The #1 Classroom-Proven IT Training and Exam Prep Tool

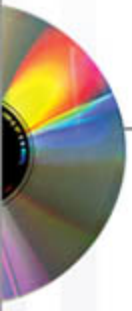
New for Java 6 from the Lead Developers of the Exam

Sun® Certified Programmer for Java™ 6 Study Guide

SCJP

Exam 310-065

100% Coverage—360+ Practice Exam Questions



Kathy Sierra
SCJP

Bert Bates
SCJP

- Three Full MasterExams
- Complete Electronic Book

LearnKey

McGraw Hill

Praise for the *Sun Certified Programmer & Developer for Java 2 Study Guide*

"Kathy Sierra is one of the few people in the world who can make complicated things seem damn simple. And as if that isn't enough, she can make boring things seem interesting. I always look forward to reading whatever Kathy writes—she's one of my favorite authors."

—Paul Wheaton, Trail Boss JavaRanch.com

"Who better to write a Java study guide than Kathy Sierra, the reigning queen of Java instruction? Kathy Sierra has done it again—here is a study guide that almost guarantees you a certification!"

—James Cubeta, Systems Engineer, SGI

"The thing I appreciate most about Kathy is her quest to make us all remember that we are teaching people and not just lecturing about Java. Her passion and desire for the highest quality education that meets the needs of the individual student is positively unparalleled at SunEd. Undoubtedly there are hundreds of students who have benefited from taking Kathy's classes."

—Victor Peters, founder Next Step Education & Software Sun Certified Java Instructor

"I want to thank Kathy for the EXCELLENT Study Guide. The book is well written, every concept is clearly explained using a real life example, and the book states what you specifically need to know for the exam. The way it's written, you feel that you're in a classroom and someone is actually teaching you the difficult concepts, but not in a dry, formal manner. The questions at the end of the chapters are also REALLY good, and I am sure they will help candidates pass the test. Watch out for this Wickedly Smart book."

—Alfred Raouf, Web Solution Developer

"The Sun Certification exam was certainly no walk in the park, but Kathy's material allowed me to not only pass the exam, but Ace it!"

—Mary Whetsel, Sr. Technology Specialist,
Application Strategy and Integration, The St. Paul Companies

"Bert has an uncanny and proven ability to synthesize complexity into simplicity offering a guided tour into learning what's needed for the certification exam."

—Thomas Bender, President, Gold Hill Software Design, Inc.

"With his skill for clearly expressing complex concepts to his training audience, every student can master what Bert has to teach."

—David Ridge, CEO, Ridge Associates

"I found this book to be extremely helpful in passing the exam. It was very well written with just enough light-hearted comments to make you forget that you were studying for a very difficult test. HIGHLY RECOMMENDED!!"

— Nicole Y. McCullough

"I have never enjoyed reading a technical book as much as I did this one…This morning I took the SCJP test and got 98% (60 out of 61) correct. Such success would not have been possible without this book!"

— Yurie Nagorny

"I gave SCJP 1.4 in July 2004 & scored 95% (58/61). Kathy & Bert have an awesome writing style & they literally burnt the core concepts into my head."

— Bhushan P. Madan (Kansas, United States)

"I just took my certification test last week and passed with a score of 95%. Had I not gone through this book, there would have been little chance of doing so well on the test. Thank you Kathy and Bert for a wonderful book!"

— Jon W. Kinsting (Saratoga, California United States)

"Don't hesitate to make this book your primary guide for SCJP 1.4 preparation. The authors have made a marvellous job about delivering the vital facts you need to know for the exam while leaving out tons of otherwise valuable data that fall beyond the scope. Both authors have participated in creating the real questions for the real exam thus providing an invaluable insight to discern the true nature of what you are up to doing. Unlike many other certification guides…this one makes perfect reading. The most boring Sun objectives in the book are nicely interwoven with the gems of refreshingly spicy humor."

— Vad Fogel (Ontario, Canada)

SCJP Sun® Certified Programmer for Java™ 6 Study Guide

(Exam 310-065)

This page intentionally left blank

SCJP Sun® Certified Programmer for Java™ 6 Study Guide

Exam (310-065)

Kathy Sierra
Bert Bates

New York Chicago San Francisco Lisbon London Madrid
Mexico City Milan New Delhi San Juan Seoul Singapore Sydney Toronto

0-07-159107-9

The material in this eBook also appears in the print version of this title: 0-07-159106-0.

DOI: 10.1036/0071591060

CONTRIBUTORS

About the Authors

Kathy Sierra was a lead developer for the SCJP exam for Java 5 and Java 6. Sierra 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 Sun's Java Champions program.

Bert Bates was a lead developer for many of Sun's Java certification exams including the SCJP for Java 5 and Java 6. He is also a forum moderator on JavaRanch.com, and has been developing software for more than 20 years. Bert is the co-author of several bestselling Java books, and he's a founding member of Sun's Java Champions program.

About the Technical Review Team

Johannes de Jong has been the leader of our technical review teams for ever and ever. (He has more patience than any three people we know.) For this 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 LearnKey

LearnKey provides self-paced learning content and multimedia delivery solutions to enhance personal skills and business productivity. LearnKey claims the largest library of rich streaming-media training content that engages learners in dynamic media-rich instruction complete with video clips, audio, full motion graphics, and animated illustrations. LearnKey can be found on the Web at www.LearnKey.com.

Technical Review Superstars


Andrew


Bill M.


Burk


Devender


Gian


Jef


Jeroen


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!

The Java 6 Elite Review Team

Fred

Marc P.

Marc W.

Since the upgrade to the Java 6 exam was a 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!

Mikalai

Christophe

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.

To the Java Community

CONTENTS AT A GLANCE

1	Declarations and Access Control	1
2	Object Orientation	85
3	Assignments	183
4	Operators	287
5	Flow Control, Exceptions, and Assertions	327
6	Strings, I/O, Formatting, and Parsing	425
7	Generics and Collections	541
8	Inner Classes	661
9	Threads	701
10	Development	789
A	About the CD	831
	Index	835

This page intentionally left blank

CONTENTS

Contributors . . . . . *vii*
Acknowledgments . . . . . *xx*
Preface . . . . . *xxi*
Introduction . . . . . *xxiii*

1 Declarations and Access Control **1**

Java Refresher 2
Identifiers & JavaBeans (Objectives 1.3 and 1.4) 4
Legal Identifiers 5
Sun's Java Code Conventions 6
JavaBeans Standards 8
Declare Classes (Exam Objective 1.1) 10
Source File Declaration Rules 11
Class Declarations and Modifiers 12
Exercise 1-1: Creating an Abstract Superclass and Concrete Subclass 18
Declare Interfaces (Exam Objectives 1.1 and 1.2) 19
Declaring an Interface 19
Declaring Interface Constants 22
Declare Class Members (Objectives 1.3 and 1.4) 24
Access Modifiers 24
Nonaccess Member Modifiers 39
Constructor Declarations 47
Variable Declarations 49
Declaring Enums 60
✓ Two-Minute Drill 68
Q&A Self Test 74
Self Test Answers 79

2 Object Orientation 85

Encapsulation (Exam Objective 5.1) 86
Inheritance, Is-A, Has-A (Exam Objective 5.5) 90
IS-A 94
HAS-A 96
Polymorphism (Exam Objective 5.2) 98
Overriding / Overloading (Exam Objectives 1.5 and 5.4) 103
Overridden Methods 103
Overloaded Methods 109
Reference Variable Casting (Objective 5.2) 116
Implementing an Interface (Exam Objective 1.2) 120
Legal Return Types (Exam Objective 1.5) 126
Return Type Declarations 126
Returning a Value 128
Constructors and Instantiation
(Exam Objectives 1.6, 5.3, and 5.4) 130
Determine Whether a Default Constructor
Will Be Created 135
Overloaded Constructors 139
Statics (Exam Objective 1.3) 145
Static Variables and Methods 145
Coupling and Cohesion (Exam Objective 5.1) 151
✓ Two-Minute Drill 157
Q&A Self Test 162
Self Test Answers 171

3 Assignments 183

Stack and Heap—Quick Review 184
Literals, Assignments, and Variables
(Exam Objectives 1.3 and 7.6) 186
Literal Values for All Primitive Types 186
Assignment Operators 190
Exercise 3-1: Casting Primitives 195
Using a Variable or Array Element That Is Uninitialized
and Unassigned 203
Local (Stack, Automatic) Primitives and Objects 207

Passing Variables into Methods (Objective 7.3) . . . 213
Passing Object Reference Variables . . . 213
Does Java Use Pass-By-Value Semantics? . . . 214
Passing Primitive Variables . . . 215
Array Declaration, Construction, and Initialization (Exam Objective 1.3) . . . 219
Declaring an Array . . . 219
Constructing an Array . . . 220
Initializing an Array . . . 224
Initialization Blocks . . . 234
Using Wrapper Classes and Boxing (Exam Objective 3.1) . . . 237
An Overview of the Wrapper Classes . . . 238
Creating Wrapper Objects . . . 239
Using Wrapper Conversion Utilities . . . 240
Autoboxing . . . 244
Overloading (Exam Objectives 1.5 and 5.4) . . . 247
Garbage Collection (Exam Objective 7.4) . . . 254
Overview of Memory Management and Garbage Collection . . . 254
Overview of Java's Garbage Collector . . . 255
Writing Code That Explicitly Makes Objects Eligible for Collection . . . 257
Exercise 3-2: Garbage Collection Experiment . . . 262
✓ Two-Minute Drill . . . 265
Q&A Self Test . . . 269
Self Test Answers . . . 277

4 Operators . . . 287

Java Operators (Exam Objective 7.6) . . . 288
Assignment Operators . . . 288
Relational Operators . . . 290
instanceof Comparison . . . 295
Arithmetic Operators . . . 298
Conditional Operator . . . 304
Logical Operators . . . 305
✓ Two-Minute Drill . . . 311
Q&A Self Test . . . 313
Self Test Answers . . . 319

5 Flow Control, Exceptions, and Assertions 327

if and switch Statements (Exam Objective 2.1) 328
if-else Branching 329
switch Statements 334
Exercise 5-1: Creating a switch-case Statement 342
Loops and Iterators (Exam Objective 2.2) 343
Using while Loops 343
Using do Loops 344
Using for Loops 345
Using break and continue 352
Unlabeled Statements 353
Labeled Statements 354
Exercise 5-2: Creating a Labeled while Loop 356
Handling Exceptions (Exam Objectives 2.4 and 2.5) 356
Catching an Exception Using try and catch 357
Using finally 359
Propagating Uncaught Exceptions 362
Exercise 5-3: Propagating and Catching an Exception 364
Defining Exceptions 365
Exception Hierarchy 366
Handling an Entire Class Hierarchy of Exceptions 368
Exception Matching 369
Exception Declaration and the Public Interface 371
Rethrowing the Same Exception 376
Exercise 5-4: Creating an Exception 377
Common Exceptions and Errors(Exam Objective 2.6) 378
Working with the Assertion Mechanism (Exam Objective 2.3) . . . 383
Assertions Overview 384
Enabling Assertions 387
Using Assertions Appropriately 391
✓ Two-Minute Drill 397
Q&A Self Test 401
Self Test Answers 411

6 Strings, I/O, Formatting, and Parsing 425

String, StringBuilder, and StringBuffer (Exam Objective 3.1) . . . 426
The String Class . . . 426
Important Facts About Strings and Memory . . . 433
Important Methods in the String Class . . . 434
The StringBuffer and StringBuilder Classes . . . 438
Important Methods in the StringBuffer and StringBuilder Classes . . . 440
File Navigation and I/O (Exam Objective 3.2) . . . 443
The java.io.Console Class . . . 457
Serialization (Exam Objective 3.3) . . . 459
Dates, Numbers, and Currency (Exam Objective 3.4) . . . 473
Working with Dates, Numbers, and Currencies . . . 474
Parsing, Tokenizing, and Formatting (Exam Objective 3.5) . . . 487
A Search Tutorial . . . 488
Locating Data via Pattern Matching . . . 498
Tokenizing . . . 501
Formatting with printf() and format() . . . 506
✓ Two-Minute Drill . . . 511
Q&A Self Test . . . 515
Self Test Answers . . . 526

7 Generics and Collections 541

Overriding hashCode() and equals() (Objective 6.2) . . . 542
Overriding equals() . . . 544
Overriding hashCode() . . . 549
Collections (Exam Objective 6.1) . . . 556
So What Do You Do with a Collection? . . . 556
List Interface . . . 561
Set Interface . . . 562
Map Interface . . . 563
Queue Interface . . . 564
Using the Collections Framework (Objectives 6.3 and 6.5) . . . 566
ArrayList Basics . . . 567
Autoboxing with Collections . . . 568
Sorting Collections and Arrays . . . 568
Navigating (Searching) TreeSets and TreeMaps . . . 586
Other Navigation Methods . . . 587
Backed Collections . . . 589

Generic Types (Objectives 6.3 and 6.4) . . . 595
Generics and Legacy Code . . . 600
Mixing Generic and Non-generic Collections . . . 601
Polymorphism and Generics . . . 607
Generic Methods . . . 609
Generic Declarations . . . 622
✓ Two-Minute Drill . . . 631
Q&A Self Test . . . 636
Self Test Answers . . . 647

8 Inner Classes . . . 661

Inner Classes . . . 663
Coding a "Regular" Inner Class . . . 664
Referencing the Inner or Outer Instance from Within the Inner Class . . . 668
Method-Local Inner Classes . . . 670
What a Method-Local Inner Object Can and Can't Do . . . 671
Anonymous Inner Classes . . . 673
Plain-Old Anonymous Inner Classes, Flavor One . . . 673
Plain-Old Anonymous Inner Classes, Flavor Two . . . 677
Argument-Defined Anonymous Inner Classes . . . 678
Static Nested Classes . . . 680
Instantiating and Using Static Nested Classes . . . 681
✓ Two-Minute Drill . . . 683
Q&A Self Test . . . 685
Self Test Answers . . . 692

9 Threads . . . 701

Defining, Instantiating, and Starting Threads (Objective 4.1) . . . 702
Defining a Thread . . . 705
Instantiating a Thread . . . 706
Starting a Thread . . . 709
Thread States and Transitions (Objective 4.2) . . . 718
Thread States . . . 718
Preventing Thread Execution . . . 720
Sleeping . . . 721
Exercise 9-1: Creating a Thread and Putting It to Sleep . . . 723
Thread Priorities and yield( ) . . . 724

Synchronizing Code (Objective 4.3) 728
Synchronization and Locks 735
Exercise 9-2: Synchronizing a Block of Code 738
Thread Deadlock 745
Thread Interaction (Objective 4.4) 746
Using notifyAll( ) When Many Threads May Be Waiting 752
✓ Two-Minute Drill 758
Q&A Self Test 761
Self Test Answers 772
Exercise Answers 787

10 Development 789

Using the javac and java Commands (Exam Objectives 7.1, 7.2, and 7.5) 790
Compiling with javac 790
Launching Applications with java 793
Searching for Other Classes 796
JAR Files (Objective 7.5) 802
JAR Files and Searching 803
Using Static Imports (Exam Objective 7.1) 806
Static Imports 806
✓ Two-Minute Drill 809
Q&A Self Test 811
Self Test Answers 820

A About the CD 831

System Requirements 832
Installing and Running Master Exam 832
Master Exam 832
Electronic Book 833
Help 833
Removing Installation(s) 833
Technical Support 833
LearnKey Technical Support 833

Index 835

ACKNOWLEDGMENTS

Kathy and Bert would like to thank the following people:

- All the incredibly hard-working folks at McGraw-Hill: Tim Green, Jim Kussow, Jody McKenzie, Madhu Bhardwaj, and Jennifer Housh for all their help, and for being so responsive and patient—well, okay, not all that patient—but so professional and the nicest group of people you could hope to work with.
- To our saviors Solveig Haugland and Midori Batten, for coming to our rescue when we were really in a bind!
- Some of the software professionals and friends who helped us in the early days: Tom Bender, Peter Loerincs, Craig Matthews, Dave Gustafson, Leonard Coyne, Morgan Porter, and Mike Kavenaugh.
- The 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, Bryan Basham, Kathy Collina, and Simon Roberts.
- 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.
- To Darren and Mary, thanks for keeping us sane and for helping us with our new furry friends Andi, Kara, Birta, Sola, Draumur, and Tjara.
- Finally, to Eric and Beth Freeman for your continued inspiration.

PREFACE

This book's primary objective is to help you prepare for and pass Sun Microsystem's SCJP certification for Java 6 or Java 5. The Java 6 and Java 5 exams are almost identical in scope, and they are both much broader than their predecessor, the Java 1.4 exam. For the remainder of this book we'll typically reference the Java 6 exam, but remember that other than the addition of the System.Console class and Navigable collections, the Java 5 and Java 6 exams are identical in scope. We recommend that you take the Java 6 exam and not the Java 5 exam, but if you do decide to take the Java 5 exam, this book is still appropriate. The new exam's objectives touch on many of the more commonly used of Java's APIs. The key word here is "touch." The exam's creators intended that passing the exam will demonstrate that the candidate understands the *basics* of APIs such as those for file I/O and regular expressions. This book follows closely both the breadth and the depth of the real exam. For instance, after reading this book, you probably won't emerge as a regex guru, but if you study the material, and do well on the self tests, you'll have a basic understanding of regex, 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 Sun has established for the exam.

In This Book

This book is organized to optimize your learning of the topics covered by the SCJP Java 6 exam. Whenever possible, we've organized the chapters to parallel the Sun 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 addition to fully covering the SCJP Java 6 exam, we have also included on the CD eight chapters that cover important aspects of Sun's SCJD exam.

In Every Chapter

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 every chapter:

- 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!

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

- **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.

- **Exercises** are interspersed throughout the chapters. They 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.

- **From the Classroom** sidebars describe the issues that come up most often in the training classroom setting. These sidebars give you a valuable perspective into certification- and product-related topics. They point out common mistakes and address questions that have arisen from classroom discussions.

- 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.

- 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 Sun Certified Programmer for both the Java 6 and Java 5 exams, for 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. The CD included with the book also includes an in-depth review of the essential ingredients for a successful assessment of a project submitted for the Sun Certified Java Developer exam.

What This Book Is Not

You will not find a beginner's guide to learning Java in this book. All 800 pages of this book are dedicated solely to helping you pass the exams. If you are brand new to Java, we suggest you spend a little time learning the basics. You shouldn't start with this book until you know how to write, compile, and run simple Java programs. 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.

On the CD

For more information on the CD-ROM, please see Appendix A.

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!

Introduction to the Material in the Book

The Sun Certified Java Programmer (SCJP) 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 go 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 different 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.

The Sun Certified Developer Exam (covered in chapters that are contained on the CD) is unique to the IT certification realm, because it actually evaluates your skill as a developer rather than simply your knowledge of the language or tools. Becoming a Certified Java Developer is, by definition, a development experience.

Who Cares About Certification?

Employers do. Headhunters do. Programmers do. Sun's programmer exam has been considered the fastest-growing certification in the IT world, and the number of candidates taking the exam continues to grow each year. 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 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. To really demonstrate your ability to develop (as opposed to just your knowledge of the language), you should consider pursuing the Developer Exam, where you're given an assignment to build a program, start to finish, and submit it for an assessor to evaluate and score.

Sun's Certification Program

Currently there are eight Java certification exams (although several of them might have more than one live version). The Associate exam, the Programmer exam, and the Developer exam are all associated with the Java Standard Edition. The Web Component exam, the Business Component exam, the Web Services exam, and the Enterprise Architect exam are all associated with the Java Enterprise Edition. The Mobile Application exam is associated with the Java Micro Edition.

The Associate, Programmer, Web Component, Business Component, Web Services, and Mobile Application exams are exclusively multiple-choice and drag-and-drop exams taken at a testing center, while the Developer and Architect exams also involve submitting a project.

The Associate Exam (CX-310-019)

Sun Certified Java Associate (SCJA)
The Associate exam is for candidates just entering an application development or a software project management career using Java technologies. This exam tests basic knowledge of object-oriented concepts, the basics of UML, the basics of the Java programming language, and general knowledge of Java Platforms and Technologies. This exam has no prerequisites.

The Programmer Exams (CX-310-065)

Sun Certified Java Programmer (SCJP) for Java 6
The Programmer exam is designed to test your knowledge of the Java programming language itself. It requires detailed knowledge of language syntax, core concepts, and a number of common application programming interfaces (APIs). This exam also tests intermediate knowledge of object-oriented design concepts. It does not test any issues related to architecture, and it does not ask why one approach is better than another, but rather it asks whether the given approach works in a particular situation. This exam has no prerequisites. As of May, 2008, two older versions of this exam are still available, the 1.4 and the 5.0.

The Developer Exam (CX-310-252A, CX-310-027)

Sun Certified Java Developer (SCJD)
The Developer exam picks up where the Programmer exam leaves off. Passing the Programmer exam is required before you can start the Developer exam. The Developer exam requires you to develop an actual program and then defend your design decisions. It is designed to test your understanding of why certain approaches are better than others in certain circumstances, and to prove your ability to follow a specification and implement a correct, functioning, and user-friendly program.

The Developer exam consists of two pieces: a project assignment and a follow-up essay exam. Candidates have an unlimited amount of time to complete the project, but once the project is submitted, the candidate then must go to a testing center and complete a short follow-up essay exam, designed primarily to validate and verify that it was you who designed and built the project.

The Web Component Developer Exam (CX-310-083)

Sun Certified Web Component Developer for Java EE Platform (SCWCD)
The web developer exam is for those who are using Java servlet and JSP (Java Server Pages) technologies to build Web applications. It's based on the Servlet and JSP specifications defined in the Java Enterprise Edition (Java EE). This exam requires that the candidate is a Sun Certified Java Programmer.

The Business Component Developer Exam (CX-310-091)

Sun Certified Business Component Developer for Java EE Platform (SCBCD)
The business component developer exam is for those candidates who are using Java EJB technology to build business-tier applications. The exam is based on the EJB specification defined in the Java Enterprise Edition (Java EE). This exam requires that the candidate is a Sun Certified Java Programmer.

The Web Services Developer Exam (CX-310-220)

Sun Certified Developer for Web Services for Java EE Platform (SCDJWS)
The web services exam is for those candidates who are building applications using Java EE and Java Web Services Developer Pack technologies. This exam requires that the candidate is a Sun Certified Java Programmer.

The Architect Exam (CX-310-052, CX-310-301A, CX-310-062)

Sun Certified Enterprise Architect for J2EE Technology (SCEA)
This certification is for enterprise architects, and thus does not require that the candidate pass the Programmer exam. The Architect exam is in three pieces: a knowledge-based multiple-choice exam, an architectural design assignment, and a follow-up essay exam. You must successfully pass the multiple-choice exam before registering and receiving the design assignment.

The Mobile Exam (CX-310-110)

Sun Certified Mobile Application Developer for Java ME (SCMAD)
The mobile application developer exam is for candidates creating applications for cell phones or other Java enabled devices. The exam covers the Java Technology for Wireless Industry (JTWI) specification, the Wireless Messaging API, and Mobile Media APIs. This exam requires that the candidate is an SCJP.

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, Sun evaluates candidates using a computer-based testing service operated by Sylvan Prometric. This service is quite popular in the industry, and it is used for a number of vendor certification programs, including Novell's CNE and Microsoft's MCSE. Thanks to Sylvan Prometric's large number of facilities, exams can be administered worldwide, generally in the same town as a prospective candidate.

For the most part, Sylvan Prometric exams work similarly from vendor to vendor. However, there is an important fact to know about Sun's exams: they use the traditional Sylvan Prometric test format, not the newer adaptive format. This gives the candidate an advantage, since the traditional format allows answers to be reviewed and revised during the test.

exam Watch

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 second-guess every answer and talk yourself out of a correct one.

To discourage simple memorization, Sun 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 Sun exam has a specific number of questions (the Programmer's exam contains 72 questions) and test duration (210 minutes for the Programmer's exam) is designed to be generous. The time remaining is always displayed in the corner of the testing screen, along with the number of remaining questions. If time expires during an exam, the test terminates, and incomplete answers are counted as incorrect.

At the end of the exam, your test is immediately graded, and the results are displayed on the screen. Scores for each subject area are also provided, but the system will not indicate which specific questions were missed. A report is automatically printed at the proctor's desk for your files. The test score is electronically transmitted back to Sun.

exam Watch

When you find yourself stumped answering multiple-choice questions, use your scratch paper 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 re-think 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.

Question Format

Sun's Java exams pose questions in either multiple-choice or drag-and-drop formats.

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 are closely matched to 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.)

