to generate a `String` message that displays in the stack trace to give you a little more debugging information. It works much like `System.out.println()` in that you can pass it a primitive or an object and it will convert it into a `String` representation. It must resolve to a value!

The following code lists legal and illegal expressions for both parts of an `assert` statement. Remember, `expression2` (the value to print in the stack trace) is used only with the simple `assert` statement; the second expression exists solely to give you a little more debugging detail:

```
void noReturn() { }
int aReturn() { return 1; }
void go() {
    int x = 1;
    boolean b = true;

    // the following six are legal assert statements
    assert(x == 1);
    assert(b);
    assert(true);
    assert(x == 1) : x;
    assert(x == 1) : aReturn();
    assert(x == 1) : new ValidAssert();

    // the following six are ILLEGAL assert statements
    assert(x = 1);           // none of these are booleans
    assert(x);
    assert(0);
    assert(x == 1) : ;        // none of these return a value
    assert(x == 1) : noReturn();
    assert(x == 1) : ValidAssert va;
}
```



If you see the word "expression" in a question about assertions and the

question doesn't specify whether it means expression1 (the boolean test) or expression2 (the value to print in the stack trace), always assume the word "expression" refers to expression1, the boolean test. For example, consider the following question:

Exam Question: An assert expression must result in a boolean value, true or false?

Assume that the word "expression" refers to expression1 of an assert, so the question statement is correct. If the statement were referring to expression2, however, the statement would be incorrect because expression2 can be anything that results in a value, not just a boolean.

Using Assertions

If you want to use assertions, the first step is to put them in your code. Next, every time you run your code, you can choose whether to enable the assertions or not.

Running with Assertions

Here's where it gets cool. Once you've written your assertion-aware code, you can choose to enable or disable your assertions at runtime! The first thing to remember is that at runtime assertions are **disabled by default**.

Enabling Assertions at Runtime

You enable assertions at runtime with a command like this:

```
java -ea com.geeksanonymous.TestClass
```

or

```
java -enableassertions com.geeksanonymous.TestClass
```

The preceding command-line switches tell the JVM to run with assertions enabled.

Disabling Assertions at Runtime

You must also know the command-line switches for disabling assertions:

```
java -da com.geeksanonymous.TestClass
```

or

```
java -disableassertions com.geeksanonymous.TestClass
```

Because assertions are disabled by default, using the disable switches might seem unnecessary. Indeed, using the switches the way we do in the preceding example just gives you the default behavior (in other words, you get the same result, regardless of whether you use the disabling switches). But...you can also selectively enable and disable assertions in such a way that they're enabled for some classes and/or packages and disabled for others while a particular program is running.

Selective Enabling and Disabling

The command-line switches for assertions can be used in various ways:

- **With no arguments (as in the preceding examples)** Enables or disables assertions in all classes, except for the system classes.
- **With a package name** Enables or disables assertions in the package specified and in any packages below this package in the same directory hierarchy (more on that in a moment).
- **With a class name** Enables or disables assertions in the class specified.

You can combine switches to, say, disable assertions in a single class but keep them enabled for all others as follows:

```
java -ea -da:com.geeksanonymous.Foo
```

The preceding command line tells the JVM to enable assertions in general, but disable them in the class `com.geeksanonymous.Foo`. You can do the same selectivity for a package as follows:

```
java -ea -da:com.geeksanonymous...
```

The preceding command line tells the JVM to enable assertions in general,

but disable them in the package `com.geeksanonymous` and all of its subpackages! You may not be familiar with the term “subpackages,” since that term wasn’t used much prior to assertions. A subpackage is any package in a subdirectory of the named package. For example, look at the following directory tree:

```
com
  | _geeksanonymous
    | _Foo.class
    | _twelvesteps
      | _StepOne.class
      | _StepTwo.class
```

This tree lists three directories:

```
com
geeksanonymous
twelvesteps
and three classes:
com.geeksanonymous.Foo
com.geeksanonymous.twelvesteps.StepOne
com.geeksanonymous.twelvesteps.StepTwo
```

The subpackage of `com.geeksanonymous` is the `twelvesteps` package. Remember that in Java, the `com.geeksanonymous.twelvesteps` package is treated as a completely distinct package that has no relationship with the packages above it (in this example, the `com.geeksanonymous` package), except they just happen to share a couple of directories. [Table 3-1](#) lists examples of command-line switches for enabling and disabling assertions.

TABLE 3-1 Assertion Command-Line Switches

Command-Line Example	What It Means
<code>java -ea</code> <code>java -enableassertions</code>	Enable assertions.
<code>java -da</code> <code>java -disableassertions</code>	Disable assertions (the default behavior).
<code>java -ea:com.foo.Bar</code>	Enable assertions in class com.foo.Bar.
<code>java -ea:com.foo...</code>	Enable assertions in package com.foo and any of its subpackages.
<code>java -ea -dsa</code>	Enable assertions in general, but disable assertions in system classes.
<code>java -ea -da:com.foo...</code>	Enable assertions in general, but disable assertions in package com.foo and any of its subpackages.

Using Assertions Appropriately

Not all legal uses of assertions are considered appropriate. As with so much of Java, you can abuse the intended use of assertions, despite the best efforts of Oracle's Java engineers to discourage you from doing so. For example, you're never supposed to handle an assertion failure. That means you shouldn't catch it with a catch clause and attempt to recover. Legally, however, `AssertionError` is a subclass of `Throwable`, so it can be caught. But just don't do it! If you're going to try to recover from something, it should be an exception. To discourage you from trying to substitute an assertion for an exception, the `AssertionError` doesn't provide access to the object that generated it. All you get is the `String` message.

So who gets to decide what's appropriate? Oracle. The exam uses Oracle's "official" assertion documentation to define appropriate and inappropriate uses.

Don't Use Assertions to Validate Arguments to a public Method

The following is an inappropriate use of assertions:

```
public void doStuff(int x) {  
    assert (x > 0);           // inappropriate !  
    // do things with x  
}
```



*If you see the word “appropriate” on the exam, do not mistake that for “legal.” “Appropriate” always refers to the way in which something is supposed to be used, according to either the developers of the mechanism or best practices officially embraced by Oracle. If you see the word “correct” in the context of assertions, as in “Line 3 is a correct use of assertions,” you should also assume that correct is referring to how assertions **SHOULD** be used rather than how they legally **COULD** be used.*

A public method might be called from code that you don't control (or from code you have never seen). Because public methods are part of your interface to the outside world, you're supposed to guarantee that any constraints on the arguments will be enforced by the method itself. But since assertions aren't guaranteed to actually run (they're typically disabled in a deployed application), the enforcement won't happen if assertions aren't enabled. You don't want publicly accessible code that works only conditionally, depending on whether assertions are enabled.

If you need to validate public method arguments, you'll probably use exceptions to throw, say, an `IllegalArgumentException` if the values passed to the public method are invalid.

Do Use Assertions to Validate Arguments to a private Method

If you write a `private` method, you almost certainly wrote (or control) any code that calls it. When you assume that the logic in code calling your `private` method is correct, you can test that assumption with an assertion as follows:

```
private void doMore(int x) {  
    assert (x > 0);  
    // do things with x  
}
```

The only difference that matters between the preceding example and the one before it is the access modifier. So, do enforce constraints on `private` methods' arguments, but do not enforce constraints on `public` methods. You're certainly free to compile assertion code with an inappropriate validation of `public` arguments, but for the exam (and real life), you need to know that you shouldn't do it.

Don't Use Assertions to Validate Command-Line Arguments

This is really just a special case of the “Do not use assertions to validate arguments to a `public` method” rule. If your program requires command-line arguments, you’ll probably use the exception mechanism to enforce them.

Do Use Assertions, Even in public Methods, to Check for Cases That You Know Are Never, Ever Supposed to Happen

This can include code blocks that should never be reached, including the default of a `switch` statement as follows:

```
switch(x) {  
    case 1: y = 3; break;  
    case 2: y = 9; break;  
    case 3: y = 27; break;  
    default: assert false; // we're never supposed to get here!  
}
```

If you assume that a particular code block won't be reached, as in the preceding example where you assert that `x` must be 1, 2, or 3, then you can use `assert false` to cause an `AssertionError` to be thrown immediately if you ever do reach that code. So in the `switch` example, we're not performing a boolean test—we've already asserted that we should never be there, so just getting to that point is an automatic failure of our assertion/assumption.

Don't Use `assert` Expressions That Can Cause Side Effects!

The following would be a very bad idea:

```
public void doStuff() {  
    assert (modifyThings());  
    // continues on  
}  
public boolean modifyThings() {  
    y = x++;  
    return true;  
}
```

The rule is that an `assert` expression should leave the program in the same state it was in before the expression! Think about it. `assert` expressions aren't guaranteed to always run, so you don't want your code to behave differently depending on whether assertions are enabled. Assertions must not cause any side effects. If assertions are enabled, the only change to the way your program runs

is that an `AssertionError` can be thrown if one of your assertions (think *assumptions*) turns out to be false.



Using assertions that cause side effects can cause some of the most maddening and hard-to-find bugs known to man or woman! When a hot-tempered QA analyst is screaming at you that your code doesn't work, trotting out the old "well, it works on MY machine" excuse won't get you very far.

CERTIFICATION OBJECTIVE

Working with Exception Handling (OCP Objectives 6.1, 6.2, 6.3, and 6.4)

- 6.1 Use try-catch, and throw statements.
- 6.2 Use catch, multi-catch, and finally clauses.
- 6.3 Use Autoclose resources with a try-with-resources statement.
- 6.4 Create custom exceptions and Auto-closeable resources.

You should already know the basics of `try`, `catch`, and `throw`, but if you need a refresher, head back to [Chapter 5](#) of the *OCA Java SE 8 Programmer I Exam Guide*. For this section, we're assuming you know that material, so we're going to dive right into Java's more advanced exception-handling features.

Use the try Statement with multi-catch and finally Clauses

Sometimes we want to handle different types of exceptions the same way. Especially when all we can do is log the exception and declare defeat. But we don't want to repeat code. So what to do? When you were studying for the OCA 8 exam, you already saw that having a single catch-all exception handler is a bad idea. Prior to Java 7, the best we could do was

```
try {
    // access the database and write to a file
} catch (SQLException e) {
    handleErrorCase(e);
} catch (IOException e) {
    handleErrorCase(e);
}
```

You may be thinking that it is only one line of duplicate code. But what happens when you are catching six different exception types? That's a lot of duplication. Luckily, Java 7 made handling this sort of situation nice and easy with a feature called multi-catch:

```
try {
    // access the database and write to a file
} catch (SQLException | IOException e) {
    handleErrorCase(e);
}
```

No more duplication. This is great. As you might imagine, multi-catch is short for “multiple catch.” You just list out the types you want the multi-catch to handle separated by pipe (|) characters. This is easy to remember because | is the “or” operator in Java, which means the catch can be read as “SQLException or IOException e.”



You can't use the variable name multiple times in a multi-catch. The following won't compile:

```
catch(Exception1 e1 | Exception2 e2)
```

It makes sense that this example doesn't compile. After all, the code in the exception handler needs to know which variable name to refer to.

```
catch(Exception1 e | Exception2 e)
```

This one is tempting. When we declare variables, we normally put the variable name right after the type. Try to think of it as a list of types. We are declaring variable e to be caught and it must be one of Exception1 or Exception2 types.

With multi-catch, order doesn't matter. The following two snippets are equivalent to each other:

```
catch(SQLException | IOException e) // these two statements are  
// equivalent  
catch(IOException | SQLException e)
```

Just like with exception matching in a regular catch block, you can't just throw any two exceptions together. With multi-catch, you have to make sure a given exception can only match one type. The following will not compile:

```
catch(FileNotFoundException | IOException e)  
catch(IOException | FileNotFoundException e)
```

You'll get a compiler error that looks something like:

The exception FileNotFoundException is already caught by the alternative IOException

Since FileNotFoundException is a subclass of IOException, we could have just written that in the first place! There was no need to use multi-catch. The simplified and working version simply says:

```
catch(IOException e)
```

Remember, multi-catch is only for exceptions in different inheritance hierarchies. To make sure this is clear, what do you think happens with the following code?

```
catch (IOException | Exception e)
```

That's right. It won't compile because `IOException` is a subclass of `Exception`. Which means it is redundant and the compiler won't accept it.

To summarize, we use multi-catch when we want to reuse an exception handler. We can list as many types as we want so long as none of them have a superclass/subclass relationship with each other.

Multi-catch and catch Parameter Assignment

There is one tricky thing with multi-catch. And we know the exam creators like tricky things!

The following LEGAL code demonstrates assigning a new value to the single catch parameter:

```
try {
    // access the database and write to a file
} catch (IOException e) {
    e = new IOException();
}
```



Don't assign a new value to the `catch` parameter. It is not good practice and creates confusing, hard-to-maintain code. But it is legal Java code to assign a new value to the `catch` block's parameter when there is only one type listed, and it will compile.

The following ILLEGAL code demonstrates trying to assign a value to the final multi-catch parameter:

```
try {
    // access the database and write to a file
} catch (SQLException | IOException e) {
    e = new IOException();
}
```

At least you get a clear compiler error if you try to do this. The compiler tells you:

```
The parameter e of a multi-catch block cannot be assigned
```

Since multi-catch uses multiple types, there isn't a clearly defined type for the variable that you can set. Java solves this by making the catch parameter `final` when that happens. And then the code doesn't compile because you can't assign a new value to a `final` variable.

Rethrowing Exceptions

Sometimes we want to do something with the thrown exceptions before we rethrow them:

```
public void couldThrowAnException() throws IOException, SQLException {}

public void rethrow() throws SQLException, IOException {
    try {
        couldThrowAnException();
    } catch (SQLException | IOException e) {
        log(e);
        throw e;
    }
}
```

This is a common pattern called “handle and declare.” We want to do something with the exception—log it. We also want to acknowledge we couldn’t completely handle it, so we declare it and let the caller deal with it. (As an aside, many programmers believe that logging an exception and rethrowing it is a bad practice, but you never know—you might see this kind of code on the exam.)

You may have noticed that `couldThrowAnException()` doesn’t actually throw an exception. The compiler doesn’t know this. The method signature is key to the compiler. It can’t assume that no exception gets thrown, as a subclass could override the method and throw an exception.

There is a bit of duplicate code here. We have the list of exception types thrown by the methods we call typed twice. Multi-catch was introduced to avoid having duplicate code, yet here we are with duplicate code.

Lucky for us, Java helps us out here as well with a feature added in Java 7. This example is a nicer way of writing the previous code:

```
1. public void rethrow() throws SQLException, IOException {  
2.     try {  
3.         couldThrowAnException();  
4.     } catch (Exception e) {    // watch out: this isn't really  
5.                         // catching all exception subclasses  
6.         log(e);  
7.         throw e;           // note: won't compile in Java 6  
8.     }  
9. }
```

Notice the multi-catch is gone and replaced with `catch(Exception e)`. It’s not bad practice here, though, because we aren’t really catching all exceptions. The compiler is treating `Exception` as “any exceptions that the called methods happen to throw.” (You’ll see this idea of code shorthand again with the diamond operator when you get to generics.)

This is very different from Java 6 code that catches `Exception`. In Java 6, we’d need the `rethrow()` method signature to be `throws Exception` in order to make this code compile.

In Java 7 and later, } catch (Exception e) { doesn't really catch ANY Exception subclass. The code may say that, but the compiler is translating for you. The compiler says, "Well, I know it can't be just any exception because the throws clause won't let me. I'll pretend the developer meant to only catch SQLException and IOException. After all, if any others show up, I'll just fail compilation on throw e;—just like I used to in Java 6." Tricky, isn't it?

At the risk of being too repetitive, remember that catch (Exception e) doesn't necessarily catch all Exception subclasses. In Java 7 and later, it means catch all Exception subclasses that would allow the method to compile.

Got that? Now why on earth would Oracle do this to us? It sounds more complicated than it used to be! Turns out they were trying to solve another problem at the same time they were changing this stuff. Suppose the API developer of couldThrowAnException() decided the method will never throw a SQLException and removes SQLException from the signature to reflect that.

Imagine we were using the Java 6 style of having one catch block per exception or even the multi-catch style of

```
} catch (SQLException | IOException e) {
```

Our code would stop compiling with an error like:

Unreachable catch block for SQLException

It is reasonable for code to stop compiling if we add exceptions to a method. But we don't want our code to break if a method's implementation gets LESS brittle. And that's the advantage of using

```
} catch (Exception e) {
```

Java infers what we mean here and doesn't say a peep when the API we are calling removes an exception.



Don't go changing your API signatures on a whim. Most code was written before Java 7 and will break if you change signatures. Your callers won't

thank you when their code suddenly fails compilation because they tried to use your new, shiny, “cleaner” API.

You've probably noticed by now that Oracle values backward compatibility and doesn't change the behavior or “compiler worthiness” of code from older versions of Java. That still stands. In Java 6, we can't write `catch (Exception e)` and merely throw specific exceptions. If we tried, the compiler would still complain:

```
Unhandled exception type Exception.
```

Backward compatibility only needs to work for code that compiles! It's OK for the compiler to get less strict over time.

To make sure you understand what is going on here, think about what happens in this example:

```
public class A extends Exception{}  
public class B extends Exception{}  
public void rain() throws A, B {}
```

[Table 3-2](#) summarizes handling changes to the exception-related parts of method signatures in Java 6, Java 7, and Java 8.

TABLE 3-2 Exceptions and Signatures

	What happens if rain() adds a new checked exception?	What happens if rain() removes a checked exception from the signature?
Java 6 style:	Add another catch block to handle the new exception.	Remove a catch block to avoid compiler error about unreachable code.
<pre>public void ahh() throws A, B { try { rain(); } catch (A e) { throw e; } catch (B e) { throw e; } }</pre>		
Java 7 and 8, with duplication:	Add another exception to the multi-catch block to handle the new exception.	Remove an expression from the multi-catch block to avoid compiler error about unreachable code.
<pre>public void ahh() throws A, B { try { rain(); } catch (A B e) { throw e; } }</pre>		
Java 7 and 8, without duplication:	Add another exception to the method signature to handle the new exception that can be thrown.	No code changes needed.
<pre>public void ahh() throws A, B { try { rain(); } catch (Exception e) { throw e; } }</pre>		

There is one more trick. If you assign a value to the catch parameter, the code no longer compiles:

```
public void rethrow() throws SQLException, IOException {
    try {
        couldThrowAnException();
    } catch (Exception e) {
        e = new IOException();
        throw e;
    }
}
```

As with multi-catch, you shouldn't be assigning a new value to the catch parameter in real life anyway. The difference between this and multi-catch is where the compiler error occurs. For multi-catch, the compiler error occurs on the line where we attempt to assign a new value to the parameter, whereas here, the compiler error occurs on the line where we `throw e`. It is different because code written prior to Java 7 still needs to compile. Because the multi-catch syntax is still relatively new, there is no legacy code to worry about.

AutoCloseable Resources with a try-with-resources Statement

The `finally` block is a good place for closing files and assorted other resources, but real-world clean-up code is easy to get wrong. And when correct, it is verbose. Let's look at the code to close our one resource when closing a file:

```
1: Reader reader = null;
2: try {
3:     // read from file
```

```
4: } catch(IOException e) {
5:   log(); throw e;
6: } finally {
7:   if (reader != null) {
8:     try {
9:       reader.close();
10:    } catch (IOException e) {
11:      // ignore exceptions on closing file
12:    }
13:  }
14: }
```

That's a lot of code just to close a single file! But it's all necessary. First, we need to check if the reader is null on line 7. It is possible the `try` block threw an exception before creating the reader, or while trying to create the reader if the file we are trying to read doesn't exist. It isn't until line 9 that we get to the one line in the whole `finally` block that does what we care about—closing the file. Lines 8 and 10 show a bit more housekeeping. We can get an `IOException` on attempting to close the file. While we could try to handle that exception, there isn't much we can do, thus making it common to just ignore the exception. This gives us nine lines of code (lines 6–14) just to close a file.

Developers typically write a helper class to close resources, or they use the open-source Apache Commons helper to get this mess down to three lines:

```
6: } finally {
7:   HelperClass.close(reader);
8: }
```

Which is still three lines too many.

Lucky for us, we have *Automatic Resource Management* using “`try-with-resources`” to get rid of even these three lines. The following code is equivalent

to the previous example:

```
1: try (Reader reader =  
2:       new BufferedReader(new FileReader(file))) { // note the new syntax  
3:       // read from file  
4:     } catch (IOException e) { log(); throw e; }
```

No `finally` left at all! We don't even mention closing the reader. Automatic Resource Management takes care of it for us. Let's take a look at what happens here. **We start out by declaring the reader inside the try declaration. Think of the parentheses as a for loop in which we declare a loop index variable that is scoped to just the loop. Here, the reader is scoped to just the try block. Not the catch block, just the try block.**

The actual `try` block does the same thing as before. It reads from the file. Or, at least, it comments that it would read from the file. The `catch` block also does the same thing as before. And just like in our traditional `try` statement, `catch` is optional.

Remember that a `try` must have `catch` or `finally`. Time to learn something new about that rule.

This is ILLEGAL code because it demonstrates a `try` without a `catch` or `finally`:

```
1: try {  
2:   // do stuff  
3: } // need a catch or finally here
```

The following LEGAL code demonstrates a `try-with-resources` with no `catch` or `finally`:

```
1: try (Reader reader =  
2:       new BufferedReader(new FileReader(file))) {  
3:       // do stuff  
4:     }
```

What's the difference? The legal example does have a `finally` block; you just don't see it. **The try-with-resources statement is logically calling a finally block to close the reader.** And just to make this even trickier, you can add your own `finally` block to `try-with-resources` as well. Both will get called. We'll take a look at how this works shortly.

Since the syntax is inspired from the `for` loop, we get to use a semicolon when declaring multiple resources in the `try`. For example:

```
try (MyResource mr = MyResource.createResource(); // first resource  
     MyThingy mt = mr.createThingy()) {           // second resource  
    // do stuff  
}
```

There is something new here. Our declaration calls methods. Remember that the `try-with-resources` is just Java code. It is restricted to only declarations. This means if you want to do anything more than one statement long, you'll need to put it into a method.

To review, [Table 3-3](#) lists the big differences that are new for `try-with-resources`.

TABLE 3-3 Comparing Traditional `try` Statement to `try-with-resources`

	<code>try-catch-finally</code>	<code>try-with-resources</code>
Resource declared	Before <code>try</code> keyword	In parentheses within <code>try</code> declaration
Resource initialized	In <code>try</code> block	In parentheses within <code>try</code> declaration
Resource closed	In <code>finally</code> block	Nowhere—happens automatically
Required keywords	<code>try</code> <code>catch</code> or <code>finally</code>	<code>try</code>

AutoCloseable and Closeable

Because Java is a statically typed language, it doesn't let you declare just any type in a try-with-resources statement. The following code will not compile:

```
try (String s = "hi") {}
```

You'll get a compiler error that looks something like:

```
The resource type String does not implement  
java.lang.AutoCloseable
```

AutoCloseable only has one method to implement. Let's take a look at the simplest code we can write using this interface:

```
public class MyResource implements AutoCloseable {  
    public void close() {  
        // take care of closing the resource  
    }  
}
```

There's also an interface called `java.io.Closeable`, which is similar to `AutoCloseable` but with some key differences. Why are there two similar interfaces, you may wonder? The `Closeable` interface was introduced in Java 5. When `try-with-resources` was invented in Java 7, the language designers wanted to change some things but needed backward compatibility with all existing code. So they created a superinterface with the rules they wanted.

One thing the language designers wanted to do was make the signature more generic. `Closeable` allows implementors to throw only an `IOException` or a `RuntimeException`. `AutoCloseable` allows any `Exception` at all to be thrown. Look at some examples:

```

// ok because AutoCloseable allows throwing any Exception
class A implements AutoCloseable { public void close() throws Exception{}}

// ok because subclasses or implementing methods can throw
// a subclass of Exception or none at all
class B implements AutoCloseable { public void close() {}}
class C implements AutoCloseable { public void close() throws IOException {}}

// ILLEGAL - Closeable only allows IOExceptions or subclasses
class D implements Closeable { public void close() throws Exception{}}

// ok because Closeable allows throwing IOException
class E implements Closeable { public void close() throws IOException{}}

```

In your code, Oracle recommends throwing the narrowest Exception subclass that will compile. However, they do limit Closeable to IOException, and you must use AutoCloseable for anything more.

The next difference is even trickier. What happens if we call the `close()` multiple times? It depends. For classes that implement Closeable, the implementation is required to be *idempotent*—which means you can call `close()` over and over again and nothing will happen the second time and beyond. It will not attempt to close the resource again and it will not blow up. For classes that implement AutoCloseable, there is no such guarantee.

If you look at the JavaDoc, you'll notice many classes implement both AutoCloseable and Closeable. These classes use the stricter signature rules and are idempotent. They still need to implement Closeable for backward compatibility, but added AutoCloseable for the new contract.

To review, [Table 3-4](#) shows the differences between AutoCloseable and Closeable. Remember the exam creators like to ask about “similar but not quite the same” things!

TABLE 3-4 Comparing AutoCloseable and Closeable

	AutoCloseable	Closeable
Extends	None	AutoCloseable
close method throws	Exception	IOException
Must be idempotent (can call more than once without side effects)	No, but encouraged	Yes

A Complex try-with-resources Example The following example is as complicated as try-with-resources gets:

```

1: class One implements AutoCloseable {
2:     public void close() {
3:         System.out.println("Close - One");
4:     }
5: class Two implements AutoCloseable {
6:     public void close() {
7:         System.out.println("Close - Two");
8:     }

```

```
9: class TryWithResources {  
10:    public static void main(String[] args) {  
11:        try (One one = new One(); Two two = new Two()) {  
12:            System.out.println("Try");  
13:            throw new RuntimeException();  
14:        } catch (Exception e) {  
15:            System.out.println("Catch");  
16:        } finally {  
17:            System.out.println("Finally");  
18:        } } }
```

Running the preceding code will print:

```
Try  
Close - Two  
Close - One  
Catch  
Finally
```

It's actually more logical than it looks at first glance. We first enter the `try` block on line 11, and Java creates our two resources. Line 12 prints `Try`. When we throw an exception on line 13, the first interesting thing happens. The `try` block “ends,” and Automatic Resource Management automatically cleans up the resources before moving on to the `catch` or `finally`. The resources get cleaned up, “backward” printing `Close - Two` and then `Close - One`. The `close()` method gets called in the reverse order in which resources are declared to allow for the fact that resources might depend on each other. Then we are back to the regular `try` block order, printing `Catch` and `Finally` on lines 15 and 17.

If you only remember two things from this example, remember that `try-with-resources` is part of the `try` block, and resources are cleaned up in the reverse order in which they were created.

Suppressed Exceptions

We’re almost done with exceptions. There’s only one more wrinkle to cover in

exception handling. Now that we have an extra step of closing resources in the try, it is possible for multiple exceptions to get thrown. Each close() method can throw an exception in addition to the try block itself.

```
1: public class Suppressed {
2:     public static void main(String[] args) {
3:         try (One one = new One()) {
4:             throw new Exception("Try");
5:         } catch (Exception e) {
6:             System.err.println(e.getMessage());
7:             for (Throwable t : e.getSuppressed()) {
8:                 System.err.println("suppressed:" + t);
9:             }
10:        }
11:    }
12:
13:    class One implements AutoCloseable {
14:        public void close() throws IOException {
15:            throw new IOException("Closing");
16:        }
17:    }
18: }
```

We know that after the exception in the try block gets thrown on line 4, the try-with-resources still calls close() on line 3 and the catch block on line 5 catches one of the exceptions. Running the code prints:

```
Try
suppressed:java.io.IOException: Closing
```

This tells us the exception we thought we were throwing still gets treated as most important. Java also adds any exceptions thrown by the close() methods to a suppressed array in that main exception. The catch block or caller can deal with any or all of these. If we remove line 4, the code just prints closing.

In other words, the exception thrown in `close()` doesn't always get suppressed. It becomes the main exception if there isn't already one existing. As one more example, think about what the following prints:

```
class Bad implements AutoCloseable {  
    String name;  
    Bad(String n) { name = n; }  
    public void close() throws IOException {  
        throw new IOException("Closing - " + name);  
    } }  
  
public class Suppressed {  
    public static void main(String[] args) {  
        try (Bad b1 = new Bad("1"); Bad b2 = new Bad("2")) {  
            // do stuff  
        } catch (Exception e) {  
            System.err.println(e.getMessage());  
            for (Throwable t : e.getSuppressed()) {  
                System.err.println("suppressed:" + t);  
            } } } }
```

The answer is:

```
Closing - 2  
  
suppressed:java.io.IOException: Closing - 1
```

Until `try-with-resources` calls `close()`, everything is going just dandy. When Automatic Resource Management calls `b2.close()`, we get our first exception. This becomes the main exception. Then, Automatic Resource Management calls

`b1.close()` and throws another exception. Since there was already an exception thrown, this second exception gets added as a second exception.

If the `catch` or `finally` block throws an exception, no suppressions happen. The last exception thrown gets sent to the caller rather than the one from the `try`—just like before `try-with-resources` was created.

CERTIFICATION SUMMARY

Assertions are a useful debugging tool. You learned how you can use them for testing by enabling them but keep them disabled when the application is deployed.

You learned how `assert` statements always include a boolean expression, and if the expression is `true`, the code continues on, but if the expression is `false`, an `AssertionError` is thrown. If you use the two-expression `assert` statement, then the second expression is evaluated, converted to a `String` representation, and inserted into the stack trace to give you a little more debugging info. Finally, you saw why assertions should not be used to enforce arguments to public methods and why `assert` expressions must not contain side effects!