Drag-and-Drop Questions

Although many of the other Sun Java certification exams have been using drag-and-drop questions for several years, this is the first version of the SCJP exam that includes drag-and-drop questions. As we discussed earlier, the exam questions you receive are randomized, but you should expect that about 20–25% of the questions you encounter will be drag-and-drop style.

Drag-and-drop questions typically consist of three components:

- **A scenario** A short description of the task you are meant to complete.
- **A partially completed task** A code listing, a table, or a directory tree. The partially completed task will contain empty slots, which are indicated with (typically yellow) boxes. These boxes need to be filled to complete the task.
- **A set of possible "fragment" answers** You will click on fragments (typically blue boxes) and drag-and-drop them into the correct empty slots. The question's scenario will tell you whether you can reuse fragments.

Most drag-and-drop questions will have anywhere from 4 to 10 empty slots to fill, and typically a few more fragments than are needed (usually some fragments are left unused). Drag-and-drop questions are often the most complex on the exam, and the number of possible answer combinations makes them almost impossible to guess.

exam Watch

In regards to drag-and-drop questions, there is a huge problem with the testing software at many of the Prometric centers world-wide. In general, the testing software allows you to review questions you've already answered as often as you'd like.

In the case of drag-and-drop questions, however, many candidates have reported that if they choose to review a question, the software will erase their previous answer! BE CAREFUL! Until this problem is corrected, we recommend that you keep a list of which questions are drag and drop, so that you won't review one unintentionally. Another good idea is to write down your drag-and-drop answers so that if one gets erased it will be less painful to recreate the answer.

This brings us to another issue that some candidates have reported. The testing center is supposed to provide you with sufficient writing implements so that you can work problems out "on paper." In some cases, the centers have provided inadequate markers and dry-erase boards which are too small and cumbersome to use effectively. We recommend that you call ahead and verify that you will be supplied with actual pencils and at least several sheets of blank paper.

Tips on Taking the Exam

There are 72 questions on the 310-065 (Java 6) exam. You will need to get at least 47 of them correct to pass—around 65%. You are given over three hours to complete the exam. This information is subject to change. Always check with Sun before taking the exam, at www.suned.sun.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 parenthesis 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.

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 exam. A flash card is simply a 3 x 5 or 4 x 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%), they almost always report that they wrote a lot of code.

Scheduling Your Exam

The Sun exams are purchased directly from Sun, but are scheduled through Sylvan Prometric. For locations outside the United States, your local number can be found on Sylvan's Web site at http://www.2test.com. Sylvan representatives can schedule your exam, but they don't have information about the certification programs. Questions about certifications should be directed to Sun's Worldwide Training department. These representatives are familiar enough with the exams to find them by name, but it's best if you have the exam number handy when you call. You wouldn't want to be scheduled and charged for the wrong exam.

Exams can be scheduled up to a year in advance, although it's really not necessary. Generally, scheduling a week or two ahead is sufficient to reserve the day and time you prefer. When scheduling, operators will search for testing centers in your area. For convenience, they can also tell which testing centers you've used before.

When registering for the exam, you will be asked for your ID number. This number is used to track your exam results back to Sun. It's important that you use the same ID number each time you register, so that Sun can follow your progress. Address information provided when you first register is also used by Sun to ship certificates and other related material. In the United States, your Social Security Number is commonly used as your ID number. However, Sylvan can assign you a unique ID number if you prefer not to use your Social Security Number.

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 sign in with the exam administrator. In order to sign in, you need to provide two forms of identification. Acceptable forms include government-issued IDs (for example, passport or driver's license), credit cards, and company ID badges. One form of ID must include a photograph. 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 paper and a pencil. Some testing centers may provide a small marker board instead (we recommend that you don't settle for a whiteboard). We do recommend that you bring a water bottle. Three hours is a long time to keep your brain active, and it functions much better when well hydrated.

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, the exam administrator logs onto your exam, and you have to verify that your ID number and the exam number are correct. If this is the first time you've taken a Sun test, you can select a brief tutorial of the exam software. Before the test begins, you will be provided with facts about the exam, including the duration, the number of questions, and the score required for passing. The odds are good that you will be asked to fill out a brief survey before the exam actually begins. This survey will ask you about your level of Java experience. 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 is Windows-based, but you won't have access to the main desktop or any of the accessories. The exam is presented in full screen, with a single question per screen. Navigation buttons allow you to move forward and backward between questions. In the upper-right corner of the screen, counters show the number of questions and time remaining. Most important, there is a Mark check box in the upper-left corner of 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 side-tracked 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 25 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 50–60 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 10–20 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

(although you'll probably want to skip a review of the drag-and-drop questions!), 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 what will seem like the longest 10 seconds of your life, the testing software will respond with your score. This is usually displayed as a bar graph, showing the minimum passing score, your score, and a PASS/FAIL indicator.

If you're curious, you can review the statistics of your score at this time. Answers to specific questions are not presented; rather, questions are lumped into categories, and results are tallied for each category. This detail is also on a report that has been automatically printed at the exam administrator's desk.

As you leave, you'll need to leave your scratch paper behind or return it to the administrator. (Some testing centers track the number of sheets you've been given, so be sure to return them all.) In exchange, you'll receive a copy of the test report.

This report will be embossed with the testing center's seal, and you should keep it in a safe place. Normally, the results are automatically transmitted to Sun, but occasionally you might need the paper report to prove that you passed the exam.

In a few weeks, Sun will send you a package in the mail containing a nice paper certificate, a lapel pin, and a letter. You may also be sent instructions for how to obtain artwork for a logo that you can use on personal business cards.

Re-Testing

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 know the format of the test a little better, and the report shows which areas you need to strengthen.

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 Sun 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!

1 Declarations and Access Control

CERTIFICATION OBJECTIVES

- Declare Classes & Interfaces
- Develop Interfaces & Abstract Classes
- Use Primitives, Arrays, Enums, & Legal Identifiers
- Use Static Methods, JavaBeans Naming, & Var-Args

✓ Two-Minute Drill

Q&A Self Test

We assume that because you're planning on becoming certified, you already know the basics of Java. If you're completely new to the language, this chapter—and the rest of the book—will be confusing; so be sure you know at least the basics of the language before diving into this book. That said, we're starting with a brief, high-level refresher to put you back in the Java mood, in case you've been away for awhile.

Java Refresher

A Java program is mostly a collection of *objects* talking to other objects by invoking each other's *methods*. Every object is of a certain *type*, and that type is defined by a *class* or an *interface*. Most Java programs use a collection of objects of many different types.

- **Class** A template that describes the kinds of state and behavior that objects of its type support.
- **Object** At runtime, when the Java Virtual Machine (JVM) encounters the `new` keyword, it will use the appropriate class to make an object which is an instance of that class. That object will have its own state, and access to all of the behaviors defined by its class.
- **State (instance variables)** Each object (instance of a class) will have its own unique set of instance variables as defined in the class. Collectively, the values assigned to an object's instance variables make up the object's state.
- **Behavior (methods)** When a programmer creates a class, she creates methods for that class. Methods are where the class' logic is stored. Methods are where the real work gets done. They are where algorithms get executed, and data gets manipulated.

Identifiers and Keywords

All the Java components we just talked about—classes, variables, and methods—need names. In Java these names are called *identifiers*, and, as you might expect, there are rules for what constitutes a legal Java identifier. Beyond what's *legal*,

though, Java programmers (and Sun) have created *conventions* for naming methods, variables, and classes.

Like all programming languages, Java has a set of built-in *keywords*. These keywords must *not* be used as identifiers. Later in this chapter we'll review the details of these naming rules, conventions, and the Java keywords.

Inheritance

Central to Java and other object-oriented (OO) languages is the concept of *inheritance*, which allows code defined in one class to be reused in other classes. In Java, you can define a general (more abstract) superclass, and then extend it with more specific subclasses. The superclass knows nothing of the classes that inherit from it, but all of the subclasses that inherit from the superclass must explicitly declare the inheritance relationship. A subclass that inherits from a superclass is automatically given accessible instance variables and methods defined by the superclass, but is also free to override superclass methods to define more specific behavior.

For example, a Car *super*class class could define general methods common to all automobiles, but a Ferrari *sub*class could override the `accelerate()` method.

Interfaces

A powerful companion to inheritance is the use of interfaces. Interfaces are like a 100-percent abstract superclass that defines the methods a subclass must support, but not *how* they must be supported. In other words, an `Animal` interface might declare that all `Animal` implementation classes have an `eat()` method, but the `Animal` interface doesn't supply any logic for the `eat()` method. That means it's up to the classes that implement the Animal interface to define the actual code for how that particular Animal type behaves when its `eat()` method is invoked.

Finding Other Classes

As we'll see later in the book, it's a good idea to make your classes *cohesive*. That means that every class should have a focused set of responsibilities. For instance, if you were creating a zoo simulation program, you'd want to represent aardvarks with one class, and zoo visitors with a different class. In addition, you might have a Zookeeper class, and a Popcorn vendor class. The point is that you don't want a class that has both Aardvark *and* Popcorn behaviors (more on that in Chapter 2).

Even a simple Java program uses objects from many different classes: some that *you* created, and some built by others (such as Sun's Java API classes). Java organizes classes into *packages*, and uses *import* statements to give programmers a consistent

way to manage naming of, and access to, classes they need. The exam covers a *lot* of concepts related to packages and class access; we'll explore the details in this—and later—chapters.

CERTIFICATION OBJECTIVE

Identifiers & JavaBeans (Objectives 1.3 and 1.4)

1.3 Develop code that declares, initializes, and uses primitives, arrays, enums, and objects as static, instance, and local variables. Also, use legal identifiers for variable names.

1.4 Develop code that declares both static and non-static methods, and—if appropriate—use method names that adhere to the JavaBeans naming standards. Also develop code that declares and uses a variable-length argument list.

Remember that when we list one or more Certification Objectives in the book, as we just did, it means that the following section covers at least some part of that objective. Some objectives will be covered in several different chapters, so you'll see the same objective in more than one place in the book. For example, this section covers declarations, identifiers, and JavaBeans naming, but *using* the things you declare is covered primarily in later chapters.

So, we'll start with Java identifiers. The three aspects of Java identifiers that we cover here are

- **Legal Identifiers** The rules the compiler uses to determine whether a name is legal.
- **Sun's Java Code Conventions** Sun's recommendations for naming classes, variables, and methods. We typically adhere to these standards throughout the book, except when we're trying to show you how a tricky exam question might be coded. You won't be asked questions about the Java Code Conventions, but we strongly recommend that programmers use them.
- **JavaBeans Naming Standards** The naming requirements of the JavaBeans specification. You don't need to study the JavaBeans spec for the exam, but you do need to know a few basic JavaBeans naming rules we cover in this chapter.

Legal Identifiers

Technically, legal identifiers must be composed of only Unicode characters, numbers, currency symbols, and connecting characters (like underscores). The exam doesn't dive into the details of which ranges of the Unicode character set are considered to qualify as letters and digits. So, for example, you won't need to know that Tibetan digits range from `\u0420` to `\u0f29`. Here are the rules you *do* need to know:

- Identifiers must start with a letter, a currency character ($), or a connecting character such as the underscore (_). Identifiers cannot start with a number!
- After the first character, identifiers can contain any combination of letters, currency characters, connecting characters, or numbers.
- In practice, there is no limit to the number of characters an identifier can contain.
- You can't use a Java keyword as an identifier. Table 1-1 lists all of the Java keywords including one new one for 5.0, `enum`.
- Identifiers in Java are case-sensitive; `foo` and `FOO` are two different identifiers.

Examples of legal and illegal identifiers follow, first some legal identifiers:

```
int _a;
int $c;
int ______2_w;
int _$;
int this_is_a_very_detailed_name_for_an_identifier;
```

The following are illegal (it's your job to recognize why):

```
int :b;
int -d;
int e#;
int .f;
int 7g;
```

TABLE 1-1 Complete List of Java Keywords (`assert` added in 1.4, `enum` added in 1.5)

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

Sun's Java Code Conventions

Sun estimates that over the lifetime of a standard piece of code, 20 percent of the effort will go into the original creation and testing of the code, and 80 percent of the effort will go into the subsequent maintenance and enhancement of the code. Agreeing on, and coding to, a set of code standards helps to reduce the effort involved in testing, maintaining, and enhancing any piece of code. Sun has created a set of coding standards for Java, and published those standards in a document cleverly titled "Java Code Conventions," which you can find at `java.sun.com`. It's a great document, short and easy to read and we recommend it highly.

That said, you'll find that many of the questions in the exam don't follow the code conventions, because of the limitations in the test engine that is used to deliver the exam internationally. One of the great things about the Sun certifications is that the exams are administered uniformly throughout the world. In order to achieve that, the code listings that you'll see in the real exam are often quite cramped, and do not follow Sun's code standards. In order to toughen you up for the exam, we'll often present code listings that have a similarly cramped look and feel, often indenting our code only two spaces as opposed to the Sun standard of four.

We'll also jam our curly braces together unnaturally, and sometimes put several statements on the same line…ouch! For example:

```
1. class Wombat implements Runnable {
2.   private int i;
3.   public synchronized void run() {
4.     if (i%5 != 0) { i++; }
5.     for(int x=0; x<5; x++, i++)
```

```
 6.       { if (x > 1) Thread.yield(); }
 7.     System.out.print(i + " ");
 8.   }
 9.   public static void main(String[] args) {
10.     Wombat n = new Wombat();
11.     for(int x=100; x>0; --x) { new Thread(n).start(); }
12. } }
```

Consider yourself forewarned—you'll see lots of code listings, mock questions, and real exam questions that are this sick and twisted. Nobody wants you to write your code like this. Not your employer, not your coworkers, not us, not Sun, and not the exam creation team! Code like this was created only so that complex concepts could be tested within a universal testing tool. The one standard that *is* followed as much as possible in the real exam are the naming standards. Here are the naming standards that Sun recommends, and that we use in the exam and in most of the book:

- **Classes and interfaces** The first letter should be capitalized, and if several words are linked together to form the name, the first letter of the inner words should be uppercase (a format that's sometimes called "camelCase"). For classes, the names should typically be nouns. For example:

  ```
  Dog
  Account
  PrintWriter
  ```

  For interfaces, the names should typically be adjectives like

  ```
  Runnable
  Serializable
  ```

- **Methods** The first letter should be lowercase, and then normal camelCase rules should be used. In addition, the names should typically be verb-noun pairs. For example:

  ```
  getBalance
  doCalculation
  setCustomerName
  ```

- **Variables** Like methods, the camelCase format should be used, starting with a lowercase letter. Sun recommends short, meaningful names, which sounds good to us. Some examples:

```
buttonWidth
accountBalance
myString
```

- **Constants** Java constants are created by marking variables `static` and `final`. They should be named using uppercase letters with underscore characters as separators:

```
MIN_HEIGHT
```

JavaBeans Standards

The JavaBeans spec is intended to help Java developers create Java components that can be easily used by other Java developers in a visual Integrated Development Environment (IDE) tool (like Eclipse or NetBeans). As a Java programmer, you want to be able to use components from the Java API, but it would be great if you could also buy the Java component you want from "Beans 'R Us," that software company down the street. And once you've found the components, you'd like to be able to access them through a development tool in such a way that you don't have to write all your code from scratch. By using naming rules, the JavaBeans spec helps guarantee that tools can recognize and use components built by different developers. The JavaBeans API is quite involved, but you'll need to study only a few basics for the exam.

First, JavaBeans are Java classes that have *properties*. For our purposes, think of properties as `private` instance variables. Since they're `private`, the only way they can be accessed from outside of their class is through *methods* in the class. The methods that change a property's value are called *setter* methods, and the methods that retrieve a property's value are called *getter* methods. The JavaBean naming rules that you'll need to know for the exam are the following:

JavaBean Property Naming Rules

- If the property is not a boolean, the getter method's prefix must be *get*. For example, `getSize()` is a valid JavaBeans getter name for a property named "size." Keep in mind that you do not need to have a variable named *size*

(although some IDEs expect it). The name of the property is *inferred* from the getters and setters, not through any variables in your class. What you return from `getSize()` is up to you.

- If the property is a boolean, the getter method's prefix is either `get` or `is`. For example, `getStopped()` or `isStopped()` are both valid JavaBeans names for a boolean property.
- The setter method's prefix must be *set*. For example, `setSize()` is the valid JavaBean name for a property named *size*.
- To complete the name of a getter or setter method, change the first letter of the property name to uppercase, and then append it to the appropriate prefix (`get`, `is`, or `set`).
- Setter method signatures must be marked `public`, with a `void` return type and an argument that represents the property type.
- Getter method signatures must be marked `public`, take no arguments, and have a return type that matches the argument type of the setter method for that property.

Second, the JavaBean spec supports *events*, which allow components to notify each other when something happens. The event model is often used in GUI applications when an event like a mouse click is multicast to many other objects that may have things to do when the mouse click occurs. The objects that receive the information that an event occurred are called *listeners*. For the exam, you need to know that the methods that are used to add or remove listeners from an event must also follow JavaBean naming standards:

JavaBean Listener Naming Rules

- Listener method names used to "register" a listener with an event source must use the prefix `add`, followed by the listener type. For example, `addActionListener()` is a valid name for a method that an event source will have to allow others to register for Action events.
- Listener method names used to remove ("unregister") a listener must use the prefix `remove`, followed by the listener type (using the same rules as the registration add method).
- The type of listener to be added or removed must be passed as the argument to the method.
- Listener method names must end with the word "Listener".

Examples of valid JavaBean method signatures are

```
public void setMyValue(int v)
public int getMyValue()
public boolean isMyStatus()
public void addMyListener(MyListener m)
public void removeMyListener(MyListener m)
```

Examples of *invalid* JavaBean method signatures are

```
void setCustomerName(String s)          // must be public
public void modifyMyValue(int v)        // can't use 'modify'
public void addXListener(MyListener m) // listener type mismatch
```

exam Watch

The objective says you have to know legal identifiers only for variable names, but the rules are the same for ALL Java components. So remember that a legal identifier for a variable is also a legal identifier for a method or a class. However, you need to distinguish between legal identifiers and naming conventions, such as the JavaBeans standards, that indicate how a Java component should be named. In other words, you must be able to recognize that an identifier is legal even if it doesn't conform to naming standards. If the exam question is asking about naming conventions—not just whether an identifier will compile—JavaBeans will be mentioned explicitly.

CERTIFICATION OBJECTIVE

Declare Classes (Exam Objective 1.1)

1.1 Develop code that declares classes (including abstract and all forms of nested classes), interfaces, and enums, and includes the appropriate use of package and import statements (including static imports).

When you write code in Java, you're writing classes or interfaces. Within those classes, as you know, are variables and methods (plus a few other things). How you declare your classes, methods, and variables dramatically affects your code's behavior. For example, a `public` method can be accessed from code running anywhere in your application. Mark that method `private`, though, and it vanishes from everyone's radar (except the class in which it was declared). For this objective, we'll study the ways in which you can declare and modify (or not) a class. You'll find that we cover modifiers in an extreme level of detail, and though we know you're already familiar with them, we're starting from the very beginning. Most Java programmers think they know how all the modifiers work, but on closer study often find out that they don't (at least not to the degree needed for the exam). Subtle distinctions are everywhere, so you need to be absolutely certain you're completely solid on everything in this section's objectives before taking the exam.

Source File Declaration Rules

Before we dig into class declarations, let's do a quick review of the rules associated with declaring classes, `import` statements, and `package` statements in a source file:

- There can be only one `public` class per source code file.
- Comments can appear at the beginning or end of any line in the source code file; they are independent of any of the positioning rules discussed here.
- If there *is* a `public` class in a file, the name of the file must match the name of the `public` class. For example, a class declared as `public class Dog { }` must be in a source code file named `Dog.java`.
- If the class is part of a package, the `package` statement must be the first line in the source code file, before any `import` statements that may be present.
- If there are `import` statements, they must go *between* the `package` statement (if there is one) and the class declaration. If there isn't a `package` statement, then the `import` statement(s) must be the first line(s) in the source code file. If there are no `package` or `import` statements, the class declaration must be the first line in the source code file.
- `import` and `package` statements apply to *all* classes within a source code file. In other words, there's no way to declare multiple classes in a file and have them in different packages, or use different imports.
- A file can have more than one nonpublic class.

- Files with no public classes can have a name that does not match any of the classes in the file.

In Chapter 10 we'll go into a lot more detail about the rules involved with declaring and using imports, packages, and a feature new to Java 5, static imports.

Class Declarations and Modifiers

Although nested (often called inner) classes are on the exam, we'll save nested class declarations for Chapter 8. You're going to love that chapter. 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. Modifiers fall into two categories:

- Access modifiers: `public`, `protected`, `private`.
- Non-access 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, 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. Sun 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 Sun'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:

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

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.

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 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 we 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 give you a compiler error.

So why would you ever mark a class `final`? After all, doesn't that violate the whole object-oriented (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.

A benefit of having nonfinal classes is this scenario: Imagine 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, then 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, if we try to compile the Tea subclass:

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

We get an error something like

```
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. So unless you have a serious safety or security issue, assume that some day 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 nonabstract, 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 nonabstract 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 nonabstract 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 class path 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

Declare Interfaces (Exam Objectives 1.1 and 1.2)

1.1 Develop code that declares classes (including abstract and all forms of nested classes), interfaces, and enums, and includes the appropriate use of package and import statements (including static imports).

1.2 Develop code that declares an interface. Develop code that implements or extends one or more interfaces. Develop code that declares an abstract class. Develop code that extends an abstract class.

Declaring an Interface

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. An interface is a contract. You could write an interface `Bounceable`, for example, that says in effect, "This is the `Bounceable` interface. Any class type that implements this interface must agree to write the code for the `bounce()` and `setBounceFactor()` methods."

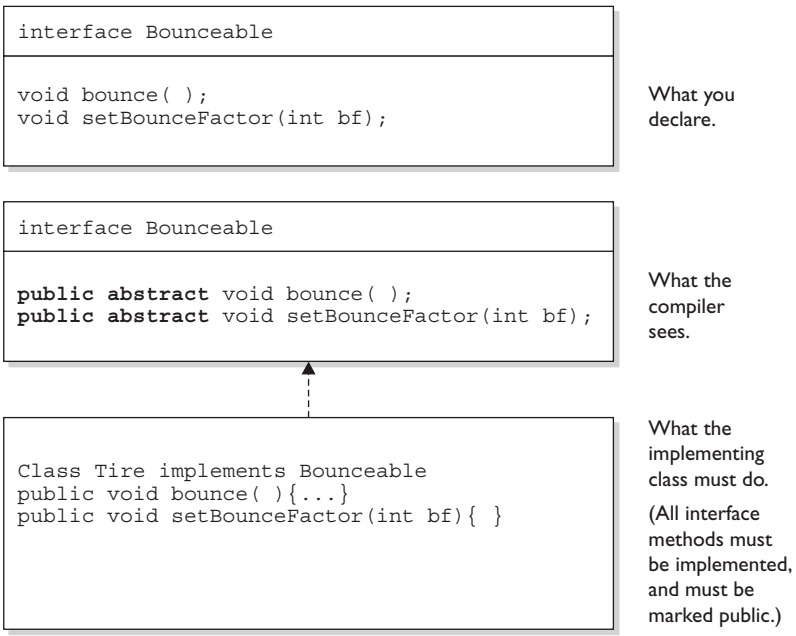
By defining an interface for Bounceable, any class that wants to be treated as a Bounceable thing can simply implement the `Bounceable` interface and provide code for the interface's two methods.

Interfaces can be implemented by any class, 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 while 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



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 while an `abstract` class can define both `abstract` and non-`abstract` methods, an interface can have 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` and `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 must not be `static`.
- Because interface methods are abstract, they cannot be marked `final`, `strictfp`, or `native`. (More on these modifiers later.)
- 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, since all interface methods are implicitly `public` and `abstract`. Given that rule, you can see that 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 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
static void bounce();   // interfaces define instance methods
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 have to actually 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.

exam Watch

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.***

CERTIFICATION OBJECTIVE

Declare Class Members (Objectives 1.3 and 1.4)

1.3 Develop code that declares, initializes, and uses primitives, arrays, enums, and objects as static, instance, and local variables. Also, use legal identifiers for variable names.

1.4 Develop code that declares both static and non-static methods, and—if appropriate—use method names that adhere to the JavaBeans naming standards. Also develop code that declares and uses a variable-length argument list.

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 will be mentioned later.

It's crucial that you know access control inside and out for the exam. There will be quite a few questions with access control playing a role. Some questions test

several concepts of access control at the same time, so not knowing one small part of access control could 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 is 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.

```
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.