Exception handling was enhanced in Java version 7, making exceptions easier to use. First you learned that you can specify multiple exception types to share a `catch` block using the new multi-catch syntax. The major benefit is in reducing code duplication by having multiple exception types share the same exception handler. The variable name is listed only once, even though multiple types are listed. You can't assign a new exception to that variable in the `catch` block. Then you saw the “handle and declare” pattern where the exception types in the multi-catch are listed in the method signature and Java translates “`catch Exception e`” into that exception type list.

Next, you learned about the `try-with-resources` syntax where Java will take care of calling `close()` for you. The objects are scoped to the `try` block. Java treats them as a `finally` block and closes these resources for you in the opposite order to which they were opened. If you have your own `finally` block, it is executed after `try-with-resources` closes the objects. You also learned the difference between `AutoCloseable` and `Closeable`. `Closeable` was introduced in Java 5, allowing only `IOException` (and `RuntimeException`) to be thrown. `AutoCloseable` was added in Java 7, allowing any type of `Exception`.



TWO-MINUTE DRILL

Here are some of the key points from the certification objectives in this chapter.

Test Invariants Using Assertions (OCP Objective 6.5)

- Assertions give you a way to test your assumptions during development and debugging.
- Assertions are typically enabled during testing but disabled during deployment.
- Assertions are disabled at runtime by default. To enable them, use a command-line flag: `-ea` or `-enableassertions`.
- Selectively disable assertions by using the `-da` or `-disableassertions` flag.
- If you enable or disable assertions using the flag without any arguments, you're enabling or disabling assertions in general. You can combine enabling and disabling switches to have assertions enabled for some classes and/or packages, but not others.
- You can enable and disable assertions on a class-by-class basis, using the following syntax:
`java -ea -da:MyClass TestClass`
- You can enable and disable assertions on a package-by-package basis, and any package you specify also includes any subpackages (packages further down the directory hierarchy).
- Do not use assertions to validate arguments to `public` methods.
- Do not use `assert` expressions that cause side effects. Assertions aren't guaranteed to always run, and you don't want behavior that changes depending on whether assertions are enabled.
- Do use assertions—even in `public` methods—to validate that a particular code block will never be reached. You can use `assert false;` for code that should never be reached so that an assertion error is thrown immediately if the `assert` statement is executed.

Use the `try` Statement with Multi-catch and `finally` Clauses (OCP Objective 6.2)

- If two catch blocks have the same exception handler code, you can merge them with multi-catch using `catch (Exception1 | Exception2 e)`.
- The types in a multi-catch list must not extend one another.
- When using multi-catch, the catch block parameter is final and cannot have a new value assigned in the catch block.
- If you catch a general exception as shorthand for specific subclass exceptions and rethrow the caught exception, you can still list the specific subclasses in the method signature. The compiler will treat it as if you had listed them out in the catch.

AutoCloseable Resources with a try-with-resources Statement (OCP Objectives 6.3 and 6.4)

- try-with-resources automatically calls `close()` on any resources declared in the `try` as `try(Resource r = new Foo())`.
- A `try` must have at least a `catch` or `finally` unless it is a `try-with-resources`. For `try-with-resources`, it can have neither, one, or both of the keywords.
- `AutoCloseable`'s `close()` method throws `Exception` and may be but is not required to be idempotent. `Closeable`'s `close()` throws `IOException` and must be idempotent.
- `try-with-resources` are closed in reverse order of creation and before going on to `catch` or `finally`.
- If more than one exception is thrown in a `try-with-resources` block, it gets added as a suppressed exception.
- The type used in a `try-with-resources` statement must implement `AutoCloseable`.

Q SELF TEST

The following questions will help you measure your understanding of the material presented in this chapter. Read all of the choices carefully, as there may be more than one correct answer. Choose all correct answers for each question. Stay focused.

- 1.** Which are true? (Choose all that apply.)
 - A.** It is appropriate to use assertions to validate arguments to methods marked `public`
 - B.** It is appropriate to catch and handle assertion errors
 - C.** It is NOT appropriate to use assertions to validate command-line arguments
 - D.** It is appropriate to use assertions to generate alerts when you reach code that should not be reachable
 - E.** It is NOT appropriate for assertions to change a program's state

2. Given:

```
3. public class Clumsy {  
4.     public static void main(String[] args) {  
5.         int j = 7;  
6.         assert(++j > 7);  
7.         assert(++j > 8) : "hi";  
8.         assert(j > 10) : j=12;  
9.         assert(j==12) : doStuff();  
10.        assert(j==12) : new Clumsy();  
11.    }  
12.    static void doStuff() { }  
13. }
```

Which are true? (Choose all that apply.)

- A.** Compilation succeeds
- B.** Compilation fails due to an error on line 6
- C.** Compilation fails due to an error on line 7
- D.** Compilation fails due to an error on line 8
- E.** Compilation fails due to an error on line 9

F. Compilation fails due to an error on line 10

3. Given:

```
class AllGoesWrong {  
    public static void main(String[] args) {  
        AllGoesWrong a = new AllGoesWrong();  
        try {  
            a.blowUp();  
            System.out.print("a");  
  
        } catch (IOException e | SQLException e) {  
            System.out.print("c");  
        } finally {  
            System.out.print("d");  
        }  
    }  
    void blowUp() throws IOException, SQLException {  
        throw new SQLException();  
    }  
}
```

What is the result?

A. ad

B. acd

C. cd

D. d

E. Compilation fails

F. An exception is thrown at runtime

4. Given:

```

class BadIO {
    public static void main(String[] args) {
        BadIO a = new BadIO();
        try {
            a.fileBlowUp();
            a.databaseBlowUp();
            System.out.println("a");
        } // insert code here
        System.out.println("b");
    } catch (Exception e) {
        System.out.println("c");
    }
}
void databaseBlowUp() throws SQLException {
    throw new SQLException();
}
void fileBlowUp() throws IOException {
    throw new IOException();
}

```

Which, inserted independently at // insert code here, will compile and produce the output b? (Choose all that apply.)

- A. catch(Exception e) {
- B. catch(FileNotFoundException e) {
- C. catch(IOException e) {
- D. catch(IOException | SQLException e) {
- E. catch(IOException e | SQLException e) {
- F. catch(SQLException e) {
- G. catch(SQLException | IOException e) {

```
H. catch(SQLException e | IOException e) {
```

5. Given:

```
class Train {  
    class RanOutOfTrack extends Exception { }  
    class AnotherTrainComing extends Exception { }  
  
    public static void main(String[] args) throws RanOutOfTrack,  
        AnotherTrainComing {  
        Train a = new Train();  
        try {  
            a.drive();  
            System.out.println("toot! toot!");  
        } // insert code here  
        System.out.println("error locomoting");  
        throw e;  
    }  
}  
void drive() throws RanOutOfTrack, AnotherTrainComing {  
    throw new RanOutOfTrack();  
} }
```

Which, inserted independently at // insert code here, will compile and produce the output error driving before throwing an exception? (Choose all that apply.)

- A. catch(AnotherTrainComing e) {
- B. catch(AnotherTrainComing | RanOutOfTrack e) {
- C. catch(AnotherTrainComing e | RanOutOfTrack e) {

- D. catch(Exception e) {
- E. catch(IllegalArgumentException e) {
- F. catch(RanOutOfTrack e) {
- G. None of the above—code fails to compile for another reason

6. Given:

```
class Conductor {
    static String s = "-";
    class Whistle implements AutoCloseable {
        public void toot() {      s += "t";      }
        public void close() {     s += "c";      }
    }
    public static void main(String[] args) {
        new Conductor().run();
        System.out.println(s);
    }
    public void run() {
        try (Whistle w = new Whistle()) {
            w.toot();
            s += "1";
            throw new Exception();
        } catch (Exception e) { s += "2";
        } finally { s += "3"; }
    }
}
```

What is the result?

- A. -t123t

- B. -t12c3
- C. -t123
- D. -t1c3
- E. -t1c23
- F. None of the above; main() throws an exception
- G. Compilation fails

7. Given:

```
public class MultipleResources {
    class Lamb implements AutoCloseable {
        public void close() throws Exception {
            System.out.print("l");
        }
    }
    class Goat implements AutoCloseable {
        public void close() throws Exception {
            System.out.print("g");
        }
    }
    public static void main(String[] args) throws Exception {
        new MultipleResources().run();
    }
    public void run() throws Exception {
        try (Lamb l = new Lamb();
             System.out.print("t");
             Goat g = new Goat();) {
            System.out.print("2");
        } finally {
            System.out.print("f");
        }
    }
}
```

What is the result?

- A. 2g1f
- B. 2lgf
- C. tglf
- D. t2lgf
- E. t2lgf

F. None of the above; `main()` throws an exception

G. Compilation fails

8. Given:

```
1: public class Animals {  
2:     class Lamb {  
3:         public void close() throws Exception { }  
4:     }  
5:     public static void main(String[] args) throws Exception {  
6:         new Animals().run();  
7:     }  
8:  
9:     public void run() throws Exception {  
10:        try (Lamb l = new Lamb();) {  
11:        }  
12:    }  
13: }
```

And the following possible changes:

C1. Replace line 2 with `class Lamb implements AutoCloseable {`

C2. Replace line 2 with `class Lamb implements Closeable {`

C3. Replace line 11 with `} finally {}`

What change(s) allow the code to compile? (Choose all that apply.)

A. Just C1 is sufficient

B. Just C2 is sufficient

C. Just C3 is sufficient

D. Both C1 and C3 are required

E. Both C2 and C3 are required

F. The code compiles without any changes

9. Given:

```
public class Animals {  
    class Lamb implements Closeable {  
        public void close() {  
            throw new RuntimeException("a");  
        } }  
    public static void main(String[] args) {  
        new Animals().run();  
    }  
    public void run() {  
        try (Lamb l = new Lamb();) {  
            throw new IOException();  
        } catch(Exception e) {  
            throw new RuntimeException("c");  
        } } }
```

Which exceptions will the code throw?

- A. IOException with suppressed RuntimeException a
- B. IOException with suppressed RuntimeException c
- C. RuntimeException a with no suppressed exception
- D. RuntimeException c with no suppressed exception
- E. RuntimeException a with suppressed RuntimeException c
- F. RuntimeException c with suppressed RuntimeException a
- G. Compilation fails

10. Given:

```
public class Animals {  
    class Lamb implements AutoCloseable {  
        public void close() {  
            throw new RuntimeException("a");  
        } }  
    public static void main(String[] args) throws IOException {  
        new Animals().run();  
    }  
    public void run() throws IOException {  
        try (Lamb l = new Lamb();) {  
            throw new IOException();  
        } catch(Exception e) {  
            throw e;  
        } } }
```

Which exceptions will the code throw?

- A. IOException with suppressed RuntimeException a
- B. IOException with suppressed Exception e
- C. RuntimeException a with no suppressed exception
- D. Exception e with no suppressed exception
- E. RuntimeException a with suppressed Exception e
- F. RuntimeException c with suppressed RuntimeException a
- G. Compilation fails

11. Given:

```
public class Concert {  
    static class PowerOutage extends Exception {}  
    static class Thunderstorm extends Exception {}  
    public static void main(String[] args) {  
        try {  
            new Concert().listen();  
            System.out.print("a");  
  
        } catch(PowerOutage | Thunderstorm e) {  
            e = new PowerOutage();  
            System.out.print("b");  
        } finally { System.out.print("c"); }  
    }  
    public void listen() throws PowerOutage, Thunderstorm{}  
}
```

What will this code print?

- A. a
- B. ab
- C. ac
- D. abc
- E. bc
- F. Compilation fails

A SELF TEST ANSWERS

1. **C, D, and E** are correct statements.
 - A** is incorrect. It is acceptable to use assertions to test the arguments of private methods. **B** is incorrect. While assertion errors can be caught,

Oracle discourages you from doing so. (OCP Objective 6.5)

2. **E** is correct. When an assert statement has two expressions, the second expression must return a value. The only two-expression assert statement that doesn't return a value is on line 9.
 - A, B, C, D, and F** are incorrect based on the above. (OCP Objective 6.5)
3. **E** is correct. `catch (IOException e | SQLException e)` doesn't compile. While multiple exception types can be specified in the multi-catch, only one variable name is allowed. The correct syntax is `catch (IOException | SQLException e)`. Other than this, the code is valid. Note that it is legal for `blowUp()` to have `IOException` in its signature even though that `Exception` can't be thrown.
 - A, B, C, D, and F** are incorrect based on the above. If the catch block's syntax error were corrected, the code would output `cd`. The multi-catch would catch the `SQLException` from `blowUp()` since it is one of the exception types listed. And, of course, the `finally` block runs at the end of the try/catch. (OCP Objective 6.2)
4. **C, D, and G** are correct. Since order doesn't matter, both **D** and **G** show correct use of the multi-catch block. And **C** catches the `IOException` from `fileBlowUp()` directly. Note that `databaseBlowUp()` is never called at runtime. However, if you remove the call, the compiler won't let you catch the `SQLException` since it would be impossible to be thrown.
 - A, B, E, H, and F** are incorrect. **A** is incorrect because it will not compile. Since there is already a catch block for `Exception`, adding another will make the compiler think there is unreachable code. **B** is incorrect because it will print `c` rather than `b`. Since `FileNotFoundException` is a subclass of `IOException`, the thrown `IOException` will not match the catch block for `FileNotFoundException`. **E** and **H** are incorrect because they are invalid syntax for multi-catch. The catch parameter `e` can only appear once. **F** is incorrect because it will print `c` rather than `b`. Since the `IOException` thrown by `fileBlowUp()` is never caught, the thrown exception will match the catch block for `Exception`. (OCP Objective 6.2)
5. **B, D, and F** are correct. **B** uses multi-catch to identify both exceptions `drive()` may throw. **D** still compiles since it uses the new enhanced exception typing to recognize that `Exception` may only refer to `AnotherTrainComing` and `RanOutOfTrack`. **F** is the simple case that catches

a single exception. Since `main` declares that it can throw `AnotherTrainComing`, the catch block doesn't need to handle it.

A, C, E, and G are incorrect. **A** and **E** are incorrect because the catch block will not handle `RanOutOfTrack` when `drive()` throws it. The `main` method will still throw the exception, but the `println()` will not run. **C** is incorrect because it is invalid syntax for multi-catch. The catch parameter `e` can only appear once. **G** is incorrect because of the above. (OCP Objective 6.2)

6. **E** is correct. After the exception is thrown, Automatic Resource Management calls `close()` before completing the `try` block. From that point, `catch` and `finally` execute in the normal order.

F is incorrect because the catch block catches the exception and does not rethrow it.
A, B, C, D, and G are incorrect because of the above. (OCP Objective 6.3)
7. **G** is correct. `System.out.println` cannot be in the declaration clause of a try-with-resources block because it does not declare a variable. If the `println` was removed, the answer would be **A** because resources are closed in the opposite order in which they are created.

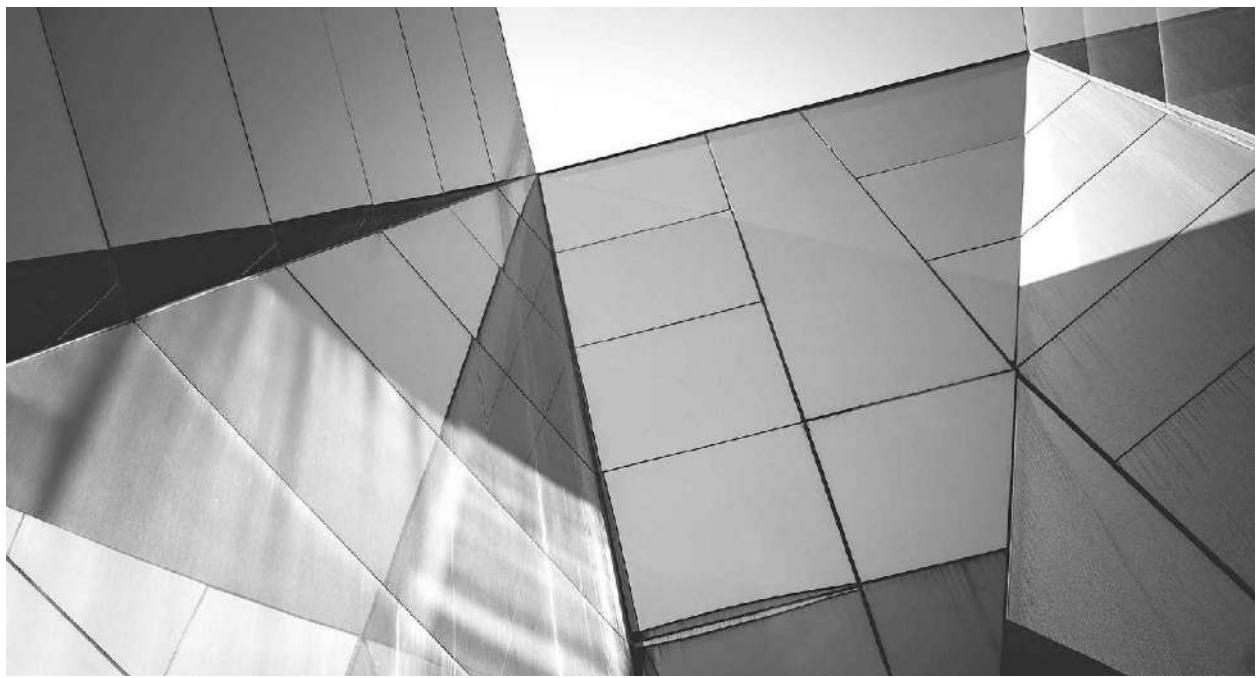
A, B, C, D, E, and F are incorrect because of the above. (OCP Objective 6.3)
8. **A** is correct. If the code is left with no changes, it will not compile because `try-with-resources` requires `Lamb` to implement `AutoCloseable` or a subinterface. If `C2` is implemented, the code will not compile because `close()` throws `Exception` instead of `IOException`. Unlike the traditional `try`, `try-with-resources` does not require `catch` or `finally` to be present.

B, C, D, E, and F are incorrect because of the above. (OCP Objective 6.3)
9. **D** is correct. While the exception caught by the catch block matches choice **A**, it is ignored by the catch block. The catch block just throws `RuntimeException c` without any suppressed exceptions.

A, B, C, E, F, and G are incorrect because of the above. (OCP Objective 6.3)
10. **A** is correct. After the `try` block throws an `IOException`, Automatic Resource Management calls `close()` to clean up the resources. Since an

exception was already thrown in the try block, `RuntimeException` a gets added to it as a suppressed exception. The catch block merely rethrows the caught exception. The code does compile, even though the catch block catches an `Exception` and the method merely throws an `IOException`. In Java 7, the compiler is able to pick up on this.

- B, C, D, E, F, and G** are incorrect because of the above. (OCP Objective 6.3)
- 11. **F** is correct. The exception variable in a catch block may not be reassigned when using multi-catch. It CAN be reassigned if we are only catching one exception.
 - C** would have been correct if `e = new PowerOutage();` were removed. **A, B, D, and E** are incorrect because of the above. (OCP Objectives 6.2 and 6.4)



4

Dates, Times, Locales, and Resource Bundles

CERTIFICATION OBJECTIVES

- Create and Manage Date-Based and Time-Based Events Including a Combination of Date and Time into a Single Object Using LocalDate, LocalTime, LocalDateTime, Instant, Period, and Duration
- Work with Dates and Times Across Timezones and Manage Changes Resulting from Daylight Savings Including Format Date and Times Values
- Define and Create and Manage Date-Based and Time-Based Events Using Instant, Period, Duration, and TemporalUnit
- Read and Set the Locale by Using the Locale Object
- Create and Read a Properties File
- Build a Resource Bundle for Each Locale and Load a Resource Bundle in an Application



Two-Minute Drill

Q&A Self Test

This chapter focuses on the exam objectives related to working with date- and time-related events, formatting dates and times, and using resource bundles for localization and internationalization tasks. Many of these topics could fill an entire book. Fortunately, you won't have to become a guru to do well on the exam. The intention of the exam team was to include just the basic aspects of these technologies, and in this chapter, we cover *more* than you'll need to get through the related objectives on the exam.

CERTIFICATION OBJECTIVE

Dates, Times, and Locales (OCP Objectives 7.1, 7.2, 7.3, and 12.1)

7.1 *Create and manage date-based and time-based events including a combination of date and time into a single object using LocalDate, LocalTime, LocalDateTime, Instant, Period, and Duration.*

7.2 *Work with dates and times across timezones and manage changes resulting from daylight savings including Format date and times values.*

7.3 *Define, create and manage date-based and time-based events using Instant, Period, Duration, and TemporalUnit.*

12. 1 *Read and set the locale by using the Locale object.*

The Java API provides an extensive (perhaps a little *too* extensive) set of classes to help you work with dates and times. The exam will test your knowledge of the basic classes and methods you'll use to work with dates and such. When you've finished this section, you should have a solid foundation in tasks such as creating date and time objects, and creating, manipulating, and formatting dates and times, and doing all of this for locations around the globe. In fact, a large part of why this section was added to the exam was to test whether you can do some basic internationalization (often shortened to "i18n").

Note: In this section, we'll introduce the `Locale` class. Later in the chapter, we'll be discussing resource bundles, and you'll learn more about `Locale` then.

Working with Dates and Times

If you want to work with dates and times from around the world (and who doesn't?), you'll need to be familiar with several classes from the `java.time` package. `java.time` is new in Java 8, so if you're looking for the old familiar `java.util.Date` and `java.util.Calendar`, you won't find them on the exam, although we'll briefly mention them here for comparison purposes.

Here's an overview of how the classes in `java.time` are organized:

- **Local dates and times** These dates and times are local to your time zone and so don't have time-zone information associated with them. These are represented by classes `java.time.LocalDate`, `java.time.LocalTime`,

and `java.time.LocalDateTime`.

- **Zoned dates and times** These dates and times include time-zone information. They are represented by classes `java.time.ZonedDateTime` and `java.time.OffsetDateTime`.
- **Formatters for dates and times** With `java.time.format.DateTimeFormatter`, you can parse and print dates and times with patterns and in a variety of styles.
- **Adjustments to dates and times** With `java.time.temporal.TemporalAdjusters` and `java.time.temporal.ChronoUnit`, you can adjust and manipulate dates and times by handy increments.
- **Periods, Durations, and Instants** `java.time.Periods` and `java.time.Durations` represent an amount of time, periods for days or longer and durations for shorter periods like minutes or seconds. `java.time.Instants` represent a specific instant in time, so you can, say, compute the number of minutes between two instants.

If you’re used to working with `java.util.Date`, `java.util.Calendar`, and `java.text.DateFormat`, you’re going to have to forget most of what you’ve learned and start over (although the concepts are similar, so you have a bit of a head start). All those classes are still around (although some methods are marked as deprecated), but they are considered “old,” and the classes in `java.time` are designed to replace them completely. Hopefully this design will stick!

The Date Class

The API design of the `java.util.Date` class didn’t do a good job of handling internationalization and localization situations, so it’s been largely replaced by the classes in `java.time`, like `java.time.LocalDateTime` and `java.time.ZonedDateTime`. You might find `Date` used in legacy code, but, for the most part, it’s time to leave it behind.

The Calendar Class

Likewise, the `java.util.Calendar` class has largely been replaced with the classes in `java.time`. You’ll still find plenty of legacy `Calendar` code around, but if you’ve worked with `Calendar` before, you’ll likely find the `java.time` classes easier and less convoluted to work with.

With that, we’ll dive right into the `java.time` classes.

The `java.time.*` Classes for Dates and Times

To make learning about dates and times more fun, let's imagine you are a solar eclipse hunter. You love chasing solar eclipses around the country and around the world. This example will be U.S.-centric, but you can easily apply all these ideas to dates and times in other countries too (and doing so is great practice for the exam).

Let's begin by figuring out the current date and time where you are, right now:

```
import java.time.*;
public class Eclipse {
    public static void main(String[] args) {
        LocalDate nowDate = LocalDate.now();
        LocalTime nowTime = LocalTime.now();
        LocalDateTime nowDateTime = LocalDateTime.of(nowDate, nowTime);
        System.out.println("It's currently " + nowDateTime + " where I am");
    }
}
```

Here we're using the static method `LocalDate.now()` to get the current date. It has no time zone, so think of it as a description of “the date,” whatever that date is for you today, wherever you are, as you try this code. Similarly, we're using the static method `LocalTime.now()` to get the current time, so that will be whatever the time is for you right now, wherever you are, as you try this code. We then use the date and time of “now” to create a `LocalDateTime` object using the `of()` static method and then display it.

When we run this code we see

```
It's currently 2017-10-11T14:51:19.982 where I am
```

The string `2017-10-11T14:51:19.982` represents the date, October 11, 2017, and time, 14:51:19.982, which is 2:51 PM and 19 seconds and 982 milliseconds. Notice that Java displays a “T” between the date and the time when converting

the `LocalDateTime` to a string.

Of course, you'll see a completely different date and time because you're running this code in your own date and time, wherever and whenever that is.

We could also write:

```
LocalDateTime nowDateTime = LocalDateTime.now();
```

to get the current date and time of now as a `LocalDateTime`.

What if you want to set a specific date and time rather than "now"?

Let's say you went to Madras, Oregon, in the United States to see the solar eclipse on August 21, 2017. Here are a couple of ways you can set that specific date:

```
// The day of the eclipse in Madras, OR
LocalDate eclipseDate1 = LocalDate.of(2017, 8, 21);
LocalDate eclipseDate2 = LocalDate.parse("2017-08-21");
System.out.println("Eclipse date: " + eclipseDate1 + ", " +
eclipseDate2);
```

We're creating the same date in two slightly different ways: first, by specifying a year, month, and day as arguments to the `LocalDate.of()` static method; and second, by using the `LocalDate.parse()` method to parse a string that matches the date. `LocalDate` represents a date in the ISO-8601 calendar system (which specifies a format of YYYY-MM-DD). You don't need to know that, except to know the kinds of strings that the `parse()` method can parse correctly. (For a full list of the formats of the dates and times that can be parsed and represented, check out the documentation for

`java.time.format.DateTimeFormatter`, which we'll talk more about in a little bit. See

<https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html>

When we run this code, we see the default display (using that ISO format) for both dates:

```
Eclipse date: 2017-08-21, 2017-08-21
```

An eclipse happens at a specific time of day. The eclipse begins awhile before

totality (when the moon almost completely obscures the sun), so let's create `LocalTime` objects to represent the time the eclipse begins and the time of totality:

```
// Eclipse begins in Madras, OR
LocalTime begins = LocalTime.of(9, 6, 43);           // 9:06:43
// Totality starts in Madras, OR
LocalTime totality = LocalTime.parse("10:19:36");    // 10:19:36
System.out.println("Eclipse begins at " + begins +
    " and totality is at " + totality);
```

As with `LocalDate`, a `LocalTime` has no time zone associated with it, so in this case, we need to know that we're creating times that are valid in Madras, OR (on U.S. Pacific time). For these times, we again use the static `of()` and `parse()` methods to demonstrate two different ways to create `LocalTime` objects.

When we print the times, we see:

```
Eclipse begins at 09:06:43 and totality is at 10:19:36
```

If you want to be precise about the format of the date and time you're parsing into a `LocalDate` or `LocalTime` or `LocalDateTime`, you can use `DateTimeFormatter`. You can either use one of several predefined formats or create your own format for parsing using a sequence of letters and symbols. In the following example, we create a date and time in a string and then tell the `LocalDateTime` how to parse that using a `DateTimeFormatter`:

```
String eclipseDateTime = "2017-08-21 10:19";
DateTimeFormatter formatter =
    DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm");
LocalDateTime eclipseDay =
    LocalDateTime.parse(eclipseDateTime, formatter); // use formatter
System.out.println("Eclipse day: " + eclipseDay);
```

This creates a `LocalDateTime` object from the string, formatted using the pattern we specified, and when we print out the `LocalDateTime`, we see the correct date and time, printed in the standard ISO format we saw before:

```
Eclipse day: 2017-08-21T10:19
```

Of course, you can also use `DateTimeFormatter` to change the format of the output (again using letters and symbols):

```
System.out.println("Eclipse day, formatted: " +  
eclipseDay.format(DateTimeFormatter.ofPattern("dd, mm, yy hh,  
mm")));
```

This code results in the output:

```
Eclipse day, formatted: 21, 19, 17 10, 19
```

We'll come back to `DateTimeFormatter` in a bit.

`LocalDateTime` has several methods that make it easy to add to and subtract from dates and times. For instance, let's say your Mom calls from Nashville, TN (on U.S. Central time) and asks, "What time will it be here when you're seeing the eclipse there?" You know that Central time is two hours ahead of Pacific time, so to answer that question you can write:

```
System.out.println("Mom time: " + eclipseDay.plusHours(2));
```

which reveals that it will be 12:19 PM where she is in Tennessee when you're watching the eclipse at 10:19 AM in Oregon:

```
Mom time: 2017-08-21T12:19:36
```

Then she asks, "When are you coming home?" You can tell her, "In three days" by writing the following code:

```
System.out.println("Going home: " + eclipseDay.plusDays(3));
```

which means you'll be going home on August 24:

```
Going home: 2017-08-24T10:19:36
```

Of course, there are loads of other handy methods too, like `getDayOfWeek()` to find out what day of the week the eclipse occurs:

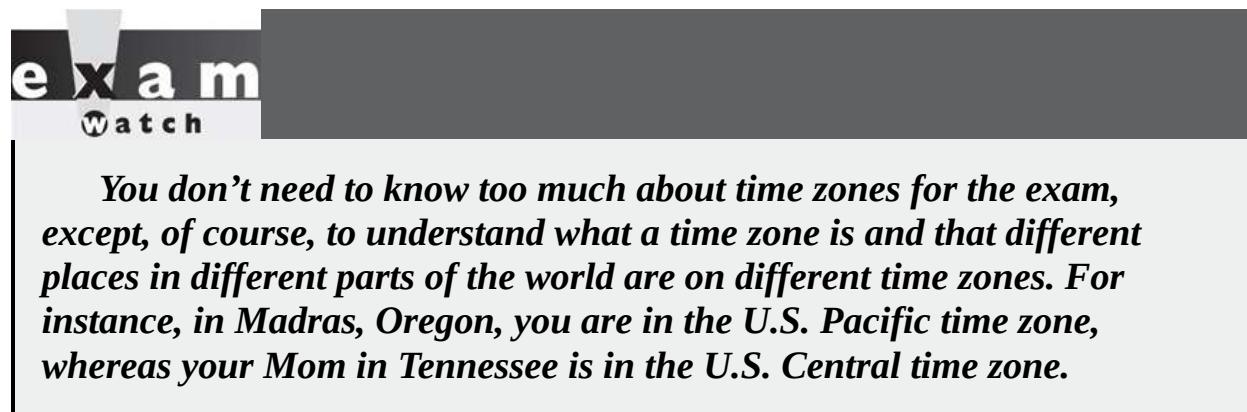
```
System.out.println("What day of the week is eclipse? " +  
eclipseDay.getDayOfWeek());
```

which lets you know the eclipse was on a Monday:

```
What day of the week is eclipse? MONDAY
```

Zoned Dates and Times

Local dates and times are great when you don't need to worry about the time zone, but sometimes we need to share dates and times with people in other time zones, so knowing which time zone you're in or they're in becomes important.



All time zones are based on *Greenwich Mean Time (GMT)*, the time in Greenwich, England. GMT is a time zone. The name of the time standard that uses GMT as the basis for all other time zones is *Coordinated Universal Time (UTC)*.

Your time zone will either be ahead of or behind GMT. For instance, in Madras, OR, for the eclipse, you are GMT-7, meaning you are seven hours behind GMT. That's for summer; in winter, you'll be GMT-8, or eight hours behind GMT because the United States has daylight savings time in summer and standard time in winter. Yes, zoned dates and times can get complicated fast, but rest assured, the exam is not about your depth of understanding of time zones. As long as you know the basics and you can create and use zoned dates and times, you'll be fine.

Let's create a zoned date and time for the date and time of the eclipse:

```
ZonedDateTime zTotalityDateTime =  
    ZonedDateTime.of(eclipseDay, ZoneId.of("US/Pacific"));  
System.out.println("Date and time totality begins with time zone: "  
    + zTotalityDateTime);
```

Looking at the output, we see:

```
Date and time totality begins with time zone:  
2017-08-21T10:19:36-07:00[US/Pacific]
```

A `ZonedDateTime` is a `LocalDateTime` plus a time zone, which is represented as a `ZoneId`. In this example, the `ZoneId` is “US/Pacific,” which happens to be GMT-7 (which you may also see written as UTC-7). You can use either “US/Pacific” or “GMT-7” as the `ZoneId`.

You might be asking: How did you know the name of the `ZoneId`? Good question. The names of the zones are not listed in the documentation page for `ZoneId`, so one good way to find them is to write some code to display them:

```
Set<String> zoneIds = ZoneId.getAvailableZoneIds();  
List<String> zoneList = new ArrayList<String>(zoneIds);  
Collections.sort(zoneList);  
for (String zoneId : zoneList) {  
    if (zoneId.contains("US")) {  
        System.out.println(zoneId);  
    }  
}
```

With this code, we’re displaying only the U.S. `zoneIds`. If you, say, want to display the `zoneIds` for Great Britain, then use “GB” in place of “US”. The list includes some `zoneIds` by country, some by city, and some by other names.

Let’s get back to daylight savings time. Recall that the U.S. Pacific time zone is either GMT-7 (in winter, standard time) or GMT-8 (in summer, daylight savings time). That means when you’re creating a `ZonedDateTime` for Madras,

Oregon (or any other place that uses daylight savings), you’re going to need to know if you’re currently in daylight savings time.

You can find out if you’re in daylight savings time by using `ZoneRules`. All you need to know about `ZoneRules` is that the class captures the current rules about daylight savings in various parts of the world. Unfortunately, this tends to change (as politicians tend to change their minds—what a shocker!) so the rules are, as the documentation states, only as accurate as the information provided. Let’s assume the rules are up to date (as you should for the exam) and write some code to find out if the “US/Pacific” time zone is currently in daylight savings time:

```
ZoneId pacific = ZoneId.of("US/Pacific");
// pacific.getRules() returns a ZoneRules object that has all the rules
// about time zones, including daylight savings and standard time.
System.out.println("Is Daylight Savings in effect at time of totality: " +
    pacific.getRules().isDaylightSavings(zTotalityDateTime.toInstant()));
```

In this code, we first get the `ZoneId` for the “US/Pacific” time zone. We can then use that `ZoneId` to get the `ZoneRules` with the `getRules()` method. A `ZoneRules` object has a method `isDaylightSavings()`, which takes an `Instant` and determines whether that `Instant` is currently in daylight savings. We’ll discuss `Instant`s in more detail shortly; for now, just know that we can convert the `ZonedDateTime` representing the date and time of the eclipse into an `Instant` with the `toInstant()` method. The result is

```
Is Daylight Savings in effect at time of totality: true
```

The result is `true` because the date of the eclipse, `zTotalityDateTime` (that is, August 21, 2017, in the U.S. Pacific zone), was in daylight savings time (summer time).

Date and Time Adjustments

Once you’ve created a `LocalDateTime` or `ZonedDateTime`, you can’t modify it. The documentation describes datetime objects as “immutable.” However, you can create a new datetime object from an existing datetime object, and `java.time.*` provides plenty of adjusters that make it easy to do so. In other

words, rather than modifying an existing datetime, you just make a new datetime from the existing one.

Let's say you want to find the date of the Thursday following the eclipse. We can take the `ZonedDateTime` for the eclipse we made above, `zTotalityDateTime`, and create a new `ZonedDateTime` from it to represent "the following Thursday" like this:

```
ZonedDateTime followingThursdayDateTime =  
    zTotalityDateTime.with(TemporalAdjusters.next( // adjust date time  
        DayOfWeek.THURSDAY)); // to next Thursday  
System.out.println("Thursday following the totality: " +  
    followingThursdayDateTime);
```

The output is:

```
Thursday following the totality: 2017-08-24T10:19:36-  
07:00[US/Pacific]
```

We can see that the Thursday following the eclipse (which, remember, was on Monday, August 21, 2017) is August 24.

The class `TemporalAdjusters` has a whole slew of handy methods to make a `TemporalAdjuster` for a variety of scenarios, such as `firstDayOfNextYear()`, `lastDayOfMonth()`, and more.

You've already seen how you can add days and hours from a datetime; many other adjustments, including `plusMinutes()`, `plusYears()`, `minusWeeks()`, `minusSeconds()`, and so on, are available as methods in `LocalDateTime` and `ZonedDateTime`. You'll also find a variety of adjustments like `withHour()` and `withYear()`, which you can use to create a new datetime object from an existing one, but with a different hour or year. Each of these methods creates a new adjusted datetime from an existing one.

`ZonedDateTime`s are subject to `ZoneRules` when you adjust them. So, if you, say, add a month to an existing `ZonedDateTime`, the `ZoneRules` will be used to determine if the new `ZonedDateTime` is GMT-7 or GMT-8, for instance, (depending on daylight savings time). If you want to create a datetime with a zone offset from GMT that does not use the `ZoneRules`, then you can use an `OffsetDateTime`. An `OffsetDateTime` is a fixed datetime and offset that doesn't

change even if the ZoneRules change.

Periods, Durations, and Instants

So far, we've looked at how to create specific dates and times. The `java.time.*` package also includes ways to represent a period of time, whether that's a period of days, months, or years (a `Period`), a short period of minutes or hours (a `Duration`), or an instant in time (an `Instant`).

Periods You're looking forward to the next eclipse on April 8, 2024, which you're going to watch in Austin, Texas, and you want to set a reminder for yourself one month in advance, so you don't forget. You could just say, well, I'll remind myself on March 8, 2024, but what fun would that be? Let's compute one month before the eclipse in code:

```
// Totality begins in Austin, TX in 2024 at 1:35pm and 56 seconds;
// Specify year, month, dayOfMonth, hour, minute, second, nano, zone
ZonedDateTime totalityAustin =
    ZonedDateTime.of(2024, 4, 8, 13, 35, 56, 0, ZoneId.of("US/Central"));
System.out.println("Next total eclipse in the US, date/time in Austin, TX: " +
    totalityAustin);
```

To create the reminder, we first create a `ZonedDateTime` for when totality begins in Austin. We use the `ZonedDateTime.of()` static method to create the datetime with the arguments year, month, day, hours, minutes, seconds, nanoseconds, and zone id. We have no idea what the nanoseconds are for the totality beginning, so we just put 0. Notice we used 13 to specify 1 PM. Austin is in the U.S. Central time zone, so we get the `ZoneId` using that name (which we got earlier when we displayed all the U.S. time-zone names). When we print this `ZonedDateTime`, we see

```
Next total eclipse in the US, date/time in Austin, TX:
2024-04-08T13:35:56-05:00[US/Central]
```

Now let's create the reminder for one month before this date and time by creating a `Period` that represents one month and subtract it from the date and

time for the eclipse:

```
// Reminder for a month before  
Period period = Period.ofMonths(1);  
System.out.println("Period is " + period);  
ZonedDateTime reminder = totalityAustin.minus(period);  
System.out.println("DateTime of 1 month reminder: " + reminder);
```

Here's the output:

```
Period is P1M  
DateTime of 1 month reminder: 2024-03-08T13:35:56-  
06:00[US/Central]
```

Notice how the period is displayed, with “P” meaning period and “1M” meaning month.

While we're here, let's see how to create a `LocalDateTime` from the `ZonedDateTime` for people who are in Austin:

```
System.out.println("Local DateTime (Austin, TX) of reminder: " +  
    reminder.toLocalDateTime());
```

Notice the difference in the `LocalDateTime`—there's no time zone:

```
Local DateTime (Austin, TX) of reminder: 2024-03-08T13:35:56
```

And finally, let's figure out when we'll see the reminder in Madras, Oregon:

```
System.out.println("Zoned DateTime (Madras, OR) of reminder: " +  
    reminder.withZoneSameInstant(ZoneId.of("US/Pacific")));
```

We'll see the reminder at 11 AM, two hours earlier than the reminder in Austin, because Madras is two hours behind.

```
Zoned DateTime (Madras, OR) of reminder: 2024-03-08T11:35:56-  
08:00[US/Pacific]
```

One more thing to notice about this code. The eclipse is happening in April 2024. April happens to be in summer time, or daylight savings time, so notice that during daylight savings, U.S. Central time is five hours behind GMT:

```
Next total eclipse in the US, date/time in Austin, TX:  
2024-04-08T13:35:56-05:00[US/Central]
```

When we subtracted the one-month period from this date and time to get the date and time for our reminder, we compute that the reminder is on March 8, which is in winter time, or standard time:

```
DateTime of 1 month reminder: 2024-03-08T13:35:56-  
06:00[US/Central]
```

So, on March 8, 2024, Austin will be six hours behind GMT. Nice for us that Java correctly computed the time using `ZoneRules` behind the scenes.

Durations How many minutes from the time the eclipse begins to the time totality begins? We can compute the time in a couple of ways; we're going to do it using `ChronoUnit` and `Duration`. `ChronoUnit` is an enum in `java.time.temporal` that provides a set of predefined units of time periods. For instance, `ChronoUnit.MINUTES` represents the concept of a minute. `ChronoUnit` also supplies a method `between()` that we can use to compute a `ChronoUnit` time period between two times. Once we have the number of minutes between two times, we can use that to create a `Duration`. `Durations` have all kinds of handy methods for computing things, like adding and subtracting hours and minutes and seconds, or converting a `Duration` into a number of seconds or milliseconds, and so on. Think of `ChronoUnit` as a unit of time and `Duration` as specifying a period of time (like a `Period`, only for period lengths less than a day).

First, let's create two `LocalTimes` to represent the start of the eclipse (when the moon first starts to cross the sun) and the time of totality (when the moon completely obscures the sun):

```
// Eclipse begins in Austin, TX
LocalTime begins = LocalTime.of(12, 17, 32);      // 12:17:32
// Totality in Austin, TX
LocalTime totality = LocalTime.of(13, 35, 56);    // 13:35:56
System.out.println("Eclipse begins at " + begins +
    " and totality is at " + totality);
```

Notice we're just using `LocalTime` here, not `ZonedDateTime`, so we don't have to specify a time zone. The output looks like this:

```
Eclipse begins at 12:17:32 and totality is at 13:35:56
```

Now, let's use a `ChronoUnit` to compute the number of minutes between `begins` and `totality`:

```
// How many minutes between when the eclipse begins and totality?
long betweenMins = ChronoUnit.MINUTES.between(begins, totality);
System.out.println("Minutes between begin and totality: " + betweenMins);
```

The minutes returned by the `between()` method is a `long`. When we look at the output, we see we have 78 minutes between the beginning of the eclipse and the beginning of totality. Notice that we lost the number of seconds in this computation because we asked for the number of minutes between the two times.

Let's turn this into a `Duration`. As you might expect, we can turn the number of minutes into a `Duration` using `Duration.ofMinutes()`:

```
Duration betweenDuration = Duration.ofMinutes(betweenMins);
System.out.println("Duration: " + betweenDuration);
```

Looking at the output we see:

```
Duration: PT1H18M
```

PT means “period of time,” meaning Duration (rather than Period), and then 1H18M means “1 hour and 18 minutes” corresponding to our 78 minutes.

Just to double-check ourselves, let’s take the begin LocalTime we created before and add back our Duration using the LocalTime.plus() method. We could also do this with our betweenMins value, using LocalTime.plusMinutes(). The plus() method takes a TemporalAmount (like a Duration or a Period), whereas plusMinutes() takes minutes as a long. Either way will work.

```
LocalTime totalityBegins = begins.plus(betweenDuration);
System.out.println("Totality begins, computed: " + totalityBegins);
```

The result is

```
Totality begins, computed: 13:35:32
```

This time is slightly different than our original begins time of 13:35:56 because we lost the seconds when we created the Duration from the minutes between begins and totality.

Instants An Instant represents an instant in time. Makes sense, right? But how is it different from a DateTime? If you’re used to timestamps, then you’ll probably recognize an Instant as the number of seconds (and nanoseconds) since January 1, 1970—the standard Java epoch. Instants can’t be represented as just one long, like you might be used to, because an Instant includes nanoseconds, so the seconds plus the nanoseconds is too big for a long. However, once you’ve created an Instant, you can always get the number of seconds as a long value from the Instant.

ZonedDateTime can be converted to Instants using the toInstant() method:

```
ZonedDateTime totalityAustin =
    ZonedDateTime.of(2024, 4, 13, 13, 56, 0, ZoneId.of("US/Central"));
Instant totalityInstant = totalityAustin.toInstant();
System.out.println("Austin's eclipse instant is: " + totalityInstant);
```

Looking at the output we see

```
Austin's eclipse instant is: 2024-04-08T18:35:56Z
```

Even though we created a `ZonedDateTime` for Austin at 1:35 PM, in the US/Central time zone, the instant displays as 6:35 PM and shows a Z at the end. That datetime represents 6:35 PM GMT. The Z is how you know the time displayed is for the GMT zone, rather than the U.S. Central zone. This format is the ISO_INSTANT format of displaying a datetime.

Note that if you want to call the `toInstant()` method on a `LocalDateTime`, you'll need to supply a `ZoneOffset` as an argument. To create a unique instant in time that works globally, a time zone is required when the `Instant` is created. If we don't include a time zone, then *your* instant and *our* instant may mean two different things.

Let's once again compute the number of minutes between two times using `ChronoUnit.MINUTES`. This time we'll compute the minutes between now and the Austin eclipse as represented by `Instants` and then use that to create a `Duration`. We'll use the `totalityInstant` we created above for the instant of the totality, and we'll use the `Instant.now()` method to create an instant representing right now:

```
Instant nowInstant = Instant.now(); // represents now
Instant totalityInstant = totalityAustin.toInstant(); // same as above
long minsBetween =
    ChronoUnit.MINUTES.between(nowInstant, totalityInstant);
Duration durationBetweenInstants = Duration.ofMinutes(minsBetween);
System.out.println("Minutes between " + minsBetween +
    ", is duration " + durationBetweenInstants);
```

The output is

```
Minutes between 3405250, is duration PT56754H10M
```

As you can see (reading the `Duration`), between now and the next eclipse in Austin, we have only 56,754 hours and 10 minutes to wait. That eclipse will be

here in no time.

Lastly, if you want to get the number of seconds since January 1, 1970, from an `Instant`, use the method `getEpochSecond()`:

```
Instant now = Instant.now();
System.out.println("Seconds since epoch: " + now.getEpochSecond());
```

The number of seconds is

```
Seconds since epoch: 1508286832
```

A Few Other Handy Methods and Examples

Let's add another reminder before the next eclipse, say, for three days before the eclipse, and then let's figure out what day of the week this reminder will occur:

```
// Another reminder 3 days before
System.out.println("DateTime of 3 day reminder: " +
    totalityAustin.minus(Period.ofDays(3)));
// What day of the week is that?
System.out.println("Day of week for 3 day reminder: " +
    totalityAustin.minus(Period.ofDays(3)).getDayOfWeek());
```

We see that the three-day reminder is on April 5, which is a Friday:

```
DateTime of 3 day reminder: 2024-04-05T13:35:56-05:00[US/Central]
Day of week for 3 day reminder: FRIDAY
```

And we really should call our sister in Paris a couple of hours after the next eclipse to tell her how it was:

```
ZonedDateTime localParis =  
    totalityAustin.withZoneSameInstant(ZoneId.of("Europe/Paris"));  
System.out.println("Eclipse happens at " + localParis + " Paris time");  
System.out.println("Phone sister at 2 hours after totality: " +  
    totalityAustin.plusHours(2) + ", " +  
    localParis.plusHours(2) + " Paris time");
```

From the output, we can see that the eclipse happens at 8:35 Paris time, and when we call our sister two hours after the eclipse, it will be 3:35 PM Austin time and 10:35 PM Paris time:

```
Eclipse happens at 2024-04-08T20:35:56+02:00[Europe/Paris] Paris time  
Phone sister at 2 hours after totality:  
2024-04-08T15:35:56-05:00[US/Central],  
2024-04-08T22:35:56+02:00[Europe/Paris] Paris time
```

If you tend to lose track of time, but you really, really don't want to miss the eclipse, you can check to make sure the eclipse is still in the future with this code:

```
// compare two ZonedDateTime (must be the same type!)  
System.out.println("Is the 2024 eclipse still in the future? " +  
    ZonedDateTime.now().isBefore(totalityAustin));
```

Since we're writing this in 2017, the 2024 eclipse is still far in the future, so we see

```
Is the 2024 eclipse still in the future? true
```

And finally, we'd better do one more check about 2024. How about checking to see if 2024 is a leap year? You definitely don't want to miss the eclipse by a day:

```
System.out.println("Is 2024 a leap year? " +  
totalityAustin.isLeapYear());
```

Try this code and you'll get a compile-time error:

The method `isLeapYear()` is undefined for the type `ZonedDateTime`

Hmm. It turns out `isLeapYear()` is defined only for `LocalDate`, not for `LocalDateTime` or `ZonedDateTime`. We can fix the code by converting `totalityAustin` to a `LocalDate`:

```
System.out.println("Is 2024 a leap year? " +  
totalityAustin.toLocalDate().isLeapYear());
```

Another way to check for a leap year is

```
System.out.println("Is 2024 a leap year? " +  
Year.of(2024).isLeap());
```

The output from both lines of code shows that 2024 is, indeed, a leap year.

Formatting Output with `DateTimeFormatter`

Earlier we used `DateTimeFormatter` to specify a pattern when parsing a date string. We can also use `DateTimeFormatter` when we display a datetime as a string.

You might know that in the United States, we tend to write month/day/year, and in the European Union, they tend to write day/month/year. (Yes, that can get a bit confusing at times!)

Let's format and display the datetime of the eclipse in Austin using the European-preferred format. There are a couple of different ways we can do that. First, we can specify exactly the format we want using letters and symbols, as we described earlier:

```
System.out.println("Totality date/time written for sister in Europe: " +  
totalityAustin.format(  
DateTimeFormatter.ofPattern("dd/MM/yyyy hh:mm")));
```

Here, we're using the `format()` method of the `ZonedDateTime`, `totalityAustin`, and passing in a formatter. The formatter specifies a format to use for formatting the datetime, using allowed letters and symbols (see the `DateTimeFormatter` documentation for all the options). When we look at the output, we see

```
Totality date/time written for sister in Europe: 08/04/2024 01:35
```

Alternatively, we could specify a format style and a locale:

```
System.out.println("Totality date/time in UK Locale: " +  
    totalityAustin.format(  
        DateTimeFormatter.ofLocalizedDateTime(  
            FormatStyle.SHORT)  
        .withLocale(Locale.UK)));
```

Now when we look at the output, we see

```
Totality date/time in UK Locale: 08/04/24 13:35
```

You'll learn more about Locales shortly; essentially, Locales are designed to tailor data for a specific region. Here, we're creating a `DateTimeFormatter` by specifying a built-in style and then using that to create a new formatter for a specific locale, the UK locale. Creating a formatter with a specific locale means the formatter is adjusted appropriately for that locale. We see that the UK locale uses the day/month/year format for the date and the 24-hour format for the time. As you can see, there is a lot to the `java.time.*` package. There's no way you can memorize everything in the package for the exam, so we recommend you focus on the classes, properties, and methods in [Tables 4-1](#) and [4-2](#) (later in the chapter) and familiarize yourself with the rest of the package by looking over [the documentation](#) to get a sense of what's there (see <https://docs.oracle.com/javase/8/docs/api/java/time/package-summary.html>).

You've probably noticed a pattern in the method names used in the `java.time.*` package. For instance, `of()` methods create a new date from, typically, a sequence of numbers specifying the year, month, day, and so on. `parse()` methods create a new date by parsing a string that's either in a standard ISO format already or by using a formatter. `with()` methods allow you to adjust a

date with a `TemporalAdjuster` to make a new date. `plusX()` and `minusX()` methods create a new `Datetime` object from an existing one by adding and subtracting `TemporalUnits` or longs representing weeks, minutes, and so on. Study the `LocalDateTime` and `ZonedDateTime` methods enough to get the hang of this pattern so you can recognize the methods on the exam (without having to memorize them all).

Using Dates and Times with Locales

The `Locale` class is your ticket to understanding how to internationalize your code. Both the `DateTimeFormatter` class and the `NumberFormat` class can use an instance of `Locale` to customize formatted output for a specific locale (and you just got a taste of this with `DateTimeFormatter` in the previous section).

You might ask how Java defines a locale. The API says a locale is “a specific geographical, political, or cultural region.” The two `Locale` constructors you’ll need to understand for the exam are

```
Locale(String language)
Locale(String language, String country)
```

The `language` argument represents an ISO 639 Language code, so, for instance, if you want to format your dates or numbers in Walloon (the language sometimes used in southern Belgium), you’d use "wa" as your language string. There are over 500 ISO Language codes, including one for Klingon ("t1h"), although, unfortunately, Java doesn’t yet support the Klingon locale. We thought about telling you that you’d have to memorize all these codes for the exam...but we didn’t want to cause any heart attacks. So rest assured, you will *not* have to memorize any ISO Language codes or ISO Country codes (of which there are about 240) for the exam.

Let’s get back to how you might use these codes. If you want to represent basic Italian in your application, all you need is the Language code. If, on the other hand, you want to represent the Italian used in Switzerland, you’d want to indicate that the country is Switzerland (yes, the Country code for Switzerland is "CH"), but that the language is Italian:

```
Locale locIT = new Locale("it");           // Italian
Locale locCH = new Locale("it", "CH");      // Switzerland
```

Using these two locales on a date could give us output like this:

```
sabato 1 ottobre 2005  
sabato, 1. ottobre 2005
```

Now let's put this all together in some code that creates a `ZonedDateTime` object and sets its date. We'll then take that datetime object and print it using locales from around the world:

```
Locale myLocale = Locale.getDefault();  
System.out.println("My locale: " + myLocale);  
LocalDateTime aDateTime = LocalDateTime.of(2024, 4, 8, 13, 35, 56);  
System.out.println("The date and time: " +  
    aDateTime.format(DateTimeFormatter.ofLocalizedDateTime(  
        TextStyle.MEDIUM)));  
ZonedDateTime zDateTime = ZonedDateTime.of(  
    aDateTime, ZoneId.of(  
        "US/Pacific"));
```

```
Locale locIT = new Locale("it", "IT");           // Italy
Locale locPT = new Locale("pt");                  // Portugal
Locale locBR = new Locale("pt", "BR");            // Brazil
Locale locIN = new Locale("hi", "IN");            // India
Locale locJA = new Locale("ja");                  // Japan
Locale locDK = new Locale("da", "DK");            // Denmark
System.out.println("Italy (Long) " +
zDateTime.format(
    DateTimeFormatter.ofLocalizedDateTime(FormatStyle.LONG)
    .withLocale(Locale.ITALY)));
System.out.println("Italy (Short) " +
aDateTime.format(
    DateTimeFormatter.ofLocalizedDateTime(FormatStyle.SHORT)
    .withLocale(locIT));

System.out.println("Japan (Long) " +
zDateTime.format(
    DateTimeFormatter.ofLocalizedDateTime(FormatStyle.LONG)
    .withLocale(Locale.JAPAN)));

System.out.println("Portugal (Long) " +
zDateTime.format(
    DateTimeFormatter.ofLocalizedDateTime(FormatStyle.LONG)
    .withLocale(locPT)));

System.out.println("India (Long) " +
zDateTime.format(
    DateTimeFormatter.ofLocalizedDateTime(FormatStyle.LONG)
    .withLocale(locIN)));

System.out.println("Denmark (Medium) " +
zDateTime.format(
    DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM)
    .withLocale(locDK)));
```

This code, on our JVM, produces the output:

```
My locale: en_US
The date and time: Apr 8, 2024 1:35:56 PM
Italy (Long) 8 aprile 2024 13.35.56 PDT
Italy (Short) 08/04/24 13.35
Japan (Long) 2024/04/08 13:35:56 PDT
Portugal (Long) 8 de Abril de 2024 13:35:56 PDT
India (Long) 8 अप्रैल, 2024 1:35:56 अपराह्न PDT
Denmark (Medium) 08-04-2024 13:35:56
```

So you can see how a single `ZonedDateTime` object can be formatted to work for many locales and varying amounts of detail. (Note that you'll need Eclipse in UTF-8 format to see the Indian output properly.)

There are a couple more methods in `Locale` (`getDisplayCountry()` and `getDisplayLanguage()`) that you need to know for the exam. These methods let you create strings that represent a given locale's country and language in terms of both the default locale and any other locale:

```
Locale locBR = new Locale("pt", "BR"); // Brazil
Locale locDK = new Locale("da", "DK"); // Denmark
Locale locIT = new Locale("it", "IT"); // Italy

System.out.println("Denmark, country: " + locDK.getDisplayCountry());
System.out.println("Denmark, country, local: " +
    locDK.getDisplayCountry(locDK));
System.out.println("Denmark, language: " + locDK.getDisplayLanguage());
System.out.println("Denmark, language, local: " +
    locDK.getDisplayLanguage(locDK));

System.out.println("Brazil, country: " + locBR.getDisplayCountry());
System.out.println("Brazil, country, local: " +
    locBR.getDisplayCountry(locBR));
System.out.println("Brazil, language: " + locBR.getDisplayLanguage());
System.out.println("Brazil, language, local: " +
    locBR.getDisplayLanguage(locBR));
System.out.println("Italy, Danish language is: " +
    locDK.getDisplayLanguage(locIT));
```

This code, on our JVM, produces the output:

```
Denmark, country: Denmark
Denmark, country, local: Danmark
Denmark, language: Danish
Denmark, language, local: Dansk
Brazil, country: Brazil
Brazil, country, local: Brasil
Brazil, language: Portuguese
Brazil, language, local: português
Italy, Danish language is: danese
```

Our JVM's locale (the default for us, which we saw displayed earlier) is `en_US`, and when we display the country name for Brazil, in our locale, we get `Brazil`. In Brazil, however, the country is `Brasil`. Same with the language; for us the language of Brazil is `Portuguese`; for people in `Brasil`, it's `português`. Likewise, for Denmark, you can see how we have different names for the country and the language than the Danish do.

Finally, just for fun, we discovered that in Italy, the Danish language is called `danese`.

Orchestrating Date- and Time-Related Classes

When you work with dates and times, you'll often use several classes together. It's important to understand how the classes described earlier relate to each other and when to use which classes in combination. For instance, you need to know that if you're creating a new `ZonedDateTime` from an existing `LocalDate` and `LocalTime`, you need a `ZoneId` too; if you want to do date formatting for a specific locale, you need to create your `Locale` object before your `DateTimeFormatter` object because you'll need your `Locale` object as an argument to your `DateTimeFormatter` method; and so on. [Tables 4-1](#) and [4-2](#) provide a quick overview and summary of common date- and time-related use cases: how to create datetime objects and how to adjust them. We are by no means including all of the many available methods to work with dates and times, however, so make sure you peruse the documentation too (see <https://docs.oracle.com/javase/8/docs/api/java/time/package-summary.html>).