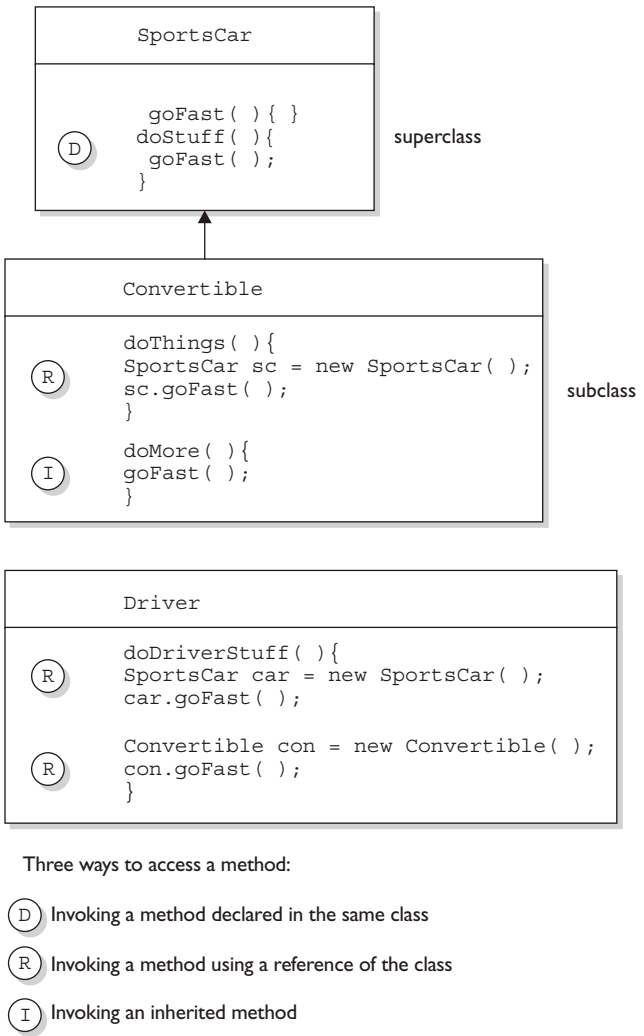
Much of access control (both types) centers on whether the two classes involved are in the same or different packages. Don't forget, though, if class A *itself* can't be accessed by class B, then no members within class A can be accessed by class B.

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 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).

FIGURE 1-2 Comparison of inheritance vs. dot operator for member access.



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.

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

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

```
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, it's 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 anything else you want to do with it.

```
package cert;
public class Roo {
   private String doRooThings() {
    // imagine the fun code that goes here, but no other class
    // will know
    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
```

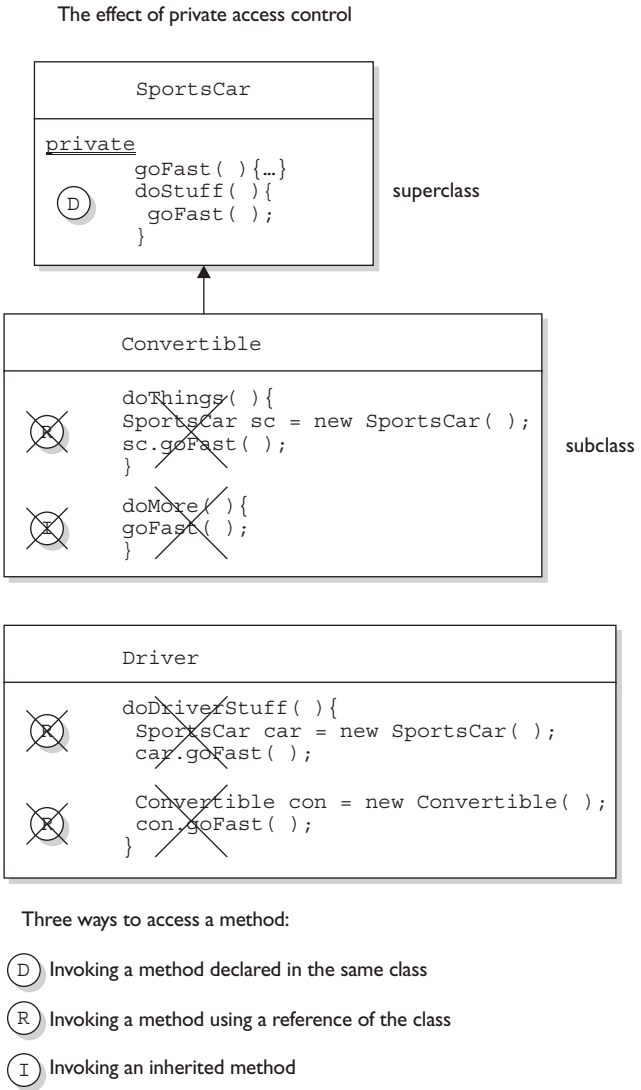
on the Job

Although you're allowed to mark instance variables as `public`***, in practice it's nearly always best to keep all variables*** `private` ***or*** `protected`***. If variables need to be changed, set, or read, programmers should use public accessor methods, so that code in any other class has to ask to get or set a variable (by going through a method), rather than access it directly. JavaBean-compliant accessor methods take the form get<propertyName> or, for booleans, is<propertyName> and set<propertyName>, and provide a place to check and/or validate before returning or modifying a value.***

Without this protection, the weight variable of a Cat object, for example, could be set to a negative number if the offending code goes straight to the public variable as in `someCat.weight = -20`***. But an accessor method,*** `setWeight(int wt)`***, could check for an inappropriate number. (OK, wild speculation, but we're guessing a negative weight might be inappropriate for a cat. Or not.) Chapter 2 will discuss this data protection (encapsulation) in more detail.***

Can a `private` method be overridden by a subclass? That's an interesting question, but the answer is technically no. Since 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 section as well as in Chapter 2, 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



Protected and Default Members

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 we'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 that 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' 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? So are we. Hang in there and it will all become clear with the next batch of code examples. (And don't worry; we're not actually confused. We're just trying to make you feel better if you are. You know, like it's OK for you to feel as though nothing makes sense, and that it isn't your fault. Or is it? <insert evil laugh>)

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' 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.Par-
ent
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' 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.

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

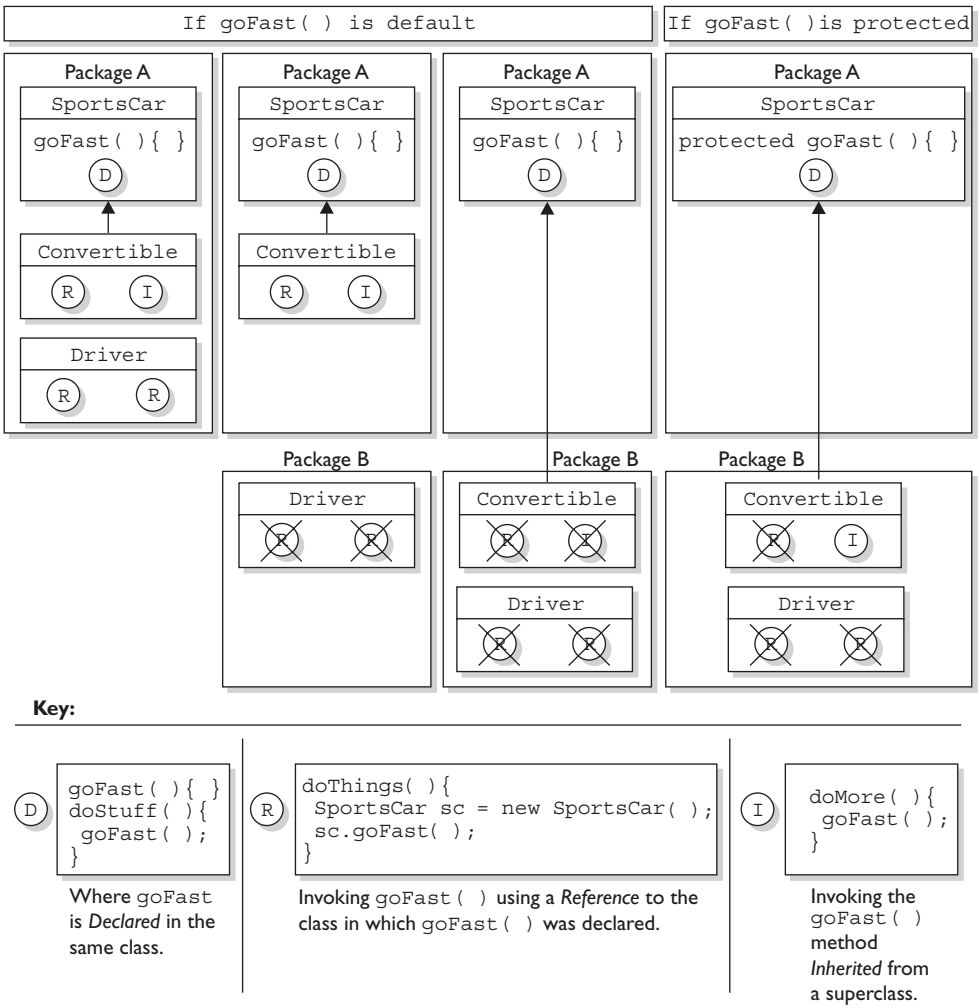
Let's start with the default behavior of a member in a superclass. We'll modify the Parent's member `x` to make it default.

```
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 that 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.

FIGURE 1-4

Effects of `protected` access



When we compile the child file, we get an error something like this:

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

The compiler gives the same 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 1-2 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 1-2 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	Yes, *through inheritance*	No	No
From any non-subclass 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`, `synchronized`, `native`, `strictfp`, and then a long look at the Big One—`static`.

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 recordNumber) {}
```

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` argument must keep the same value that the parameter had when it was passed into the method.

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 for where the implementation code goes, 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 provides actual implementation code.

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:

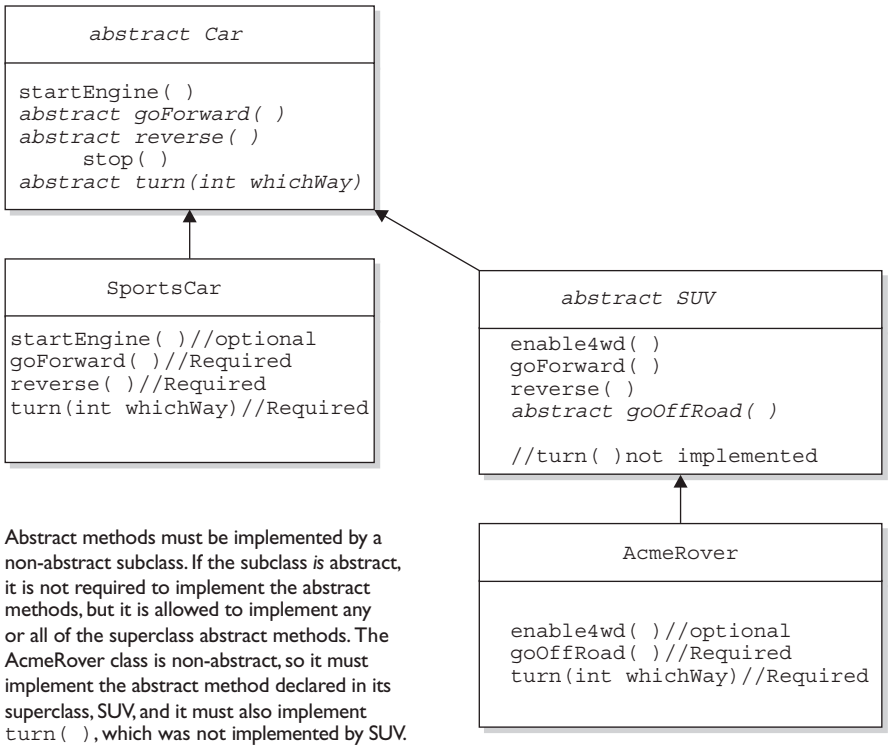
The first concrete subclass of an abstract class must implement *all* abstract methods of the superclass.

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, since 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' 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' `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 the `abstract` modifier can never be combined with the `static` modifier. We'll cover `static` methods 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. We'll discuss this nearly to death in Chapter 9, 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).

Native Methods

The `native` modifier indicates that a method is implemented in platform-dependent code, often in C. You don't need to know how to use `native` methods for the exam, other than knowing that `native` is a modifier (thus a reserved keyword) and that native can be applied only to *methods*—not classes, not variables, just methods. Note that a `native` method's body must be a semicolon (;) (like abstract methods), indicating that the implementation is omitted.

Strictfp Methods

We looked earlier at using `strictfp` as a class modifier, but even if you don't declare a class as `strictfp`, you can still declare an individual method as `strictfp`. Remember, `strictfp` forces floating points (and any floating-point operations) to adhere to the IEEE 754 standard. With `strictfp`, you can predict how your floating points will behave regardless of the underlying platform the JVM is running on. The downside is that if the underlying platform is capable of supporting greater precision, a `strictfp` method won't be able to take advantage of it.

You'll want to study the IEEE 754 if you need something to help you fall asleep. For the exam, however, you don't need to know anything about `strictfp` other than what it's used for, that it can modify a class or method declaration, and that a variable can never be declared `strictfp`.

Methods with Variable Argument Lists (var-args)

As of 5.0, 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 parameter." 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:

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

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

```
void doStuff(String s, int a) { } // we're expecting two
                                  // parameters: String and int
```

We'll cover using var-arg methods more in the next few chapters, for now 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, 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-args 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 Animals
```

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), 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. In Chapter 3 we will discuss the various ways in which they can be initialized, but for now we'll leave you with a few examples of primitive variable declarations:

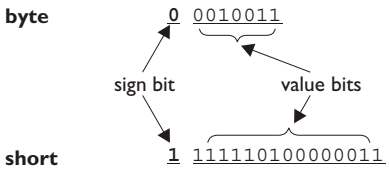
```
byte b;
boolean myBooleanPrimitive;
int x, y, z;                  // declare three int primitives
```

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`, `long`, and that `doubles` are bigger than `floats`.

You will also need to know that the number types (both integer and floating-point 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 1-3 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 zero is stored as a positive binary number. We use the formula $-2^{(bits-1)}$ to calculate the negative range, and we use $2^{(bits-1)}-1$ 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 1-3 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

The range for floating-point numbers is complicated to determine, 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).

For `boolean` types there is not a range; 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. You'll learn in Chapter 3 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. In Chapter 3 we will discuss the various ways in which they can be initialized, but for now we'll leave you with a few examples of reference variable declarations:

```
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 only initialized 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
}
```

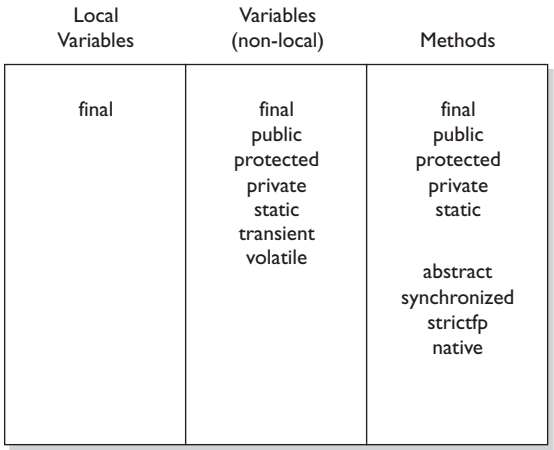
The preceding Employee class says that each employee instance will know its own name, title, and manager. In other words, each instance can have its own unique values for those three fields. If you see the term "field," "instance variable," "property," or "attribute," they mean virtually the same thing. (There actually are subtle but occasionally important distinctions between the terms, but those distinctions aren't used on the exam.)

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.

We've already covered the effects of applying access control to instance variables (it works the same way as it does for member methods). A little later in this chapter we'll look at what it means to apply the `final` or `transient` modifier to an instance variable. First, though, we'll take a quick look at the difference between instance and local variables. Figure 1-7 compares the way in which modifiers can be applied to methods vs. variables.

FIGURE 1-7 Comparison of modifiers on variables vs. methods



Local (Automatic/Stack/Method) Variables

Local variables are variables 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. (We'll talk more about the stack and the heap in Chapter 3). 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 a programmer use that expression, you'll know that he's just too lazy to phrase it in a technically precise way. You can tell him we said that—unless he knows 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 we saw earlier, local variables can be marked `final`. And as you'll learn in Chapter 3 (but here's a preview), 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 reinitialize 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 the 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)

For this objective, you only need to know how to declare an array, we'll cover constructing and initializing arrays in Chapter 3.

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's a Collection type 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? Chapter 7 covers them 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, and 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.

exam Watch

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.

In Chapter 3, we'll spend a lot of time discussing arrays, how to initialize and use them, and how to deal with multi-dimensional arrays…stay tuned!

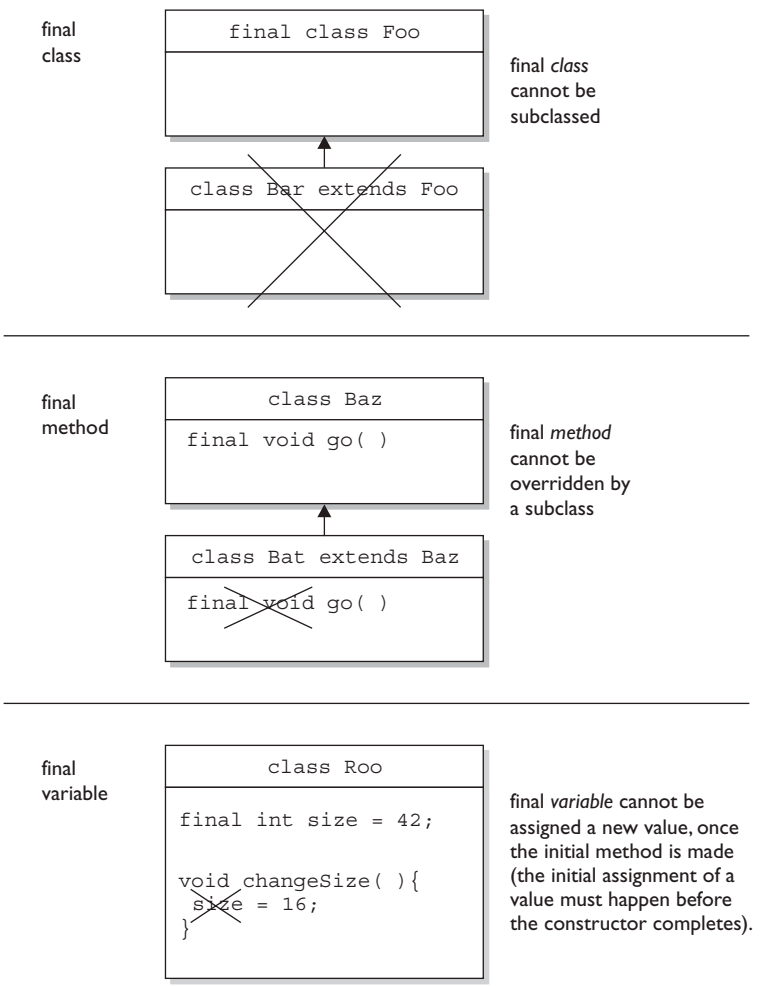
Final Variables

Declaring a variable with the `final` keyword makes it impossible to reinitialize 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't ever 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'll explain this in more detail in Chapter 3.

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



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 the coolest features of Java; 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. Serialization has been added to the exam as of Java 5, and we'll cover it in great detail in Chapter 6.

Volatile Variables

The `volatile` modifier tells the JVM that a thread accessing the variable must always reconcile its own private copy of the variable with the master copy in memory. Say what? Don't worry about it. For the exam, all you need to know about `volatile` is that, as with `transient`, it can be applied only to instance variables. Make no mistake, the idea of multiple threads accessing an instance variable is scary stuff, and very important for any Java programmer to understand. But as you'll see in Chapter 9, you'll probably use synchronization, rather than the `volatile` modifier, to make your data thread-safe.

The `volatile` modifier may also be applied to project managers :)

Static Variables and Methods

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 (more on this in Chapter 8).
- 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
- Method local inner classes (we'll explore this in Chapter 8)
- Inner class methods and instance variables
- Local variables

Declaring Enums

As of 5.0, Java lets you restrict a variable to having one of only a few pre-defined 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, in your coffee shop application you might want to restrict your size 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 Sun code convention that constants are named in caps, it's a good idea.

The basic components of an `enum` are its constants (i.e., 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!

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. (Remember, 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 types are actually instances 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);
```

```
public 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, are instances 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. (Don't worry about that in this chapter.)

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 some kind of 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 that can each have their 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.

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 CoffeeType 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 (which we talk about in Chapter 8). 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 saying, "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.

There will be many questions dealing with keywords indirectly, so be sure you can identify which are keywords and which aren't.

Although naming conventions like the use of camelCase won't be on the exam directly, you will need to understand the basics of JavaBeans naming, which uses camelCase.

You need to understand the rules associated with creating legal identifiers, and the rules associated with source code declarations, including the use of `package` and `import` statements.

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. 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.

Remember that as of Java 5, 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.

Make sure you're familiar with the relative sizes of the numeric primitives. Remember that while the values of non-final 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 a feature new to Java 5, enums. An `enum` is a much safer and more flexible way to implement constants than was possible in earlier versions of Java. 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, or *top-level* classes. We'll devote all of Chapter 8 to inner classes.

Identifiers (Objective 1.3)

- ❑ Identifiers can begin with a letter, an underscore, or a currency character.
- ❑ After the first character, identifiers can also include digits.
- ❑ Identifiers can be of any length.
- ❑ JavaBeans methods must be named using camelCase, and depending on the method's purpose, must start with `set`, `get`, `is`, `add`, or `remove`.

Declaration Rules (Objective 1.1)

- ❑ A source code file can have only one `public` class.
- ❑ If the source file contains a `public` class, the filename must match the `public` class name.
- ❑ A file can have only one `package` statement, but multiple `imports`.
- ❑ The `package` statement (if any) must be the first (non-comment) line in a source file.
- ❑ The `import` statements (if any) must come after the `package` and before the class declaration.
- ❑ If there is no `package` statement, `import` statements must be the first (non-comment) statements in the source file.
- ❑ `package` and `import` statements apply to all classes in the file.
- ❑ A file can have more than one nonpublic class.
- ❑ Files with no `public` classes have no naming restrictions.

Class Access Modifiers (Objective 1.1)

- ❑ 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) (Objective 1.2)

- 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 nonabstract methods.
- The first concrete class to extend an `abstract` class must implement all of its `abstract` methods.

Interface Implementation (Objective 1.2)

- 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.
- An interface is like a 100-percent `abstract` class, and is implicitly abstract whether you type the `abstract` modifier in the declaration or not.
- An interface can have only abstract methods, no concrete methods allowed.
- Interface methods are by default `public` and `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.
- 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 (Objectives 1.3 and 1.4)

- Methods and instance (nonlocal) variables are known as "members."
- Members can use all four access levels: public, protected, default, 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 plus 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' own subclasses.

Local Variables (Objective 1.3)

- 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 (Objective 1.3)

- `final` methods cannot be overridden in a subclass.
- `abstract` methods are declared, with a signature, a return type, and an optional throws clause, but are not implemented.
- `abstract` methods end in a semicolon—no curly braces.
- Three ways to spot a non-abstract method:
  - The method is not marked `abstract`.
  - The method has curly braces.
  - The method has code between the curly braces.
- The first nonabstract (concrete) class to extend an `abstract` class must implement all of the `abstract` class' `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`.
- The `native` modifier applies only to methods.
- The `strictfp` modifier applies only to classes and methods.

Methods with var-args (Objective 1.4)

- As of Java 5, 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.

Variable Declarations (Objective 1.3)

- Instance variables can
  - Have any access control
  - Be marked `final` or `transient`
- Instance variables can't be `abstract`, `synchronized`, `native`, or `strictfp`.
- 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 reinitialized once assigned a value.
  - `final` reference variables cannot refer to a different object once the object has been assigned to the `final` variable.
  - `final` reference 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 is immutable.
- The `transient` modifier applies only to instance variables.
- The `volatile` modifier applies only to instance variables.

Array Declarations (Objective 1.3)

- 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 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 (Objective 1.4)

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

Enums (Objective 1.3)

- 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 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.

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.

If you have a rough time with these at first, don't beat yourself up. Be positive. Repeat nice affirmations to yourself like, "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 is 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. Which method names follow the JavaBeans standard? (Choose all that apply.)

A. `addSize`

B. `getCust`

C. `deleteRep`

D. `isColorado`

E. `putDimensions`

3. Given:

```
1. class Voop {
2.   public static void main(String [] args) {
3.     doStuff(1);
4.     doStuff(1,2);
5.   }
6.   // insert code here
7. }
```

Which, inserted independently at line 6, will compile? (Choose all that apply.)