TABLE 4-1 Instance Creation for `java.time` Classes

java.time Class	Key Instance Creation Options
LocalDate	LocalDate.now(); LocalDate.of(2017, 8, 21); LocalDate.parse("2017-08-21");
LocalTime	LocalTime.now(); LocalTime.of(10, 19, 36); LocalTime.parse("10:19:36");
LocalDateTime	LocalDateTime.now(); LocalDateTime.of(aDate, aTime); LocalDateTime.parse("2017-04-08T10:19:36"); LocalDateTime.parse(aDateTime, aFormatter); LocalDateTime.parse("2017-08-21T10:19", aformatter);
ZonedDateTime	ZonedDateTime.now(); ZonedDateTime.of(aDateTime, ZoneId.of(aZoneString)); ZonedDateTime.parse("2017-04-08T10:19:36-05:00");
OffsetDateTime	OffsetDateTime.now(); OffsetDateTime.of(aDateTime, ZoneOffset.of("-05:00")); OffsetDateTime.parse("2017-04-08T10:19:36-05:00");
format. DateTimeFormatter	DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm"); DateTimeFormatter.ofLocalizedDateTime(FormatStyle.SHORT).withLocale(aLocale);
Instant	Instant.now(); zonedDateTime.toInstant(); aDateTime.toInstant(ZoneOffset.of("+5"));
Duration	Duration.between(aTimel, aTime2); Duration.ofMinutes(5);
Period	Period.between(aDate1, aDate2); Period.ofDays(3);
util.Locale	Locale.getDefault(); new Locale(String language); new Locale(String language, String country);

TABLE 4-2 Adjustment Options for `java.time` Classes

Class	Key Adjustment Options and Examples (all methods create a new <code>datetime</code> object)
<code>LocalDate</code>	<code>aDate.minusDays(3);</code> <code>aDate.plusWeeks(1);</code> <code>aDate.withYear(2018);</code>
<code>LocalTime</code>	<code>aTime.minus(3, ChronoUnit.MINUTES);</code> <code>aTime.plusMinutes(3);</code> <code>aTime.withHour(12);</code>
<code>LocalDateTime</code>	<code>aDateTime.minusDays(3);</code> <code>aDateTime.plusMinutes(10);</code> <code>aDateTime.plus(Duration.ofMinutes(5));</code> <code>aDateTime.withMonth(2);</code>
<code>ZonedDateTime</code>	<code>zonedDateTime.withZoneSameInstant(ZoneId.of("US/Pacific"));</code>

CERTIFICATION OBJECTIVE

Properties Files (OCP Objective 12.2)

12.2 *Create and read a Properties file.*

Property files are typically used to externally store configuration settings and operating parameters for your applications. In the Java world, there are at least three variations on property files:

1. There is a system-level properties file that holds system information like hardware info, software versions, classpaths, and so on. The `java.lang.System` class has methods that allow you to update this file and view its contents. This property file is not on the exam.
2. There is a class called `java.util.Properties` that makes it easy for a programmer to create and maintain property files for whatever applications the programmer chooses. We'll talk about the `java.util.Properties` class in this section. In this section, when we say "property" files, we're referring to files that are compliant with the `java.util.Properties` class.
3. There is a class called `java.util.ResourceBundle` that *can*—optionally—use `java.util.Properties` files to make it easier for a programmer to add localization and/or internationalization features to applications. After we discuss `java.util.Properties`, we'll discuss `java.util.ResourceBundle`.

The `java.util.Properties` class is used to create and/or maintain human-readable text files. It's also possible to create well-formed, `Properties`-compliant text files using a text editor. If you're using a `Properties` file for applications other than to support resource bundles, you can give them whatever legal filenames you want. Typically their names end in ".properties," e.g., "MyApp.properties." Other suffixes like ".props" are also common. The basic structure of a `Properties` file is a set of comments (usually comment lines begin with "#") at the top of the file, followed by a number of rows of text data, each row representing a key/value pair, with the key and value usually separated with an "=".

Almost everyone uses # for comments and = to separate key/value pairs. There are alternative syntax choices, though, which you should understand if you come across them.

Property files can use two styles of commenting:

```
! comment
```

or

```
# comment
```

Property files can define key/value pairs in any of the following formats:

```
key=value  
key:value  
key value
```

Let's refresh what we've learned about property files and take a closer look. Aside from comments, a property file contains key/value pairs:

```
# this file contains a single key/value  
hello=Hello Java
```

A *key* is the first string on a line. Keys and values are usually separated by an equal sign. If you want to break up a single line into multiple lines, you use a backslash. Given an entry in a property file:

```
hello1 = Hello \  
World!
```

The code and output would be

```
System.out.println(rb.getString("hello1"));  
Hello World!
```

If you actually want a line break, you use the standard Java \n escape sequence. Given an entry in a property file:

```
hello2 = Hello \nWorld !
```

The code and output would be

```
System.out.println(rb.getString("hello2"));  
Hello  
World !
```

You can mix and match these to your heart's content. Java helpfully ignores any whitespace before subsequent lines of a multiline property, so you can use indentation for clarity:

```
hello3 =      123\  
45
```

Given the above entry in a properties file, the code and output would be

```
System.out.println(rb.getString("hello3"));
12345
```

As we mentioned earlier, `java.lang.System` provides access to a property file. Although this file isn't on the exam, the following code

```
import java.util.*;
public class SysProps {
    public static void main(String[] args) {
        Properties p = System.getProperties(); // open system properties file
        p.setProperty("myProp", "myValue"); // add an entry
        p.list(System.out); // list the file's contents
    }
}
```

will produce output that contains entries like these:

```
myProp=myValue
java.version=1.8.0_45
os.name=Mac OS X
os.version=10.12.6
..
..
```

Again, what we're seeing here is a list of key/value pairs.

Let's move on to creating and working with our own property file. Here's some code that creates a new `Properties` object, adds a few properties, and then stores the contents of the `Properties` object to a file on disk:

```
import java.util.*;
import java.io.*;
class Props1 {
    public static void main(String[] args) {
        Properties p = new Properties();
        p.setProperty("k1", "v1");
        p.setProperty("k2", "v2");
        p.list(System.out); // what's in the object
        try {
            // creates or replaces file
            FileOutputStream out = new FileOutputStream("myProps1.props");
            p.store(out, "test-comment"); // adds header comment
            out.close();
        } catch (IOException e) {
            System.out.println("exc 1");
        }
    }
}
```

which produces the following output:

```
-- listing properties --
k2=v2
k1=v1
```

and a file named `myProps1.props`, which contains

```
#test-comment
#Fri Feb 02 14:53:30 PST 2018
k2=v2
k1=v1
```

Note that there are a couple of comments at the top of the file. Now let's run a second program that opens up the file we just created, adds a new key/value pair, then saves the result to a second file on disk:

```
import java.util.*;
import java.io.*;
class Props2 {
    public static void main(String[] args) {
        Properties p2 = new Properties();
        try {
            FileInputStream in = new FileInputStream("myProps1.props");
            p2.load(in);
            p2.list(System.out);
```

```
        p2.setProperty("newProp", "newData");
        p2.list(System.out);
        FileOutputStream out = new FileOutputStream("myProps2.props");
        p2.store(out, "myUpdate");
        in.close();
        out.close();
    } catch (IOException e) {
        System.out.println("exc 2");
    }
}
}
```

which produces

```
-- listing properties --
newProp=newData
k2=v2
k1=v1
```

and a file named `myProps2.props`, which contains

```
#myUpdate
#Fri Feb 02 14:53:58 PST 2018
newProp=newData
k2=v2
k1=v1
```

It's important to know that `java.util.Properties` inherits from `java.util.Hashtable`. Technically, when mucking around with property files, you could use methods from `Hashtable` like `put()` and `get()`, but Oracle encourages

you to stick with the methods provided in the `Properties` class since those methods will force you to use arguments of type `String`. For the exam you should know the following methods from the `Properties` class:

```
String getProperty(String key)
```

```
void list(PrintStream out)
```

```
void load(InputStream inStream)
```

```
Object setProperty(String key, String value)
```

```
void store(OutputStream out, String headerComment)
```

Next, let's move on to the resource bundles to see how they work and how you can (if you want to) use property files to support your resource bundles.

CERTIFICATION OBJECTIVE

Resource Bundles (OCP Objectives 12.1, 12.2, and 12.3)

12.1 Read and set the locale by using the `Locale` object.

12.2 Create and read a `Properties` file.

12.3 Build a resource bundle for each locale and load a resource bundle in an application.

Earlier, we used the `Locale` class to display dates for basic localization. For full-fledged localization, we also need to provide language- and country-specific strings for display. There are only two parts to building an application with resource bundles:

- `Locale` You can use the same `Locale` we used for `DateFormat` and

`NumberFormat` to identify which resource bundle to choose.

- `ResourceBundle` Think of a `ResourceBundle` as a map. You can use property files or Java classes to specify the mappings.

Let's build a simple application to be used in Canada. Since Canada has two official languages, we want to let the user choose her favorite language. Designing our application, we decided to have it just output "Hello Java" to show off how cool it is. We can always add more text later.

We are going to externalize everything language specific to special property files. They're just property files that contain keys and string values to display, but they follow very specific, `ResourceBundle`-required naming conventions. Here are two simple resource bundle files:

A file named `Labels_en.properties` that contains a single line of data:

```
hello=Hello Java!
```

A second file named `Labels_fr.properties` that contains a single line of data:

```
hello=Bonjour Java!
```

It's critical to understand that when you use `Properties` files to support `ResourceBundle` objects, the naming of the files MUST follow two rules:

1. These files must end in ".properties."
2. The end of the name before the .properties suffix must be a string that starts with an underscore and then declares the Locale the file represents (e.g., `MyApp_en.properties` or `MyApp_fr.properties` or `MyApp_fr_CA.properties`). `ResourceBundle` only knows how to find the appropriate file via the filename. There is no requirement for the data in the file to contain locale information.

Using a resource bundle requires three steps: obtaining the `Locale`, getting the `ResourceBundle`, and looking up a value from the resource bundle. First, we create a `Locale` object. To review, this means one of the following:

```
new Locale("en")           // language - English  
new Locale("en", "CA")    // language and country - Canadian English  
Locale.CANADA             // constant for common locales - Canadian English
```

Next, we need to create the resource bundle. We need to know the bundle name of the resource bundle and the locale. The bundle name of the resource bundle is that part of the filename up to (but not including) the underscore that is the start of the locale info. For example, if a `Properties` file is named `MyApp_en.properties`, then the bundle name is “`MyApp`.” Then we pass those values to a factory, which creates the resource bundle. The `getBundle()` method looks in the classpath for bundles that match the bundle name (in the code below, the bundle name is “`Labels`”) and the provided `locale`.

```
ResourceBundle rb = ResourceBundle.getBundle("Labels", locale);
```

Finally, we use the resource bundle like a map and get a value based on the key:

```
rb.getString("hello");
```

So, back to our example, we have two files: `Labels_en.properties` and `Labels_fr.properties`. The following code takes a `locale` argument and builds a `ResourceBundle` object that’s tied to the `Properties` file containing data for that `Locale` and read from the “resource bundle”:

```
import java.util.Locale;  
import java.util.ResourceBundle;  
  
public class WhichLanguage {  
    public static void main(String[] args) {
```

```

        Locale locale = new Locale(args[0]);
        ResourceBundle rb = ResourceBundle.getBundle("Labels", locale);
        System.out.println(rb.getString("hello"));
    }
}

```

Running the code twice, we get

```

> java WhichLanguage en
Hello Java!
> java WhichLanguage fr
Bonjour Java!

```



The Java API for `java.util.ResourceBundle` lists three good reasons to use resource bundles. Using resource bundles “allows you to write programs that can

- *Be easily localized, or translated, into different languages*
- *Handle multiple locales at once*
- *Be easily modified later to support even more locales”*

If you encounter any questions on the exam that ask about the advantages of using resource bundles, this quote from the API will serve you well.



The most common use of localization in Java is web applications. You can get the user’s locale from information passed in the request rather than hard-

coding it.

Java Resource Bundles

When we need to move beyond simple property file key to string value mappings, we can use resource bundles that are Java classes. We write Java classes that extend `ListResourceBundle`. The class name is similar to the one for property files. Only the extension is different.

```
import java.util.ListResourceBundle;
public class Labels_en_CA extends ListResourceBundle {
    protected Object[][] getContents() {
        return new Object[][] {
            { "hello", new StringBuilder("from Java") }
        };
    }
}
```

We implement `ListResourceBundle`'s one required method that returns an array of arrays. The inner array is key/value pairs. The outer array accumulates such pairs. Notice that now we aren't limited to `String` values. We can call `getObject()` to get a non-`String` value:

```
Locale locale = new Locale("en", "CA");
 ResourceBundle rb = ResourceBundle.getBundle("Labels", locale);
 System.out.println(rb.getObject("hello"));
```

which prints "from Java".

Default Locale

What do you think happens if we call `ResourceBundle.getBundle("Labels")` without any locale? It depends. Java will pick the resource bundle that matches

the locale the JVM is using. Typically, this matches the locale of the machine running the program, but it doesn't have to. You can even change the default locale at runtime, which might be useful if you are working with people in different locales so you can get the same behavior on all machines.

Let's explore the API to get and set the default locale:

```
// store locale so can put it back at end
Locale initial = Locale.getDefault();
System.out.println(initial);

// set locale to Germany
Locale.setDefault(Locale.GERMANY);
System.out.println(Locale.getDefault());

// put original locale back
Locale.setDefault(initial);
System.out.println(Locale.getDefault());
```

which on our computer prints:

```
en_US
de_DE
en_US
```

For the first and last line, you may get different output depending on where you live. The key is that the middle of the program executes as if it were in Germany, regardless of where it is actually being run. It is good practice to restore the default unless your program is ending right away. That way, the rest of your code works normally—it probably doesn't expect to be in Germany.

Choosing the Right Resource Bundle

There are two main ways to get a resource bundle:

```
ResourceBundle.getBundle(baseName)  
ResourceBundle.getBundle(baseName, locale)
```

Luckily, `ResourceBundle.getBundle(baseName)` is just shorthand for `ResourceBundle.getBundle(baseName, Locale.getDefault())`, and you only have to remember one set of rules. There are a few other overloaded signatures for `getBundle()`, such as taking a `ClassLoader`. But don't worry—these aren't on the exam.

Now on to the rules. How does Java choose the right resource bundle to use? In a nutshell, Java chooses the most specific resource bundle it can while giving preference to Java `ListResourceBundle`.

Going back to our Canadian application, we decide to request the Canadian French resource bundle:

```
Locale locale = new Locale("fr", "CA");  
ResourceBundle rb = ResourceBundle.getBundle("RB", locale);
```

Java will look for the following files in the classpath in this order:

```
RB_fr_CA.java           // exactly what we asked for  
RB_fr_CA.properties  
  
RB_fr.java             // couldn't find exactly what we asked for  
RB_fr.properties       // now trying just requested language  
RB_en_US.java          // couldn't find French  
RB_en_US.properties    // now trying default Locale  
RB_en.java              // couldn't find full default Locale country  
RB_en.properties        // now trying default Locale language  
RB.java                 // couldn't find anything matching Locale,  
RB.properties           // now trying default bundle
```

If none of these files exist, Java gives up and throws a `MissingResourceException`. Although this is a lot of things for Java to try, it is pretty easy to remember. Start with the full `Locale` requested. Then fall back to just language. Then fall back to the default `Locale`. Then fall back to the default bundle. Then cry.

Make sure you understand this because it is about to get more complicated.

You don't have to specify all the keys in all the property files. They can inherit from each other. This is a good thing, as it reduces duplication.

```
RB_en.properties
```

```
    ride.in=Take a ride in the
```

```
RB_en_US.properties
```

```
    elevator=elevator
```

```
RB_en_UK.properties
```

```
    elevator=lift
```

```
Locale locale = new Locale("en", "UK");
```

```
ResourceBundle rb = ResourceBundle.getBundle("RB", locale);
```

```
System.out.println(rb.getString("ride.in") + " " +
```

```
rb.getString("elevator"));
```

Outputs:

```
Take a ride in the lift
```

The common "ride.in" property comes from the parent noncountry-specific bundle "RB_en.properties." The "elevator" property is different by country and comes from the UK version that we specifically requested.

The parent hierarchy is more specific than the search order. A bundle's parent always has a shorter name than the child bundle. If a parent is missing, Java just skips along that hierarchy. `ListResourceBundles` and

PropertyResourcesBundles do not share a hierarchy. Similarly, the default locale's resource bundles do not share a hierarchy with the requested locale's resource bundles. [Table 4-3](#) shows examples of bundles that do share a hierarchy.

TABLE 4-3 Resource Bundle Lookups

Name of Resource Bundle	Hierarchy
RB_fr_CA.java	RB.java RB_fr.java RB_fr_CA.java
RB_fr_CA.properties	RB.properties RB_fr.properties RB_fr_CA.properties
RB_en_US.java	RB.java RB_en.java RB_en_US.java
RB_en_US.properties	RB.properties RB_en.properties RB_en_US.properties

Remember that searching for a property file uses a linear list. However, once a matching resource bundle is found, keys can only come from that resource bundle's hierarchy.

One more example to make this clear. Think about which resource bundles will be used from the previous code if we use the following code to request a resource bundle:

```
Locale locale = new Locale("fr", "FR");
ResourceBundle rb = ResourceBundle.getBundle("RB", locale);
```

First, Java looks for `RB_fr_FR.java` and `RB_fr_FR.properties`. Because neither is found, Java falls back to using `RB_fr.java`. Then as we request keys from `rb`, Java starts looking in `RB_fr.java` and additionally looks in `RB.java`. Java started out looking for a matching file and then switched to searching the hierarchy of that file.

CERTIFICATION SUMMARY

Dates and Times The `Date` and `Calendar` classes, as well as `DateFormat`, have all been replaced by classes in the `java.time` package, so pay close attention if you're transitioning from the old classes to the new. The key datetime classes to know from `java.time` are `LocalDate`, `LocalTime`, `LocalDateTime`, and `ZonedDateTime`. Each has a variety of methods to create and adjust datetime objects. You also need to know about `TemporalAdjusters` (like `TemporalAdjuster.firstDayOfMonth()`) and `TemporalUnits` (like `ChronoUnit.DAYS`), both from the `java.time.temporal` package, and `Instants`, `Periods`, and `Durations`, in the `java.time` package. `DateFormat` has been replaced with `DateTimeFormatter` in the `java.time.format` package, which is used to parse, format, and print datetime objects. The `Locale` class is used with `DateTimeFormatter` to generate a variety of output styles that are language and/or country specific.

Make sure you are clear on how to work with time zones and daylight savings time. Fortunately, the `ZonedDateTime` and related classes handle most of the hard work for you, but pay close attention to the format of the datetimes when they are represented as strings so you can recognize a local datetime from a zoned datetime and so you know how to create a `ZonedDateTime` using a `ZoneId`.

Locales, Properties Files, and Resource Bundles Resource bundles allow you to move locale-specific information (usually strings) out of your code and into external files where they can easily be amended. This provides an easy way for you to localize your applications across many locales. Properties files allow you to create text files formatted as key/value pairs to store application customization parameters and such external to your application. The `ResourceBundle` class

provides convenient ways to use files that are `Properties` class-compatible to store internationalization and localization values.



TWO-MINUTE DRILL

Here are some of the key points from the certification objectives in this chapter.

Dates and Times (OCP Objectives 7.1, 7.2, and 7.3)

- The classes you need to understand are those in `java.time`, `java.time.temporal`, and `java.time.format`, as well as `java.util.Locale`.
- `Date` and `Calendar` are no longer used, and most of the `Date` class's methods have been deprecated.
- A `LocalDate` is a date, and a `LocalTime` is a time. Combine the two to make a `LocalDateTime`. None of these types have a time zone associated with them.
- A `ZonedDateTime` is a datetime object with a time zone. All zoned datetimes are relative to Greenwich Mean Time (GMT). You may sometimes see GMT written as UTC.
- A `ZoneId` can be created from a string representing a time zone (e.g. "US/Pacific").
- When you adjust `ZonedDateTime`s, daylight savings time will be automatically handled using the `ZoneRules`.
- If you want a datetime object with a time zone that is independent of zone rules, use an `OffsetDateTime`.
- A `Period` is a period of time that is a day or longer.
- A `Duration` is a period of time that is shorter than a day.
- An `Instant` is an instant in time and represents the number of seconds and nanoseconds since January 1, 1970. You can get the number of seconds as a `long` value from an `Instant` and convert any `ZonedDateTime` object into an `Instant`.
- There are several format “styles” available in the `java.format` class. You can use format styles such as `FormatStyle.SHORT` with `DateTimeFormatter` to format datetime objects.

- The `DateTimeFormatter` class is used to parse and create strings containing properly formatted dates.
- The `Locale` class is used in conjunction with `DateFormat` and `NumberFormat`.
- A `DateTimeFormatter` object can be constructed with a specific, immutable `Locale`.
- For the exam, you should understand creating `Locales` using either language or a combination of language and country.

Locales, Properties Files, and Resource Bundles (OCP 12.1, 12.2, and 12.3)

- The `java.util.Properties` class gives you a convenient way to create and maintain text files that are external to your applications and can hold configuration values.
- A file that is `java.util.Properties`-compliant and has a name that ends with a locale and a suffix of `.properties` can be used by `ResourceBundle.getBundle()`.
- A `ListResourceBundle` comes from Java classes, and a `PropertyResourceBundle` comes from `.properties` files.
- `ResourceBundle.getBundle(name)` uses the default `Locale`.
- `Locale.getDefault()` returns the JVM's default `Locale`. `Locale.setDefault(locale)` can change the JVM's `locale`.
- Java searches for resource bundles in this order: requested language/country, requested language, default locale language/country, default locale language, default bundle. Within each item, Java `ListResourceBundle` is favored over `PropertyResourceBundle`.
- Once a `ResourceBundle` is found, only parents of that bundle can be used to look up keys.

SELF TEST

1. Given the code fragment:

```
ZonedDateTime zd = ZonedDateTime.parse("2020-05-
```

```
04T08:05:00");
System.out.println(zd.getMonth() + " " + zd.getDayOfMonth());
```

What is the result? (Choose all that apply.)

- A. MAY 4
- B. APRIL 5
- C. MAY 4 2020
- D. APRIL 5 2020
- E. Compilation fails
- F. Runtime exception

2. Given the code fragment:

```
LocalTime t1 = LocalTime.of(9, 0);
LocalTime t2 = LocalTime.of(10, 5);
```

Which of the following code fragment(s) will produce a new `LocalTime` `t3` that represents the same time as `t2`? (Choose all that apply.)

- A. `LocalTime t3 = t1.plus(65, ChronoUnit.MINUTES)`
- B. `LocalTime t3 = t1.plusMinutes(65);`
- C. `LocalTime t3 = t1.plusHours(1);`
- D. `LocalTime t3 = t1.plusDays(1);`
- E. `LocalTime t3 = t1.plus(Duration.ofMinutes(65));`

3. Given the code fragment:

1. `LocalDate d1 = LocalDate.of(2018, 1, 1);`
2. `LocalDate d2 = LocalDate.of(2018, 6, 15);`
3. _____ `r = _____ .between(d1, d2);`
4. `System.out.println("Months and days: " + r.getMonths() + ", " + r.getDays());`

What are the correct types to fill in the blanks on line 3?

- A. Duration, Duration
 - B. Instant, Period
 - C. Period, Instant
 - D. Period, ChronoUnit
 - E. Period, Period
 - F. Duration, LocalDate
4. How would you use nowzdt from the code fragment below to compute the equivalent time in Berlin, Germany? (Choose all that apply.)

```
ZonedDateTime nowzdt =  
    ZonedDateTime.of(LocalDateTime.now(), ZoneId.of("US/Pacific"));
```

A.

```
ZonedDateTime berlinZdt = ZonedDateTime.from(nowzdt, ZoneId.of("Europe/Berlin"));
```

B.

```
ZonedDateTime berlinZdt = nowzdt.withZoneSameInstant(ZoneId.of("Europe/Berlin"));
```

C.

```
ZonedDateTime berlinZdt =  
    ZonedDateTime.ofInstant(nowzdt.toInstant(), ZoneId.of("Europe/Berlin"));
```

D.

```
ZonedDateTime berlinZdt =  
    nowzdt.withZoneId("Europe/Berlin"));
```

E.

```
ZonedDateTime berlinZdt = nowzdt.now(ZoneId.of("Europe/Berlin"));
```

5. The next total solar eclipse visible in South America is on July 2, 2019, at 16:55 UTC. Which code fragment will correctly compute and display the time in San Juan, Argentina, for this solar eclipse?

A.

```
ZonedDateTime totalityUTC = ZonedDateTime.of(  
    LocalDateTime.of(2019, 7, 2, 16, 55));  
ZonedDateTime totalitySanJuan =  
    totalityUTC.withZoneSameInstant(ZoneId.of("America/Argentina/San_Juan"));  
System.out.println(totalitySanJuan);
```

B.

```
ZonedDateTime totalityUTC = ZonedDateTime.of(  
    LocalDateTime.of(2019, 7, 2, 4, 55, "PM") , ZoneId.of("Z"));  
ZonedDateTime totalitySanJuan =  
    totalityUTC.withZoneSameInstant(ZoneId.of("America/Argentina/San_Juan"));  
System.out.println(totalitySanJuan);
```

C.

```
ZonedDateTime totalityUTC = ZonedDateTime.of(  
    LocalDateTime.of(2019, 7, 2, 16, 55), ZoneId.of("Z"));  
ZonedDateTime totalitySanJuan =  
    totalityUTC.withZoneSameInstant(ZoneId.of("America/Argentina/San_Juan"));  
System.out.println(totalitySanJuan);
```

D.

```
ZonedDateTime totalityUTC = ZonedDateTime.of(  
    LocalDateTime.of(2019, 7, 2, 16, 55),  
    ZoneId.of("America/Argentina/San_Juan"));  
ZonedDateTime totalitySanJuan =  
    totalityUTC.withZoneSameInstant(ZoneId.of("America/Argentina/San_Juan"));  
System.out.println(totalitySanJuan);
```

E.

```
ZonedDateTime totalityUTC = ZonedDateTime.of(  
    LocalDateTime.of(2019, 7, 2, 16, 55), ZoneId.of("Z "));  
LocalDateTime totalitySanJuan =  
    totalityUTC.withZoneSameInstant(ZoneId.of("America/Argentina/San_Juan"));  
System.out.println(totalitySanJuan);
```

6. Given:

```
public class Canada {  
    public static void main(String[] args) {  
        ResourceBundle rb = ResourceBundle.getBundle("Flag",  
            new Locale("en", "CA"));  
        System.out.println(rb.getString("key"));  
    }  
}
```

Assume the default Locale is Italian. If each of the following is the only resource bundle on the classpath and contains key=value, which will be

used? (Choose all that apply.)

- A. Flag.java
- B. Flag_CA.properties
- C. Flag_en.java
- D. Flag_en.properties
- E. Flag_en_CA.properties
- F. Flag_fr_CA.properties

7. Given three resource bundles and a Java class:

Train_en_US.properties: train=subway

Train_en_UK.properties: train=underground

Train_en.properties: ride = ride

```
1: public class ChooChoo {  
2:     public static void main(String[] args) {  
3:         Locale.setDefault(new Locale("en", "US"));  
4:         ResourceBundle rb = ResourceBundle.getBundle("Train",  
5:             new Locale("en", "US"));  
6:         System.out.print(rb.getString("ride")  
7:                         + " " + rb.getString("train"));  
8:     }  
9: }
```

Which of the following, when made independently, will change the output to “ride underground”? (Choose all that apply.)

- A. Add train=underground to Train_en.properties
- B. Change line 3 to Locale.setDefault(new Locale("en", "UK"));
- C. Change line 5 to Locale.ENGLISH;
- D. Change line 5 to new Locale("en", "UK"));

E. Delete file Train_en_US.properties

8. Let's say you want to print the day of the week and the date of Halloween (October 31) 2018, at 5 PM in German, using the LONG style. Complete the code below using the following fragments. Note: You can use each fragment either zero or more times, and you might not need to fill all of the slots. You probably won't encounter a fill-in-the-blank question on the exam, but just in case, we put a few in the book, like this one.

Code:

```
import java._____  
import java._____  
import java._____
```

```
public class DateHalloween {  
    public static void main(String[] args) {  
        ZonedDateTime d = _____  
        Locale locDE = new Locale("de");  
        DayOfWeek day = _____  
        String df = _____  
        System.out.println(day + " " + df);  
    }  
}
```

Fragments:

```
io.*;  
nio.*;  
util.*;  
time.*;  
date.*;  
time.format.*;  
new ZonedDateTime(2018, 10, 31, 17, 0);  
new LocalDate(2018, 10, 31, 17, 0);  
ZonedDateTime.of(2018, 10, 31, 17, 0, 0, ZoneId.of("Europe/Berlin"));  
ZonedDateTime.of(2018, 10, 31, 17, 0, 0, 0,  
                 ZoneId.of("Europe/Berlin"));  
d.getDayOfWeek();  
d.getDay();  
DateTimeFormatter.of(FormatStyle.LONG).withLocale(locDE);  
d.format(DateTimeFormatter.ofLocalizedDateTime(FormatStyle.LONG)  
        .withLocale(locDE));
```

9. Given two files:

```
package rb;
public class Bundle extends java.util.ListResourceBundle {
    protected Object[][] getContents() {
        return new Object[][] { { "123", 456 } };
    }
}
```

```
package rb;
import java.util.*;
public class KeyValue {
    public static void main(String[] args) {
        ResourceBundle rb = ResourceBundle.getBundle("rb.Bundle",
            Locale.getDefault());
        // insert code here
    }
}
```

Which, inserted independently, will compile? (Choose all that apply.)

- A. Object obj = rb.getInteger("123");
- B. Object obj = rb.getInteger(123);
- C. Object obj = rb.getObject("123");
- D. Object obj = rb.getObject(123);
- E. object obj = rb.getString("123");
- F. object obj = rb.getString(123);

10. Given the following code fragment:

```
LocalDateTime now = LocalDateTime.of(2017,10,27,14,22,54,0);
DateTimeFormatter formatter =
    DateTimeFormatter.ofPattern("_____"); // L1
String formattedDateTime = now.format(formatter);
System.out.println("Formatted DateTime: " + formattedDateTime);
```

Which String inserted as an argument to
DateTimeFormatter.ofPattern() at // L1 will produce the output?
(Choose all that apply.)

Formatted DateTime: 2017-10-27 14:22:54

- A. "yyyy-MM-dd hh:mm:ss a"
- B. "yyyy-MM-dd hh:mm:ss"
- C "yyyy-mm-dd HH:MM:ss"
- D. "yyyy-MM-dd HH:mm:ss"
- E. "yyyy-MM-dd HH:mm:ss Z"

11. Given the following code fragment:

```
LocalDate d1 = LocalDate.of(2017, Month.NOVEMBER, 28);
System.out.print(d1 + ", ");
LocalDate d2 = d1.with(TemporalAdjusters.lastDayOfYear());
System.out.print(d2 + ", ");
LocalDate d3 = d1.plusDays(3).with(TemporalAdjusters.firstDayOfNextMonth());
System.out.print(d3 + ", ");
LocalDate d4 = d1.minusMonths(11).with(TemporalAdjusters.firstDayOfNextYear());
System.out.print(d4 + ", ");
LocalDate d5 = LocalDate.ofEpochDay(d1.plusDays(27).toEpochDay());
System.out.print(d5 + ", ");
LocalDate d6 = d1.minus(Period.ofDays(5));
System.out.println(d6);
```

What output will you see?

- A. 2017-11-28, 2017-12-31, 2017-12-01, 2017-01-01, 2017-12-25, 2017-11-23
 - B. 2017-11-28T00:00, 2017-12-31T00:00, 2017-12-01T00:00, 2017-01-01T00:00, 2017-12-25T00:00, 2017-11-23T00:00
 - C. 2017-11-28, 2017-12-31, 2018-01-01, 2017-01-01, 2017-12-25, 2017-11-23
 - D.**2017-11-28T00:00, 2017-12-31T00:00, 2018-01-01T00:00, 2017-01-01T00:00, 2017-12-25T00:00, 2017-11-23T00:00
 - E.2017-11-28, 2017-12-31, 2018-01-01, 2018-01-01, 2017-12-25, 2017-11-23
12. If it is 19:12:53 on October 27, 2017, in the US/Pacific Zone (which is GMT-8:00, summer time), then what does the following code fragment produce? (Choose all that apply.)
- ```
ZoneId zid = ZoneId.of("US/Eastern"); // GMT-5:00
Instant i = Instant.now();
ZonedDateTime zdt = i.atZone(zid);
System.out.println(zdt.format(
 DateTimeFormatter.ofLocalizedTime(FormatStyle.MEDIUM)));
```
- A. 10:12:53 PM
  - B. 20:12:53
  - C. 19:12:53
  - D. 7:12:53 PM
  - E. 2017-10-27 10:12:53 PM

## A SELF TEST ANSWERS

1.  F is correct. The string we are parsing has no time zone, so the parse will fail at runtime.  
 A, B, C, D, and E are incorrect based on the above. (OCP Objective 7.2)
2.  A, B, and E are correct. Each adds 1 hour and 5 minutes to t1 to make a

new `LocalTime` `t3`, which represents 10:05, the same time as `t2`. The `plus()` method takes an amount to add as a `long`, and a unit (**A**) or a `TemporalAmount` (**E**).

**C** and **D** are incorrect. **C** adds only 1 hour to make 10 AM instead of 10:05 AM. **D** generated a compile error because `LocalTime` does not have a `plusDays()` method. (OCP Objective 7.1 and 7.3)

3.  **E** is correct. `Period` is the correct type to measure a period of time in days.

**A, B, C, D**, and **F** are incorrect based on the above. (OCP Objective 7.1 and 7.3)

4.  **B** and **C** are correct. In both cases, we're creating an `Instant` from `nowZdt` and then creating a new `ZonedDateTime` from that `Instant`, representing the same time as `nowZdt`, in Berlin.

**A, D**, and **E** are incorrect. **A** is incorrect because, although you can create a new `ZonedDateTime` from an existing `ZonedDateTime` with `from()`, you can't change the zone when you do. **D** is incorrect because `withZoneId()` is not a valid method. **E** is almost correct, except that it is not precisely the same time as `nowZdt` because you're calling the `now()` method again, though it may only be slightly different (perhaps only a few nanoseconds). (OCP Objective 7.2)

5.  **C** is correct. We first create a `ZonedDateTime` for the UTC time with zone "z" (corresponding to GMT zone) and then create the equivalent `ZonedDateTime` for the San Juan, Argentina, zone.

**A, B, D**, and **E** are incorrect. **A** is missing the time zone on the UTC time. **B** includes incorrect arguments to the `LocalDateTime.of()` method. **D** has the incorrect time zone on the UTC time. **E** has the incorrect type for `totalitySanJuan`. (OCP Objective 7.2)

6.  **A, C, D**, and **E** are correct. The default `Locale` is irrelevant here since none of the choices use Italian. **A** is the default resource bundle. **C** and **D** use the language but not the country from the requested locale. **E** uses the exact match of the requested locale.

**B** is incorrect because the language code of `CA` does not match `en`. And `CA` isn't a valid language code. **F** is incorrect because the language code "`fr`" does not match `en`. Even though the country code of `CA` does match, the language code is more important. (OCP Objectives 12.2 and 12.3)

7.  **D** is correct. As is, the code finds resource bundle `Train_en_US.properties`, which uses `Train_en.properties` as a parent. Choice **D** finds resource bundle `Train_en_UK.properties`, which uses `Train_en.properties` as a parent.
- A, B, C, E, and F** are incorrect. **A** is incorrect because both the parent and child have the same property. In this scenario, the more specific one (child) gets used. **B** is incorrect because the default locale only gets used if the requested resource bundle can't be found. **C** is incorrect because it finds the resource bundle `Train_en.properties`, which does not have any "train" key. **E** is incorrect because there is no "ride" key once we delete the parent. **F** is incorrect based on the above. (OCP Objectives 12.2 and 12.3)

8. Answer:

```
import java.util.*;
import java.time.*;
import java.time.format.*;
public class DateHalloween {
 public static void main(String[] args) {
 ZonedDateTime d = ZonedDateTime.of(2018, 10, 31, 17, 0, 0, 0,
 ZoneId.of("Europe/Berlin"));
 Locale locDE = new Locale("de");
 DayOfWeek day = d.getDayOfWeek();
 String df = d.format(DateTimeFormatter
 .ofLocalizedDateTime(FormatStyle.LONG).withLocale(locDE));
 System.out.println(day + " " + df);
 }
}
```

Reminders: To create a `ZonedDateTime` with the `of()` method, you must include all portions of the date and time (including nanoseconds) and a zone. `DateTimeFormatter`

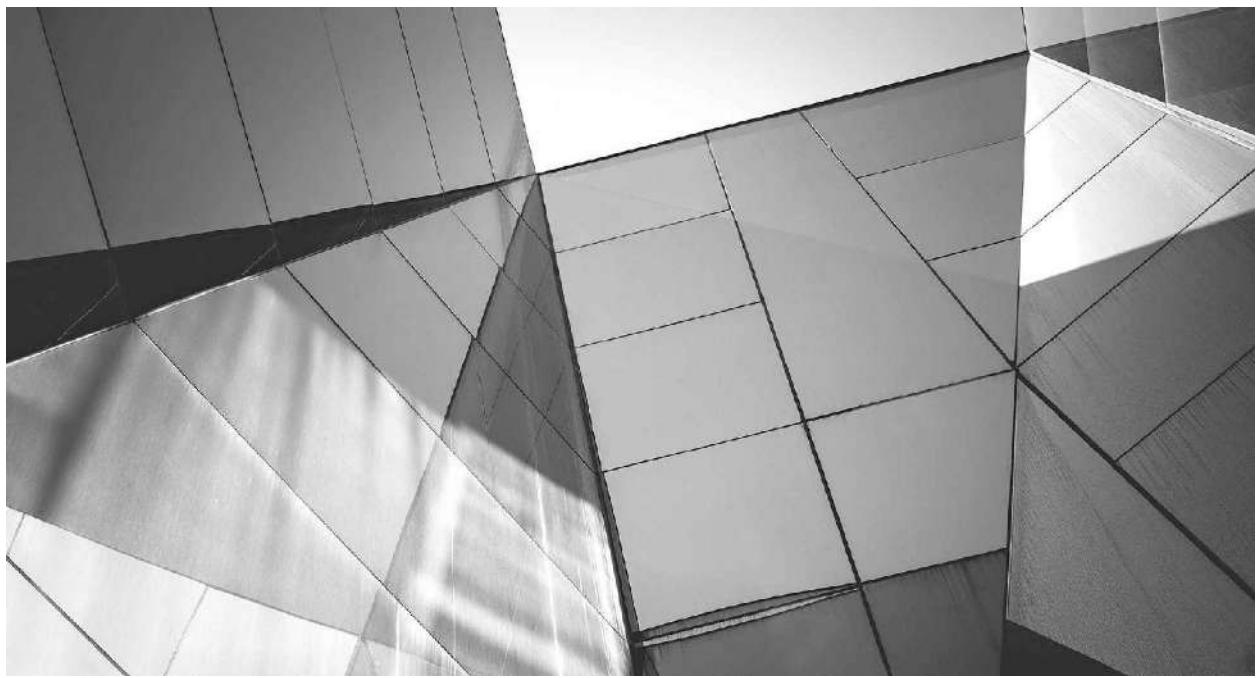
`.ofLocalizedDateTime()` returns a locale-specific date-time formatter, and `withLocale()` returns a copy of this formatter with a new locale. (OCP Objectives 7.2 and 12.1)

9.  **C** and **E** are correct. When getting a key from a resource bundle, the key must be a string. The returned result must be a string or an object. While that object may happen to be an integer, the API is still `getObject()`. **E** will throw a `ClassCastException` since 456 is not a string, but it will compile.  
 **A, B, D**, and **F** are incorrect because of the above. (OCP Objectives 12.2 and 12.3)
10.  **D** is correct; this string corresponds to the format shown in the output.  
 **A, B, C**, and **E** are incorrect. **A** uses `hh` for the hour, which will show 02 instead of 14 (that is, a 12-hour format instead of a 24-hour format), and displays the AM/PM at the end, which is great if we're using 12-hour format, but that's not what we're looking for. **B** results in 12-hour format instead of 24-hour format. **C** switches months and minutes. **E** requires a `ZonedDateTime` instead of a `LocalDateTime`, and using this `String` will throw a runtime exception when we try to format now with this formatter. (OCP Objectives 7.1 and 7.2)
11.  **C** is correct because of the below.  
 **A, B, D**, and **E** are incorrect. **B** and **D** show the time, and we are displaying `LocalDate` values that have no time associated with them. **A** has the incorrect value for d3, and **E** has the wrong value for d4. (OCP Objectives 7.1 and 7.3)

q

12.  **A** is correct. We first get the `zoneId` for "US/Eastern" time, which is GMT-5:00, and the locale to US. We then create an `Instant` for "now," which is 19:12:53 on October 27, 2017 (7:12:53 PM PDT, which is 10:12:53 PM EDT). We then create a `ZonedDateTime` from the `Instant`, using the `zoneId` for "US/Eastern" and format it using `DateTimeFormatter.ofLocalizedTime()`, which turns the `ZonedDateTime` into a `LocalTime` (dropping the date and zone information) and display it in the `MEDIUM` format style for the US locale, resulting in 10:12:53 PM. Format styles depend on local configuration, but we know this answer is correct because **B**, **C**, and **D** show the incorrect times, and **E** shows the date.  
 **B, C, D**, and **E** are incorrect. **B, C**, and **D** show the incorrect times, and

**E** shows the date, which we dropped when we formatted `zdt` to a localized time. (OCP Objectives 7.2)



# 5

## I/O and NIO

### CERTIFICATION OBJECTIVES

- Read and Write Data from the Console
  - Use BufferedReader, BufferedWriter, File, FileReader, FileWriter, FileInputStream, FileOutputStream, ObjectOutputStream, ObjectInputStream, and PrintWriter in the java.io Package
  - Use Path Interface to Operate on File and Directory Paths
  - Use Files Class to Check, Read, Delete, Copy, Move, Manage Metadata of a File or Directory
  - Use Stream API with NIO.2
-  Two-Minute Drill

#### Q&A Self Test

**I**/O (input/output) has been around since the beginning of Java. You could read and write files along with some other common operations. Then with Java 1.4, Java added more I/O functionality and cleverly named it NIO. That stands for “new I/O.” Don’t worry—you won’t be asked about those Java 1.4 additions on the exam.

The APIs prior to Java 7 still had a few limitations when you had to write applications that focused heavily on files and file manipulation. Trying to write a little routine listing all the files created in the past day within a directory tree would have given you some headaches. There was no support for navigating directory trees, and just reading attributes of a file was also quite hard. As of Java 7, this whole routine is fewer than 15 lines of code!

Now what to name yet another I/O API? The name “new I/O” was taken, and “new new I/O” would just sound silly. Since the Java 7 functionality was added to package names that began with `java.nio`, the new name was NIO.2. For the purposes of this chapter and the exam, NIO is shorthand for NIO.2.

Since NIO (or NIO.2 if you like) builds on the original I/O, some of those concepts are still tested on the exam in addition to the new parts. Fortunately, you won’t have to become a total I/O or NIO guru to do well on the exam. The intention of the exam team was to include just the basic aspects of these technologies, and in this chapter, we cover *more* than you’ll need to get through these objectives on the exam.

## CERTIFICATION OBJECTIVE

### File Navigation and I/O (OCP Objectives 8.1 and 8.2)

*8.1 Read and write data from the console.*

*8.2 Use BufferedReader, BufferedWriter, File, FileReader, FileWriter, FileInputStream, FileOutputStream, ObjectOutputStream, ObjectInputStream, and PrintWriter in the java.io package.*

I/O has had a strange history with the OCP certification. It was included in all the versions of the exam, up to and including 1.2, then removed from the 1.4 exam, reintroduced for Java 5, extended for Java 6, and extended still more for Java 7 and 8.

I/O is a huge topic in general, and the Java APIs that deal with I/O in one fashion or another are correspondingly huge. A general discussion of I/O could include topics such as file I/O, console I/O, thread I/O, high-performance I/O, byte-oriented I/O, character-oriented I/O, I/O filtering and wrapping, serialization, and more. Luckily for us, the I/O topics included in the Java 8 exam are fairly well restricted to file I/O for characters and serialization.

Here’s a summary of the I/O classes you’ll need to understand for the exam:

- **File** The API says that the `File` class is “an abstract representation of file and directory pathnames.” The `File` class isn’t used to actually read or write data; it’s used to work at a higher level, making new empty files, searching for files, deleting files, making directories, and working with

paths.

- **FileReader** This class is used to read character files. Its `read()` methods are fairly low-level, allowing you to read single characters, the whole stream of characters, or a fixed number of characters. `FileReaders` are usually *wrapped* by higher-level objects such as `BufferedReaders`, which improve performance and provide more convenient ways to work with the data.
- **BufferedReader** This class is used to make lower-level `Reader` classes like `FileReader` more efficient and easier to use. Compared to `FileReaders`, `BufferedReaders` read relatively large chunks of data from a file at once and keep this data in a buffer. When you ask for the next character or line of data, it is retrieved from the buffer, which minimizes the number of times that time-intensive file-read operations are performed. In addition, `BufferedReader` provides more convenient methods, such as `readLine()`, that allow you to get the next line of characters from a file.
- **FileWriter** This class is used to write to character files. Its `write()` methods allow you to write character(s) or strings to a file. `Filewriters` are usually *wrapped* by higher-level `writer` objects, such as `BufferedWriters` or `Printwriters`, which provide better performance and higher-level, more flexible methods to write data.
- **BufferedWriter** This class is used to make lower-level classes like `Filewriters` more efficient and easier to use. Compared to `Filewriters`, `BufferedWriters` write relatively large chunks of data to a file at once, minimizing the number of times that slow file-writing operations are performed. The `BufferedWriter` class also provides a `newLine()` method to create platform-specific line separators automatically.
- **PrintWriter** This class has been enhanced significantly in Java 5. Because of newly created methods and constructors (like building a `PrintWriter` with a `File` or a `String`), you might find that you can use `PrintWriter` in places where you previously needed a `writer` to be wrapped with a `FileWriter` and/or a `BufferedWriter`. New methods like `format()`, `printf()`, and `append()` make `Printwriters` quite flexible and powerful.
- **FileInputStream** This class is used to read bytes from files and can be used for binary as well as text. Like `FileReader`, the `read()` methods are low-level, allowing you to read single bytes, a stream of bytes, or a

fixed number of bytes. We typically use `FileInputStream` with higher-level objects such as `ObjectInputStream`.

- **FileOutputStream** This class is used to write bytes to files. We typically use `FileOutputStream` with higher-level objects such as `ObjectOutputStream`.
- **ObjectInputStream** This class is used to read an input stream and deserialize objects. We use `ObjectInputStream` with lower-level classes like `FileInputStream` to read from a file. `ObjectInputStream` works at a higher level so that you can read objects rather than characters or bytes. This process is called *deserialization*.



***Classes with “Stream” in their name are used to read and write bytes, and Readers and Writers are used to read and write characters.***

- **ObjectOutputStream** This class is used to write objects to an output stream and is used with classes like `FileOutputStream` to write to a file. This is called *serialization*. Like `ObjectInputStream`, `ObjectOutputStream` works at a higher level to write objects, rather than characters or bytes.
- **Console** This Java 6 convenience class provides methods to read input from the console and write formatted output to the console.

## Creating Files Using the `File` Class

Objects of type `File` are used to represent the actual files (but not the data in the files) or directories that exist on a computer's physical disk. Just to make sure we're clear, when we talk about an object of type `File`, we'll say `File`, with a capital *F*. When we're talking about what exists on a hard drive, we'll call it a file with a lowercase *f* (unless it's a variable name in some code). Let's start with a few basic examples of creating files, writing to them, and reading from them. First, let's create a new file and write a few lines of data to it:

```
import java.io.*; // The section 8 objectives
// focus on classes from
// java.io

class Writer1 {
 public static void main(String [] args) {
 File file = new File("fileWrite1.txt"); // There's no
 // file yet!
 }
}
```

If you compile and run this program, when you look at the contents of your current directory, you'll discover absolutely no indication of a file called `fileWrite1.txt`. When you make a new instance of the class `File`, *you're not yet making an actual file; you're just creating a filename*. Once you have a `File object`, there are several ways to make an actual file. Let's see what we can do with the `File` object we just made:

```
import java.io.*;

class Writer1 {
 public static void main(String [] args) {
 try { // warning: exceptions possible
 boolean newFile = false;
 File file = new File // it's only an object
 ("fileWriter1.txt");
 System.out.println(file.exists()); // look for a real file
 newFile = file.createNewFile(); // maybe create a file!
 System.out.println(newFile); // already there?
 System.out.println(file.exists()); // look again
 } catch(IOException e) { }
 }
}
```

This produces the output

```
false
true
true
```

And also produces an empty file in your current directory. If you run the code a *second* time, you get the output

```
true
false
true
```

Let's examine these sets of output:

- **First execution** The first call to `exists()` returned `false`, which we expected...remember, `new File()` doesn't create a file on the disk! The `createNewFile()` method created an actual file and returned `true`, indicating that a new file was created and that one didn't already exist. Finally, we called `exists()` again, and this time it returned `true`, indicating the file existed on the disk.
- **Second execution** The first call to `exists()` returns `true` because we built the file during the first run. Then the call to `createNewFile()` returns `false` since the method didn't create a file this time through. Of course, the last call to `exists()` returns `true`.

A couple of other new things happened in this code. First, notice that we had to put our file creation code in a try/catch. This is true for almost all of the file I/O code you'll ever write. I/O is one of those inherently risky things. We're keeping it simple for now and ignoring the exceptions, but we still need to follow the handle-or-declare rule, since most I/O methods declare checked exceptions. We'll talk more about I/O exceptions later. We used a couple of `File`'s methods in this code:

## exam watch

*Remember, the exam creators are trying to jam as much code as they can into a small space, so in the previous example, instead of these three lines of code:*

```
boolean newFile = false;
...
newFile = file.createNewFile();
System.out.println(newFile);
```

*you might see something like the following single line of code, which is a bit harder to read, but accomplishes the same thing:*

```
System.out.println(file.createNewFile());
```

- **boolean exists()** This method returns `true` if it can find the actual file.
- **boolean createNewFile()** This method creates a new file if it doesn't already exist.

## Using `FileWriter` and `FileReader`

In practice, you probably won't use the `FileWriter` and `FileReader` classes without wrapping them (more about "wrapping" very soon). That said, let's go ahead and do a little "naked" file I/O:

```
import java.io.*;
class Writer2 {
 public static void main(String [] args) {
 char[] in = new char[50]; // to store input
 int size = 0;
 try {
 File file = new File(// just an object
 "fileWrite2.txt");
 FileWriter fw =
 new FileWriter(file); // create an actual file
 // & a FileWriter obj
 fw.write("howdy\nfolks\n"); // write characters to
 // the file
 fw.flush(); // flush before closing
 fw.close(); // close file when done
 FileReader fr =
 new FileReader(file); // create a FileReader
 // object
 size = fr.read(in); // read the whole file!
 System.out.print(size + " "); // how many characters read
 for(char c : in) // print the array
 System.out.print(c);
 fr.close(); // again, always close
 } catch(IOException e) { }
 }
}
```

which produces the output:

```
12 howdy
folks
```

Here's what just happened:

1. `FileWriter fw = new FileWriter(file)` did three things:
  - a. It created a `FileWriter` reference variable, `fw`.
  - b. It created a `FileWriter` object and assigned it to `fw`.
  - c. It created an actual empty file out on the disk (and you can prove it).
2. We wrote 12 characters to the file with the `write()` method, and we did a `flush()` and a `close()`.
3. We made a new `FileReader` object, which also opened the file on disk for reading.
4. The `read()` method read the whole file, a character at a time, and put it into the `char[] in`.
5. We printed out the number of characters we read in `size`, and we looped through the `in` array, printing out each character we read, and then we closed the file.

Before we go any further, let's talk about `flush()` and `close()`. When you write data out to a stream, some amount of buffering will occur, and you never know for sure exactly when the last of the data will actually be sent. You might perform many write operations on a stream before closing it, and invoking the `flush()` method guarantees that the last of the data you thought you had already written actually gets out to the file. Whenever you're done using a file, either reading it or writing to it, you should invoke the `close()` method. When you are doing file I/O, you're using expensive and limited operating system resources, and so when you're done, invoking `close()` will free up those resources.

Now, back to our last example. This program certainly works, but it's painful in a couple of different ways:

1. When we were writing data to the file, we manually inserted line separators (in this case `\n`) into our data.
2. When we were reading data back in, we put it into a character array. It

being an array and all, we had to declare its size beforehand, so we'd have been in trouble if we hadn't made it big enough! We could have read the data in one character at a time, looking for the end of the file after each `read()`, but that's pretty painful too.

Because of these limitations, we'll typically want to use higher-level I/O classes like `BufferedWriter` or `BufferedReader` in combination with `FileWriter` or `FileReader`.

## Using `FileInputStream` and `FileOutputStream`

Using `FileInputStream` and `FileOutputStream` is similar to using `FileReader` and `FileWriter`, except you're working with byte data instead of character data. That means you can use `FileInputStream` and `FileOutputStream` to read and write binary data as well as text data.

We've rewritten the previous example to use `FileInputStream` and `FileOutputStream`; the code does exactly the same thing, but because we're working with bytes instead of characters, we made a few small modifications, which we'll point out:

```
import java.io.*;
class Writer3 {
 public static void main(String [] args) {
 byte[] in = new byte[50]; // bytes, not chars!
 int size = 0;
 FileOutputStream fos = null;
 FileInputStream fis = null;
 File file = new File("fileWrite3.txt");
 try {
 fos = new FileOutputStream(file); // create a FileOutputStream
 String s = "howdy\nfolks\n";
 fos.write(s.getBytes("UTF-8")); // write characters (bytes)
 // to the file
 fos.flush(); // flush before closing
 fos.close(); // close file when done

 fis = new FileInputStream(file); // create a FileInputStream
 size = fis.read(in); // read the file into in
 System.out.print(size + " ");
 for(byte b : in) { // print the array
 System.out.print((char)b);
 }
 fis.close(); // again, always close
 } catch(IOException e) {
 e.printStackTrace();
 }
 }
}
```

As you can see, this example is almost exactly like the previous one, except we’re using bytes rather than chars. That means we convert the `String` we write to the file to bytes for the `write()` method, and when we read, we read into an array of bytes, rather than an array of chars, and convert each byte to a char before we print it.

And like the previous example, this one is painful in the same ways. You’ll typically find you use higher-level I/O classes like `ObjectInputStream` and `ObjectOutputStream`, rather than `FileInputStream` and `FileOutputStream`, unless you really need to read binary data byte by byte. We talk about `ObjectInputStream` and `ObjectOutputStream` in the section on serialization later in the chapter.

## Combining I/O Classes

Java’s entire I/O system was designed around the idea of using several classes in combination. Combining I/O classes is sometimes called *wrapping* and sometimes called *chaining*. The `java.io` package contains about 50 classes, 10 interfaces, and 15 exceptions. Each class in the package has a specific purpose (i.e., highly specialized), and the classes are designed to be combined with each other in countless ways to handle a wide variety of situations.

When it’s time to do some I/O in real life, you’ll undoubtedly find yourself poring over the `java.io` API, trying to figure out which classes you’ll need and how to hook them together. For the exam, you’ll need to do the same thing, but Oracle artificially reduced the API (phew!). In terms of studying for Exam Objective 8.2, we can imagine that the entire `java.io` package—consisting of the classes listed in Exam Objective 8.2 and summarized in [Table 5-1](#)—is our mini I/O API.

**TABLE 5-1** `java.io` Mini API

| java.io Class    | Extends From | Key Constructor(s)                                                     | Arguments                                                                                               | Key Methods |
|------------------|--------------|------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------|-------------|
| File             | Object       | File, String<br>String<br>String, String                               | createNewFile()<br>delete()<br>exists()<br>isDirectory()<br>isFile()<br>list()<br>mkdir()<br>renameTo() |             |
| FileWriter       | Writer       | File<br>String                                                         | close()<br>flush()<br>write()                                                                           |             |
| BufferedWriter   | Writer       | Writer                                                                 | close()<br>flush()<br>newLine()<br>write()                                                              |             |
| PrintWriter      | Writer       | File (as of Java 5)<br>String (as of Java 5)<br>OutputStream<br>Writer | close()<br>flush()<br>format(), printf()<br>print(), println()<br>write()                               |             |
| FileOutputStream | OutputStream | File<br>String                                                         | close()<br>write()                                                                                      |             |
| FileReader       | Reader       | File<br>String                                                         | read()                                                                                                  |             |
| BufferedReader   | Reader       | Reader                                                                 | read()<br>readLine()                                                                                    |             |
| FileInputStream  | InputStream  | File<br>String                                                         | read()<br>close()                                                                                       |             |

Now let's say we want to find a less painful way to write data to a file and read the file's contents back into memory. Starting with the task of writing data to a file, here's a process for determining what classes we'll need and how we'll hook them together:

1. We know that ultimately we want to hook to a `File` object. So whatever other class or classes we use, one of them must have a constructor that takes an object of type `File`.
2. Find a method that sounds like the most powerful, easiest way to accomplish the task. When we look at [Table 5-1](#) we can see that `BufferedWriter` has a `newLine()` method. That sounds a little better than having to manually embed a separator after each line, but if we look further, we see that `PrintWriter` has a method called `println()`. That sounds like the easiest approach of all, so we'll go with it.
3. When we look at `PrintWriter`'s constructors, we see that we can build a `PrintWriter` object if we have an object of type `File`, so all we need to do to create a `PrintWriter` object is the following:

```
File file = new File("fileWrite2.txt"); // create a File
PrintWriter pw = new PrintWriter(file); // pass file to
 // the PrintWriter
 // constructor
```

Okay, time for a pop quiz. Prior to Java 5, `PrintWriter` did not have constructors that took either a `String` or a `File`. If you were writing some I/O code in Java 1.4, how would you get a `PrintWriter` to write data to a file? Hint: You can figure this out by studying the mini I/O API in [Table 5-1](#).

Here's one way to go about solving this puzzle: First, we know that we'll create a `File` object on one end of the chain and that we want a `PrintWriter` object on the other end. We can see in [Table 5-1](#) that a `PrintWriter` can also be built using a `Writer` object. Although `Writer` isn't a *class* we see in the table, we can see that several other classes extend `Writer`, which, for our purposes, is just as good; any class that extends `Writer` is a candidate. Looking further, we can see that `FileWriter` has the two attributes we're looking for:

- It can be constructed using a `File`.
- It extends `Writer`.

Given all of this information, we can put together the following code (remember, this is a Java 1.4 example):

```
File file = new File("fileWrite2.txt"); // create a File object
FileWriter fw = new FileWriter(file); // create a FileWriter
// that will send its
// output to a File

PrintWriter pw = new PrintWriter(fw); // create a PrintWriter
// that will send its
// output to a Writer

pw.println("howdy"); // write the data
pw.println("folks");
```

At this point, it should be fairly easy to put together the code to more easily read data from the file back into memory. Again, looking through the table, we see a method called `readLine()` that sounds like a much better way to read data. Going through a similar process, we get the following code:

```
File file =
 new File("fileWrite2.txt"); // create a File object AND
 // open "fileWrite2.txt"

FileReader fr =
 new FileReader(file); // create a FileReader to get
 // data from 'file'

BufferedReader br =
 new BufferedReader(fr); // create a BufferReader to
 // get its data from a Reader

String data = br.readLine(); // read some data
```



**You're almost certain to encounter exam questions that test your knowledge of how I/O classes can be chained. If you're not totally clear on this last section, we recommend using [Table 5-1](#) as a reference and writing code to experiment with which chaining combinations are legal and which are illegal.**

## Working with Files and Directories

Earlier, we touched on the fact that the `File` class is used to create files and directories. In addition, `File`'s methods can be used to delete files, rename files, determine whether files exist, create temporary files, change a file's attributes, and differentiate between files and directories. A point that is often confusing is that an object of type `File` is used to represent *either a file or a directory*. We'll talk about both cases next.