A. `static void doStuff(int... doArgs) { }`

B. `static void doStuff(int[] doArgs) { }`

C. `static void doStuff(int doArgs...) { }`

D. `static void doStuff(int... doArgs, int y) { }`

E. `static void doStuff(int x, int... doArgs) { }`

4. 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

5. 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

6. 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

7. Given:

```
4. class Announce {
5.   public static void main(String[] args) {
6.     for(int __x = 0; __x < 3; __x++) ;
7.     int #lb = 7;
```

```
 8.     long [] x [5];
 9.     Boolean []ba[];
10.     enum Traffic { RED, YELLOW, GREEN };
11.   }
12. }
```

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

A. Compilation succeeds

B. Compilation fails with an error on line 6

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

8. 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

9. Given:

```
 4. public class Frodo extends Hobbit {
 5.   public static void main(String[] args) {
 6.     Short 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

SELF TEST ANSWERS

1. Which is 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

Answer:

☑ **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. (Objective 1.2)

2. Which method names follow the JavaBeans standard? (Choose all that apply.)

A. `addSize`

B. `getCust`

C. `deleteRep`

D. `isColorado`

E. `putDimensions`

Answer:

☑ **B** and **D** use the valid prefixes '`get`' and '`is`'.

☒ **A** is incorrect because '`add`' can be used only with Listener methods. **C** and **E** are incorrect because '`delete`' and '`put`' are not standard JavaBeans name prefixes. (Objective 1.4)

3. Given:

```
1. class Voop {
2.   public static void main(String[] args) {
3.     doStuff(1);
4.     doStuff(1,2);
5.   }
6.   // insert code here
7. }
```

Which, inserted independently at line 6, will compile? (Choose all that apply.)

A. `static void doStuff(int... doArgs) { }`
B. `static void doStuff(int[] doArgs) { }`
C. `static void doStuff(int doArgs...) { }`
D. `static void doStuff(int... doArgs, int y) { }`
E. `static void doStuff(int x, int... doArgs) { }`

Answer:

☑ **A** and **E** use valid var-args syntax.

☒ **B** and **C** are invalid var-arg syntax, and **D** is invalid because the var-arg must be the last of a method's arguments. (Objective 1.4)

4. 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

Answer:

☑ **A** is correct; enums can have constructors and variables.

☒ **B, C, D, E,** and **F** are incorrect; these lines all use correct syntax. (Objective 1.3)

5. 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.print(" " + 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

Answer:

☑ **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. (Objective 1.1)

6. 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

Answer:

☑ **A** is correct; all of these are legal declarations.

☒ **B, C, D, E,** and **F** are incorrect based on the above information. (Objective 1.2)

7. Given:

```
4. class Announce {
5.   public static void main(String[] args) {
6.     for(int __x = 0; __x < 3; __x++) ;
7.     int #lb = 7;
8.     long [] x [5];
9.     Boolean []ba[];
10.    enum Traffic { RED, YELLOW, GREEN };
11.  }
12. }
```

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

A. Compilation succeeds
B. Compilation fails with an error on line 6
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

Answer:

☑ **C, D,** and **F** are correct. Variable names cannot begin with a #, an array declaration can't include a size without an instantiation, and enums can't be declared within a method.

☒ **A, B,** and **E** are incorrect based on the above information. (Objective 1.3)

8. 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

Answer:

☑ **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. (Objective 1.3)

9. Given:

```
4. public class Frodo extends Hobbit {
5.   public static void main(String[] args) {
6.     Short 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

Answer:

☑ **D** is correct. The `Short myGold` is autoboxed correctly, but the `countGold()` method cannot be invoked from a static context.

☒ **A, B, C,** and **E** are incorrect based on the above information. (Objective 1.4)

2

Object Orientation

CERTIFICATION OBJECTIVES

- Declare Interfaces
- Declare, Initialize, and Use Class Members
- Use Overloading and Overriding
- Develop Constructors
- Describe Encapsulation, Coupling, and Cohesion
- Use Polymorphism
- Relate Modifiers and Inheritance
- Use Superclass Constructors and Overloaded Constructors
- Use IS-A and HAS-A Relationships

✓ Two-Minute Drill

Q&A Self Test

Being an SCJP 6 means you must be at one with the object-oriented aspects of Java. You must dream of inheritance hierarchies, the power of polymorphism must flow through you, cohesion and loose coupling must become second nature to you, and composition must be your bread and butter. This chapter will prepare you for all of the object-oriented objectives and questions you'll encounter on the exam. We have heard of many experienced Java programmers who haven't really become fluent with the object-oriented tools that Java provides, so we'll start at the beginning.

CERTIFICATION OBJECTIVE

Encapsulation (Exam Objective 5.1)

5.1 Develop code that implements tight encapsulation, loose coupling, and high cohesion in classes, and describe the benefits.

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 just 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 Object Orientation (OO): 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 if 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 ExploitBad00 {
    public static void main (String [] args) {
        Bad00 b = new Bad00();
        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 protect 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.

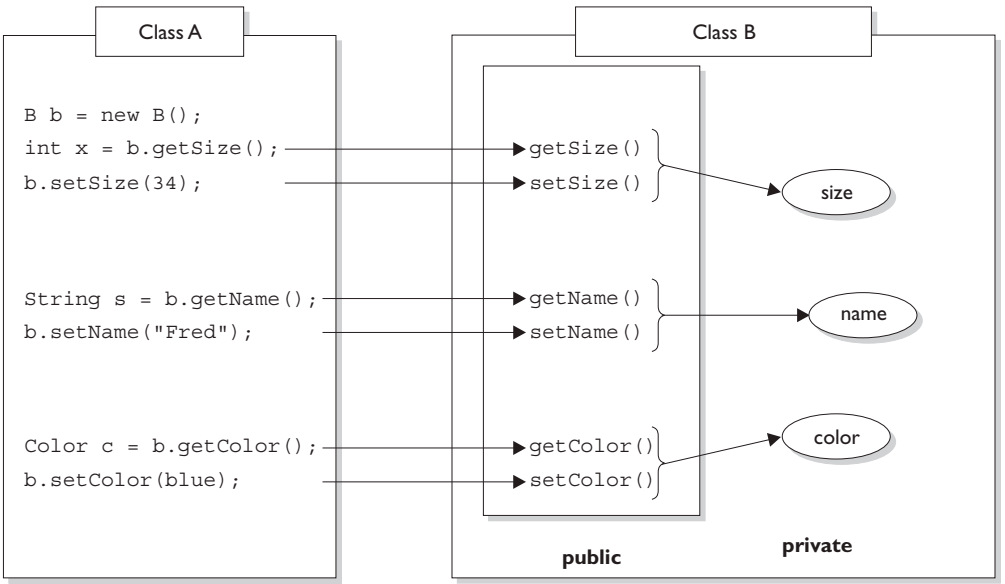
If you want maintainability, flexibility, and extensibility (and of course, you do), your design must include encapsulation. How do you do that?

- Keep instance variables protected (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.
- For the methods, use the JavaBeans 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 {
   // protect 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 additional 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.

exam Watch

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, Is-A, Has-A (Exam Objective 5.5)

5.5 Develop code that implements "is-a" and/or "has-a" relationships.

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. In order to explore this topic we're going to use the `instanceof` operator, which we'll discuss in more detail in Chapter 4. For now, just remember that `instanceof` returns `true` if the reference variable being tested is of the type being compared to. 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 the 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, 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, a `clone` method, `notify`, `wait`, 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 very 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.)

For the exam you'll need to know that you can create inheritance relationships in Java by *extending* a class. 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 `PlayingPiece` class inherits the generic `display()` method from the less-specialized class `GameShape`, and also adds its own method, `movePiece()`. Code reuse through inheritance means that methods with generic functionality (like `display()`)—that 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 generic superclass. You don't want to have to rewrite the `display()` 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 `display()` 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 `display()` 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`, that 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

For the exam you need to be able to look at code and determine whether the code demonstrates an IS-A or HAS-A relationship. The rules are simple, so this should be one of the few areas where answering the questions correctly is almost a no-brainer.

IS-A

In OO, the concept of IS-A is based on class 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 the superclass of Car.
Car is the subclass of Vehicle.
Car is the superclass of Subaru.
Subaru is the 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' arrow points toward the class from which it extends.

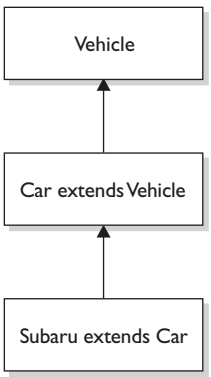


FIGURE 2-2

Inheritance tree for Vehicle, Car, Subaru

HAS-A

HAS-A relationships are based on usage, 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,

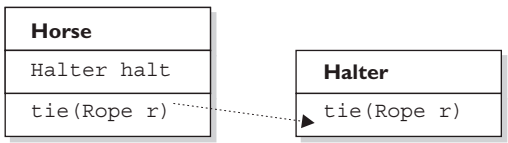
A Horse IS-A Animal. A Horse HAS-A Halter.
The code might look like this:

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

In the preceding code, the Horse class has an instance variable of type Halter, so you can say that "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.

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 object-oriented 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. Having seen the ramifications of poor designs, I can assure you that this estimate is not far-fetched.

Even the best object-oriented designers 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 has to be rewritten can sometimes cause programming teams to start over from scratch.

The software industry has evolved to aid the designer. Visual object modeling languages, like the Unified Modeling Language (UML), allow designers to design and easily modify classes without having to write code first, because object-oriented components are represented graphically. This allows the designer to create a map of the class relationships and helps them recognize errors before coding begins. Another innovation in object-oriented design is design patterns. Designers noticed that many object-oriented designs apply 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. Object-oriented 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 object-oriented design this thoroughly, hopefully this background information will help you better appreciate why the test writers chose to include encapsulation, and IS-A, and HAS-A relationships on the exam.

–Jonathan Meeks, Sun Certified Java Programmer

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
   }
}
```

In OO, we don't want callers to worry about which class or which 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 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 (Exam Objective 5.2)

5.2 Given a scenario, develop code that demonstrates the use of polymorphism. Further, determine when casting will be necessary and recognize compiler vs. runtime errors related to object reference casting.

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 that the only way to access an object is through a reference variable, and there are a few key things to remember 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 game board. But not *all* shapes can be animatable, 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, while the TilePiece would extend only GameShape. But no, this won't work! Java supports only single 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, since 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.

on the Job

Some languages (like 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 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 known as 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.

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, including it's more error-prone, it makes the `GameShape` class less cohesive (more on cohesion in a minute), 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 successfully run the `animate()` method.

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 that 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 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.

Since this definition depends on a clear understanding of overriding, and the distinction between static methods and instance methods, we'll cover those next.

CERTIFICATION OBJECTIVE

Overriding / Overloading (Exam Objectives 1.5 and 5.4)

1.5 Given a code example, determine if a method is correctly overriding or overloading another method, and identify legal return values (including covariant returns), for the method.

5.4 Given a scenario, develop code that declares and/or invokes overridden or overloaded methods and code that declares and/or invokes superclass, overridden, or overloaded constructors.

Overridden Methods

Any time you have a class that inherits a method from a superclass, 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 subclass type. 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 superclass, you have no choice. You *must* implement the method in the subclass ***unless the subclass is also abstract.*** Abstract methods must be *implemented* by the concrete subclass, but this is a lot like saying that the concrete subclass *overrides* the abstract methods of the superclass. So you could think of abstract methods as methods you're forced to override.

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, specific way. Polymorphically, when someone has an `Animal` reference that refers not to an `Animal` instance, but to an `Animal` subclass instance, the caller should be able to invoke `eat()` on the `Animal` reference, but 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 (non-abstract), 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 refer 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 subclass 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 subclass.) Let's modify the polymorphic example we 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 superclass reference to a subclass instance possible is that the subclass is guaranteed to be able to do everything the superclass 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.

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 the overridden method's.
- 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 subclass. A subclass within the same package as the instance's superclass can override any superclass method that is not marked `private` or `final`. A subclass in a different package can override only those non-`final` methods marked `public` or `protected` (since `protected` methods are inherited by the subclass).
- The overriding method CAN throw any unchecked (runtime) exception, regardless of whether the overridden method declares the exception. (More in Chapter 5.)
- 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 subclass' 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 Superclass Version of an Overridden Method

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

Note: Using `super` to invoke an overridden method only applies to instance methods. (Remember, `static` methods can't be overridden.)

exam Watch

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 (more in Chapter 5). Let's take a 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 Legal and Illegal Method Overrides

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

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

Table 2-1 lists examples of illegal overrides of the Animal `eat()` method, given the preceding version of the Animal class.

TABLE 2-1 Examples of Illegal Overrides

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

Overloaded Methods

You're wondering what overloaded methods are doing in an OO chapter, but we've included them here since one of the things newer Java developers are most confused about are all of the subtle differences between over*loaded* and over*ridden* methods.

Over*loaded* 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 are simple:

- 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* class or in a *subclass*. In other words, if class A defines a `doStuff(int i)` method, the 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 classes can still be considered overloaded, if the subclass inherits one version of the method and then declares another overloaded version in its class definition.

exam Watch

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 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:

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

Invoking Overloaded Methods

Note that there's a lot more to this discussion on how the compiler knows which method to invoke, but the rest is covered in Chapter 3 when we look at boxing and var-args—both of which have a huge impact on overloading. (You still have to pay attention to the part covered here, though.)

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.

Deciding which of the matching methods to invoke is 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 the preceding 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 say, "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:

```
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 over*ridden* 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 over*loaded* 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 there's actually a Horse being passed.

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 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 2-2 shows which version of the three eat() methods will run depending on how they are invoked.

TABLE 2-2 Examples of 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 (Horse), not the reference type (Animal), is used to determine which eat() is called.
`Horse he = new Horse();` `he.eat("Apples");`	`Horse eating Apples` The overloaded eat(String s) method is invoked.
`Animal a2 = new Animal();` `a2.eat("treats");`	Compiler error! Compiler sees that Animal class doesn't have an eat() method that takes a String.
`Animal ah2 = new Horse();` `ah2.eat("Carrots");`	Compiler error! Compiler *still* looks only at the reference, and sees that Animal doesn't have an eat() method that takes a String. Compiler doesn't care that the actual object might be a Horse at runtime.

exam Watch

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 2-3 summarizes the difference between overloaded and overridden methods.

TABLE 2-3 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.
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	*Reference* type determines which overloaded version (based on declared argument types) is selected. Happens at *compile* time. The actual *method* that's invoked is still a virtual method invocation that happens at runtime, but the compiler will already know the *signature* of the method to be invoked. So at runtime, the argument match will already have been nailed down, just not the *class* in which the method lives.	*Object* type (in other words, *the type of the actual instance on the heap*) determines which method is selected. Happens at *runtime*.

The current objective (5.4) covers both method and constructor overloading, but we'll cover constructor overloading in the next section, 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.

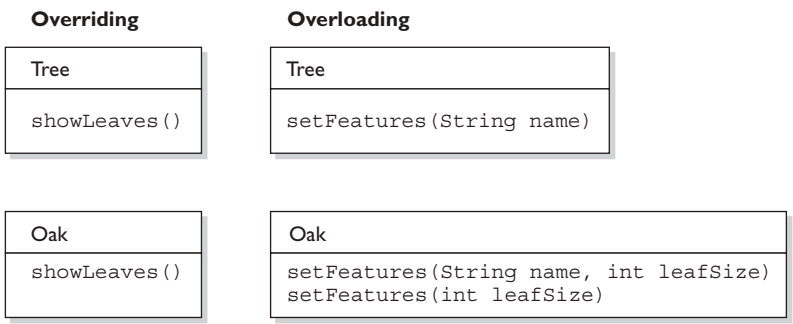


FIGURE 2-4 Overloaded and overridden methods in class relationships

CERTIFICATION OBJECTIVE

Reference Variable Casting (Objective 5.2)

5.2 Given a scenario, develop code that demonstrates the use of polymorphism. Further, determine when casting will be necessary and recognize compiler vs. runtime errors related to object reference casting.

We've seen how it's both possible and common to use generic 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 of this code is OK, except that 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! 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 that 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
```

Unlike downcasting, upcasting (casting *up* the inheritance tree to a more general type) works implicitly (i.e., 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 *down*casting, which implies that later on, you might want to invoke a more *specific* method. For instance:

```
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 Animal, which means that anything an Animal can do, a Dog can do. A Dog can do more, of course, but the point is—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 of itself, and all its children. The Beagle class can always override any methods it inherits from Dog, including methods that Dog implemented to fulfill its interface contract.

And just one more thing…if Beagle does declare 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 OK to make it more obvious."

So don't be fooled by code that shows a concrete class that declares that 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 superclasses of this implementing class have declared. If any superclass in its inheritance tree has already provided concrete (i.e., non-abstract) method implementations, then, regardless of whether the superclass declares that it implements the interface, the subclass is under no obligation to re-implement (override) those methods.

exam Watch

The exam creators will tell you that they're forced to jam tons of code into little spaces "because of the exam engine." While 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 of those parentheses, otherwise it thinks it's been handed an incomplete statement.

CERTIFICATION OBJECTIVE

Implementing an Interface (Exam Objective 1.2)

1.2 Develop code that declares an interface…

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 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 that 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 method in the interface (with a few exceptions 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) { }
}
```

OK, 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 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. The compiler will never say to you, "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. In order to be a legal implementation class, a nonabstract implementation class must do the following:

- Provide concrete (nonabstract) implementations for all methods from the declared interface.
- Follow all the rules for legal overrides.
- 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.)

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 OK. 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 will have to provide all the 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 methods defined in the interface.

Two more rules you need to know 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 implement many interfaces. But remember that 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, but never implement anything. 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 methods of Bounceable, plus all the 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 J2EE where you'll often have to build your own interface that extends one of the J2EE 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 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:

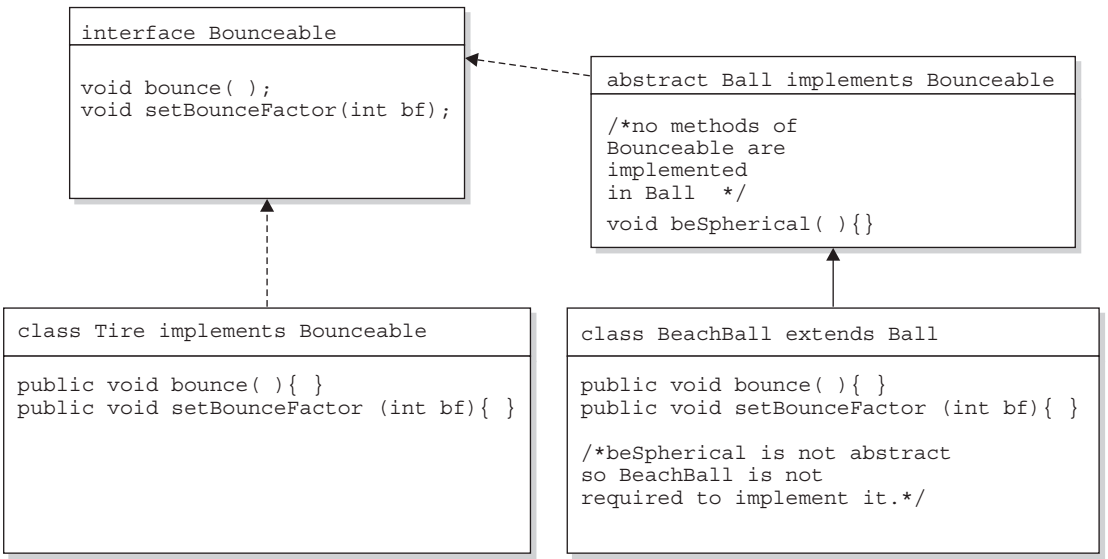
```
class Ball implements Bounceable {

   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 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 any, all, or none of the 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.

exam Watch

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! Interface can't
                                       // implement an interface
interface Zee implements Foo { }       // No! Interface can't
                                       // implement a class
interface Zoo extends Foo { }          // No! Interface can't
                                       // extend a class
interface Boo extends Fi { }           // OK. Interface can extend
                                       // an interface
class Toon extends Foo, Button { }     // No! Class can't extend
                                       // multiple classes
class Zoom implements Fi, Baz { }      // OK. class can implement
                                       // multiple interfaces
interface Vroom extends Fi, Baz { }    // OK. interface can extend
                                       // multiple interfaces
class Yow extends Foo implements Fi { } // OK. Class can do both
                                        // (extends must be 1st)
```

Burn these in, 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?)

CERTIFICATION OBJECTIVE

Legal Return Types (Exam Objective 1.5)

1.5 Given a code example, determine if a method is correctly overriding or overloading another method, and identify legal return values (including covariant returns), for the method.

This objective 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 polymorphic 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 subclass, 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 superclass 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 subclass wants to change the method implementation of an inherited method (an override), the subclass must define a method that matches the inherited version exactly. Or, as of 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 in Java 1.5
    return new Beta();
  }
}
```

As of Java 5, this code will compile. 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 something like this:

```
attempting to use incompatible return type
```

(We'll talk more about compiler flags in Chapter 10.)

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!!
}
```

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
   }
}
```

exam Watch

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, the return value is a subtype.

CERTIFICATION OBJECTIVE

Constructors and Instantiation (Exam Objectives 1.6, 5.3, and 5.4)

1.6 Given a set of classes and superclasses, develop constructors for one or more of the classes. Given a class declaration, determine if a default constructor will be created, and if so, determine the behavior of that constructor. Given a nested or nonnested class listing, write code to instantiate the class.

5.3 Explain the effect of modifiers on inheritance with respect to constructors, instance or static variables, and instance or static methods.

5.4 Given a scenario, develop code that declares and/or invokes overridden or overloaded methods and code that declares and/or invokes superclass, overridden, or overloaded constructors.

Objects are constructed. You can't 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`. OK, to be a bit more accurate, there can also be initialization blocks that run when you say `new`, but we're going to cover them (init blocks), and their static initialization counterparts, in the next chapter. 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 one, 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 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, because 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.)

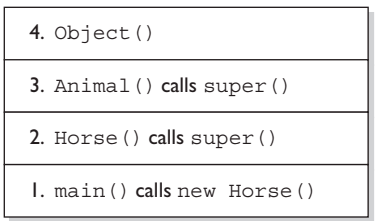
1. Horse constructor is invoked. Every constructor invokes the constructor of its superclass with an (implicit) call to super(), unless the constructor invokes an overloaded constructor of the same class (more on that in a minute).
2. Animal constructor is invoked (Animal is the superclass of Horse).
3. 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. Object instance variables are given their explicit values. By *explicit* values, we mean values that are assigned at the time the variables are declared, like "`int x = 27`", where "`27`" is the explicit value (as opposed to the default value) of the instance variable.
5. Object constructor completes.
6. Animal instance variables are given their explicit values (if any).
7. Animal constructor completes.

8. Horse instance variables are given their explicit values (if any).
9. 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 type 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 that 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 be either a no-arg call or can include arguments passed to the super constructor.
- A no-arg constructor is not necessarily the default (i.e., compiler-supplied) constructor, although the default constructor is always a no-arg constructor. The default constructor is the one the compiler provides! While 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 the class above? 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 the preceding class, because the class doesn't have any constructors defined.
OK, what about this class?

```
class Horse {
   void Horse() { }
}
```

It might look like the compiler won't create one, since there already is a constructor in the Horse class. Or is there? 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, and 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 2-4 shows what the compiler will (or won't) generate for your class.

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 that 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.

TABLE 2-4 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();`
`  }`
`}`
	Nothing, compiler doesn't need to insert anything.