We saw earlier that the statement

```
File file = new File("foo");
```

always creates a `File` object and then does one of two things:

1. If "foo" does NOT exist, no actual file is created.
2. If "foo" *does* exist, the new `File` object refers to the existing file.

Notice that `File file = new File("foo");` NEVER creates an actual file. There are two ways to create a file:

1. Invoke the `createNewFile()` method on a `File` object. For example:

```
File file = new File("foo"); // no file yet
file.createNewFile(); // make a file, "foo" which
 // is assigned to 'file'
```

2. Create a `Writer` or a `Stream`. Specifically, create a `FileWriter`, a `PrintWriter`, or a `FileOutputStream`. Whenever you create an instance of one of these classes, you automatically create a file, unless one already exists, for instance:

```
File file = new File("foo"); // no file yet
PrintWriter pw =
 new PrintWriter(file); // make a PrintWriter object AND
 // make a file, "foo" to which
 // 'file' is assigned, AND assign
 // 'pw' to the PrintWriter
```

Creating a directory is similar to creating a file. Again, we'll use the convention of referring to an object of type `File` that represents an actual directory as a `Directory` object, with a capital *D* (even though it's of type `File`). We'll call an actual directory on a computer a `directory`, with a small *d*. Phew! As with creating a file, creating a directory is a two-step process; first we create a `Directory` (`File`) object; then we create an actual directory using the following `mkdir()` method:

```
File myDir = new File("mydir"); // create an object
myDir.mkdir(); // create an actual directory
```

Once you've got a directory, you put files into it and work with those files:

```
File myFile = new File(myDir, "myFile.txt");
myFile.createNewFile();
```

This code is making a new file in a subdirectory. Since you provide the subdirectory to the constructor, from then on, you just refer to the file by its reference variable. In this case, here's a way that you could write some data to the file `myFile`:

```
PrintWriter pw = new PrintWriter(myFile);
pw.println("new stuff");
pw.flush();
pw.close();
```

Be careful when you're creating new directories! As we've seen, constructing a `Writer` or a `Stream` will often create a file for you automatically if one doesn't exist, but that's not true for a directory.

```
File myDir = new File("mydir");
// myDir.mkdir(); // call to mkdir() omitted!
File myFile = new File(
 myDir, "myFile.txt");
myFile.createNewFile(); // exception if no mkdir!
```

This will generate an exception that looks something like

```
java.io.IOException: No such file or directory
```

You can refer a `File` object to an existing file or directory. For example, assume we already have a subdirectory called `existingDir` in which an existing file `existingDirFile.txt` resides. This file contains several lines of text. When we run the following code:

```
File existingDir = new File("existingDir"); // assign a dir
System.out.println(existingDir.isDirectory());

File existingDirFile = new File(
 existingDir, "existingDirFile.txt"); // assign a file
System.out.println (existingDirFile.isFile());

FileReader fr = new FileReader(existingDirFile);
BufferedReader br = new BufferedReader(fr); // make a Reader

String s;
while((s = br.readLine()) != null) // read data
 System.out.println(s);

br.close();
```

the following output will be generated:

```
true
true
existing sub-dir data
line 2 of text
line 3 of text
```

Take special note of what the `readLine()` method returns. When there is no more data to read, `readLine()` returns a `null`—this is our signal to stop reading the file. Also, notice that we didn't invoke a `flush()` method. When reading a file, no flushing is required, so you won't even find a `flush()` method in a Reader kind of class.

In addition to creating files, the `File` class lets you do things like renaming and deleting files. The following code demonstrates a few of the most common

ins and outs of deleting files and directories (via `delete()`) and renaming files and directories (via `renameTo()`):

```
File delDir = new File("deldir"); // make a directory
delDir.mkdir();

File delFile1 = new File(
 delDir, "delFile1.txt"); // add file to directory
delFile1.createNewFile();

File delFile2 = new File(
 delDir, "delFile2.txt"); // add file to directory
delFile2.createNewFile();
delFile1.delete(); // delete a file
System.out.println("delDir is "
 + delDir.delete()); // attempt to delete
 // the directory

File newName = new File(
 delDir, "newName.txt"); // a new object
delFile2.renameTo(newName); // rename file

File newDir = new File("newDir"); // rename directory
delDir.renameTo(newDir);
```

This outputs

```
delDir is false
```

and leaves us with a directory called `newDir` that contains a file called `newName.txt`. Here are some rules that we can deduce from this result:

- **delete()** You can't delete a directory if it's not empty, which is why the invocation `delDir.delete()` failed.
- **renameTo()** You must give the existing `File` object a valid new `File` object with the new name that you want. (If `newName` had been `null`, we would have gotten a `NullPointerException`.)
- **renameTo()** It's okay to rename a directory, even if it isn't empty.

There's a lot more to learn about using the `java.io` package, but as far as the exam goes, we only have one more thing to discuss, and that is how to search for a file. Assuming we have a directory named `searchThis` that we want to search through, the following code uses the `File.list()` method to create a `String` array of files and directories. We then use the enhanced `for` loop to iterate through and print.

```
String[] files = new String[100];
File search = new File("searchThis");
files = search.list(); // create the list

for(String fn : files) // iterate through it
 System.out.println("found " + fn);
```

On our system, we got the following output:

```
found dir1
found dir2
found dir3
found file1.txt
found file2.txt
```

Your results will almost certainly be different!

In this section, we've scratched the surface of what's available in the `java.io` package. Entire books have been written about this package, so we're obviously covering only a very small (but frequently used) portion of the API. On the other hand, if you understand everything we've covered in this section, you will be in

great shape to handle any `java.io` questions you encounter on the exam, except for the `Console` class, which we'll cover next.

## The `java.io.Console` Class

Java 6 added the `java.io.Console` class. In this context, the *console* is the physical device with a keyboard and a display (like your Mac or PC). If you're running Java SE 6 from the command line, you'll typically have access to a `console` object, to which you can get a reference by invoking `System.console()`. Keep in mind that it's possible for your Java program to be running in an environment that doesn't have access to a `console` object, so be sure that your invocation of `System.console()` actually returns a valid `console` reference and not null.

The `Console` class makes it easy to accept input from the command line, both echoed and nonechoed (such as a password), and makes it easy to write formatted output to the command line. It's a handy way to write test engines for unit testing or if you want to support a simple but secure user interaction and you don't need a GUI.

On the input side, the methods you'll have to understand are `readLine` and `readPassword`. The `readLine` method returns a string containing whatever the user keyed in—that's pretty intuitive. However, the `readPassword` method doesn't return a string; it returns a character array. Here's the reason for this: Once you've got the password, you can verify it and then absolutely remove it from memory. If a string was returned, it could exist in a pool somewhere in memory, and perhaps some nefarious hacker could find it.

Let's take a look at a small program that uses a `console` to support testing another class:

```
import java.io.Console;

public class NewConsole {
 public static void main(String[] args) {
 String name = "";
 Console c = System.console(); // #1: get a Console
 char[] pw;
 pw = c.readPassword("%s", "pw: "); // #2: return a char[]
 for(char ch: pw)
 c.format("%c ", ch); // #3: format output
 c.format("\n");

 MyUtility mu = new MyUtility();
 while(true) {
 name = c.readLine("%s", "input?: "); // #4: return a String
 c.format("output: %s \n", mu.doStuff(name));
 }
 }
}
```

```
class MyUtility { // #5: class to test
 String doStuff(String arg1) {
 // stub code
 return "result is " + arg1;
 }
}
```

Let's review this code:

- At line 1, we get a new `Console` object. Remember that we can't say this:

```
Console c = new Console();
```

- At line 2, we invoke `readPassword`, which returns a `char[]`, not a string. You'll notice when you test this code that the password you enter isn't echoed on the screen.
- At line 3, we're just manually displaying the password you keyed in, separating each character with a space. Later on in this chapter, you'll read about the `format()` method, so stay tuned.
- At line 4, we invoke `readLine`, which returns a string.
- At line 5 is the class that we want to test. We recommend that you use something like `NewConsole` to test the concepts that you're learning.

The `Console` class has more capabilities than are covered here, but if you understand everything discussed so far, you'll be in good shape for the exam.

## CERTIFICATION OBJECTIVE

### Files, Path, and Paths (OCP Objectives 9.1 and 9.2)

9.1 Use `Path` interface to operate on file and directory paths.

9.2 Use `Files` class to check, read, delete, copy, move, manage metadata of a file or directory.

The OCP 8 exam has two sections devoted to I/O. The previous section Oracle refers to as “Java I/O Fundamentals” (which we’ve referred to as the 8.x objectives), and it was focused on the `java.io` package. Now we’re going to look at the set of objectives Oracle calls “Java File I/O (NIO.2),” whose specific objectives we’ll refer to as 9.x. The term *NIO.2* is a bit loosely defined, but most people (and the exam creators) define NIO.2 as being the key new features introduced in Java 7 that reside in two packages:

- `java.nio.file`
- `java.nio.file.attribute`

We’ll start by looking at the important classes and interfaces in the `java.nio.file` package, and then we’ll move to the `java.nio.file.attribute` package later in the chapter.

As you read earlier in the chapter, the `File` class represents a file or directory at a high level. NIO.2 adds three new central classes that you’ll need to understand well for the exam:

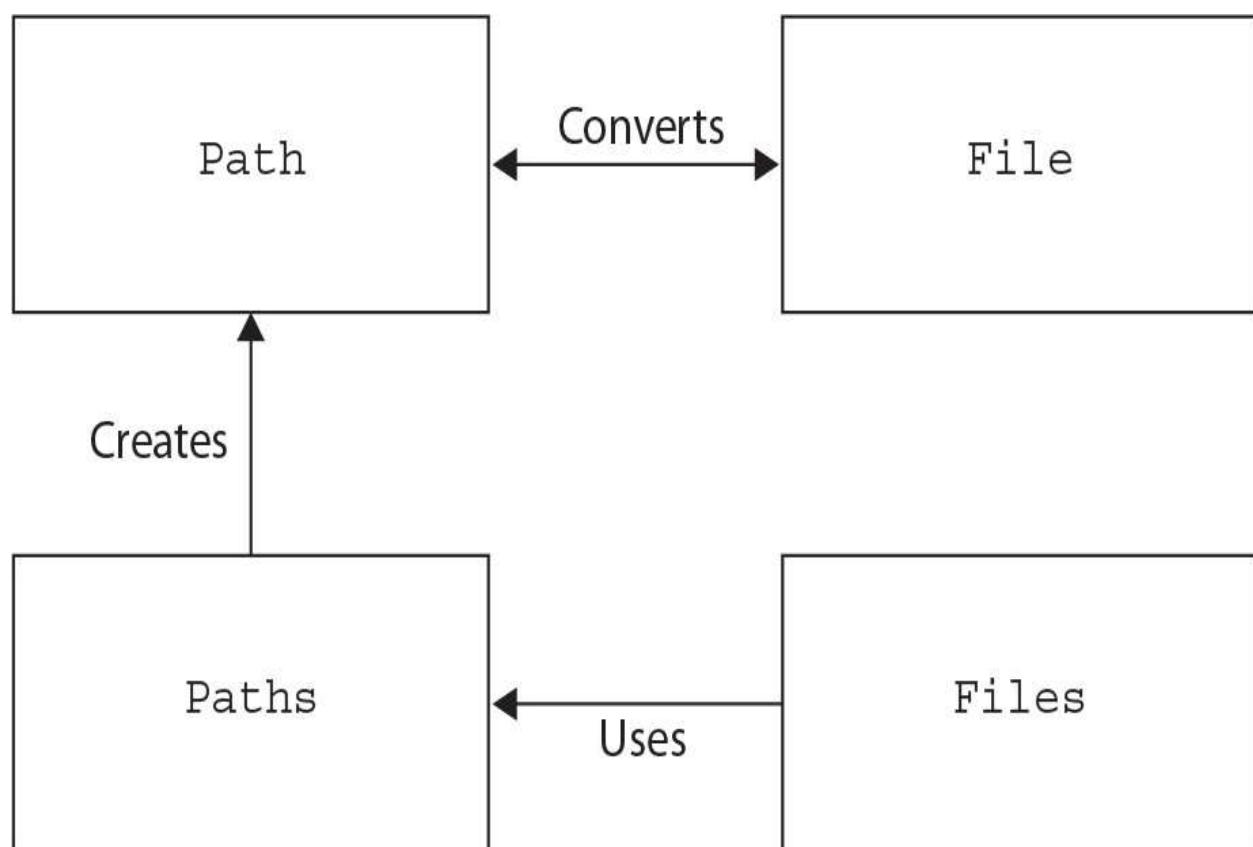
- **Path** This interface replaces `File` as the representation of a file or a directory when working in NIO.2. It is a lot more powerful than a `File`.
- **Paths** This class contains static methods that create `Path` objects.
- **Files** This class contains static methods that work with `Path` objects. You’ll find basic operations in here like copying or deleting files.

The interface `java.nio.file.Path` is one of the key classes of file-based I/O under NIO.2. Just like the good old `java.io.File`, a `Path` represents only a location in the file system, like `C:\java\workspace\ocpj7` (a Windows directory) or `/home/nblack/docs` (the `docs` directory of user `nblack` on UNIX). When you create a `Path` to a new file, that file does not exist until you actually create the file using `Files.createFile(Path target)`. The `Files` utility class will be covered in depth in the next section.



***The difference between `File`, `Files`, `Path`, and `Paths` is really important. Read carefully on the exam. A one-letter difference can mean a big difference in what the class does.***

Let's take a look at these relationships another way. The `Paths` class is used to create a class implementing the `Path` interface. The `Files` class uses `Path` objects as parameters. All three of these were introduced in Java 7. Then there is the `File` class. It's been around since the beginning. `File` and `Path` objects know how to convert to the other. This lets any older code interact with the new APIs in `Files`. But notice what is missing. In the figure, there is no line between `File` and `Files`. Despite the similarity in name, these two classes do not know about each other.



To make sure you know the difference between these key classes backward and forward, make sure you can fill in the four rightmost columns in [Table 5-2](#).

**TABLE 5-2** Comparing the Core Classes

|                              | File           | Files          | Path      | Paths          |
|------------------------------|----------------|----------------|-----------|----------------|
| Existed in Java 6?           | Yes            | No             | No        | No             |
| Concrete class or interface? | Concrete class | Concrete class | Interface | Concrete class |
| Create using "new"           | Yes            | No             | No        | No             |
| Contains only static methods | No             | Yes            | No        | Yes            |

## Creating a Path

A `Path` object can be easily created by using the `get` methods from the `Paths` helper class. Remember you are calling `Paths.get()` and not `Path.get()`. If you don't remember why, study the last section some more. It's important to have this down cold.

Taking a look at two simple examples, we have:

```
Path p1 = Paths.get("/tmp/file1.txt"); // on UNIX
Path p2 = Paths.get("c:\\temp\\test"); // On Windows
```

The actual method we just called is `Paths.get(String first, String... more)`. This means we can write it out by separating the parts of the path.

```
Path p3 = Paths.get("/tmp", "file1.txt"); // same as p1
Path p4 = Paths.get("c:", "temp", "test"); // same as p2
Path p5 = Paths.get("c:\\temp", "test") ; // also same as p2
```

As you can see, you can separate out folder and filenames as much or as little as you want when calling `Paths.get()`. For Windows, that is particularly cool because you can make the code easier to read by getting rid of the backslash and

escape character.

Be careful when creating paths. The previous examples are absolute paths since they begin with the root (/ on UNIX or c: on Windows). When you don't begin with the root, the Path is considered a relative path, which means Java looks from the current directory. Which file1.txt do you think p6 has in mind?

```
Path p6 = Paths.get("tmp", "file1.txt"); // relative path - NOT same as p1
/ (root)
|-- tmp
| - file1.txt
| - tmp
| - file1.txt
```

It depends. If the program is run from the root, it is the one in /tmp/file1.txt. If the program is run from /tmp, it is the one in /tmp/tmp/file1.txt. If the program is run from anywhere else, p6 refers to a file that does not exist.

One more thing to watch for. If you are on Windows, you might deal with a URL that looks like file:///c:/temp. The file:// is a protocol just like http:// is. This syntax allows you to browse to a folder in Internet Explorer. Your program might have to deal with such a String that a user copied/pasted from the browser. No problem, right? We learned to code:

```
Path p = Paths.get("file:///c:/temp/test");
```

Unfortunately, this doesn't work, and you get an exception about the colon being invalid that looks something like this:

```
Exception in thread "main" java.nio.file.InvalidPathException:
Illegal char <:>
at index 4: file:///c:/temp
```

Paths provides another method that solves this problem. Paths.get(URI uri) lets you (indirectly) convert the String to a URI (Uniform Resource Identifier) before trying to create a Path:

```
Path p = Paths.get(URI.create("file:///C:/temp"));
```

The last thing you should know is that the `Paths.get()` method we've been discussing is really a shortcut. You won't need to code the longer version, but it is good to understand what is going on under the hood. First, Java finds out what the default file system is. For example, it might be `WindowsFileSystemProvider`. Then Java gets the path using custom logic for that file system. Luckily, this all goes on without us having to write any special code or even think about it.

```
Path short = Paths.get("c:", "temp");
Path longer = FileSystems.getDefault() // get default file system
 .getPath("c:", "temp"); // then get the Path
```

Now that you know how to create a `Path` instance, you can manipulate it in various ways. We'll get back to that in a bit.



**As far as the exam is concerned, `Paths.get()` is how to create a `Path` initially. There is another way that is useful when working with code that was written before Java 7:**

```
Path convertedPath = file.toPath();
File convertedFile = path.toFile();
```

**If you are updating older code that uses `File`, you can convert it to a `Path` and start calling the new classes. And if your newer code needs to call older code, it can convert back to a `File`.**

## Creating Files and Directories

With I/O, we saw that a file doesn't exist just because you have a `File` object. You have to call `createNewFile()` to bring the file into existence and `exists()` to check if it exists. Rewriting the example from earlier in the chapter to use NIO.2 methods, we now have:

```
Path path = Paths.get("fileWrite1.txt"); // it's only an object
System.out.println(Files.exists(path)); // look for a real file
Files.createFile(path); // create a file!
System.out.println(Files.exists(path)); // look again
```

NIO.2 has equivalent methods with two differences:

- You call static methods on `Files` rather than instance methods on `File`.
- Method names are slightly different.

See [Table 5-3](#) for the mapping between old class/method names and new ones. You can still continue to use the older I/O approach if you happen to be dealing with `File` objects.

**TABLE 5-3** I/O vs. NIO.2

| Description                                                  | I/O Approach                                                   | NIO.2 Approach                                                             |
|--------------------------------------------------------------|----------------------------------------------------------------|----------------------------------------------------------------------------|
| Create an empty file                                         | <pre>File file = new File("test"); file.createNewFile();</pre> | <pre>Path path = Paths.get("test"); Files.createFile(path);</pre>          |
| Create an empty directory                                    | <pre>File file = new File("dir"); file.mkdir()</pre>           | <pre>Path path = Paths.get("dir"); Files.createDirectory(path);</pre>      |
| Create a directory, including any missing parent directories | <pre>File file = new File("/a/b/c"); file.mkdirs();</pre>      | <pre>Path path = Paths.get("/a/b/c"); Files.createDirectories(path);</pre> |
| Check if a file or directory exists                          | <pre>File file = new File("test"); file.exists();</pre>        | <pre>Path path = Paths.get("test"); Files.exists(path);</pre>              |



**The method *Files.notExists()* supplements *Files.exists()*. In some incredibly rare situations, Java won't have enough permissions to know whether the file exists. When this happens, both methods return false.**

You can also create directories in Java. Suppose we have a directory named /java and we want to create the file /java/source/directory/Program.java.

We could do this one at a time:

```
Path path1 = Paths.get("/java/source");
Path path2 = Paths.get("/java/source/directory");
Path file = Paths.get("/java/source/directory/Program.java");
Files.createDirectory(path1); // create first level of directory
Files.createDirectory(path2); // create second level of directory
Files.createFile(file); // create file
```

Or we could create all the directories in one go:

```
Files.createDirectories(path2); // create all levels of directories
Files.createFile(file); // create file
```

Although both work, the second is clearly better if you have a lot of directories to create. And remember that the directory needs to exist by the time the file is created.

## Copying, Moving, and Deleting Files

We often copy, move, or delete files when working with the file system. Up until Java 7, this was hard to do. Now, however, each is one line. Let's look at some examples:

```
Path source = Paths.get("/temp/test1.txt"); // exists
Path target = Paths.get("/temp/test2.txt"); // doesn't yet exist
Files.copy(source, target); // now two copies of the file
Files.delete(target); // back to one copy
Files.move(source, target); // still one copy
```

This is all pretty self-explanatory. We copy a file, delete the copy, and then move the file. Now, let's try another example:

```
Path one = Paths.get("/temp/test1.txt"); // exists
Path two = Paths.get("/temp/test2.txt"); // exists
Path targ = Paths.get("/temp/test23.txt"); // doesn't yet exist
Files.copy(one, targ); // now two copies of the file
Files.copy(two, targ); // oops,
 // FileAlreadyExistsException
```

Java sees it is about to overwrite a file that already exists. Java doesn't want us to lose the file, so it "asks" if we are sure by throwing an exception. `copy()` and `move()` actually take an optional third parameter—zero or more `CopyOptions`. The most useful option you can pass is

`StandardCopyOption.REPLACE_EXISTING`.

```
Files.copy(two, target, // ok. You know what
 StandardCopyOption.REPLACE_EXISTING); // you are doing
```

We have to think about whether a file exists when deleting the file too. Let's say we wrote this test code:

```
Path path = Paths.get("/java/out.txt");
try {
 methodUnderTest(); // might throw an exception
 Files.createFile(path); // file only gets created
 // if methodUnderTest() succeeds
} finally {
 Files.delete(path); // NoSuchElementException if no file
}
```

We don't know whether `methodUnderTest` works properly yet. If it does, the code works fine. If it throws an exception, we never create the file and `Files.delete()` throws a `NoSuchFileException`. This is a problem, as we only want to delete the file if it was created so we aren't leaving stray files around.

There is an alternative. `Files.deleteIfExists(path)` returns true and deletes the file only if it exists. If not, it just quietly returns false. Most of the time, you can ignore this return value. You just want the file to not be there. If it never existed, mission accomplished.



**If you have to work on pre-Java 7 code, you can use the `FileUtils` class in Apache Commons IO (<http://commons.apache.org/io>). It has methods similar to many of the copy, move, and delete methods that are now built into Java.**

To review, **Table 5-4** lists the methods on `Files` that you are likely to come across on the exam. Luckily, the exam doesn't expect you to know all 30 methods in the API. The important thing to remember is to check the `Files` JavaDoc when you find yourself dealing with files.

**TABLE 5-4** `Files` Methods

| Method                                                     | Description                                                      |
|------------------------------------------------------------|------------------------------------------------------------------|
| Path copy(Path source, Path target, CopyOption... options) | Copy the file from source to target and return target            |
| Path move(Path source, Path target, CopyOption... options) | Move the file from source to target and return target            |
| void delete(Path path)                                     | Delete the file and throw an exception if it does not exist      |
| boolean deleteIfExists(Path path)                          | Delete the file if it exists and return whether file was deleted |
| boolean exists(Path path, LinkOption... options)           | Return true if file exists                                       |
| boolean notExists(Path path, LinkOption... options)        | Return true if file does not exist                               |

## Retrieving Information about a Path

The `Path` interface defines a bunch of methods that return useful information about the path that you're dealing with. In the following code listing, a `Path` is created referring to a directory and then we output information about the `Path` instance:

```
Path path = Paths.get("C:/home/java/workspace");
System.out.println("getFileName: " + path.getFileName());
System.out.println("getName(1): " + path.getName(1));
System.out.println("getNameCount: " + path.getNameCount());
```

```
System.out.println("getParent: " + path.getParent());
System.out.println("getRoot: " + path.getRoot());
System.out.println("subpath(0, 2): " + path.subpath(0, 2));
System.out.println("toString: " + path.toString());
```

When you execute this code snippet on Windows, the following output is printed:

```
getFileName: workspace
getName(1): java
getNameCount: 3
getParent: C:\home\java
getRoot: C:\
subpath(0, 2): home\java
toString: C:\home\java\workspace
```

Based on this output, it is fairly simple to describe what each method does. [Table 5-5](#) does just that.

**TABLE 5-5** | Path Methods

| Method                                                  | Description                                                                                                                                                                      |
|---------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>String getFileName()</code>                       | Returns the filename or the last element of the sequence of name elements.                                                                                                       |
| <code>Path getName(int index)</code>                    | Returns the path element corresponding to the specified index. The 0th element is the one closest to the root. (On Windows, the root is usually C:\ and on UNIX, the root is /.) |
| <code>int getNameCount()</code>                         | Returns the number of elements in this path, excluding the root.                                                                                                                 |
| <code>Path getParent()</code>                           | Returns the parent path, or null if this path does not have a parent.                                                                                                            |
| <code>Path getRoot()</code>                             | Returns the root of this path, or null if this path does not have a root.                                                                                                        |
| <code>Path subpath(int beginIndex, int endIndex)</code> | Returns a subsequence of this path (not including a root element) as specified by the beginning (included) and ending (not included) indexes.                                    |
| <code>String toString()</code>                          | Returns the string representation of this path.                                                                                                                                  |

Here is yet another interesting fact about the `Path` interface: It extends from `Iterable<Path>`. At first sight, this seems anything but interesting. But every class that (correctly) implements the `Iterable<?>` interface can be used as an expression in the enhanced `for` loop. You know you can iterate through an array

or a `List`, but you can iterate through a `Path` as well. That's pretty cool!

Using this functionality, it's easy to print the hierarchical tree structure of a file (or directory), as the following example shows:

```
int spaces = 1;
Path myPath = Paths.get("tmp", "dir1", "dir2", "dir3", "file.txt");
for (Path subPath : myPath) {
 System.out.format("%" + spaces + "s%n", "", subPath);
 spaces += 2; }
```

When you run this example, a (simplistic) tree is printed. Thanks to the variable `spaces` (which is increased with each iteration by 2), the different subpaths are printed like a directory tree.

```
tmp
 dir1
 dir2
 dir3
 file.txt
```

## Normalizing a Path

Normally (no pun intended), when you create a `Path`, you create it in a direct way. However, all three of these return the same logical `Path`:

```
Path p1 = Paths.get("myDirectory");
Path p2 = Paths.get("./myDirectory"); // one dot means
 // current directory
Path p3 = Paths.get("anotherDirectory", "..",
 "myDirectory"); // two dots means go up
 // one directory
```

`p1` is probably what you would type if you were coding. `p2` is just plain redundant. `p3` is more interesting. The two directories—`anotherDirectory` and

`myDirectory`—are on the same level, but we have to go up one level to get there:

```
/ (root)
| -- anotherDirectory
| -- myDirectory
```

You might be wondering why on earth we wouldn't just type `myDirectory` in the first place. And you would if you could. Sometimes, that doesn't work out. Let's look at a real example of why this might be.

```
/ (root)
| -- Build_Project
| | -- scripts
| | -- buildScript.sh
| -- My_Project
| | -- source
| | -- MyClass.java
```

If you wanted to compile `MyClass`, you would `cd` to `/My_Project/source` and run `javac MyClass.java`. Once your program gets bigger, it could be thousands of classes and have hundreds of jar files. You don't want to type in all of those just to compile, so someone writes a script to build your program.

`buildScript.sh` now finds everything that is needed to compile and runs the `javac` command for you. The problem is that the current directory is now `/Build_Project/scripts`, not `/My_Project/source`. The build script helpfully builds a path for you by doing something like this:

```
String buildProject // build scripts to express
= "/Build_Project/scripts"; // paths in relation to themselves

String upTwoDirectories = "../.."; // remember what .. means?

String myProject = "/My_Project/source";
Path path = Paths.get(buildProject,
 upTwoDirectories, myProject); // build path from variables
System.out.println("Original: " + path);
System.out.println("Normalized: " + path.normalize());
```

which outputs:

```
Original:/Build_Project/scripts/../../My_Project/source
Normalized:/My_Project/source
```

Whew. The second one is much easier to read. The `normalize()` method knows that a single dot can be ignored. It also knows that any directory followed by two dots can be removed from a path.

Be careful when using this `normalize()`! It just looks at the `String` equivalent of the path and doesn't check the file system to see whether the directories or files actually exist.

Let's practice and see what `normalize` returns for these paths. This time, we aren't providing a directory structure to show that the directories and files don't need to be present on the computer. What do you think the following prints out?

```
System.out.println(Paths.get("/a/./b/./c").normalize());
System.out.println(Paths.get(".classpath").normalize());
System.out.println(Paths.get("/a/b/c/..").normalize());
System.out.println(Paths.get("../a/b/c").normalize());
```

The output is

```
/a/b/c
.classpath
/a/b
../a/b/c
```

The first one removes all the single dots since they just point to the current directory. The second doesn't change anything since the dot is part of a filename and not a directory. The third sees one set of double dots, so it only goes up one directory. The last one is a little tricky. The two dots do say to go up one directory. But since there isn't a directory before it, `Path` can't simplify it.