`class Foo {`
`  void Foo() { }`
`}`
	`class Foo {`
	`  void Foo() { }`
	`  **Foo() {**`
	`  **super();**`
	`  }`
	`}`
	`(void Foo()` *is a method, not a constructor.*`)`

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 the stunningly lucid:

```
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!
   }           // Invokes 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, you've already seen that they can be overloaded, and typically are.

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 alternate 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",
                                    "Rover", "Spike",
                                    "Gigi"}[x];
14.     return name;
15.   }
16.
17.   public static void main (String [] args) {
18.     Animal a = new Animal();
19.     System.out.println(a.name);
20.     Animal b = new Animal("Zeus");
21.     System.out.println(b.name);
22.   }
23. }
```

Running the code four times produces this output:

```
% 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.

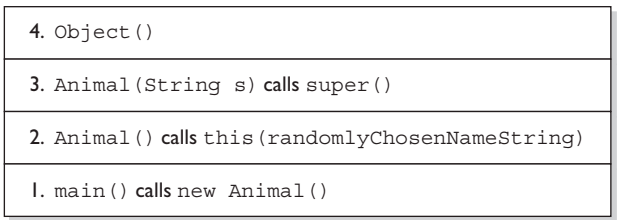


FIGURE 2-7

Overloaded constructors on the call stack

- **Line 2** Declare a String instance variable name.
- **Lines 3–5** Constructor that takes a String, and assigns it to 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 you'll 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 name, `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 since 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 anything).
  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. We could have made it much easier to read if we'd just written

```
String[] nameList = {"Fluffy", "Fido", "Rover", "Spike",
                     "Gigi"};
String name = nameList[x];
```

But where's the fun in that? Throwing in unusual syntax (especially for code wholly unrelated to the real question) is in the spirit of the exam. Don't be

startled! (OK, 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. OK, 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();
   }
}
```

Your compiler may not actually catch the problem (it varies depending on your compiler, but most won't catch the problem). It assumes 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 out of 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

```
% 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, so we're holding the rest of the discussion on instantiating inner classes until Chapter 8.

CERTIFICATION OBJECTIVE

Statics (Exam Objective 1.3)

1.3 Develop code that declares, initializes, and uses primitives, arrays, enums, and objects as static, instance, and local variables. Also, use legal identifiers for variable names.

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 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 class 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 class, rather than to any particular instance. In fact, you can use a `static` method or variable without having any instances of that class at all. You need only have the class 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.) 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, rather just 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.

exam 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.***

exam Watch

(continued) ***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, 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, then 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 class name, as opposed to using it on a reference to an instance, as follows:

```
class Frog {
    static int frogCount = 0;  // Declare and initialize
                               // static variable
    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.print("frogCount:"+Frog.frogCount); //Access
                                                    // static variable
    }
}
```

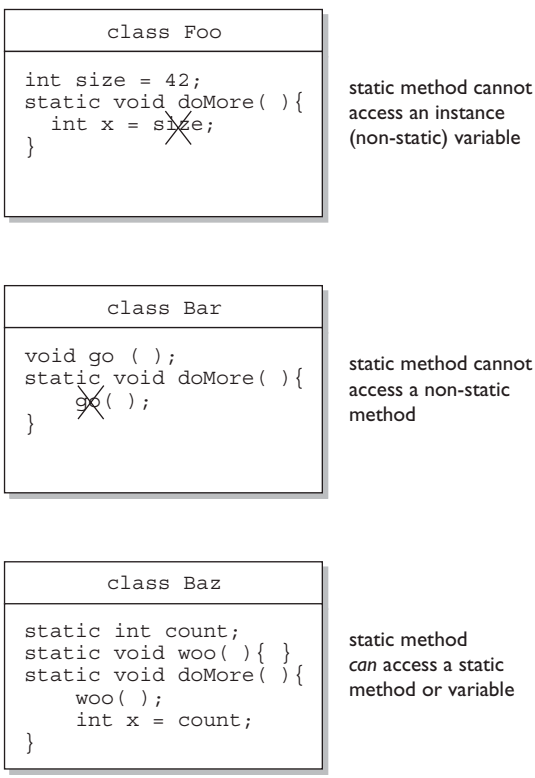
But just to make it really confusing, the Java language also allows you to use an object reference variable to access a static member:

```
Frog f = new Frog();
int frogs = f.frogCount; // Access static variable
                         // FrogCount using f
```

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. 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



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 take a 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,
                                   // not an override
    System.out.print("d ");
  }
  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
  }
}
```

Running this code produces the output:

```
a a a
```

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 that we didn't use the Java 1.5 *enhanced* `for` *loop* here (covered in Chapter 5), even though we could have. Expect to see a mix of both Java 1.4 and Java 5 coding styles and practices on the exam.

CERTIFICATION OBJECTIVE

Coupling and Cohesion (Exam Objective 5.1)

5.1 Develop code that implements tight encapsulation, loose coupling, and high cohesion in classes, and describe the benefits.

We're going to admit it up front. The Sun exam's definitions for cohesion and coupling are somewhat subjective, so what we discuss in this chapter is from the perspective of the exam, and by no means The One True Word on these two OO design principles. It may not be exactly the way that you've learned it, but it's what you need to understand to answer the questions. You'll have very few questions about coupling and cohesion on the real exam.

These two topics, coupling and cohesion, have to do with the quality of an OO design. In general, good OO design calls for *loose coupling* and shuns tight coupling, and good OO design calls for *high cohesion*, and shuns low cohesion. As with most OO design discussions, the goals for an application are

- Ease of creation
- Ease of maintenance
- Ease of enhancement

Coupling

Let's start by making an attempt at a definition of coupling. Coupling is the degree to which one class knows about another class. If the only knowledge that class A has about class B, is what class B has exposed through its interface, then class A and class B are said to be loosely coupled…that's a good thing. If, on the other hand, class A relies on parts of class B that are not part of class B's interface, then the coupling between the classes is tighter…*not* a good thing. In other words, if A knows more than it should about the way in which B was implemented, then A and B are tightly coupled.

Using this second scenario, imagine what happens when class B is enhanced. It's quite possible that the developer enhancing class B has no knowledge of class A, why would she? Class B's developer ought to feel that any enhancements that don't break the class's interface should be safe, so she might change some noninterface part of the class, which then causes class A to break.

At the far end of the coupling spectrum is the horrible situation in which class A knows non-API stuff about class B, and class B knows non-API stuff about class A… this is REALLY BAD. If either class is ever changed, there's a chance that the other class will break. Let's look at an obvious example of tight coupling, which has been enabled by poor encapsulation:

```
class DoTaxes {
  float rate;
```

```
  float doColorado() {
    SalesTaxRates str = new SalesTaxRates();
    rate = str.salesRate;     // ouch
                              // this should be a method call:
                               // rate = str.getSalesRate("CO");
    // do stuff with rate
  }
}

class SalesTaxRates {
  public float salesRate;            // should be private
  public float adjustedSalesRate;    // should be private

  public float getSalesRate(String region) {
    salesRate = new DoTaxes().doColorado();    // ouch again!
    // do region-based calculations
    return adjustedSalesRate;
  }
}
```

All nontrivial OO applications are a mix of many classes and interfaces working together. Ideally, all interactions between objects in an OO system should use the APIs, in other words, the contracts, of the objects' respective classes. Theoretically, if all of the classes in an application have well-designed APIs, then it should be possible for all interclass interactions to use those APIs exclusively. As we discussed earlier in this chapter, an aspect of good class and API design is that classes should be well encapsulated.

The bottom line is that coupling is a somewhat subjective concept. Because of this, the exam will test you on really obvious examples of tight coupling; you won't be asked to make subtle judgment calls.

Cohesion

While coupling has to do with how classes interact with each other, cohesion is all about how a single class is designed. The term *cohesion* is used to indicate the degree to which a class has a single, well-focused purpose. Keep in mind that cohesion is a subjective concept. The more focused a class is, the higher its cohesiveness—a good thing. The key benefit of high cohesion is that such classes are typically much easier to maintain (and less frequently changed) than classes with low cohesion. Another benefit of high cohesion is that classes with a well-focused purpose tend to be more reusable than other classes. Let's take a look at a pseudo-code example:

```
class BudgetReport {
  void connectToRDBMS(){ }
  void generateBudgetReport() { }
  void saveToFile() { }
  void print() { }
}
```

Now imagine your manager comes along and says, "Hey you know that accounting application we're working on? The clients just decided that they're also going to want to generate a revenue projection report, oh and they want to do some inventory reporting also. They do like our reporting features however, so make sure that all of these reports will let them choose a database, choose a printer, and save generated reports to data files..." Ouch!

Rather than putting all the printing code into one report class, we probably would have been better off with the following design right from the start:

```
class BudgetReport {
  Options getReportingOptions() { }
  void generateBudgetReport(Options o) { }
}

class ConnectToRDBMS {
  DBconnection getRDBMS() { }
}

class PrintStuff {
  PrintOptions getPrintOptions() { }
}

class FileSaver {
  SaveOptions getFileSaveOptions() { }
}
```

This design is much more cohesive. Instead of one class that does everything, we've broken the system into four main classes, each with a very specific, or *cohesive*, role. Because we've built these specialized, reusable classes, it'll be much easier to write a new report, since we've already got the database connection class, the printing class, and the file saver class, and that means they can be reused by other classes that might want to print a report.

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 subclass inherits a method from a superclass, and then reimplements the method to add more specialized behavior. We learned that, at runtime, the JVM will invoke the subclass version on an instance of a subclass, and the superclass version on an instance of the superclass. 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, as of Java 5, they can return a subtype of the declared return type of the superclass 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 learned the mechanics of casting (mostly downcasting), reference variables, when it's necessary, and how to use the `instanceof` operator.

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. 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 while 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 there is even a single constructor 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, since constructors do not have return types.

We saw how all of 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).

We looked at `static` methods and variables. Static members are tied to the class, 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 class name with the dot operator to access `static` members.

We discussed the OO concepts of coupling and cohesion. Loose coupling is the desirable state of two or more classes that interact with each other only through their respective API's. Tight coupling is the undesirable state of two or more classes that know inside details about another class, details not revealed in the class's API. High cohesion is the desirable state of a single class whose purpose and responsibilities are limited and well-focused.

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 (Objective 5.1)

- ❑ 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`.
- ❑ 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.

Inheritance (Objective 5.5)

- ❑ Inheritance allows a class to be a subclass of a superclass, and thereby inherit `public` and `protected` variables and methods of the superclass.
- ❑ 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 (Objective 5.2)

- ❑ 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 (Objectives 1.5 and 5.4)

- Methods can be overridden or overloaded; constructors can be overloaded but not overridden.
- Abstract methods must be overridden by the first concrete (non-abstract) subclass.
- With respect to the method it overrides, the overriding method
  - Must have the same argument list.
  - Must have the same return type, except that as of Java 5, the return type can be a subclass—this is 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.
- 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 superclass can be overloaded in a subclass.
- 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.

Reference Variable Casting (Objective 5.2)

- ❑ 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 (Objective 1.2)

- ❑ When you implement an interface, you are fulfilling its contract.
- ❑ You implement an interface by properly and concretely overriding all of the methods defined by the interface.
- ❑ A single class can implement many interfaces.

Return Types (Objective 1.5)

- ❑ 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 (Objectives 1.6 and 5.4)

- ❑ A constructor is always invoked when a new object is created.

- Each superclass in an object's inheritance tree will have a constructor called.
- 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 to either `this()` (an overloaded constructor) or `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()`).
- Issues with calls to `this()`
  - May appear only as the first statement in a constructor.
  - The argument list determines which overloaded constructor is called.

- ❑ Constructors can call 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.

Statics (Objective 1.3)

- ❑ 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();
```

- ❑ `static` methods can't be overridden, but they can be redefined.

Coupling and Cohesion (Objective 5.1)

- ❑ Coupling refers to the degree to which one class knows about or uses members of another class.
- ❑ Loose coupling is the desirable state of having classes that are well encapsulated, minimize references to each other, and limit the breadth of API usage.
- ❑ Tight coupling is the undesirable state of having classes that break the rules of loose coupling.
- ❑ Cohesion refers to the degree in which a class has a single, well-defined role or responsibility.
- ❑ High cohesion is the desirable state of a class whose members support a single, well-focused role or responsibility.
- ❑ Low cohesion is the undesirable state of a class whose members support multiple, unfocused roles or responsibilities.

SELF TEST

1. Given:

```
public abstract interface Frobnicate { 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

4. Using the **fragments** below, complete the following **code** so it compiles. Note, you may not have to fill 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. Which statement(s) are true? (Choose all that apply.)
   A. Cohesion is the OO principle most closely associated with hiding implementation details
   B. Cohesion is the OO principle most closely associated with making sure that classes know about other classes only through their APIs
   C. Cohesion is the OO principle most closely associated with making sure that a class is designed with a single, well-focused purpose
   D. Cohesion is the OO principle most closely associated with allowing a single object to be seen as having many types

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, 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:

   1. ClassA has a ClassD
   2. Methods in ClassA use public methods in ClassB
   3. Methods in ClassC use public methods in ClassA
   4. Methods in ClassA use public variables in ClassB

   Which is most likely true? (Choose the most likely.)

   A. ClassD has low cohesion
   B. ClassA has weak encapsulation
   C. ClassB has weak encapsulation
   D. ClassB has strong encapsulation
   E. ClassC is tightly coupled to ClassA

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. You're designing a new online board game in which Floozels are a type of Jammers, Jammers can have Quizels, Quizels are a type of Klakker, and Floozels can have several Floozets. Which of the following fragments represent this design? (Choose all that apply.)

A.
```
import java.util.*;
interface Klakker { }
class Jammer { Set<Quizel> q; }
class Quizel implements Klakker { }
public class Floozel extends Jammer { List<Floozet> f; }
interface Floozet { }
```

B.
```
import java.util.*;
class Klakker { Set<Quizel> q; }
class Quizel extends Klakker { }
class Jammer { List<Floozel> f; }
class Floozet extends Floozel { }
public class Floozel { Set<Klakker> k; }
```

C.
```
import java.util.*;
class Floozet { }
class Quizel implements Klakker { }
class Jammer { List<Quizel> q; }
interface Klakker { }
class Floozel extends Jammer { List<Floozet> f; }
```

D.
```
import java.util.*;
interface Jammer extends Quizel { }
interface Klakker { }
interface Quizel extends Klakker { }
interface Floozel extends Jammer, Floozet { }
interface Floozet { }
```

15. Given:

```
 3. class A { }
 4. class B extends A { }
 5. public class ComingThru {
 6.   static String s = "-";
 7.   public static void main(String[] args) {
 8.     A[] aa = new A[2];
 9.     B[] ba = new B[2];
10.     sifter(aa);
11.     sifter(ba);
12.     sifter(7);
13.     System.out.println(s);
14.   }
```

```
15.   static void sifter(A[]... a2)    { s += "1"; }
16.   static void sifter(B[]... b1)    { s += "2"; }
17.   static void sifter(B[] b1)       { s += "3"; }
18.   static void sifter(Object o)     { s += "4"; }
19. }
```

What is the result?

A. `-124`

B. `-134`

C. `-424`

D. `-434`

E. `-444`

F. Compilation fails

SELF TEST ANSWERS

1. Given:

```
public abstract interface Frobnicate { 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) { }
}
```

Answer:

☑ **B** is correct, an `abstract` class need not implement any or all of an interface's methods. **E** is correct, the class implements the interface method and additionally overloads the `twiddle()` method.

☒ **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.
(Objective 5.4)

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

Answer:

☑ **E** is correct. The implied `super()` call in Bottom2's constructor cannot be satisfied because there isn't a 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. (Objective 1.6)

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

Answer:

☑ **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. (Objective 5.3)

4. Using the **fragments** below, complete the following **code** so it compiles. Note, you may not have to fill 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	
(	)	{	}
;			

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-argument 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-argument version, this constructor must be created. It will not be created by the compiler because there is another constructor already present.
(Objective 5.4)

5 Which statement(s) are true? (Choose all that apply.)

A. Cohesion is the OO principle most closely associated with hiding implementation details

B. Cohesion is the OO principle most closely associated with making sure that classes know about other classes only through their APIs

C. Cohesion is the OO principle most closely associated with making sure that a class is designed with a single, well-focused purpose

D. Cohesion is the OO principle most closely associated with allowing a single object to be seen as having many types

Answer:

☑ Answer **C** is correct.

☒ **A** refers to encapsulation, **B** refers to coupling, and **D** refers to polymorphism.
(Objective 5.1)

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.  }
11. }
```

Which, 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

Answer:

☑ **C** is correct. Before you can invoke `Y`'s `do2` method you have to cast `x2` to be of type `Y`. Statement **B** looks like a proper cast but without the second set of parentheses, the compiler thinks it's an incomplete statement.

☒ **A, B** and **D** are incorrect based on the preceding.
(Objective 5.2)

7. Given:

1. ClassA has a ClassD
2. Methods in ClassA use public methods in ClassB
3. Methods in ClassC use public methods in ClassA
4. Methods in ClassA use public variables in ClassB

Which is most likely true? (Choose the most likely.)

A. ClassD has low cohesion

B. ClassA has weak encapsulation

C. ClassB has weak encapsulation

D. ClassB has strong encapsulation

E. ClassC is tightly coupled to ClassA

Answer:

☑ **C** is correct. Generally speaking, public variables are a sign of weak encapsulation.

☒ **A, B, D,** and **E** are incorrect, because based on the information given, none of these statements can be supported.
(Objective 5.1)

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

Answer:

☑ **F** is correct. Class `Dog` doesn't have a `sniff` method.

☒ **A, B, C, D,** and **E** are incorrect based on the above information. (Objective 5.2)

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

Answer:

☑ **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. (Objective 5.2)

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

Answer:

☑ **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. (Objective 5.2)

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

Answer:

☑ **C** is correct. Watch out, `SubSubAlpha` extends `Alpha`! Since 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. (Objective 5.3)

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

Answer:

☑ **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. (Objectives 1.6, 5.4)

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

Answer:

☑ **A** is correct. Polymorphism is only for instance methods.

☒ **B, C, D, E,** and **F** are incorrect based on the above information. (Objectives 1.5, 5.4)

14. You're designing a new online board game in which Floozels are a type of Jammers, Jammers can have Quizels, Quizels are a type of Klakker, and Floozels can have several Floozets. Which of the following fragments represent this design? (Choose all that apply.)

A.
```
import java.util.*;
interface Klakker { }
class Jammer { Set<Quizel> q; }
class Quizel implements Klakker { }
public class Floozel extends Jammer { List<Floozet> f; }
interface Floozet { }
```

B.
```
import java.util.*;
class Klakker { Set<Quizel> q; }
class Quizel extends Klakker { }
class Jammer { List<Floozel> f; }
class Floozet extends Floozel { }
public class Floozel { Set<Klakker> k; }
```

C.
```
import java.util.*;
class Floozet { }
class Quizel implements Klakker { }
class Jammer { List<Quizel> q; }
interface Klakker { }
class Floozel extends Jammer { List<Floozet> f; }
```

D.
```
import java.util.*;
interface Jammer extends Quizel { }
interface Klakker { }
interface Quizel extends Klakker { }
interface Floozel extends Jammer, Floozet { }
interface Floozet { }
```

Answer:

☑ **A** and **C** are correct. The phrase "type of" indicates an "is-a" relationship (extends or implements), and the phrase "have" is of course a "has-a" relationship (usually instance variables).

☒ **B** and **D** are incorrect based on the above information.
(Objective 5.5)

15. Given:

```
 3. class A { }
 4. class B extends A { }
 5. public class ComingThru {
 6.   static String s = "-";
 7.   public static void main(String[] args) {
 8.     A[] aa = new A[2];
 9.     B[] ba = new B[2];
10.     sifter(aa);
11.     sifter(ba);
12.     sifter(7);
13.     System.out.println(s);
14.   }
15.   static void sifter(A[]... a2)    { s += "1"; }
16.   static void sifter(B[]... b1)    { s += "2"; }
17.   static void sifter(B[] b1)       { s += "3"; }
18.   static void sifter(Object o)     { s += "4"; }
19. }
```

What is the result?

A. `-124`

B. `-134`

C. `-424`

D. `-434`

E. `-444`

F. Compilation fails

Answer:

☑ **D** is correct. In general, overloaded var-args methods are chosen last. Remember that arrays are objects. Finally, an int can be boxed to an Integer and then "widened" to an Object.

☒ **A, B, C, E,** and **F** are incorrect based on the above information.
(Objective 1.5)

This page intentionally left blank

3

Assignments

CERTIFICATION OBJECTIVES

- Use Class Members
- Develop Wrapper Code & Autoboxing Code
- Determine the Effects of Passing Variables into Methods
- Recognize when Objects Become Eligible for Garbage Collection

✓ Two-Minute Drill

Q&A Self Test

Stack and Heap—Quick Review

For most people, understanding the basics of the stack and the heap makes it far easier to understand topics like argument passing, polymorphism, threads, exceptions, and garbage collection. In this section, we'll stick to an overview, but we'll expand these topics several more times throughout the book.

For the most part, the various pieces (methods, variables, and objects) of Java programs live in one of two places in memory: the stack or the heap. For now, we're going to worry about only three types of things: instance variables, local variables, and objects:

- Instance variables and objects live on the heap.
- Local variables live on the stack.

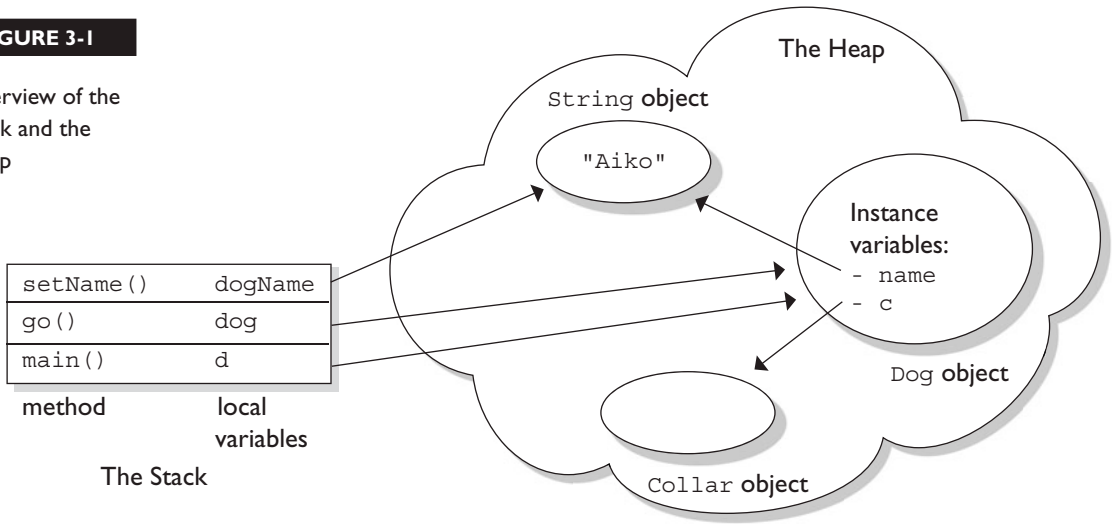
Let's take a look at a Java program, and how its various pieces are created and map into the stack and the heap:

```
 1. class Collar { }
 2.
 3. class Dog {
 4.   Collar c;          // instance variable
 5.   String name;       // instance variable
 6.
 7.   public static void main(String [] args) {
 8.
 9.     Dog d;                          // local variable: d
10.     d = new Dog();
11.     d.go(d);
12.   }
13.   void go(Dog dog) {                // local variable: dog
14.     c = new Collar();
15.     dog.setName("Aiko");
16.   }
17.   void setName(String dogName) {    // local var: dogName
18.     name = dogName;
19.     // do more stuff
20.   }
21. }
```

Figure 3-1 shows the state of the stack and the heap once the program reaches line 19. Following are some key points:

FIGURE 3-1

Overview of the Stack and the Heap



- Line 7—`main()` is placed on the stack.
- Line 9—reference variable `d` is created on the stack, but there's no `Dog` object yet.
- Line 10—a new `Dog` object is created and is assigned to the `d` reference variable.
- Line 11—a copy of the reference variable `d` is passed to the `go()` method.
- Line 13—the `go()` method is placed on the stack, with the `dog` parameter as a local variable.
- Line 14—a new `Collar` object is created on the heap, and assigned to Dog's instance variable.
- Line 17—`setName()` is added to the stack, with the `dogName` parameter as its local variable.
- Line 18—the `name` instance variable now also refers to the String object.
- Notice that two *different* local variables refer to the same `Dog` object.
- Notice that one local variable and one instance variable both refer to the same String `Aiko`.
- After Line 19 completes, `setName()` completes and is removed from the stack. At this point the local variable `dogName` disappears too, although the String object it referred to is still on the heap.

CERTIFICATION OBJECTIVE

Literals, Assignments, and Variables (Exam Objectives 1.3 and 7.6)

1.3 Develop code that declares, initializes, and uses primitives, arrays, enums, and objects as static, instance, and local variables. Also, use legal identifiers for variable names.

7.6 Write code that correctly applies the appropriate operators including assignment operators (limited to: =, +=, -=)…

Literal Values for All Primitive Types

A primitive literal is merely a source code representation of the primitive data types—in other words, an integer, floating-point number, boolean, or character that you type in while writing code. The following are examples of primitive literals:

```
'b'          // char literal
42           // int literal
false        // boolean literal
2546789.343  // double literal
```

Integer Literals

There are three ways to represent integer numbers in the Java language: decimal (base 10), octal (base 8), and hexadecimal (base 16). Most exam questions with integer literals use decimal representations, but the few that use octal or hexadecimal are worth studying for. Even though the odds that you'll ever actually use octal in the real world are astronomically tiny, they were included in the exam just for fun.

Decimal Literals Decimal integers need no explanation; you've been using them since grade one or earlier. Chances are you don't keep your checkbook in hex. (If you do, there's a Geeks Anonymous [GA] group ready to help.) In the Java language, they are represented as is, with no prefix of any kind, as follows:

```
int length = 343;
```

Octal Literals Octal integers use only the digits 0 to 7. In Java, you represent an integer in octal form by placing a zero in front of the number, as follows:

```
class Octal {
  public static void main(String [] args) {
    int six = 06;     // Equal to decimal 6
    int seven = 07;   // Equal to decimal 7
    int eight = 010;  // Equal to decimal 8
    int nine = 011;   // Equal to decimal 9
    System.out.println("Octal 010 = " + eight);
  }
}
```

Notice that when we get past seven and are out of digits to use (we are allowed only the digits 0 through 7 for octal numbers), we revert back to zero, and one is added to the beginning of the number. You can have up to 21 digits in an octal number, not including the leading zero. If we run the preceding program, it displays the following:

```
Octal 010 = 8
```

Hexadecimal Literals Hexadecimal (hex for short) numbers are constructed using 16 distinct symbols. Because we never invented single digit symbols for the numbers 10 through 15, we use alphabetic characters to represent these digits. Counting from 0 through 15 in hex looks like this:

```
0 1 2 3 4 5 6 7 8 9 a b c d e f
```

Java will accept capital or lowercase letters for the extra digits (one of the few places Java is not case-sensitive!). You are allowed up to 16 digits in a hexadecimal number, not including the prefix 0x or the optional suffix extension L, which will be explained later. All of the following hexadecimal assignments are legal:

```
class HexTest {
  public static void main (String [] args) {
    int x = 0X0001;
    int y = 0x7fffffff;
    int z = 0xDeadCafe;
    System.out.println("x = " + x + " y = " + y + " z = " + z);
  }
}
```

Running HexTest produces the following output:

```
x = 1 y = 2147483647 z = -559035650
```

Don't be misled by changes in case for a hexadecimal digit or the 'x' preceding it. `0XCAFE` and `0xcafe` are both legal *and have the same value*.

All three integer literals (octal, decimal, and hexadecimal) are defined as `int` by default, but they may also be specified as `long` by placing a suffix of `L` or `l` after the number:

```
long jo = 110599L;
long so = 0xFFFFl;  // Note the lowercase 'l'
```

Floating-Point Literals

Floating-point numbers are defined as a number, a decimal symbol, and more numbers representing the fraction.

```
double d = 11301874.9881024;
```

In the preceding example, the number 11301874.9881024 is the literal value. Floating-point literals are defined as double (64 bits) by default, so if you want to assign a floating-point literal to a variable of type float (32 bits), you must attach the suffix F or f to the number. If you don't, the compiler will complain about a possible loss of precision, because you're trying to fit a number into a (potentially) less precise "container." The F suffix gives you a way to tell the compiler, "Hey, I know what I'm doing, and I'll take the risk, thank you very much."

```
float f = 23.467890;         // Compiler error, possible loss
                             // of precision
float g = 49837849.029847F;  // OK; has the suffix "F"
```

You may also optionally attach a D or d to double literals, but it is not necessary because this is the default behavior.

```
double d = 110599.995011D; // Optional, not required
double  g = 987.897;       // No 'D' suffix, but OK because the
                           // literal is a double by default
```

Look for numeric literals that include a comma, for example,

```
int x = 25,343;  // Won't compile because of the comma
```

Boolean Literals

Boolean literals are the source code representation for boolean values. A boolean value can only be defined as `true` or `false`. Although in C (and some other languages) it is common to use numbers to represent `true` or `false`, this will not work in Java. Again, repeat after me, "Java is not C++."

```
boolean t = true;  // Legal
boolean  f = 0;    // Compiler error!
```

Be on the lookout for questions that use numbers where booleans are required. You might see an `if` test that uses a number, as in the following:

```
int x = 1;  if (x) {  } // Compiler error!
```

Character Literals

A char literal is represented by a single character in single quotes.

```
char a = 'a';
char b = '@';
```

You can also type in the Unicode value of the character, using the Unicode notation of prefixing the value with \u as follows:

```
char letterN = '\u004E'; // The letter 'N'
```

Remember, characters are just 16-bit unsigned integers under the hood. That means you can assign a number literal, assuming it will fit into the unsigned 16-bit range (65535 or less). For example, the following are all legal:

```
char a = 0x892;         // hexadecimal literal
char b = 982;           // int literal
char c = (char)70000;   // The cast is required; 70000 is
                        // out of char range
```

```
char d = (char) -98;   // Ridiculous, but legal
```

And the following are not legal and produce compiler errors:

```
char e = -29;   // Possible loss of precision; needs a cast
char f = 70000  // Possible loss of precision; needs a cast
```

You can also use an escape code if you want to represent a character that can't be typed in as a literal, including the characters for linefeed, newline, horizontal tab, backspace, and single quotes.

```
char c = '\"';    // A double quote
char d = '\n';    // A newline
```

Literal Values for Strings

A string literal is a source code representation of a value of a String object. For example, the following is an example of two ways to represent a string literal:

```
String s = "Bill Joy";
System.out.println("Bill" + " Joy");
```

Although strings are not primitives, they're included in this section because they can be represented as literals—in other words, typed directly into code. The only other nonprimitive type that has a literal representation is an array, which we'll look at later in the chapter.

```
Thread t = ???  // what literal value could possibly go here?
```

Assignment Operators

Assigning a value to a variable seems straightforward enough; you simply assign the stuff on the right side of the = to the variable on the left. Well, sure, but don't expect to be tested on something like this:

```
x = 6;
```

No, you won't be tested on the no-brainer (technical term) assignments. You will, however, be tested on the trickier assignments involving complex

expressions and casting. We'll look at both primitive and reference variable assignments. But before we begin, let's back up and peek inside a variable. What is a variable? How are the variable and its value related?

Variables are just bit holders, with a designated type. You can have an `int` holder, a `double` holder, a Button holder, and even a String[] holder. Within that holder is a bunch of bits representing the value. For primitives, the bits represent a numeric value (although we don't know what that bit pattern looks like for `boolean`, luckily, we don't care). A `byte` with a value of 6, for example, means that the bit pattern in the variable (the `byte` holder) is 00000110, representing the 8 bits.

So the value of a primitive variable is clear, but what's inside an object holder? If you say,

```
Button b = new Button();
```

what's inside the Button holder `b`? Is it the Button object? No! A variable referring to an object is just that—a *reference* variable. A reference variable bit holder contains bits representing a *way to get to the object*. We don't know what the format is. The way in which object references are stored is virtual-machine specific (it's a pointer to something, we just don't know what that something really is). All we can say for sure is that the variable's value is *not* the object, but rather a value representing a specific object on the heap. Or `null`. If the reference variable has not been assigned a value, or has been explicitly assigned a value of `null`, the variable holds bits representing—you guessed it—`null`. You can read

```
Button b = null;
```

as "The Button variable b is not referring to any object."

So now that we know a variable is just a little box o' bits, we can get on with the work of changing those bits. We'll look first at assigning values to primitives, and finish with assignments to reference variables.

Primitive Assignments

The equal (=) sign is used for assigning a value to a variable, and it's cleverly named the assignment operator. There are actually 12 assignment operators, but only the five most commonly used are on the exam, and they are covered in Chapter 4.

You can assign a primitive variable using a literal or the result of an expression.

Take a look at the following:

```
int x = 7;      // literal assignment
int y = x + 2; // assignment with an expression
               // (including a literal)
int z = x * y; // assignment with an expression
```

The most important point to remember is that a literal integer (such as 7) is always implicitly an `int`. Thinking back to Chapter 1, you'll recall that an `int` is a 32-bit value. No big deal if you're assigning a value to an `int` or a `long` variable, but what if you're assigning to a `byte` variable? After all, a `byte`-sized holder can't hold as many bits as an `int`-sized holder. Here's where it gets weird. The following is legal,

```
byte b = 27;
```

but only because the compiler automatically narrows the literal value to a `byte`. In other words, the compiler puts in the *cast*. The preceding code is identical to the following:

```
byte b = (byte) 27; // Explicitly cast the int literal to a byte
```

It looks as though the compiler gives you a break, and lets you take a shortcut with assignments to integer variables smaller than an `int`. (Everything we're saying about `byte` applies equally to `char` and `short`, both of which are smaller than an `int`.) We're not actually at the weird part yet, by the way.

We know that a literal integer is always an `int`, but more importantly, the result of an expression involving anything `int`-sized or smaller is always an `int`. In other words, add two `bytes` together and you'll get an `int`—even if those two bytes are tiny. Multiply an `int` and a `short` and you'll get an `int`. Divide a `short` by a `byte` and you'll get…an `int`. OK, now we're at the weird part. Check this out:

```
byte a = 3;     // No problem, 3 fits in a byte
byte b = 8;     // No problem, 8 fits in a byte
byte c = b + c; // Should be no problem, sum of the two bytes
                // fits in a byte
```

The last line won't compile! You'll get an error something like this:

```
TestBytes.java:5: possible loss of precision
found   : int
required: byte
    byte c = a + b;
                 ^
```

We tried to assign the sum of two bytes to a `byte` variable, the result of which (11) was definitely small enough to fit into a `byte`, but the compiler didn't care. It knew the rule about `int`-or-smaller expressions always resulting in an `int`. It would have compiled if we'd done the *explicit* cast:

```
byte c = (byte) (a + b);
```

Primitive Casting

Casting lets you convert primitive values from one type to another. We mentioned primitive casting in the previous section, but now we're going to take a deeper look. (Object casting was covered in Chapter 2.)

Casts can be implicit or explicit. An implicit cast means you don't have to write code for the cast; the conversion happens automatically. Typically, an implicit cast happens when you're doing a widening conversion. In other words, putting a smaller thing (say, a `byte`) into a bigger container (like an `int`). Remember those "`possible loss of precision`" compiler errors we saw in the assignments section? Those happened when we tried to put a larger thing (say, a `long`) into a smaller container (like a `short`). The large-value-into-small-container conversion is referred to as *narrowing* and requires an explicit cast, where you tell the compiler that you're aware of the danger and accept full responsibility. First we'll look at an implicit cast:

```
int a = 100;
long b = a; // Implicit cast, an int value always fits in a long
```

An explicit casts looks like this:

```
float a = 100.001f;
int b = (int)a; // Explicit cast, the float could lose info
```

Integer values may be assigned to a `double` variable without explicit casting, because any integer value can fit in a 64-bit `double`. The following line demonstrates this:

```
double d = 100L; // Implicit cast
```

In the preceding statement, a `double` is initialized with a `long` value (as denoted by the `L` after the numeric value). No cast is needed in this case because a `double` can hold every piece of information that a `long` can store. If, however, we want to assign a `double` value to an integer type, we're attempting a narrowing conversion and the compiler knows it:

```
class Casting {
  public static void main(String [] args) {
    int x = 3957.229; // illegal
  }
}
```

If we try to compile the preceding code, we get an error something like:

```
%javac Casting.java
Casting.java:3: Incompatible type for declaration. Explicit cast
needed to convert double to int.
      int x = 3957.229; // illegal
1 error
```

In the preceding code, a floating-point value is being assigned to an integer variable. Because an integer is not capable of storing decimal places, an error occurs. To make this work, we'll cast the floating-point number into an `int`:

```
class Casting {
  public static void main(String [] args) {
    int x = (int)3957.229; // legal cast
    System.out.println("int x = " + x);
  }
}
```

When you cast a floating-point number to an integer type, the value loses all the digits after the decimal. The preceding code will produce the following output:

```
int x = 3957
```

We can also cast a larger number type, such as a `long`, into a smaller number type, such as a `byte`. Look at the following:

```
class Casting {
  public static void main(String [] args) {
    long l = 56L;
    byte b = (byte)l;
    System.out.println("The byte is " + b);
  }
}
```

The preceding code will compile and run fine. But what happens if the `long` value is larger than 127 (the largest number a `byte` can store)? Let's modify the code:

```
class Casting {
  public static void main(String [] args) {
    long l = 130L;
    byte b = (byte)l;
    System.out.println("The byte is " + b);
  }
}
```

The code compiles fine, and when we run it we get the following:

```
%java Casting
The byte is -126
```

You don't get a runtime error, even when the value being narrowed is too large for the type. The bits to the left of the lower 8 just…go away. If the leftmost bit (the sign bit) in the `byte` (or any integer primitive) now happens to be a 1, the primitive will have a negative value.

EXERCISE 3-1

Casting Primitives

Create a `float` number type of any value, and assign it to a `short` using casting.

1. Declare a `float` variable: `float f = 234.56F;`
2. Assign the `float` to a `short`: `short s = (short)f;`

Assigning Floating-Point Numbers Floating-point numbers have slightly different assignment behavior than integer types. First, you must know that every floating-point literal is implicitly a `double` (64 bits), not a `float`. So the literal `32.3`, for example, is considered a `double`. If you try to assign a `double` to a `float`, the compiler knows you don't have enough room in a 32-bit `float` container to hold the precision of a 64-bit `double`, and it lets you know. The following code looks good, but won't compile:

```
float f = 32.3;
```

You can see that `32.3` should fit just fine into a float-sized variable, but the compiler won't allow it. In order to assign a floating-point literal to a `float` variable, you must either cast the value or append an `f` to the end of the literal. The following assignments will compile:

```
float f = (float) 32.3;
float g = 32.3f;
float h = 32.3F;
```

Assigning a Literal That Is Too Large for the Variable We'll also get a compiler error if we try to assign a literal value that the compiler knows is too big to fit into the variable.

```
byte a = 128; // byte can only hold up to 127
```

The preceding code gives us an error something like

```
TestBytes.java:5: possible loss of precision
found   : int
required: byte
byte a = 128;
```

We can fix it with a cast:

```
byte a = (byte) 128;
```

But then what's the result? When you narrow a primitive, Java simply truncates the higher-order bits that won't fit. In other words, it loses all the bits to the left of the bits you're narrowing to.

Let's take a look at what happens in the preceding code. There, 128 is the bit pattern 10000000. It takes a full 8 bits to represent 128. But because the literal 128 is an int, we actually get 32 bits, with the 128 living in the right-most (lower-order) 8 bits. So a literal 128 is actually

00000000000000000000000010000000

Take our word for it; there are 32 bits there.

To narrow the 32 bits representing 128, Java simply lops off the leftmost (higher-order) 24 bits. We're left with just the 10000000. But remember that a byte is signed, with the leftmost bit representing the sign (and not part of the value of the variable). So we end up with a negative number (the 1 that used to represent 128 now represents the negative sign bit). Remember, to find out the value of a negative number using two's complement notation, you flip all of the bits and then add 1. Flipping the 8 bits gives us 01111111, and adding 1 to that gives us 10000000, or back to 128! And when we apply the sign bit, we end up with –128.

You must use an explicit cast to assign 128 to a byte, and the assignment leaves you with the value –128. A cast is nothing more than your way of saying to the compiler, "Trust me. I'm a professional. I take full responsibility for anything weird that happens when those top bits are chopped off."

That brings us to the compound assignment operators. The following will compile,

```
byte b = 3;
b += 7;         // No problem - adds 7 to b (result is 10)
```

and is equivalent to

```
byte b = 3;
b = (byte) (b + 7);  // Won't compile without the
                     // cast, since b + 7 results in an int
```

The compound assignment operator += lets you add to the value of b, without putting in an explicit cast. In fact, +=, -=, *=, and /= will all put in an implicit cast.

Assigning One Primitive Variable to Another Primitive Variable

When you assign one primitive variable to another, the contents of the right-hand variable are copied. For example,

```
int a = 6;
int b = a;
```

This code can be read as, "Assign the bit pattern for the number 6 to the int variable `a`. Then copy the bit pattern in `a`, and place the copy into variable `b`."

So, both variables now hold a bit pattern for 6, but the two variables have no other relationship. We used the variable `a` *only* to copy its contents. At this point, `a` and `b` have identical contents (in other words, identical values), but if we change the contents of *either* `a` or `b`, the other variable won't be affected.

Take a look at the following example:

```
class ValueTest {
   public static void main (String [] args) {
      int a = 10;  // Assign a value to a
      System.out.println("a = " + a);
      int b = a;
      b = 30;
      System.out.println("a = " + a + " after change to b");
   }
}
```

The output from this program is

```
%java ValueTest
a = 10
a = 10 after change to b
```

Notice the value of `a` stayed at 10. The key point to remember is that even after you assign `a` to `b`, `a` and `b` are not referring to the same place in memory. The `a` and `b` variables do not share a single value; they have identical copies.

Reference Variable Assignments

You can assign a newly created object to an object reference variable as follows:

```
Button b = new Button();
```

The preceding line does three key things:

- Makes a reference variable named `b`, of type `Button`
- Creates a new `Button` object on the heap
- Assigns the newly created `Button` object to the reference variable `b`

You can also assign `null` to an object reference variable, which simply means the variable is not referring to any object:

```
Button c = null;
```

The preceding line creates space for the Button reference variable (the bit holder for a reference value), but doesn't create an actual Button object.

As we discussed in the last chapter, you can also use a reference variable to refer to any object that is a subclass of the declared reference variable type, as follows:

```
public class Foo {
   public void doFooStuff() { }
}
public class Bar extends Foo {
   public void doBarStuff() { }
}
class Test {
   public static void main (String [] args) {
      Foo reallyABar = new Bar();  // Legal because Bar is a
                                   // subclass of Foo
      Bar reallyAFoo = new Foo();  // Illegal! Foo is not a
                                   // subclass of Bar
   }
}
```

The rule is that you can assign a subclass of the declared type, but not a superclass of the declared type. Remember, a Bar object is guaranteed to be able to do anything a Foo can do, so anyone with a Foo reference can invoke Foo methods even though the object is actually a Bar.

In the preceding code, we see that Foo has a method `doFooStuff()` that someone with a Foo reference might try to invoke. If the object referenced by the Foo variable is really a Foo, no problem. But it's also no problem if the object is a Bar, since Bar inherited the `doFooStuff()` method. You can't make it work

in reverse, however. If somebody has a Bar reference, they're going to invoke `doBarStuff()`, but if the object is a Foo, it won't know how to respond.

Variable Scope

Once you've declared and initialized a variable, a natural question is "How long will this variable be around?" This is a question regarding the scope of variables. And not only is scope an important thing to understand in general, it also plays a big part in the exam. Let's start by looking at a class file:

```
class Layout {                          // class

  static int s = 343;                   // static variable

  int x;                                // instance variable

  { x = 7; int x2 = 5; }                // initialization block

  Layout() { x += 8; int x3 = 6;}       // constructor

  void doStuff() {                      // method

    int y = 0;                          // local variable

    for(int z = 0; z < 4; z++) {        // 'for' code block
      y += z + x;
    }
  }
}
```

As with variables in all Java programs, the variables in this program (`s`, `x`, `x2`, `x3`, `y`, and `z`) all have a scope:

- `s` is a static variable.
- `x` is an instance variable.
- `y` is a local variable (sometimes called a "method local" variable).
- `z` is a block variable.
- `x2` is an init block variable, a flavor of local variable.
- `x3` is a constructor variable, a flavor of local variable.

For the purposes of discussing the scope of variables, we can say that there are four basic scopes:

- Static variables have the longest scope; they are created when the class is loaded, and they survive as long as the class stays loaded in the Java Virtual Machine (JVM).
- Instance variables are the next most long-lived; they are created when a new instance is created, and they live until the instance is removed.
- Local variables are next; they live as long as their method remains on the stack. As we'll soon see, however, local variables can be alive, and still be "out of scope".
- Block variables live only as long as the code block is executing.

Scoping errors come in many sizes and shapes. One common mistake happens when a variable is *shadowed* and two scopes overlap. We'll take a detailed look at shadowing in a few pages. The most common reason for scoping errors is when you attempt to access a variable that is not in scope. Let's look at three common examples of this type of error:

- Attempting to access an instance variable from a static context (typically from `main()`).

```
class ScopeErrors {
  int x = 5;
  public static void main(String[] args) {
    x++;   // won't compile, x is an 'instance' variable
  }
}
```

- Attempting to access a local variable from a nested method.

  When a method, say `go()`, invokes another method, say `go2()`, `go2()` won't have access to `go()`'s local variables. While `go2()` is executing, `go()`'s local variables are still *alive*, but they are *out of scope*. When `go2()` completes, it is removed from the stack, and `go()` resumes execution. At this point, all of `go()`'s previously declared variables are back in scope. For example:

```
class ScopeErrors {
  public static void main(String [] args) {
    ScopeErrors s = new ScopeErrors();
    s.go();
  }
  void go() {
    int y = 5;
```

```
    go2();
    y++;          // once go2() completes, y is back in scope
  }
  void go2() {
    y++;          // won't compile, y is local to go()
  }
}
```

- Attempting to use a block variable after the code block has completed.
  It's very common to declare and use a variable within a code block, but be careful not to try to use the variable once the block has completed:

```
void go3() {
  for(int z = 0; z < 5; z++) {
    boolean test = false;
    if(z == 3) {
      test = true;
      break;
    }
  }
  System.out.print(test);   // 'test' is an ex-variable,
                            // it has ceased to be...
}
```

In the last two examples, the compiler will say something like this:

```
cannot find symbol
```

This is the compiler's way of saying, "That variable you just tried to use? Well, it might have been valid in the distant past (like one line of code ago), but this is Internet time baby, I have no memory of such a variable."

exam Watch

Pay extra attention to code block scoping errors. You might see them in switches, try-catches, for, do, and while loops.

Using a Variable or Array Element That Is Uninitialized and Unassigned

Java gives us the option of initializing a declared variable or leaving it uninitialized. When we attempt to use the uninitialized variable, we can get different behavior depending on what type of variable or array we are dealing with (primitives or objects). The behavior also depends on the level (scope) at which we are declaring our variable. An instance variable is declared within the class but outside any method or constructor, whereas a local variable is declared within a method (or in the argument list of the method).

Local variables are sometimes called stack, temporary, automatic, or method variables, but the rules for these variables are the same regardless of what you call them. Although you can leave a local variable uninitialized, the compiler complains if you try to use a local variable before initializing it with a value, as we shall see.

Primitive and Object Type Instance Variables

Instance variables (also called *member* variables) are variables defined at the class level. That means the variable declaration is not made within a method, constructor, or any other initializer block. Instance variables are initialized to a default value each time a new instance is created, although they may be given an explicit value after the object's super-constructors have completed. Table 3-1 lists the default values for primitive and object types.

TABLE 3-1 Default Values for Primitives and Reference Types

Variable Type	Default Value
Object reference	`null` (not referencing any object)
`byte, short, int, long`	`0`
`float, double`	`0.0`
`boolean`	`false`
`char`	`'\u0000'`

Primitive Instance Variables

In the following example, the integer `year` is defined as a class member because it is within the initial curly braces of the class and not within a method's curly braces:

```
public class BirthDate {
  int year;                                    // Instance variable
  public static void main(String [] args) {
    BirthDate bd = new BirthDate();
    bd.showYear();
  }
  public void showYear() {
    System.out.println("The year is " + year);
  }
}
```

When the program is started, it gives the variable `year` a value of zero, the default value for primitive number instance variables.

It's a good idea to initialize all your variables, even if you're assigning them with the default value. Your code will be easier to read; programmers who have to maintain your code (after you win the lottery and move to Tahiti) will be grateful.

Object Reference Instance Variables

When compared with uninitialized primitive variables, object references that aren't initialized are a completely different story. Let's look at the following code:

```
public class Book {
  private String title;          // instance reference variable
  public String getTitle() {
    return title;
  }
  public static void main(String [] args) {
    Book b = new Book();
    System.out.println("The title is " + b.getTitle());
  }
}
```

This code will compile fine. When we run it, the output is

```
The title is null
```

The title variable has not been explicitly initialized with a String assignment, so the instance variable value is `null`. Remember that `null` is not the same as an empty String (""). A `null` value means the reference variable is not referring to any object on the heap. The following modification to the Book code runs into trouble:

```
public class Book {
  private String title;          // instance reference variable
  public String getTitle() {
    return title;
  }
  public static void main(String [] args) {
    Book b = new Book();
    String s = b.getTitle();     // Compiles and runs
    String t = s.toLowerCase();  // Runtime Exception!
  }
}
```

When we try to run the Book class, the JVM will produce something like this:

```
Exception in thread "main" java.lang.NullPointerException
     at Book.main(Book.java:9)
```

We get this error because the reference variable `title` does not point (refer) to an object. We can check to see whether an object has been instantiated by using the keyword `null`, as the following revised code shows:

```
public class Book {
  private String title;          // instance reference variable
  public String getTitle() {
    return title;
  }
  public static void main(String [] args) {
    Book b = new Book();
    String s = b.getTitle(); // Compiles and runs
    if (s != null) {
      String t = s.toLowerCase();
    }
```

```
    }
  }
```

The preceding code checks to make sure the object referenced by the variable `s` is not `null` before trying to use it. Watch out for scenarios on the exam where you might have to trace back through the code to find out whether an object reference will have a value of `null`. In the preceding code, for example, you look at the instance variable declaration for title, see that there's no explicit initialization, recognize that the title variable will be given the default value of `null`, and then realize that the variable `s` will also have a value of `null`. Remember, the value of `s` is a copy of the value of `title` (as returned by the `getTitle()` method), so if title is a `null` reference, `s` will be too.

Array Instance Variables

Later in this chapter we'll be taking a very detailed look at declaring, constructing, and initializing arrays and multidimensional arrays. For now, we're just going to look at the rule for an array element's default values.

An array is an object; thus, an array instance variable that's declared but not explicitly initialized will have a value of `null`, just as any other object reference instance variable. But…if the array is initialized, what happens to the elements contained *in* the array? All array elements are given their default values—the same default values that elements of that type get when they're instance variables. *The bottom line: Array elements are always, always, always given default values, regardless of where the array itself is declared or instantiated.*

If we initialize an array, object reference elements will equal null if they are not initialized individually with values. If primitives are contained in an array, they will be given their respective default values. For example, in the following code, the array `year` will contain 100 integers that all equal zero by default:

```
public class BirthDays {
  static int [] year = new int[100];
  public static void main(String [] args) {
    for(int i=0;i<100;i++)
      System.out.println("year[" + i + "] = " + year[i]);
  }
}
```

When the preceding code runs, the output indicates that all 100 integers in the array equal zero.

Local (Stack, Automatic) Primitives and Objects

Local variables are defined within a method, and they include a method's parameters.

exam Watch

"Automatic" is just another term for "local variable." It does not mean the automatic variable is automatically assigned a value! The opposite is true. An automatic variable must be assigned a value in the code, or the compiler will complain.

Local Primitives

In the following time travel simulator, the integer year is defined as an automatic variable because it is within the curly braces of a method.