To review, `normalize()` removes unneeded parts of the `Path`, making it more like you'd normally type it. (That's not where the word "normalize" comes from, but it is a nice way to remember it.)

## Resolving a Path

So far, you have an overview of all methods that can be invoked on a single `Path` object, but what if you need to combine two paths? You might want to do this if you have one `Path` representing your home directory and another containing the `Path` within that directory.

```
Path dir = Paths.get("/home/java");
Path file = Paths.get("models/Model.pdf");
Path result = dir.resolve(file);
System.out.println("result = " + result);
```

This produces the absolute path by merging the two paths:

```
result = /home/java/models/Model.pdf
```

`path1.resolve(path2)` should be read as "resolve `path2` within `path1`'s directory." In this example, we resolved the path of the `file` within the directory provided by `dir`.

Keeping this definition in mind, let's look at some more complex examples:

```
Path absolute = Paths.get("/home/java");
Path relative = Paths.get("dir");
Path file = Paths.get("Model.pdf");
System.out.println("1: " + absolute.resolve(relative));
System.out.println("2: " + absolute.resolve(file));
System.out.println("3: " + relative.resolve(file));
System.out.println("4: " + relative.resolve(absolute)); // BAD
System.out.println("5: " + file.resolve(absolute)); // BAD
System.out.println("6: " + file.resolve(relative)); // BAD
```

The output is

```
1: /home/java/dir
2: /home/java/Model.pdf
3: dir/Model.pdf
4: /home/java
5: /home/java
6: Model.pdf/dir
```

The first three do what you'd expect. They add the parameter to resolve to the provided path object. The fourth and fifth ones try to resolve an absolute path within the context of something else. The problem is that an absolute path doesn't depend on other directories. It is absolute. Therefore, `resolve()` just returns that absolute path. The output of the sixth one looks a little bit weird, but Java does the only right thing to do here. For all it knows the `Path` referred to by `Model.pdf` may be a directory and the `Path` referred to by `dir` may be a file!

Just like `normalize()`, keep in mind that `resolve()` will not check that the directory or file actually exists. To review, `resolve()` tells you how to resolve one path within another.

**Careful with methods that come in two flavors: one with a Path parameter and the other with a String parameter such as resolve(). The tricky part here is that null is a valid value for both a Path and a String. What will happen if you pass just null as a parameter? Which method will be invoked?**

```
Path path = Paths.get("/usr/bin/zip");
path.resolve(null);
```

**The compiler can't decide which method to invoke: the one with the Path parameter or the other one with the String parameter. That's why this code won't compile, and if you see such code in an exam question, you'll know what to do.**

**The following examples will compile without any problem, because the compiler knows which method to invoke, thanks to the type of the variable other and the explicit cast to String.**

```
Path path = Paths.get("/usr/bin/zip");
Path other = null;
path.resolve(other);
path.resolve ((String) null);
```

## Relativizing a Path

Now suppose we want to do the opposite of resolve. We have the absolute path of our home directory and the absolute path of the music file in our home directory. We want to know just the music file directory and name.

```
Path dir = Paths.get("/home/java");
Path music = Paths.get("/home/java/country/Swift.mp3");
Path mp3 = dir.relativize(music);
System.out.println(mp3);
```

The output is

**country/Swift.mp3.**

Java recognized that the /home/java part is the same and returned a path of just the remainder.

`path1.relativize(path2)` should be read as “give me a path that shows how to get from `path1` to `path2`.” In this example, we determined that `music` is a file in a directory named `country` within `dir`.

Keeping this definition in mind, let’s look at some more complex examples:

```
Path absolute1 = Paths.get("/home/java");
Path absolute2 = Paths.get("/usr/local");
Path absolute3 = Paths.get("/home/java/temp/music.mp3");
Path relative1 = Paths.get("temp");
Path relative2 = Paths.get("temp/music.pdf");
System.out.println("1: " + absolute1.relativize(absolute3));
System.out.println("2: " + absolute3.relativize(absolute1));
System.out.println("3: " + absolute1.relativize(relative2));
System.out.println("4: " + relative1.relativize(relative2));
System.out.println("5: " + absolute1.relativize(relative1));//BAD
```

The output is

```
1: temp/music.mp3
2: ../..
3: ../../usr/local
4: music.pdf
Exception in thread "main" java.lang.IllegalArgumentException: 'other'
is different type of Path
```

Before you scratch your head, let's look at the logical directory structure here. Keep in mind the directory doesn't actually need to exist; this is just to visualize it.

```
/root
| - usr
 | - local
| - home
 | -- java
 | - temp
 | - music.mp3
```

Now we can trace it through. The first example is straightforward. It tells us how to get to `absolute3` from `absolute1` by going down two directories. The second is similar. We get to `absolute1` from `absolute3` by doing the opposite—going up two directories. Remember from `normalize()` that a double dot means to go up a directory.

The third output statement says that we have to go up two directories and then down two directories to get from `absolute1` to `absolute2`. Java knows this because we provided absolute paths. The worst possible case is to have to go all the way up to the root like we did here.

The fourth output statement is okay. Even though they are both relative paths, there is enough in common for Java to tell what the difference in the path is.

The fifth example throws an exception. Java can't figure out how to make a relative path out of one absolute path and one relative path.

Remember, `relativize()` and `resolve()` are opposites. And just like

`resolve()`, `relativize()` does not check that the path actually exists. To review, `relativize()` tells you how to get a relative path between two paths.

## CERTIFICATION OBJECTIVE

### File and Directory Attributes (OCP Objective 9.2)

*9.2 Use `Files` class to check, read, delete, copy, move, manage metadata of a file or directory.*



*Metadata is data about data. For a file, you can think of the stuff that's in the file as the data, and the attributes of the file, like the date the file was created, as the metadata; that is, the data about the data that's in the file. When you see "metadata" in this objective, think "attributes" of files and directories.*

### Reading and Writing Attributes the Easy Way

In this section, we'll add classes and interfaces from the `java.nio.file.attribute` package to the discussion. Prior to NIO.2, you could read and write just a handful of attributes. Just like we saw when creating files, there is a new way to do this using `Files` instead of `File`. Oracle also took the opportunity to clean up the method signatures a bit. The following example creates a file, changes the last modified date, prints it out, and deletes the file using both the old and new method names. We might do this if we want to make a file look as if it were created in the past. (As you can see, there is a lesson about not relying on file timestamps here!)

```

ZonedDateTime janFirstDateTime =
 ZonedDateTime.of(
 // create a date
 LocalDate.of(2017, 1, 1),
 LocalTime.of(10, 0), ZoneId.of("US/Pacific"));
Instant januaryFirst = janFirstDateTime.toInstant();

// old way
File file = new File("c:/temp/file");
file.createNewFile(); // create the file
file.setLastModified(
 januaryFirst.getEpochSecond()*1000); // set time
System.out.println(file.lastModified()); // get time
file.delete(); // delete the file

// new way
Path path = Paths.get("c:/temp/file2");
Files.createFile(path); // create another file
FileTime fileTime =
 FileTime.fromMillis(// convert to the new
 januaryFirst.getEpochSecond()*1000); // FileTime object
Files.setLastModifiedTime(path, fileTime); // set time
System.out.println(Files.getLastModifiedTime(path)); // get time
Files.delete(path);

```

As you can see from the output, the only change in functionality is that the new `Files.getLastModifiedTime()` uses a human-friendly date format.

1483293600000

2017-01-01T18:00:00Z

The other common type of attribute you can set are file permissions. Both Windows and UNIX have the concept of three types of permissions. Here's what they mean:

- **Read** You can open the file or list what is in that directory.
- **Write** You can make a change to the file or add a file to that directory.
- **Execute** You can run the file if it is a runnable program or go into that directory.

Printing out the file permissions is easy. Note that these permissions are just for the user who is running the program—you! There are other types of permissions as well, but these can't be set in one line.

```
System.out.println(Files.isExecutable(path)) ;
System.out.println(Files.isReadable(path)) ;
System.out.println(Files.isWritable(path)) ;
```

[Table 5-6](#) shows how to get and set these attributes that can be set in one line, both using the older I/O way and the new `Files` class. You may have noticed that setting file permissions isn't in the table. That's more code, so we will talk about it later.

**TABLE 5-6** | I/O vs. NIO.2 Permissions

| Description                                                                                  | I/O Approach                                                                 | NIO.2 Approach                                                                                                                               |
|----------------------------------------------------------------------------------------------|------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------|
| Get the last modified date/time                                                              | <pre>File file = new File("test"); file.lastModified();</pre>                | <pre>Path path = Paths.get("test"); Files.getLastModifiedTime(path);</pre>                                                                   |
| Is read permission set                                                                       | <pre>File file = new File("test"); file.canRead();</pre>                     | <pre>Path path = Paths.get("test"); Files.isReadable(path);</pre>                                                                            |
| Is write permission set                                                                      | <pre>File file = new File("test"); file.canWrite();</pre>                    | <pre>Path path = Paths.get("test"); Files.isWritable(path);</pre>                                                                            |
| Is executable permission set                                                                 | <pre>File file = new File("test"); file.canExecute();</pre>                  | <pre>Path path = Paths.get("test"); Files.isExecutable(path);</pre>                                                                          |
| Set the last modified date/time<br><br>(Note:<br>timeInMillis<br>is an appropriate<br>long.) | <pre>File file = new File("test"); file.setLastModified(timeInMillis);</pre> | <pre>Path path = Paths.get("test"); FileTime fileTime = FileTime. fromMillis(timeInMillis); Files.setLastModifiedTime(path, fileTime);</pre> |

## Types of Attribute Interfaces

The attributes you set by calling methods on `Files` are the most straightforward ones. Beyond that, Java NIO.2 added attribute interfaces so you could read attributes that might not be on every operating system.

- **BasicFileAttributes** In the JavaDoc, Oracle says these are “attributes common to many file systems.” What they mean is that you can rely on these attributes being available to you unless you are writing Java code for some funky new operating system. Basic attributes include things like creation date.
- **PosixFileAttributes** POSIX stands for Portable Operating System Interface. This interface is implemented by both UNIX- and Linux-based operating systems. You can remember this because POSIX ends in “x,” as do UNIX and Linux.
- **DosFileAttributes** DOS stands for Disk Operating System. It is part of all Windows operating systems. Even Windows 8 and 10 have a DOS prompt available.

There are also separate interfaces for setting or updating attributes. While the details aren’t in scope for the exam, you should be familiar with the purpose of each one.

- **BasicFileAttributeView** Used to set the last updated, last accessed, and creation dates.
- **PosixFileAttributeView** Used to set the groups or permissions on UNIX/Linux systems. There is an easier way to set these permissions though, so you won’t be using the attribute view.
- **DosFileAttributeView** Used to set file permissions on DOS/Windows systems. Again, there is an easier way to set these, so you won’t be using the attribute view.
- **FileOwnerAttributeView** Used to set the primary owner of a file or directory.
- **AclFileAttributeView** Sets more advanced permissions on a file or directory.

## Working with **BasicFileAttributes**

The `BasicFileAttributes` interface provides methods to get information about a file or directory.

```
BasicFileAttributes basic = Files.readAttributes(path, // assume a valid path
 BasicFileAttributes.class);
System.out.println("create: " + basic.creationTime());
System.out.println("access: " + basic.lastAccessTime());
System.out.println("modify: " + basic.lastModifiedTime());
System.out.println("directory: " + basic.isDirectory());
```

The sample output shows that all three date/time values can be different. A file is created once. It can be modified many times. And it can be last accessed for reading after that. The `isDirectory` method is the same as `Files.isDirectory(path)`. It is just an alternative way of getting the same information.

```
create: 2017-03-21T23:14:36Z
access: 2017-09-25T02:01:11Z
modify: 2017-04-12T17:38:51Z
directory: false
```

There are some more attributes on `BasicFileAttributes`, but they aren't on the exam and you aren't likely to need them when coding. Just remember to check the JavaDoc if you need more information about a file.

So far, you've noticed that all the attributes are read only. That's because Java provides a different interface for updating attributes. Let's write code to update the last accessed time:

```

BasicFileAttributes basic = Files.readAttributes(
 path, BasicFileAttributes.class); // attributes
FileTime lastUpdated = basic.lastModifiedTime(); // get current
FileTime created = basic.creationTime(); // values
FileTime now = FileTime.fromMillis(System.currentTimeMillis());
BasicFileAttributeView basicView = Files.getFileAttributeView(
 path, BasicFileAttributeView.class); // "view" this time
basicView.setTimes(lastUpdated, now, created); // set all three

```

In this example, we demonstrated getting all three times. In practice, when calling `setTimes()`, you should pass null values for any of the times you don't want to change, and only pass `FileTimes` for the times you want to change.

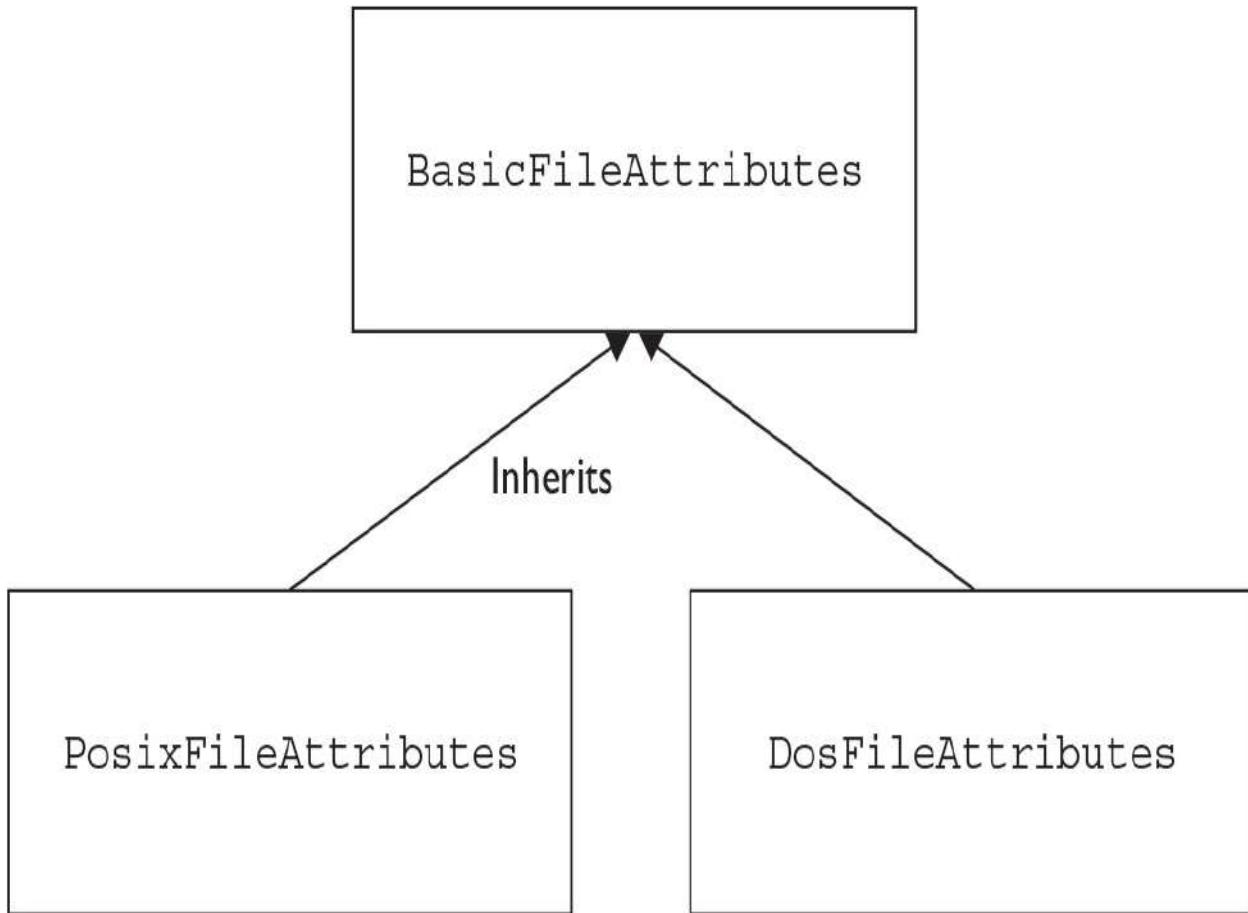
The key takeaways here are that the “`XxxFileAttributes`” classes are read only and the “`XxxFileAttributeView`” classes allow updates.



*The `BasicFileAttributes` and `BasicFileAttributeView` interfaces are a bit confusing. They have similar names but different functionality, and you get them in different ways. Try to remember these three things:*

- **`BasicFileAttributeView` is singular, but `BasicFileAttributes` is not.**
- **You get `BasicFileAttributeView` using `Files.getFileAttributeView`, and you get `BasicFileAttributes` using `Files.readAttributes`.**
- **You can ONLY update attributes in `BasicFileAttributeView`, not in `BasicFileAttributes`. Remember that the view is for updating.**

`PosixFileAttributes` and `DosFileAttributes` inherit from `BasicFileAttributes`. This means you can call Basic methods on a POSIX or DOS subinterface.



Try to use the more general type if you can. For example, if you are only going to use basic attributes, just get `BasicFileAttributes`. This lets your code remain operating system independent. If you are using a mix of basic and POSIX attributes, you can use `PosixFileAttributes` directly rather than calling `readAttributes()` twice to get two different ones.

## Working with `DosFileAttributes`

`DosFileAttributes` adds four more attributes to the basics. We'll look at the most common ones here—hidden files and read-only files. Hidden files typically begin with a dot and don't show up when you type `dir` to list the contents of a directory. Read-only files are what they sound like—files that can't be updated. (The other two attributes are “archive” and “system,” which you are quite unlikely to ever use.)

```
Path path= Paths.get("C:/test");
Files.createFile(path); // create file
Files.setAttribute(path, "dos:hidden", true); // set attribute
Files.setAttribute(path, "dos:readonly", true); // another one
DosFileAttributes dos = Files.readAttributes(path,
 DosFileAttributes.class); // dos attributes
System.out.println(dos.isHidden());
System.out.println(dos.isReadOnly());
Files.setAttribute(path, "dos:hidden", false);
Files.setAttribute(path, "dos:readonly", false);
dos = Files.readAttributes(path,
 DosFileAttributes.class); // get attributes again
System.out.println(dos.isHidden());
System.out.println(dos.isReadOnly());
Files.delete(path);
```

The output is

```
true
true
false
false
```

The first tricky thing in this code is that the `String` “`readonly`” is lowercase even though the method name is mixed case. If you forget and use the `String` “`readOnly`,” an `IllegalArgumentException` will be thrown at runtime.

The other tricky thing is that you cannot delete a read-only file. That’s why the code calls `setAttribute` a second time with `false` as a parameter, to make it no longer “read only” so the code can clean up after itself. And you can see that we had to call `readAttributes` again to see those updated values.



**There is an alternative way to set these attributes so you don't have to worry about the *String* values. However, the exam wants you to know how to use *Files*. It is good to know both ways, though.**

```
DosFileAttributeView view = Files.getFileAttributeView(path,
 DosFileAttributeView.class);

view setHidden(true);

view.setReadOnly(true);
```

## Working with PosixFileAttributes

PosixFileAttributes adds two more attributes to the basics—groups and permissions. On UNIX, every file or directory has both an owner and group name.

UNIX permissions are also more elaborate than the basic ones. Each file or directory has nine permissions set in a String. A sample is “rwxrw-r--.” Breaking this into groups of three, we have “rwx”, “rw-,” and “r--.” These sets of permissions correspond to who gets them. In this example, the “user” (owner) of the file has read, write, and execute permissions. The “group” only has read and write permissions. UNIX calls everyone who is not the owner or in the group “other.” “Other” only has read access in this example.

Now let's look at some code to set the permissions and output them in human-readable form:

```
Path path = Paths.get("/tmp/file2");
Files.createFile(path);
PosixFileAttributes posix = Files.readAttributes(path,
PosixFileAttributes.class); // get the Posix type
Set<PosixFilePermission> perms =
 PosixFilePermissions.fromString("rw-r--r--"); // UNIX style
Files.setPosixFilePermissions(path, perms); // set permissions
System.out.println(posix.permissions()); // get permissions
```

The output looks like this:

```
[OWNER_WRITE, GROUP_READ, OTHERS_READ, OWNER_READ]
```

It's not symmetric. We gave Java the permissions in cryptic UNIX format and got them back in plain English. You can also output the group name:

```
System.out.println(posix.group()); // get group
```

which outputs something like this:

```
horse
```

## Reviewing Attributes

Let's review the most common attributes information in [Table 5-7](#).

**TABLE 5-7** Common Attributes

| Type                  | Read and Write an Attribute                                                                                                                                                                                                                                                                                                                                                                                                          |
|-----------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Basic                 | <pre>// read  BasicFileAttributes basic = Files.readAttributes(path,         BasicFileAttributes.class); FileTime lastUpdated = basic.lastModifiedTime(); FileTime created = basic.creationTime(); FileTime now = FileTime.fromMillis(System.currentTimeMillis());  // write BasicFileAttributeView basicView = Files.getFileAttributeView(path, BasicFileAttributeView.class); basicView.setTimes(lastUpdated, now, created);</pre> |
| Posix<br>(UNIX/Linux) | <pre>PosixFileAttributes posix = Files.readAttributes(path,         PosixFileAttributes.class); Set&lt;PosixFilePermission&gt; perms = PosixFilePermissions. fromString("rw-r--r--"); Files.setPosixFilePermissions(path, perms); System.out.println(posix.group()); System.out.println(posix.permissions());</pre>                                                                                                                  |
| Dos<br>(Windows)      | <pre>DosFileAttributes dos = Files.readAttributes(path,         DosFileAttributes.class); System.out.println(dos.isHidden()); System.out.println(dos.isReadOnly()); Files.setAttribute(path, "dos:hidden", false); Files.setAttribute(path, "dos:readonly", false);</pre>                                                                                                                                                            |

## CERTIFICATION OBJECTIVE

### DirectoryStream (OCP Objectives 9.2 and 9.3)

9.2 Use `Files` class to check, read, delete, copy, move, manage metadata of a file or directory.

9.3 Use Stream API with NIO.2.

Now let's return to more NIO.2 capabilities that you'll find in the `java.nio.file` package... You might need to loop through a directory. Let's say you were asked to list out all the users with a home directory on this computer.

```
/home
 | - users
 | | - vafi
 | | - eyra
```

```
Path dir = Paths.get("/home/users");
try (DirectoryStream<Path> stream = // use try-with-resources
 Files.newDirectoryStream(dir)) { // so we don't have close()
 for (Path path : stream) // loop through the stream
 System.out.println(path.getFileName());
}
```

As expected, this outputs

```
vafi
eyra
```

The `DirectoryStream` interface lets you iterate through a directory. But this is

just the tip of the iceberg. Let's say we have hundreds of users and each day we want to only report on a few of them. The first day, we only want the home directories of users whose names begin with either the letter v or the letter w.

```
Path dir = Paths.get("/home/users");
try (DirectoryStream<Path> stream = Files.newDirectoryStream(
 dir, "[vw]*")) { // "v" or "w" followed by anything
 for (Path path : stream)
 System.out.println(path.getFileName());
}
```

This time, the output is

vafi

Let's examine the expression `[vw]*`. `[vw]` means either of the characters v or w. The `*` is a wildcard that means zero or more of any character. Notice this is not a regular expression. (If it were, the syntax would be `[vw].*`—see the dot in there.) `DirectoryStream` uses something called a *glob*. We will see more on globs later in the chapter.

There is one limitation with `DirectoryStream`. It can only look at one directory. One way to remember this is that it works like the `dir` command in DOS or the `ls` command in UNIX. Or you can remember that `DirectoryStream` streams one directory.

## FileVisitor

Luckily, there is another class that does, in fact, look at subdirectories. Let's say you want to get rid of all the `.class` files before zipping up and submitting your assignment. You could go through each directory manually, but that would get tedious really fast. You could write a complicated command in Windows and another in UNIX, but then you'd have two programs that do the same thing. Luckily, you can use Java and only write the code once.

Java provides a `SimpleFileVisitor`. You extend it and override one or more methods. Then you can call `Files.walkFileTree`, which knows how to recursively look through a directory structure and call methods on a visitor subclass. Let's try our example:

```
/home
| - src
| | - Test.java
| | - Test.class
| | - dir
| | | - AnotherTest.java
| | | - AnotherTest.class
```

```
public class RemoveClassFiles
 extends SimpleFileVisitor<Path> { // need to extend visitor
 public FileVisitResult visitFile(// called "automatically"
 Path file, BasicFileAttributes attrs)
 throws IOException {
 if (file.getFileName().toString().endsWith(".class"))
 Files.delete(file); // delete the file
 return FileVisitResult.CONTINUE; // go on to next file
 }
 public static void main(String[] args) throws Exception {
 RemoveClassFiles dirs = new RemoveClassFiles();
 Files.walkFileTree(// kick off recursive check
 Paths.get("/home/src"), // starting point
 dirs); // the visitor
 }
}
```

This is a simple file visitor. It only implements one method: `visitFile`. This method is called for every file in the directory structure. It checks the extension

of the file and deletes it if appropriate. In our case, two .class files are deleted.

There are two parameters to `visitFile()`. The first one is the `Path` object representing the current file. The other is a `BasicFileAttributes` interface. Do you remember what this does? That's right—it lets you find out if the current file is a directory, when it was created, and many other similar pieces of data.

Finally, `visitFile()` returns `FileVisitResult.CONTINUE`. This tells `walkFileTree()` that it should keep looking through the directory structure for more files.

Now that we have a feel for the power of this class, let's take a look at all the methods available to us with another example:

```
/home
| - a.txt
| - emptyChild
| - child
| | - b.txt
| | - grandchild
| | - c.txt
```

```
public class PrintDirs extends SimpleFileVisitor<Path> {
 public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs)
 {
 System.out.println("pre: " + dir);
 return FileVisitResult.CONTINUE;
 }
 public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) {
 System.out.println("file: " + file);
 return FileVisitResult.CONTINUE;
 }
 public FileVisitResult visitFileFailed(Path file, IOException exc) {
 return FileVisitResult.CONTINUE;
 }
 public FileVisitResult postVisitDirectory(Path dir, IOException exc) {
 System.out.println("post: " + dir);
 return FileVisitResult.CONTINUE;
 }
 public static void main(String[] args) throws Exception {
 PrintDirs dirs = new PrintDirs();
 Files.walkFileTree(Paths.get("/home"), dirs);
 }
}
```

You might get the following output:

```
pre: /home
file: /home/a.txt
pre: /home/child
file: /home/child/b.txt
pre: /home/child/grandchild
file: /home/child/grandchild/c.txt
post: /home/child/grandchild
post: /home/child
pre: /home/emptyChild
post: /home/emptyChild
post: /home
```

Note that Java goes down as deep as it can before returning back up the tree. This is called a *depth-first search*. We said “might” because files and directories at the same level can get visited in either order.

You can override as few or as many of the four methods as you’d like. Note that the second half of the methods have IOException as a parameter. This allows those methods to handle problems that came earlier when walking through the tree. [Table 5-8](#) summarizes the methods.

**TABLE 5-8** FileVisitor Methods

| Method             | Description                                                                     | IOException Parameter? |
|--------------------|---------------------------------------------------------------------------------|------------------------|
| preVisitDirectory  | Called before drilling down into the directory                                  | No                     |
| visitFile          | Called once for each file (but not for directories)                             | No                     |
| visitFileFailed    | Called only if there was an error accessing a file, usually a permissions issue | Yes                    |
| postVisitDirectory | Called when finished with the directory on the way back up                      | Yes                    |

You actually do have some control, though, through those `FileVisitResult` constants. Suppose we changed the `preVisitDirectory` method to the following:

```
public FileVisitResult preVisitDirectory(
 Path dir, BasicFileAttributes attrs) {
 System.out.println("pre: " + dir);
 String name = dir.getFileName().toString();
 if (name.equals("child"))
 return FileVisitResult.SKIP_SUBTREE;
 return FileVisitResult.CONTINUE;
}
```

Now the output is

```
pre: /home
file: /home/a.txt
pre: /home/child
pre: /home/emptyChild
post: /home/emptyChild
post: /home
```

Since we instructed the program to skip the entire child subtree—i.e., we don't see the file: b.txt, or the subdirectory: grandchild—we also don't see the post visit call.

Now what do you think would happen if we changed `FileVisitResult.SKIP_SUBTREE` to `FileVisitResult.TERMINATE`? The output might be:

```
pre: /home
file: /home/a.txt
pre: /home/child
```

We see that as soon as the “child” directory came up, the program stopped walking the tree. And again, we are using “might” in terms of the output. It's also possible for `emptyChild` to come up first, in which case, the last line of the

output would be /home/emptyChild.

There's one more result type. What do you think would happen if we changed `FileVisitResult.TERMINATE` to `FileVisitResult.SKIP_SIBLINGS`? The output happens to be the same as the previous example:

```
pre: /home
file: /home/a.txt
pre: /home/child
```

`SKIP_SIBLINGS` is a combination of `SKIP_SUBTREE` and “don't look in any folders at the same level.” This means we skip everything under `child` and also skip `emptyChild`.