```
public class TimeTravel {
  public static void main(String [] args) {
    int year = 2050;
    System.out.println("The year is " + year);
  }
}
```

Local variables, including primitives, always, always, always must be initialized *before* you attempt to use them (though not necessarily on the same line of code). Java does not give local variables a default value; you must explicitly initialize them with a value, as in the preceding example. If you try to use an uninitialized primitive in your code, you'll get a compiler error:

```
public class TimeTravel {
  public static void main(String [] args) {
    int year; // Local variable (declared but not initialized)
    System.out.println("The year is " + year); // Compiler error
  }
}
```

Compiling produces output something like this:

```
%javac TimeTravel.java
TimeTravel.java:4: Variable year may not have been initialized.
           System.out.println("The year is " + year);
1 error
```

To correct our code, we must give the integer `year` a value. In this updated example, we declare it on a separate line, which is perfectly valid:

```
public class TimeTravel {
  public static void main(String [] args) {
    int year;      // Declared but not initialized
    int day;       // Declared but not initialized
    System.out.println("You step into the portal.");
    year = 2050;   // Initialize (assign an explicit value)
    System.out.println("Welcome to the year " + year);
  }
}
```

Notice in the preceding example we declared an integer called `day` that never gets initialized, yet the code compiles and runs fine. Legally, you can declare a local variable without initializing it as long as you don't use the variable, but let's face it, if you declared it, you probably had a reason (although we have heard of programmers declaring random local variables just for sport, to see if they can figure out how and why they're being used).

The compiler can't always tell whether a local variable has been initialized before use. For example, if you initialize within a logically conditional block (in other words, a code block that may not run, such as an `if` ***block or*** `for` ***loop without a literal value of*** `true` ***or*** `false` ***in the test), the compiler knows that the initialization might not happen, and can produce an error. The following code upsets the compiler:***

```
public class TestLocal {
  public static void main(String [] args) {
    int x;
    if (args[0] != null) { // assume you know this will
                           // always be true
```

```
      x = 7;               // compiler can't tell that this
                           // statement will run
    }
    int y = x;             // the compiler will choke here
  }
}
```

The compiler will produce an error something like this:

```
TestLocal.java:9: variable x might not have been initialized
```

Because of the compiler-can't-tell-for-certain problem, you will sometimes need to initialize your variable outside the conditional block, just to make the compiler happy. You know why that's important if you've seen the bumper sticker, "When the compiler's not happy, ain't nobody happy."

Local Object References

Objects references, too, behave differently when declared within a method rather than as instance variables. With instance variable object references, you can get away with leaving an object reference uninitialized, as long as the code checks to make sure the reference isn't `null` before using it. Remember, to the compiler, `null` is a value. You can't use the dot operator on a `null` reference, because *there is no object at the other end of it*, but a `null` reference is not the same as an *uninitialized* reference. Locally declared references can't get away with checking for `null` before use, unless you explicitly initialize the local variable to `null`. The compiler will complain about the following code:

```
import java.util.Date;
public class TimeTravel {
  public static void main(String [] args) {
    Date date;
    if (date == null)
      System.out.println("date is null");
  }
}
```

Compiling the code results in an error similar to the following:

```
%javac TimeTravel.java
TimeTravel.java:5: Variable date may not have been initialized.
          if (date == null)
1 error
```

Instance variable references are always given a default value of null, until explicitly initialized to something else. But local references are not given a default value; in other words, *they aren't null*. If you don't initialize a local reference variable, then by default, its value is…well that's the whole point—it doesn't have any value at all! So we'll make this simple: Just set the darn thing to `null` explicitly, until you're ready to initialize it to something else. The following local variable will compile properly:

```
Date date = null; // Explicitly set the local reference
                  // variable to null
```

Local Arrays

Just like any other object reference, array references declared within a method must be assigned a value before use. That just means you must declare and construct the array. You do not, however, need to explicitly initialize the elements of an array. We've said it before, but it's important enough to repeat: array elements are given their default values (0, `false`, `null`, `'\u0000'`, etc.) regardless of whether the array is declared as an instance or local variable. The array object itself, however, will not be initialized if it's declared locally. In other words, you must explicitly initialize an array reference if it's declared and used within a method, but at the moment you construct an array object, all of its elements are assigned their default values.

Assigning One Reference Variable to Another

With primitive variables, an assignment of one variable to another means the contents (bit pattern) of one variable are *copied* into another. Object reference variables work exactly the same way. The contents of a reference variable are a bit pattern, so if you assign reference variable `a` to reference variable `b`, the bit pattern in `a` is *copied* and the new *copy* is placed into `b`. (Some people have created a game around counting how many times we use the word *copy* in this chapter…this copy concept is a biggie!) If we assign an existing instance of an object to a new reference variable, then two reference variables will hold the same bit pattern—a bit pattern referring to a specific object on the heap. Look at the following code:

```
import java.awt.Dimension;
class ReferenceTest {
  public static void main (String [] args) {
    Dimension a = new Dimension(5,10);
    System.out.println("a.height = " + a.height);
    Dimension b = a;
    b.height = 30;
    System.out.println("a.height = " + a.height +
                       " after change to b");
  }
}
```

In the preceding example, a Dimension object `a` is declared and initialized with a width of 5 and a height of 10. Next, Dimension `b` is declared, and assigned the value of `a`. At this point, both variables (`a` and `b`) hold identical values, because the contents of a were copied into `b`. There is still only one Dimension object—the one that both `a` and `b` refer to. Finally, the height property is changed using the `b` reference. Now think for a minute: is this going to change the height property of `a` as well? Let's see what the output will be:

```
%java ReferenceTest
a.height = 10
a.height = 30 after change to b
```

From this output, we can conclude that both variables refer to the same instance of the Dimension object. When we made a change to `b`, the height property was also changed for `a`.

One exception to the way object references are assigned is String. In Java, String objects are given special treatment. For one thing, String objects are immutable; you can't change the value of a String object (lots more on this concept in Chapter 6). But it sure looks as though you can. Examine the following code:

```
class StringTest {
  public static void main(String [] args) {
    String x = "Java";  // Assign a value to x
    String y = x;       // Now y and x refer to the same
                        // String object

    System.out.println("y string = " + y);
    x = x + " Bean";    // Now modify the object using
                        // the x reference
```

```
    System.out.println("y string = " + y);
  }
}
```

You might think String `y` will contain the characters `Java Bean` after the variable `x` is changed, because Strings are objects. Let's see what the output is:

```
%java StringTest
y string = Java
y string = Java
```

As you can see, even though `y` is a reference variable to the same object that `x` refers to, when we change `x`, it doesn't change `y`! For any other object type, where two references refer to the same object, if either reference is used to modify the object, both references will see the change because there is still only a single object. *But any time we make any changes at all to a String, the VM will update the reference variable to refer to a different object.* The different object might be a new object, or it might not, but it will definitely be a different object. The reason we can't say for sure whether a new object is created is because of the String constant pool, which we'll cover in Chapter 6.

You need to understand what happens when you use a String reference variable to modify a string:

- A new string is created (or a matching String is found in the String pool), leaving the original String object untouched.
- The reference used to modify the String (or rather, make a new String by modifying a copy of the original) is then assigned the brand new String object.

So when you say

```
1. String s = "Fred";
2. String t = s;     // Now t and s refer to the same
                     // String object
3. t.toUpperCase();  // Invoke a String method that changes
                     // the String
```

you haven't changed the original String object created on line 1. When line 2 completes, both `t` and `s` reference the same String object. But when line 3 runs, rather than modifying the object referred to by `t` (which is the one and only String

object up to this point), a brand new String object is created. And then abandoned. Because the new String isn't assigned to a String variable, the newly created String (which holds the string "FRED") is toast. So while two String objects were created in the preceding code, only one is actually referenced, and both `t` and `s` refer to it. The behavior of Strings is extremely important in the exam, so we'll cover it in much more detail in Chapter 6.

CERTIFICATION OBJECTIVE

Passing Variables into Methods (Objective 7.3)

7.3 Determine the effect upon object references and primitive values when they are passed into methods that perform assignments or other modifying operations on the parameters.

Methods can be declared to take primitives and/or object references. You need to know how (or if) the caller's variable can be affected by the called method. The difference between object reference and primitive variables, when passed into methods, is huge and important. To understand this section, you'll need to be comfortable with the assignments section covered in the first part of this chapter.

Passing Object Reference Variables

When you pass an object variable into a method, you must keep in mind that you're passing the object *reference*, and not the actual object itself. Remember that a reference variable holds bits that represent (to the underlying VM) a way to get to a specific object in memory (on the heap). More importantly, you must remember that you aren't even passing the actual reference variable, but rather a *copy* of the reference variable. A copy of a variable means you get a copy of the bits in that variable, so when you pass a reference variable, you're passing a copy of the bits representing how to get to a specific object. In other words, both the caller and the called method will now have identical copies of the reference, and thus both will refer to the same exact (*not* a copy) object on the heap.

For this example, we'll use the Dimension class from the java.awt package:

```
1. import java.awt.Dimension;
2. class ReferenceTest {
```

```
 3.   public static void main (String [] args) {
 4.     Dimension d = new Dimension(5,10);
 5.     ReferenceTest rt = new ReferenceTest();
 6.     System.out.println("Before modify() d.height = "
                           + d.height);
 7.     rt.modify(d);
 8.     System.out.println("After modify() d.height = "
                           + d.height);
 9.   }
10.   void modify(Dimension dim) {
11.     dim.height = dim.height + 1;
12.     System.out.println("dim = " + dim.height);
13.   }
14. }
```

When we run this class, we can see that the `modify()` method was indeed able to modify the original (and only) Dimension object created on line 4.

```
C:\Java Projects\Reference>java ReferenceTest
Before modify() d.height = 10
dim = 11
After modify() d.height = 11
```

Notice when the Dimension object on line 4 is passed to the `modify()` method, any changes to the object that occur inside the method are being made to the object whose reference was passed. In the preceding example, reference variables `d` and `dim` both point to the same object.

Does Java Use Pass-By-Value Semantics?

If Java passes objects by passing the reference variable instead, does that mean Java uses pass-by-reference for objects? Not exactly, although you'll often hear and read that it does. Java is actually pass-by-value for all variables running within a single VM. Pass-by-value means pass-by-variable-value. And that means, pass-by-copy-of-the-variable! (There's that word *copy* again!)

It makes no difference if you're passing primitive or reference variables, you are always passing a copy of the bits in the variable. So for a primitive variable, you're passing a copy of the bits representing the value. For example, if you pass an `int` variable with the value of `3`, you're passing a copy of the bits representing `3`. The called method then gets its own copy of the value, to do with it what it likes.

And if you're passing an object reference variable, you're passing a copy of the bits representing the reference to an object. The called method then gets its own copy of the reference variable, to do with it what it likes. But because two identical reference variables refer to the exact same object, if the called method modifies the object (by invoking setter methods, for example), the caller will see that the object the caller's original variable refers to has also been changed. In the next section, we'll look at how the picture changes when we're talking about primitives.

The bottom line on pass-by-value: the called method can't change the caller's variable, although for object reference variables, the called method can change the object the variable referred to. What's the difference between changing the variable and changing the object? For object references, it means the called method can't reassign the caller's original reference variable and make it refer to a different object, or `null`. For example, in the following code fragment,

```
void bar() {
   Foo f = new Foo();
   doStuff(f);
}
void doStuff(Foo g) {
   g.setName("Boo");
   g = new Foo();
}
```

reassigning `g` does not reassign `f`! At the end of the `bar()` method, two Foo objects have been created, one referenced by the local variable `f` and one referenced by the local (argument) variable `g`. Because the `doStuff()` method has a copy of the reference variable, it has a way to get to the original Foo object, for instance to call the `setName()` method. But, the `doStuff()` method does *not* have a way to get to the `f` reference variable. So `doStuff()` can change values within the object `f` refers to, but `doStuff()` can't change the actual contents (bit pattern) of `f`. In other words, `doStuff()` can change the state of the object that `f` refers to, but it can't make `f` refer to a different object!

Passing Primitive Variables

Let's look at what happens when a primitive variable is passed to a method:

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

```
    int a = 1;
    ReferenceTest rt = new ReferenceTest();
    System.out.println("Before modify() a = " + a);
    rt.modify(a);
    System.out.println("After modify() a = " + a);
  }
  void modify(int number) {
    number = number + 1;
    System.out.println("number = " + number);
  }
}
```

In this simple program, the variable `a` is passed to a method called `modify()`, which increments the variable by 1. The resulting output looks like this:

```
Before modify() a = 1
number = 2
After modify() a = 1
```

Notice that `a` did not change after it was passed to the method. Remember, it was a copy of `a` that was passed to the method. When a primitive variable is passed to a method, it is passed by value, which means pass-by-copy-of-the-bits-in-the-variable.

FROM THE CLASSROOM

The Shadowy World of Variables

Just when you think you've got it all figured out, you see a piece of code with variables not behaving the way you think they should. You might have stumbled into code with a shadowed variable. You can shadow a variable in several ways. We'll look at the one most likely to trip you up: hiding an instance variable by shadowing it with a local variable. Shadowing involves redeclaring a variable that's already been declared somewhere else.

The effect of shadowing is to hide the previously declared variable in such a way that it may look as though you're using the hidden variable, but you're actually using the shadowing variable. You might find reasons to shadow a variable intentionally, but typically it happens by accident and causes hard-to-find bugs. On the exam, you can expect to see questions where shadowing plays a role.

FROM THE CLASSROOM

You can shadow an instance variable by declaring a local variable of the same name, either directly or as part of an argument:

```
class Foo {
  static int size = 7;
  static void changeIt(int size) {
    size = size + 200;
    System.out.println("size in changeIt is " + size);
  }
  public static void main (String [] args) {
    Foo f = new Foo();
    System.out.println("size = " + size);
    changeIt(size);
    System.out.println("size after changeIt is " + size);
  }
}
```

The preceding code appears to change the `size` instance variable in the `changeIt()` method, but because `changeIt()` has a parameter named `size`, the local `size` variable is modified while the instance variable `size` is untouched. Running class Foo prints

```
%java Foo
size = 7
size in changeIt is 207
size after changeIt is 7
```

Things become more interesting when the shadowed variable is an object reference, rather than a primitive:

```
class Bar {
  int barNum = 28;
}
```

FROM THE CLASSROOM

```
class Foo {
  Bar myBar = new Bar();
  void changeIt(Bar myBar) {
    myBar.barNum = 99;
    System.out.println("myBar.barNum in changeIt is " + myBar.barNum);
    myBar = new Bar();
    myBar.barNum = 420;
    System.out.println("myBar.barNum in changeIt is now " + myBar.barNum);
  }
  public static void main (String [] args) {
    Foo f = new Foo();
    System.out.println("f.myBar.barNum is " + f.myBar.barNum);
    f.changeIt(f.myBar);
    System.out.println("f.myBar.barNum after changeIt is "
                       + f.myBar.barNum);
  }
}
```

The preceding code prints out this:

```
f.myBar.barNum is 28
myBar.barNum in changeIt is 99
myBar.barNum in changeIt is now 420
f.myBar.barNum after changeIt is 99
```

You can see that the shadowing variable (the local parameter `myBar` in `changeIt()`) can still affect the `myBar` instance variable, because the `myBar` parameter receives a reference to the same Bar object. But when the local `myBar` is reassigned a new Bar object, which we then modify by changing its `barNum` value, Foo's original `myBar` instance variable is untouched.

CERTIFICATION OBJECTIVE

Array Declaration, Construction, and Initialization (Exam Objective 1.3)

1.3 Develop code that declares, initializes, and uses primitives, arrays, enums, and objects as static, instance, and local variables. Also, use legal identifiers for variable names.

Arrays are objects in Java that store multiple variables of the same type. Arrays can hold either primitives or object references, but the 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 this objective, 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)

There are several different ways to do each of those, and you need to know about all of them for the exam.

on the Job

Arrays are efficient, but most of the time you'll want to use one of the Collection types from java.util (including HashMap, ArrayList, TreeSet). Collection classes offer more flexible ways to access an object (for insertion, deletion, and so on) and unlike arrays, can expand or contract dynamically as you add or remove elements (they're really managed arrays, since they use arrays behind the scenes). There's a Collection type 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? A linked list? Chapter 7 covers them in more detail.

Declaring an Array

Arrays are declared by stating the type of element the array will hold, which can be an object or a primitive, followed by square brackets to the left or right of the identifier.

Declaring an array of primitives:

```
int[] key;  // brackets before name (recommended)
int key []; // brackets after name (legal but less readable)
            // spaces between the name and [] legal, but bad
```

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, and 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;  // recommended
String[] ManagerName [];    // yucky, but legal
```

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 make it past the compiler. Remember, the JVM doesn't allocate space until you actually instantiate the array object. That's when size matters.

Constructing an Array

Constructing an array means creating the array object on the *heap* (where all objects live)—i.e., doing a `new` on the array type. To create an array object, Java must know

how much space to allocate on the heap, so you must specify the size of the array at creation time. The size of the array is the number of elements the array will hold.

Constructing One-Dimensional Arrays

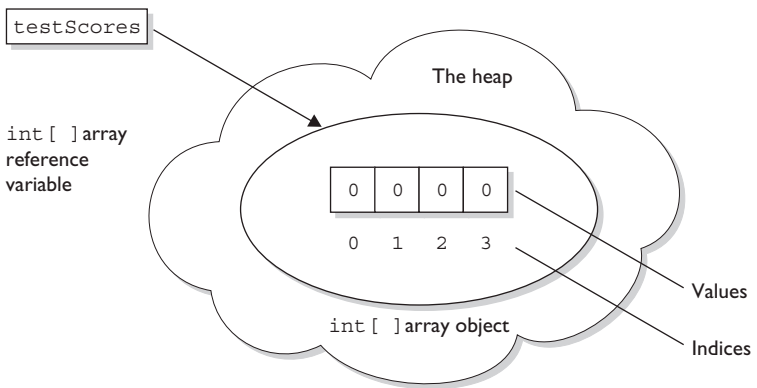
The most straightforward way to construct an array is to use the keyword new followed by the array type, with a bracket specifying how many elements of that type the array will hold. The following is an example of constructing an array of type int:

```
int[] testScores;         // Declares the array of ints
testScores = new int[4]; // constructs an array and assigns it
                          // to the testScores variable
```

The preceding code puts one new object on the heap—an array object holding four elements—with each element containing an `int` with a default value of 0. Think of this code as saying to the compiler, "Create an array object that will hold four `int`s, and assign it to the reference variable named `testScores`. Also, go ahead and set each int element to zero. Thanks." (The compiler appreciates good manners.) Figure 3-2 shows the `testScores` array on the heap, after construction.

FIGURE 3-2

A one-dimensional array on the Heap



You can also declare and construct an array in one statement as follows:

```
int[] testScores = new int[4];
```

This single statement produces the same result as the two previous statements. Arrays of object types can be constructed in the same way:

```
Thread[] threads = new Thread[5];
```

Remember that—despite how the code appears—the Thread constructor is not being invoked. We're not creating a Thread instance, but rather a single Thread array object. After the preceding statement, there are still no actual Thread objects!

exam Watch

Think carefully about how many objects are on the heap after a code statement or block executes. The exam will expect you to know, for example, that the preceding code produces just one object (the array assigned to the reference variable named threads). The single object referenced by threads holds five Thread reference variables, but no Thread objects have been created or assigned to those references.

Remember, arrays must always be given a size at the time they are constructed. The JVM needs the size to allocate the appropriate space on the heap for the new array object. It is never legal, for example, to do the following:

```
int[] carList = new int[]; // Will not compile; needs a size
```

So don't do it, and if you see it on the test, run screaming toward the nearest answer marked "Compilation fails."

exam Watch

You may see the words "construct", "create", and "instantiate" used interchangeably. They all mean, "An object is built on the heap." This also implies that the object's constructor runs, as a result of the construct/create/instantiate code. You can say with certainty, for example, that any code that uses the keyword `new`***, will (if it runs successfully) cause the class constructor and all superclass constructors to run.***

In addition to being constructed with `new`, arrays can also be created using a kind of syntax shorthand that creates the array while simultaneously initializing the array elements to values supplied in code (as opposed to default values). We'll look at that in the next section. For now, understand that because of these syntax shortcuts, objects can still be created even without you ever using or seeing the keyword `new`.

Constructing Multidimensional Arrays

Multidimensional arrays, remember, are simply arrays of arrays. So a two-dimensional array of type `int` is really an object of type `int` array (`int []`), with each element in that array holding a reference to another `int` array. The second dimension holds the actual `int` primitives. The following code declares and constructs a two-dimensional array of type `int`:

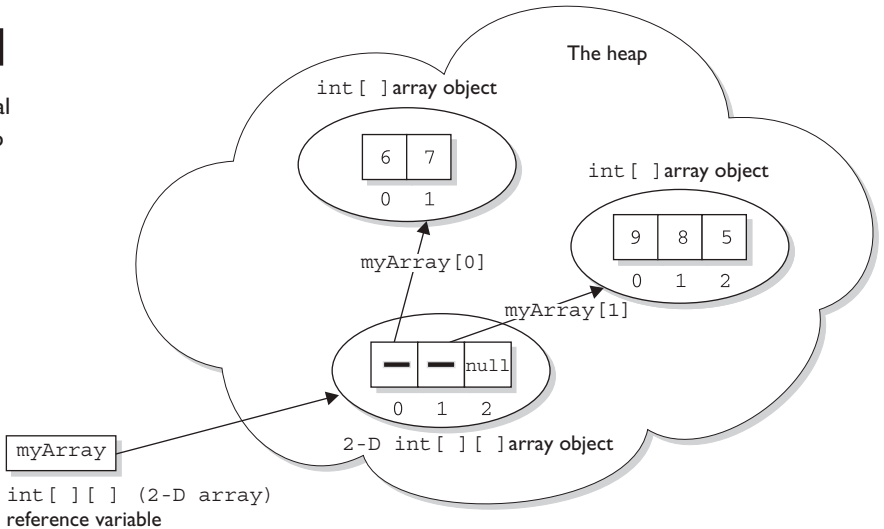
```
int[][] myArray = new int[3][];
```

Notice that only the first brackets are given a size. That's acceptable in Java, since the JVM needs to know only the size of the object assigned to the variable `myArray`.

Figure 3-3 shows how a two-dimensional `int` array works on the heap.