One more example to make sure you really understand what is going on. What do you think gets output if we use this method?

```
public FileVisitResult preVisitDirectory(Path dir,
 BasicFileAttributes attrs) {
 System.out.println("pre: " + dir);
 String name = dir.getFileName().toString();
 if (name.equals("grandchild"))
 return FileVisitResult.SKIP_SUBTREE;
 if (name.equals("emptyChild"))
 return FileVisitResult.SKIP_SIBLINGS;
 return FileVisitResult.CONTINUE;
}
```

Assuming `child` is encountered before `emptyChild`, the output is

```
pre: /home
file: /home/a.txt
pre: /home/child
file: /home/child/b.txt
pre: /home/child/grandchild
post: /home/child
pre: /home/emptyChild
post: /home
```

We don't see `file: c.txt` or `post: /home/child/grandchild` because we skip `grandchild` the subtree. We don't see `post: /home/emptyChild` because we skip siblings of `emptyChild`. But wait. Isn't `/home/child` a sibling? It is. But the visitor goes in order. Since `child` was seen before `emptyChild`, it is too late to skip it. Just like when you print a document, it is too late to prevent pages from printing that have already printed. File visitor can only skip subtrees that it has not encountered yet.

## PathMatcher

`DirectoryStream` and `FileVisitor` allowed us to go through the files that exist. Things can get complicated fast, though. Imagine you had a requirement to print out the names of all text files in any subdirectory of "password." You might be wondering why anyone would want to do this. Maybe a teammate foolishly stored passwords for everyone to see and you want to make sure nobody else did that. You could write logic to keep track of the directory structure, but that makes the code harder to read and understand. By the end of this section, you'll know a better way.

Let's start out with a simpler example to see what a `PathMatcher` can do:

```
Path path1 = Paths.get("/home/One.txt");
Path path2 = Paths.get("One.txt");
PathMatcher matcher = FileSystems.getDefault() // get the PathMatcher
 .getPathMatcher(// for the right file system
 "glob:*.txt"); // wait. What's a glob?

System.out.println(matcher.matches(path1));
System.out.println(matcher.matches(path2));
```

which outputs:

```
false
true
```

We can see that the code checks if a Path consists of any characters followed by “.txt.” To get a PathMatcher, you have to call `FileSystems.getDefault().getPathMatcher` because matching works differently on different operating systems. PathMatchers use a new type that you probably haven’t seen before called a glob. Globs are not regular expressions, although they might look similar at first. Let’s look at some more examples of globs using a common method so we don’t have to keep reading the same “boilerplate” code. (Boilerplate code is the part of the code that is always the same.)

```
public void matches(Path path, String glob) {
 PathMatcher matcher = FileSystems.getDefault().getPathMatcher(glob);
 System.out.println(matcher.matches(path));
}
```

In the world of globs, one asterisk means “match any character except for a directory boundary.” Two asterisks means “match any character, including a directory boundary.”

```
Path path = Paths.get("/com/java/One.java");
matches(path, "glob:*.java"); // false
matches(path, "glob:**/*.java"); // true
matches(path, "glob: *"); // false
matches(path, "glob: **"); // true
```



**Remember that we are using a file system–specific *PathMatcher*. This means slashes and backslashes can be treated differently, depending on what operating system you happen to be running. The previous example does print the same output on both Windows and UNIX because it uses forward slashes. However, if you change just one line of code, the output changes:**

```
Path path = Paths.get("com\\java\\One.java");
```

**Now Windows still prints:**

```
false
true
false
true
```

**However, UNIX prints:**

```
true
false
true
true
```

**Why? Because UNIX doesn't see the backslash as a directory boundary. The lesson here is to use / instead of \ so your code behaves more predictably across operating systems.**

Now let's match files with a four-character extension. A question mark matches any character. A character could be a letter or a number or anything else.

```
Path path1 = Paths.get("One.java");
Path path2 = Paths.get("One.ja^a");
matches(path1, "glob:*.????"); // true
matches(path1, "glob:*.??"); // false
matches(path2, "glob:*.????"); // true
matches(path2, "glob:*.??"); // false
```

Globs also provide a nice way to match multiple patterns. Suppose we want to match anything that begins with the names Kathy or Bert:

```
Path path1 = Paths.get("Bert-book");
Path path2 = Paths.get("Kathy-horse");
matches(path1, "glob:{Bert*,Kathy*}"); // true
matches(path2, "glob:{Bert,Kathy}*"); // true
matches(path1, "glob:{Bert,Kathy}"); // false
```

The first glob shows we can put wildcards inside braces to have multiple glob expressions. The second glob shows that we can put common wildcards outside the braces to share them. The third glob shows that without the wildcard, we will only match the literal strings “Bert” and “Kathy.”

You can also use sets of characters like [a-z] or [#\$%] in globs just like in regular expressions. You can also escape special characters with a backslash. Let's put this all together with a tricky example:

```

Path path1 = Paths.get("0*b/test/1");
Path path2 = Paths.get("9*b/test/1");
Path path3 = Paths.get("01b/test/1");
Path path4 = Paths.get("0*b/1");
String glob = "glob:[0-9]*{A*,b}/**/1";
matches(path1, glob); // true
matches(path2, glob); // false
matches(path3, glob); // false
matches(path4, glob); // false

```

Spelling out what the glob does, we have the following:

- [0-9] One single digit. Can also be read as any one character from 0 to 9.
- \\\* The literal character asterisk rather than the asterisk that means to match anything. A single backslash before \* escapes it. However, Java won't let you type a single backslash, so you have to escape the backslash itself with another backslash.
- {A\*,b} Either a capital A followed by anything or the single character *b*.
- /\*\*/ One or more directories with any name.
- 1 The single character 1.

The second path doesn't match because it has the literal backslash followed by the literal asterisk. The glob was looking for the literal asterisk by itself. The third path also doesn't match because there is no literal asterisk. The fourth path doesn't match because there is no directory between "b" and "1" for the \*\* to match. Luckily, nobody would write such a crazy, meaningless glob. But if you can understand this one, you are all set. Globs tend to be simple expressions like {\* .txt, \*.html} when used for real.

Since globs are just similar enough to regular expressions to be tricky, [Table 5-9](#) reviews the similarities and differences in common expressions. Regular expressions are more powerful, but globs focus on what you are likely to need when matching filenames.

**TABLE 5-9** Glob vs. Regular Expression

| What to Match                                                     | In a Glob               | In a Regular Expression |
|-------------------------------------------------------------------|-------------------------|-------------------------|
| Zero or more of any character, including a directory boundary     | **                      | . *                     |
| Zero or more of any character, not including a directory boundary | *                       | N/A – no special syntax |
| Exactly one character                                             | ?                       | .                       |
| Any digit                                                         | [0-9]                   | [0-9]                   |
| Begins with cat or dog                                            | {cat, dog}* {cat dog}.* | (cat dog).*             |

By now, you've probably noticed that we are dealing with `Path` objects, which means they don't actually need to exist on the file system. But we wanted to print out all the text files that actually exist in a subdirectory of `password`. Luckily, we can combine the power of `PathMatchers` with what we already know about walking the file tree to accomplish this.

```

public class MyPathMatcher extends SimpleFileVisitor<Path> {
 private PathMatcher matcher =
 FileSystems.getDefault().getPathMatcher(
 "glob:**/password/**.txt"); // ** means any subdirectory
 public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)
 throws IOException {
 if (matcher.matches(file)) {
 System.out.println(file);
 }
 return FileVisitResult.CONTINUE;
 }
 public static void main(String[] args) throws Exception {
 MyPathMatcher dirs = new MyPathMatcher();
 Files.walkFileTree(Paths.get("/"), dirs); // start with root
 }
}

```

The code looks similar, regardless of what you want to do. You just change the glob pattern to what you actually want to match.

## WatchService

The last thing you need to know about in NIO.2 is `watchService`. Suppose you are writing an installer program. You check that the directory you are about to install into is empty. If not, you want to wait until the user manually deletes that directory before continuing. Luckily, you won't have to write this code from scratch, but you should be familiar with the concepts. Here's the directory tree:

```
/dir
| - directoryToDelete
| - other
```

Here's the code snippet:

```
Path dir = Paths.get("/dir"); // get directory containing
 // file/directory we care
 // about
WatchService watcher = FileSystems.getDefault() // file system-specific code
 .newWatchService(); // create empty WatchService
dir.register(watcher, ENTRY_DELETE); // needs a static import!
 // start watching for
 // deletions
while (true) { // loop until say to stop
 WatchKey key;
 try {
 key = watcher.take(); // wait for a deletion
 } catch (InterruptedException x) {
 return; // give up if something goes
 // wrong
 }
 for (WatchEvent<?> event : key.pollEvents()) {
 WatchEvent.Kind<?> kind = event.kind();
 System.out.println(kind.name()); // create/delete/modify
 System.out.println(kind.type()); // always a Path for us
 System.out.println(event.context()); // name of the file
 String name = event.context().toString();
 if (name.equals("directoryToDelete")) { // only delete right directory
 System.out.format("Directory deleted, now we can proceed");
 }
 }
}
```

```

 return; // end program, we found what
 // we were waiting for
 }
}

key.reset(); // keep looking for events
}

```

Supposing we delete directory “other” followed by directory `directoryToDelete`, this outputs:

```

ENTRY_DELETE
interface java.nio.file.Path
other
ENTRY_DELETE
interface java.nio.file.Path
directoryToDelete
Directory deleted, now we can proceed

```

Notice that we had to watch the directory that contains the files or directories we are interested in. This is why we watched `/dir` instead of `/dir/directoryToDelete`. This is also why we had to check the context to make sure the directory we were actually interested in is the one that was deleted.

The basic flow of `watchService` stays the same, regardless of what you want to do:

1. Create a new `WatchService`.
2. Register it on a `Path` listening to one or more event types.
3. Loop until you are no longer interested in these events.
4. Get a `WatchKey` from the `WatchService`.
5. Call `key.pollEvents` and do something with the events.
6. Call `key.reset` to look for more events.

Let's look at some of these in more detail. You register the WatchService on a Path using statements like the following:

```
dir1.register(watcher, ENTRY_DELETE);
dir2.register(watcher, ENTRY_DELETE, ENTRY_CREATE);
dir3.register(watcher, ENTRY_DELETE, ENTRY_CREATE, ENTRY_MODIFY);
```

(Note: These ENTRY\_XXX constants can be found in the StandardWatchEventKinds class. Here and in later code, you'll probably want to create static imports for these constants.) You can register one, two, or three of the event types. ENTRY\_DELETE means you want your program to be informed when a file or directory has been deleted. Similarly, ENTRY\_CREATE means a new file or directory has been created. ENTRY\_MODIFY means a file has been edited in the directory. These changes can be made manually by a human or by another program on the computer.

Renaming a file or directory is interesting, as it does not show up as ENTRY\_MODIFY. From Java's point of view, a rename is equivalent to creating a new file and deleting the original. This means that two events will trigger for a rename—both ENTRY\_CREATE and ENTRY\_DELETE. Actually editing a file will show up as ENTRY\_MODIFY.

To loop through the events, we use `while(true)`. It might seem a little odd to write a loop that never ends. Normally, there is a `break` or `return` statement in the loop so you stop looping once whatever event you were waiting for has occurred. It's also possible you want the program to run until you kill or terminate it at the command line.

Within the loop, you need to get a `WatchKey`. There are two ways to do this. The most common is to call `take()`, which waits until an event is available. It throws an `InterruptedException` if it gets interrupted without finding a key. This allows you to end the program. The other way is to call `poll()`, which returns `null` if an event is not available. You can provide optional timeout parameters to wait up to a specific period of time for an event to show up.

```
watcher.take(); // wait "forever" for an event
watcher.poll(); // get event if present right NOW
watcher.poll(10, TimeUnit.SECONDS); // wait up to 10 seconds for an event
watcher.poll(1, TimeUnit.MINUTES); // wait up to 1 minute for an event
```

Next, you loop through any events on that key. In the case of rename, you'll get one key with two events—the EVENT\_CREATE and EVENT\_DELETE. Remember that you get all the events that happened since the last time you called poll() or take(). This means you can get multiple seemingly unrelated events out of the same key. They can be from different files but are for the same WatchService.

```
for (WatchEvent<?> event : key.pollEvents()) {
```

Finally, you call key.reset(). This is very important. If you forget to call reset, the program will work for the first event, but then you will not be notified of any other events.



**There are a few limitations you should be aware of with WatchService. To begin with, it is slow. You could easily wait five seconds for the event to register. It also isn't 100 percent reliable. You can add code to check whether kind == OVERFLOW, but that just tells you something went wrong. You don't know what events you lost. In practice, you are unlikely to use WatchService.**

WatchService only watches the files and directories immediately beneath it. What if we want to watch to see if either p.txt or c.txt is modified?

```
/dir
 | - parent
 | - p.txt
 | - child
 | - c.txt
```

One way is to register both directories:

```
WatchService watcher =
 FileSystems.getDefault().newWatchService();
Path dir = Paths.get("/dir/parent");
dir.register(watcher, ENTRY_MODIFY);
Path child = Paths.get("dir/parent/child");
child.register(watcher, ENTRY_MODIFY);
```

This works. You can type in all the directories you want to watch. If we had a lot of child directories, this would quickly get to be too much work. Instead, we can have Java do it for us:

```
Path myDir = Paths.get("/dir/parent");
final WatchService watcher = // final so visitor can use it
 FileSystems.getDefault().newWatchService();
Files.walkFileTree(myDir, new SimpleFileVisitor<Path>() {
 public FileVisitResult preVisitDirectory(Path dir,
 BasicFileAttributes attrs) throws IOException {
 dir.register(watcher, ENTRY_MODIFY); // watch each directory
 return FileVisitResult.CONTINUE;
 }
});
```

This code goes through the file tree recursively registering each directory with the watcher. The NIO.2 classes are designed to work together. For example, we could add `PathMatcher` to the previous example to only watch directories that have a specific pattern in their path.

## CERTIFICATION OBJECTIVE

## Serialization (Objective 8.2)

8.2 Use *BufferedReader*, *BufferedWriter*, *File*, *FileReader*, *FileWriter*, *FileInputStream*, *FileOutputStream*, *ObjectOutputStream*, *ObjectInputStream*, and *PrintWriter* in the *java.io* package.

Imagine you want to save the state of one or more objects. If Java didn't have serialization (as the earliest version did not), you'd have to use one of the I/O classes to write out the state of the instance variables of all the objects you want to save. The worst part would be trying to reconstruct new objects that were virtually identical to the objects you were trying to save. You'd need your own protocol for the way in which you wrote and restored the state of each object, or you could end up setting variables with the wrong values. For example, imagine you stored an object that has instance variables for height and weight. At the time you save the state of the object, you could write out the height and weight as two `ints` in a file, but the order in which you write them is crucial. It would be all too easy to re-create the object but mix up the height and weight values—using the saved height as the value for the new object's weight and vice versa.

Serialization lets you simply say “save this object and all of its instance variables.” Actually, it is a little more interesting than that because you can add, “...unless I've explicitly marked a variable as `transient`, which means, don't include the transient variable's value as part of the object's serialized state.”

## Working with *ObjectOutputStream* and *ObjectInputStream*

The magic of basic serialization happens with just two methods: one to serialize objects and write them to a stream, and a second to read the stream and deserialize objects.

```
ObjectOutputStream.writeObject() // serialize and write
```

```
ObjectInputStream.readObject() // read and deserialize
```

The `java.io.ObjectOutputStream` and `java.io.ObjectInputStream` classes are considered to be *higher*-level classes in the `java.io` package, and as we learned earlier, that means you'll wrap them around *lower*-level classes, such as `java.io.FileOutputStream` and `java.io.FileInputStream`. Here's a small

program that creates a `Cat` object, serializes it, and then deserializes it:

```
import java.io.*;

class Cat implements Serializable { } // 1

public class SerializeCat {
 public static void main(String[] args) {
 Cat c = new Cat(); // 2
 try {
 FileOutputStream fs = new FileOutputStream("testSer.ser");
 ObjectOutputStream os = new ObjectOutputStream(fs);
 os.writeObject(c); // 3
 os.close();
 } catch (Exception e) { e.printStackTrace(); }

 try {
 FileInputStream fis = new FileInputStream("testSer.ser");
 ObjectInputStream ois = new ObjectInputStream(fis);
 c = (Cat) ois.readObject(); // 4
 ois.close();
 } catch (Exception e) { e.printStackTrace(); }
 }
}
```

Let's take a look at the key points in this example:

1. We declare that the `Cat` class implements the `Serializable` interface.

`Serializable` is a marker interface; it has no methods to implement. (In the next several sections, we'll cover various rules about when you need to declare classes `Serializable`.)

2. We make a new `Cat` object, which as we know is serializable.
3. We serialize the `Cat` object `c` by invoking the `writeObject()` method. It took a fair amount of preparation before we could actually serialize our `Cat`. First, we had to put all of our I/O-related code in a try/catch block. Next, we had to create a `FileOutputStream` to write the object to. Then, we wrapped the `FileOutputStream` in an `ObjectOutputStream`, which is the class that has the magic serialization method that we need. Remember that the invocation of `writeObject()` performs two tasks: it serializes the object, and then it writes the serialized object to a file.
4. We de-serialize the `Cat` object by invoking the `readObject()` method. The `readObject()` method returns an `Object`, so we have to cast the deserialized object back to a `Cat`. Again, we had to go through the typical I/O hoops to set this up.

This is a bare-bones example of serialization in action. Over the next few pages, we'll look at some of the more complex issues that are associated with serialization.

## Object Graphs

What does it really mean to save an object? If the instance variables are all primitive types, it's pretty straightforward. But what if the instance variables are themselves references to *objects*? What gets saved? Clearly in Java it wouldn't make any sense to save the actual value of a reference variable, because the value of a Java reference has meaning only within the context of a single instance of a JVM. In other words, if you tried to restore the object in another instance of the JVM, even running on the same computer on which the object was originally serialized, the reference would be useless.

But what about the object that the reference refers to? Look at this class:

```
class Dog {
 private Collar theCollar;
 private int dogSize;
 public Dog(Collar collar, int size) {
 theCollar = collar;
 dogSize = size;
 }
 public Collar getCollar() { return theCollar; }
}
class Collar {
 private int collarSize;
 public Collar(int size) { collarSize = size; }
 public int getCollarSize() { return collarSize; }
}
```

Now make a dog... First, you make a `Collar` for the `Dog`:

```
Collar c = new Collar(3);
```

Then make a new `Dog`, passing it the `Collar`:

```
Dog d = new Dog(c, 8);
```

Now what happens if you save the `Dog`? If the goal is to save and then restore a `Dog`, and the restored `Dog` is an exact duplicate of the `Dog` that was saved, then the `Dog` needs a `Collar` that is an exact duplicate of the `Dog`'s `Collar` at the time the `Dog` was saved. That means both the `Dog` and the `Collar` should be saved.

And what if the `Collar` itself had references to other objects—perhaps a `color` object? This gets quite complicated very quickly. If it were up to the programmer to know the internal structure of each object the `Dog` referred to, so that the programmer could be sure to save all the state of all those objects... whew. That would be a nightmare with even the simplest of objects.

Fortunately, the Java serialization mechanism takes care of all of this. When you serialize an object, Java serialization takes care of saving that object's entire "object graph." That means a deep copy of everything the saved object needs to be restored. For example, if you serialize a `Dog` object, the `Collar` will be serialized automatically. And if the `Collar` class contained a reference to another object, *that* object would also be serialized, and so on. And the only object you have to worry about saving and restoring is the `Dog`. The other objects required to fully reconstruct that `Dog` are saved (and restored) automatically through serialization.

Remember, you do have to make a conscious choice to create objects that are serializable by implementing the `Serializable` interface. If we want to save `Dog` objects, for example, we'll have to modify the `Dog` class as follows:

```
class Dog implements Serializable {
 // the rest of the code as before
 // Serializable has no methods to implement
}
```

And now we can save the `Dog` with the following code:

```
import java.io.*;
public class SerializeDog {
 public static void main(String[] args) {
 Collar c = new Collar(3);
 Dog d = new Dog(c, 8);
 try {
 FileOutputStream fs = new FileOutputStream("testSer.ser");
 ObjectOutputStream os = new ObjectOutputStream(fs);
 os.writeObject(d);
 os.close();
 } catch (Exception e) { e.printStackTrace(); }
 }
}
```

But when we run this code we get a runtime exception, something like this

```
java.io.NotSerializableException: Collar
```

What did we forget? The `Collar` class must *also* be `Serializable`. If we modify the `Collar` class and make it serializable, then there's no problem:

```
class Collar implements Serializable {
 // same
}
```

Here's the complete listing:

```
import java.io.*;
public class SerializeDog {
 public static void main(String[] args) {
 Collar c = new Collar(3);
 Dog d = new Dog(c, 5);
 System.out.println("before: collar size is "
 + d.getCollar().getCollarSize());
 try {
 FileOutputStream fs = new FileOutputStream("testSer.ser");
 ObjectOutputStream os = new ObjectOutputStream(fs);
 os.writeObject(d);
 os.close();
 } catch (Exception e) { e.printStackTrace(); }
 try {
 FileInputStream fis = new FileInputStream("testSer.ser");
 ObjectInputStream ois = new ObjectInputStream(fis);
 d = (Dog) ois.readObject();
 ois.close();
 } catch (Exception e) { e.printStackTrace(); }

 System.out.println("after: collar size is "
 + d.getCollar().getCollarSize());
 }
}
class Dog implements Serializable {
 private Collar theCollar;
 private int dogSize;
 public Dog(Collar collar, int size) {
 theCollar = collar;
 dogSize = size;
 }
 public Collar getCollar() { return theCollar; }
}
class Collar implements Serializable {
 private int collarSize;
 public Collar(int size) { collarSize = size; }
 public int getCollarSize() { return collarSize; }
}
```

This produces the output:

```
before: collar size is 3
after: collar size is 3
```

But what would happen if we didn't have access to the `Collar` class source code? In other words, what if making the `Collar` class serializable was not an option? Are we stuck with a non-serializable `Dog`?

Obviously, we could subclass the `Collar` class, mark the subclass as `Serializable`, and then use the `Collar` subclass instead of the `Collar` class. But that's not always an option either for several potential reasons:

1. The `Collar` class might be final, preventing subclassing.  
OR
2. The `Collar` class might itself refer to other non-serializable objects, and without knowing the internal structure of `Collar`, you aren't able to make all these fixes (assuming you even wanted to *try* to go down that road).  
OR
3. Subclassing is not an option for other reasons related to your design.

So...*then* what do you do if you want to save a `Dog`?

That's where the `transient` modifier comes in. If you mark the `Dog`'s `Collar` instance variable with `transient`, then serialization will simply skip the `Collar` during serialization:

```
class Dog implements Serializable {
 private transient Collar theCollar; // add transient
 // the rest of the class as before
}

class Collar { // no longer Serializable
 // same code
}
```

Now we have a `Serializable` `Dog`, with a non-`Serializable` `Collar`, but the `Dog` has marked the `Collar` `transient`; the output is

```
before: collar size is 3
Exception in thread "main" java.lang.NullPointerException
```

So now what can we do?

## Using `writeObject` and `readObject`

Consider the problem: we have a `Dog` object we want to save. The `Dog` has a `Collar`, and the `Collar` has state that should also be saved as part of the `Dog`'s state. But...the `Collar` is not `Serializable`, so we must mark it `transient`. That means when the `Dog` is deserialized, it comes back with a null `collar`. What can we do to somehow make sure that when the `Dog` is deserialized, it gets a new `Collar` that matches the one the `Dog` had when the `Dog` was saved?

Java serialization has a special mechanism just for this—a set of private methods you can implement in your class that, if present, will be invoked automatically during serialization and deserialization. It's almost as if the methods were defined in the `Serializable` interface, except they aren't. They are part of a special callback contract the serialization system offers you that basically says, “If you (the programmer) have a pair of methods matching this exact signature (you'll see them in a moment), these methods will be called during the serialization/deserialization process.

These methods let you step into the middle of serialization and

deserialization. So they’re perfect for letting you solve the Dog/Collar problem: when a Dog is being saved, you can step into the middle of serialization and say, “By the way, I’d like to add the state of the Collar’s variable (an int) to the stream when the Dog is serialized.” You’ve manually added the state of the Collar to the Dog’s serialized representation, even though the Collar itself is not saved.

Of course, you’ll need to restore the Collar during deserialization by stepping into the middle and saying, “I’ll read that extra int I saved to the Dog stream, and use it to create a new Collar, and then assign that new Collar to the Dog that’s being deserialized.” The two special methods you define must have signatures that look *exactly* like this:

```
private void writeObject(ObjectOutputStream os) {
 // your code for saving the Collar variables
}

private void readObject(ObjectInputStream is) {
 // your code to read the Collar state, create a new Collar,
 // and assign it to the Dog
}
```

Yes, we’re going to write methods that have the same name as the ones we’ve been calling! Where do these methods go? Let’s change the Dog class:

```
class Dog implements Serializable {
 transient private Collar theCollar; // we can't serialize this
 private int dogSize;
 public Dog(Collar collar, int size) {
 theCollar = collar;
 dogSize = size;
 }
 public Collar getCollar() { return theCollar; }
 private void writeObject(ObjectOutputStream os) {
 // throws IOException { // 1
 try {
 os.defaultWriteObject(); // 2
 os.writeInt(theCollar.getCollarSize()); // 3
 } catch (Exception e) { e.printStackTrace(); }
 }
 private void readObject(ObjectInputStream is) {
 // throws IOException, ClassNotFoundException { // 4
 try {
 is.defaultReadObject(); // 5
 theCollar = new Collar(is.readInt()); // 6
 } catch (Exception e) { e.printStackTrace(); }
 }
}
```

In our scenario we've agreed that, for whatever real-world reason, we can't serialize a `Collar` object, but we want to serialize a `Dog`. To do this we're going to implement `writeObject()` and `readObject()`. By implementing these two

methods you're saying to the compiler: "If anyone invokes `writeObject()` or `readObject()` concerning a `Dog` object, use this code as part of the read and write."

Let's take a look at the preceding code.

1. Like most I/O-related methods `writeObject()` can throw exceptions. You can declare them or handle them, but we recommend handling them.
2. When you invoke `defaultWriteObject()` from within `writeObject()`, you're telling the JVM to do the normal serialization process for this object. When implementing `writeObject()`, you will typically request the normal serialization process *and* do some custom writing and reading, too.
3. In this case, we decided to write an extra `int` (the collar size) to the stream that's creating the serialized `Dog`. You can write extra stuff before and/or after you invoke `defaultWriteObject()`. But...when you read it back in, you have to read the extra stuff in the same order you wrote it.
4. Again, we chose to handle rather than declare the exceptions.
5. When it's time to deserialize, `defaultReadObject()` handles the normal deserialization you'd get if you didn't implement a `readObject()` method.
6. Finally, we build a new `Collar` object for the `Dog` using the collar size that we manually serialized. (We had to invoke `readInt()` *after* we invoked `defaultReadObject()` or the streamed data would be out of sync!)

Remember, the most common reason to implement `writeObject()` and `readObject()` is when you have to save some part of an object's state manually. If you choose, you can write and read *all* of the state yourself, but that's very rare. So, when you want to do only a *part* of the serialization/deserialization yourself, you *must* invoke the `defaultReadObject()` and `defaultWriteObject()` methods to do the rest.

Which brings up another question—why wouldn't *all* Java classes be serializable? Why isn't class `Object` serializable? There are some things in Java that simply cannot be serialized because they are runtime specific. Things like streams, threads, runtime, etc., and even some GUI classes (which are connected to the underlying OS) cannot be serialized. What is and is not serializable in the Java API is *not* part of the exam, but you'll need to keep them in mind if you're

serializing complex objects.

## How Inheritance Affects Serialization

Serialization is very cool, but in order to apply it effectively you're going to have to understand how your class's superclasses affect serialization.

**e x a m**  
watch

*If a superclass is Serializable, then, according to normal Java interface rules, all subclasses of that class automatically implement Serializable implicitly. In other words, a subclass of a class marked Serializable passes the IS-A test for Serializable and thus can be saved without having to explicitly mark the subclass as Serializable. You simply cannot tell whether a class is or is not Serializable unless you can see the class inheritance tree to see whether any other superclasses implement Serializable. If the class does not explicitly extend any other class and does not implement Serializable, then you know for certain that the class is not Serializable, because class Object does not implement Serializable.*

That brings up another key issue with serialization...what happens if a superclass is not marked Serializable, but the subclass is? Can the subclass still be serialized even if its superclass does not implement Serializable? Imagine this:

```
class Animal { }
class Dog extends Animal implements Serializable {
 // the rest of the Dog code
}
```

Now you have a Serializable Dog class with a non-Serializable superclass. This works! But there are potentially serious implications. To fully understand those implications, let's step back and look at the difference between an object that comes from deserialization versus an object created using new.

Remember, when an object is constructed using `new` (as opposed to being deserialized), the following things happen (in this order):

1. All instance variables are assigned default values.
2. The constructor is invoked, which immediately invokes the superclass constructor (or another overloaded constructor, until one of the overloaded constructors invokes the superclass constructor).
3. All superclass constructors complete.
4. Instance variables that are initialized as part of their declaration are assigned their initial value (as opposed to the default values they're given prior to the superclass constructors completing).
5. The constructor completes.

*But these things do not happen when an object is serialized.* When an instance of a serializable class is serialized, the constructor does not run and instance variables are not given their initially assigned values! Think about it—if the constructor were invoked and/or instance variables were assigned the values given in their declarations, the object you’re trying to restore would revert back to its original state, rather than coming back reflecting the changes in its state that happened sometime after it was created. For example, imagine you have a class that declares an instance variable and assigns it the `int` value `3` and includes a method that changes the instance variable value to `10`:

```
class Foo implements Serializable {
 int num = 3;
 void changeNum() { num = 10; }
}
```

Obviously, if you serialize a `Foo` instance *after* the `changeNum()` method runs, the value of the `num` variable should be `10`. When the `Foo` instance is serialized, you want the `num` variable to still be `10`! You obviously don’t want the initialization (in this case, the assignment of the value `3` to the variable `num`) to happen. Think of constructors and instance variable assignments together as part of one complete object initialization process (and, in fact, they do become one initialization method in the bytecode). The point is, when an object is serialized we do not want any of the normal initialization to happen. We don’t