FIGURE 3-3

A two-dimensional array on the Heap



Picture demonstrates the result of the following code:

```
int[ ][ ] myArray = new int[3][ ];
myArray[0] = new int[2];
myArray[0][0] = 6;
myArray[0][1] = 7;
myArray[1] = new int[3];
myArray[1][0] = 9;
myArray[1][1] = 8;
myArray[1][2] = 5;
```

Initializing an Array

Initializing an array means putting things into it. The "things" in the array are the array's elements, and they're either primitive values (2, `x`, `false`, and so on), or objects referred to by the reference variables in the array. If you have an array of objects (as opposed to primitives), the array doesn't actually hold the objects, just as any other nonprimitive variable never actually holds the object, but instead holds a *reference* to the object. But we talk about arrays as, for example, "an array of five strings," even though what we really mean is, "an array of five references to String objects." Then the big question becomes whether or not those references are actually pointing (oops, this is Java, we mean referring) to real String objects, or are simply `null`. Remember, a reference that has not had an object assigned to it is a `null` reference. And if you try to actually use that `null` reference by, say, applying the dot operator to invoke a method on it, you'll get the infamous NullPointerException.

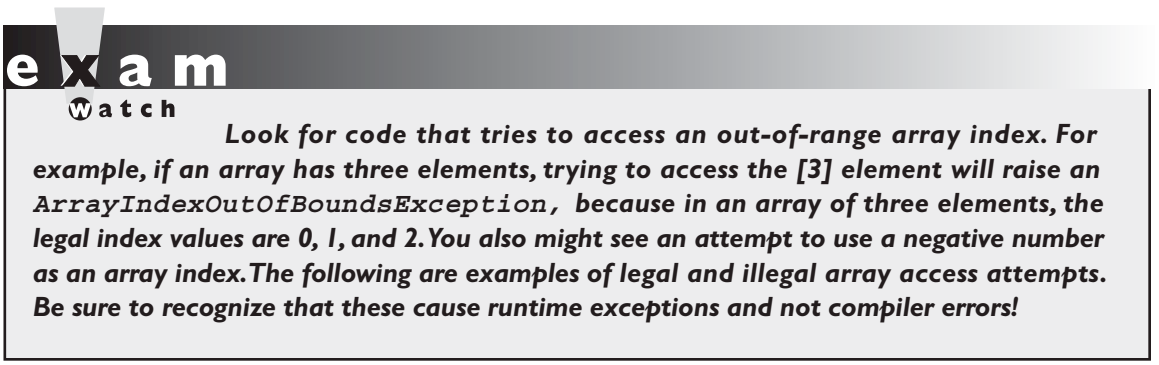
The individual elements in the array can be accessed with an index number. The index number always begins with zero, so for an array of ten objects the index numbers will run from 0 through 9. Suppose we create an array of three Animals as follows:

```
Animal [] pets = new Animal[3];
```

We have one array object on the heap, with three `null` references of type Animal, but we don't have any Animal objects. The next step is to create some Animal objects and assign them to index positions in the array referenced by `pets`:

```
pets[0] = new Animal();
pets[1] = new Animal();
pets[2] = new Animal();
```

This code puts three new Animal objects on the heap and assigns them to the three index positions (elements) in the `pets` array.



Look for code that tries to access an out-of-range array index. For example, if an array has three elements, trying to access the [3] element will raise an `ArrayIndexOutOfBoundsException`, because in an array of three elements, the legal index values are 0, 1, and 2. You also might see an attempt to use a negative number as an array index. The following are examples of legal and illegal array access attempts. Be sure to recognize that these cause runtime exceptions and not compiler errors!

exam Watch

Nearly all of the exam questions list both runtime exception and compiler error as possible answers.

```
int[] x = new int[5];
x[4] = 2; // OK, the last element is at index 4
x[5] = 3;  // Runtime  exception. There is no element at index
5!

int[] z = new int[2];
int y = -3;
z[y] = 4; // Runtime exception. y is a negative number
```

These can be hard to spot in a complex loop, but that's where you're most likely to see array index problems in exam questions.

A two-dimensional array (an array of arrays) can be initialized as follows:

```
int[][] scores = new int[3][];
// Declare and create an array holding three references
// to int arrays

scores[0] = new int[4];
// the first element in the scores array is an int array
// of four int elements

scores[1] = new int[6];
// the second element in the scores array is an int array
// of six int elements

scores[2] = new int[1];
// the third element in the scores array is an int array
// of one int element
```

Initializing Elements in a Loop

Array objects have a single public variable, `length` that gives you the number of elements in the array. The last index value, then, is always one less than the `length`. For example, if the `length` of an array is 4, the index values are from 0 through 3. Often, you'll see array elements initialized in a loop as follows:

```
Dog[] myDogs = new Dog[6]; // creates an array of 6
                           // Dog references
for(int x = 0; x < myDogs.length; x++) {
    myDogs[x] = new Dog(); // assign a new Dog to the
                           // index position x
}
```

The `length` variable tells us how many elements the array holds, but it does not tell us whether those elements have been initialized.

Declaring, Constructing, and Initializing on One Line

You can use two different array-specific syntax shortcuts to both initialize (put explicit values into an array's elements) and construct (instantiate the array object itself) in a single statement. The first is used to declare, create, and initialize in one statement as follows:

```
1.  int x = 9;
2.  int[] dots = {6,x,8};
```

Line 2 in the preceding code does four things:

- Declares an `int` array reference variable named `dots`.
- Creates an `int` array with a length of three (three elements).
- Populates the array's elements with the values 6, 9, and 8.
- Assigns the new array object to the reference variable `dots`.

The size (length of the array) is determined by the number of comma-separated items between the curly braces. The code is functionally equivalent to the following longer code:

```
int[] dots;
dots = new int[3];
int x = 9;
dots[0] = 6;
dots[1] = x;
dots[2] = 8;
```

This begs the question, "Why would anyone use the longer way?" One reason come to mind. You might not know—at the time you create the array—the values that will be assigned to the array's elements. This array shortcut alone (combined with the stimulating prose) is worth the price of this book.

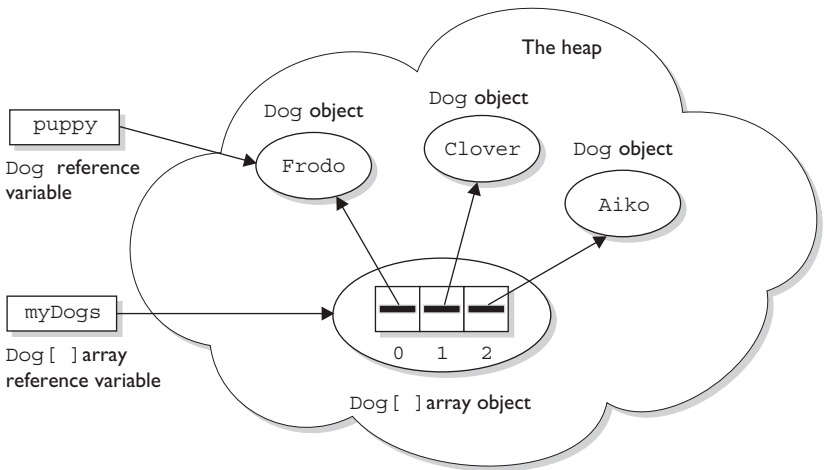
With object references rather than primitives, it works exactly the same way:

```
Dog puppy = new Dog("Frodo");
Dog[] myDogs = {puppy, new Dog("Clover"), new Dog("Aiko")};
```

The preceding code creates one Dog array, referenced by the variable `myDogs`, with a length of three elements. It assigns a previously created Dog object (assigned to the reference variable `puppy`) to the first element in the array. It also creates two new Dog objects (`Clover` and `Aiko`), and adds them to the last two Dog reference variable elements in the myDogs array. Figure 3-4 shows the result.

FIGURE 3-4

Declaring, constructing, and initializing an array of objects



Picture demonstrates the result of the following code:

```
Dog puppy = new Dog ("Frodo");
Dog[ ] myDogs = {puppy, new Dog("Clover"), new Dog("Aiko")};
```

Four objects are created:

1 `Dog` object referenced by `puppy` and by `myDogs[0]`
1 `Dog[ ]` array referenced by `myDogs`
2 `Dog` object referenced by `myDogs[1]` and `myDogs[2]`

You can also use the shortcut syntax with multidimensional arrays, as follows:

```
int[][] scores = {{5,2,4,7}, {9,2}, {3,4}};
```

The preceding code creates a total of four objects on the heap. First, an array of `int` arrays is constructed (the object that will be assigned to the `scores` reference variable). The `scores` array has a length of three, derived from the number of items (comma-separated) between the outer curly braces. Each of the three elements in the `scores` array is a reference variable to an `int` array, so the three `int` arrays are constructed and assigned to the three elements in the `scores` array.

The size of each of the three `int` arrays is derived from the number of items within the corresponding inner curly braces. For example, the first array has a length of four, the second array has a length of two, and the third array has a length of two. So far, we have four objects: one array of `int` arrays (each element is a reference to an `int` array), and three `int` arrays (each element in the three `int` arrays is an `int` value). Finally, the three `int` arrays are initialized with the actual `int` values within the inner curly braces. Thus, the first `int` array contains the values 5, 2, 4, and 7. The following code shows the values of some of the elements in this two-dimensional array:

```
scores[0] // an array of four ints
scores[1] // an array of 2 ints
scores[2] // an array of 2 ints
scores[0][1] // the int value 2
scores[2][1] // the int value 4
```

Figure 3-5 shows the result of declaring, constructing, and initializing a two-dimensional array in one statement.

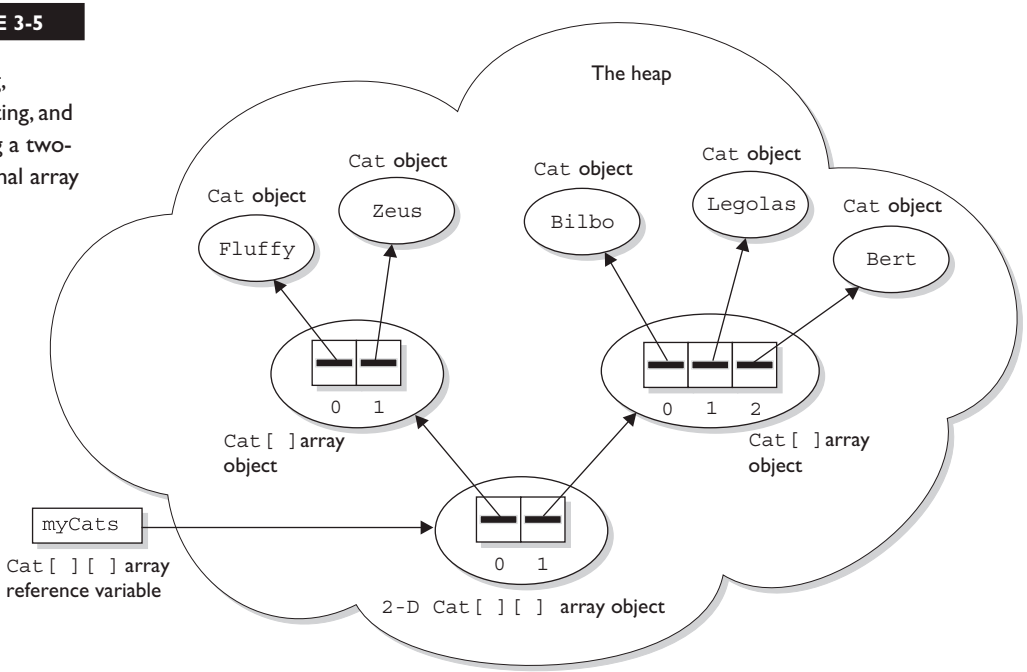
Constructing and Initializing an Anonymous Array

The second shortcut is called "anonymous array creation" and can be used to construct and initialize an array, and then assign the array to a previously declared array reference variable:

```
int[] testScores;
testScores = new int[] {4,7,2};
```

FIGURE 3-5

Declaring, constructing, and initializing a two-dimensional array



Picture demonstrates the result of the following code:

```
Cat[ ][ ] myCats = {{new Cat("Fluffy"), new Cat("Zeus")},
{new Cat("Bilbo"), new Cat("Legolas"), new Cat("Bert")}}
```

Eight objects are created:
1 `2-D Cat[ ][ ]` array object
2 `Cat[ ]` array object
5 `Cat` object

The preceding code creates a new `int` array with three elements, initializes the three elements with the values 4, 7, and 2, and then assigns the new array to the previously declared `int` array reference variable `testScores`. We call this anonymous array creation because with this syntax you don't even need to assign the new array to anything. Maybe you're wondering, "What good is an array if you don't assign it to a reference variable?" You can use it to create a just-in-time array to use, for example, as an argument to a method that takes an array parameter. The following code demonstrates a just-in-time array argument:

```
public class Foof {
  void takesAnArray(int []  someArray) {
    // use the array parameter
  }
  public static void main (String [] args) {
    Foof f = new Foof();
    f.takesAnArray(new int[] {7,7,8,2,5}); // we need an array
                                           // argument
  }
}
```

exam Watch

Remember that you do not specify a size when using anonymous array creation syntax. The size is derived from the number of items (comma-separated) between the curly braces. Pay very close attention to the array syntax used in exam questions (and there will be a lot of them). You might see syntax such as

```
new Object[3] {null, new Object(), new Object()};
  // not legal;size must not be specified
```

Legal Array Element Assignments

What can you put in a particular array? For the exam, you need to know that arrays can have only one declared type (`int []`, `Dog[]`, `String []`, and so on), but that doesn't necessarily mean that only objects or primitives of the declared type can be assigned to the array elements. And what about the array reference itself? What kind of array object can be assigned to a particular array reference? For the exam, you'll need to know the answers to all of these questions. And, as if by magic, we're actually covering those very same topics in the following sections. Pay attention.

Arrays of Primitives

Primitive arrays can accept any value that can be promoted implicitly to the declared type of the array. For example, an int array can hold any value that can fit into a 32-bit `int` variable. Thus, the following code is legal:

```
int[] weightList = new int[5];
byte b = 4;
char c = 'c';
short s = 7;
weightList[0] = b;  // OK, byte is smaller than int
weightList[1] = c;  // OK, char is smaller than int
weightList[2] = s;  // OK, short is smaller than int
```

Arrays of Object References

If the declared array type is a class, you can put objects of any subclass of the declared type into the array. For example, if Subaru is a subclass of Car, you can put both Subaru objects and Car objects into an array of type Car as follows:

```
class Car {}
class Subaru extends Car {}
class Ferrari extends Car {}
...
Car [] myCars = {new Subaru(), new Car(), new Ferrari()};
```

It helps to remember that the elements in a Car array are nothing more than Car reference variables. So anything that can be assigned to a Car reference variable can be legally assigned to a Car array element.

If the array is declared as an interface type, the array elements can refer to any instance of any class that implements the declared interface. The following code demonstrates the use of an interface as an array type:

```
interface Sporty {
  void beSporty();
}

class Ferrari extends Car implements Sporty {
  public void beSporty() {
    // implement cool sporty method in a Ferrari-specific way
  }
}
class RacingFlats extends AthleticShoe implements Sporty {
  public void beSporty() {
    // implement cool sporty method in a RacingShoe-specific way
  }
```

```
}
class GolfClub { }
class TestSportyThings {
  public static void main (String [] args) {
    Sporty[] sportyThings = new Sporty [3];
    sportyThings[0] = new Ferrari();      // OK, Ferrari
                                          // implements Sporty
    sportyThings[1] = new RacingFlats();  // OK, RacingFlats
                                          // implements Sporty
    sportyThings[2] = new GolfClub();

        // Not OK; GolfClub does not implement Sporty
        // I don't care what anyone says
  }
}
```

The bottom line is this: any object that passes the "IS-A" test for the declared array type can be assigned to an element of that array.

Array Reference Assignments for One-Dimensional Arrays

For the exam, you need to recognize legal and illegal assignments for array reference variables. We're not talking about references in the array (in other words, array elements), but rather references to the array object. For example, if you declare an `int` array, the reference variable you declared can be reassigned to any `int` array (of any size), but cannot be reassigned to anything that is not an `int` array, including an `int` value. Remember, all arrays are objects, so an `int` array reference cannot refer to an `int` primitive. The following code demonstrates legal and illegal assignments for primitive arrays:

```
int[] splats;
int[] dats = new int[4];
char[] letters = new char[5];
splats = dats; // OK, dats refers to an int array
splats = letters; // NOT OK, letters refers to a char array
```

It's tempting to assume that because a variable of type `byte`, `short`, or `char` can be explicitly promoted and assigned to an `int`, an array of any of those types could be assigned to an `int` array. You can't do that in Java, but it would be just like those cruel, heartless (but otherwise attractive) exam developers to put tricky array assignment questions in the exam.

Arrays that hold object references, as opposed to primitives, aren't as restrictive. Just as you can put a Honda object in a Car array (because Honda extends Car), you can assign an array of type Honda to a Car array reference variable as follows:

```
Car[] cars;
Honda[] cuteCars = new Honda[5];
cars = cuteCars; // OK because Honda is a type of Car
Beer[] beers = new Beer [99];
cars = beers; // NOT OK, Beer is not a type of Car
```

Apply the IS-A test to help sort the legal from the illegal. Honda IS-A Car, so a Honda array can be assigned to a Car array. Beer IS-A Car is not true; Beer does not extend Car (plus it doesn't make sense, unless you've already had too much of it).

exam Watch

You cannot reverse the legal assignments. A Car array cannot be assigned to a Honda array. A Car is not necessarily a Honda, so if you've declared a Honda array, it might blow up if you assigned a Car array to the Honda reference variable. Think about it: a Car array could hold a reference to a Ferrari, so someone who thinks they have an array of Hondas could suddenly find themselves with a Ferrari. Remember that the IS-A test can be checked in code using the `instanceof` ***operator.***

The rules for array assignment apply to interfaces as well as classes. An array declared as an interface type can reference an array of any type that implements the interface. Remember, any object from a class implementing a particular interface will pass the IS-A (`instanceof`) test for that interface. For example, if Box implements Foldable, the following is legal:

```
Foldable[]  foldingThings;
Box[] boxThings = new Box[3];
foldingThings = boxThings;
// OK, Box implements Foldable, so Box IS-A Foldable
```

Array Reference Assignments for Multidimensional Arrays

When you assign an array to a previously declared array reference, the array you're assigning must be the same dimension as the reference you're assigning it to. For

example, a two-dimensional array of `int` arrays cannot be assigned to a regular `int` array reference, as follows:

```
int[] blots;
int[][] squeegees = new int[3][];
blots = squeegees;          // NOT OK, squeegees is a
                            // two-d array of int arrays
int[] blocks = new int[6];
blots = blocks;             // OK, blocks is an int array
```

Pay particular attention to array assignments using different dimensions. You might, for example, be asked if it's legal to assign an `int` array to the first element in an array of `int` arrays, as follows:

```
int[][]  books = new int[3][];
int[] numbers = new int[6];
int aNumber = 7;
books[0] = aNumber;     // NO, expecting an int array not an int
books[0] = numbers;     // OK, numbers is an int array
```

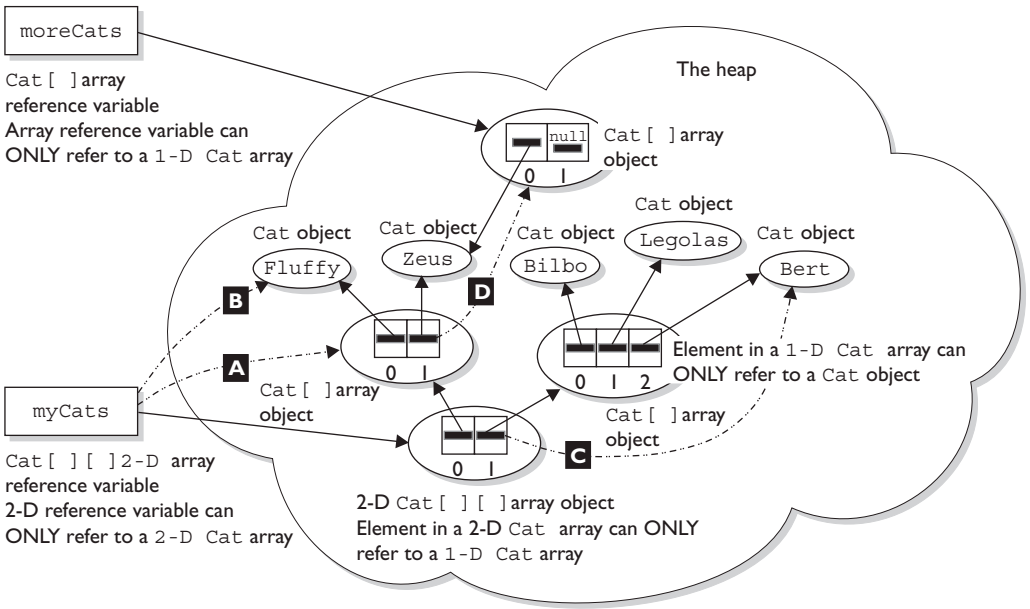
Figure 3-6 shows an example of legal and illegal assignments for references to an array.

Initialization Blocks

We've talked about two places in a class where you can put code that performs operations: methods and constructors. Initialization blocks are the third place in a Java program where operations can be performed. Initialization blocks run when the class is first loaded (a static initialization block) or when an instance is created (an instance initialization block). Let's look at an example:

```
class SmallInit {
  static int x;
  int y;

  static { x = 7 ; }        // static init block
  { y = 8; }                // instance init block
}
```

FIGURE 3-6 Legal and illegal array assignments



Illegal Array Reference Assignments

A `myCats = myCats[0];`
`// Can't assign a 1-D array to a 2-D array reference`

B `myCats = myCats[0][0];`
`// Can't assign a nonarray object to a 2-D array reference`

C `myCats[1] = myCats[1][2];`
`// Can't assign a nonarray object to a 1-D array reference`

D `myCats[0][1] = moreCats;`
`// Can't assign an array object to a nonarray reference`
`//myCats[0][1] can only refer to a Cat object`

KEY

Legal

Illegal

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, 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 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 `ExceptionInInitializationError`. 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) { }
}
```

which produces something like:

```
Exception in thread "main" java.lang.ExceptionInInitializerError
Caused by: java.lang.ArrayIndexOutOfBoundsException: 4
        at InitError.<clinit>(InitError.java:3)
```

exam Watch

By convention, init blocks usually appear near the top of the class file, somewhere around the constructors. However, this is the SCJP 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

Using Wrapper Classes and Boxing (Exam Objective 3.1)

3.1 Develop code that uses the primitive wrapper classes (such as Boolean, Character, Double, Integer, etc.), and/or autoboxing & unboxing. Discuss the differences between the String, StringBuilder, and StringBuffer classes.

The wrapper classes in the Java API serve two primary purposes:

- To provide a mechanism to "wrap" primitive values in an object so that the primitives can be included in activities reserved for objects, like being added to Collections, or returned from a method with an object return value. Note: With Java 5's addition of autoboxing (and unboxing), which we'll get to in a few pages, many of the wrapping operations that programmers used to do manually are now handled automatically.
- To provide an assortment of utility functions for primitives. Most of these functions are related to various conversions: converting primitives to and from String objects, and converting primitives and String objects to and from different bases (or radix), such as binary, octal, and hexadecimal.

An Overview of the Wrapper Classes

There is a wrapper class for every primitive in Java. For instance, the wrapper class for `int` is Integer, the class for `float` is Float, and so on. Remember that the primitive name is simply the lowercase name of the wrapper except for `char`, which maps to Character, and `int`, which maps to Integer. Table 3-2 lists the wrapper classes in the Java API.

TABLE 3-2 Wrapper Classes and Their Constructor Arguments

Primitive	Wrapper Class	Constructor Arguments
`boolean`	`Boolean`	`boolean` or `String`
`byte`	`Byte`	`byte` or `String`
`char`	`Character`	`char`
`double`	`Double`	`double` or `String`
`float`	`Float`	`float`, `double`, or `String`
`int`	`Integer`	`int` or `String`
`long`	`Long`	`long` or `String`
`short`	`Short`	`short` or `String`

Creating Wrapper Objects

For the exam you need to understand the three most common approaches for creating wrapper objects. Some approaches take a String representation of a primitive as an argument. Those that take a String throw NumberFormatException if the String provided cannot be parsed into the appropriate primitive. For example "two" can't be parsed into "2". *Wrapper objects are immutable*. Once they have been given a value, that value cannot be changed. We'll talk more about wrapper immutability when we discuss boxing in a few pages.

The Wrapper Constructors

All of the wrapper classes except Character provide two constructors: one that takes a primitive of the type being constructed, and one that takes a String representation of the type being constructed—for example,

```
Integer i1 = new Integer(42);
Integer i2 = new Integer("42");
```

or

```
Float f1 = new Float(3.14f);
Float f2 = new Float("3.14f");
```

The Character class provides only one constructor, which takes a char as an argument—for example,

```
Character c1 = new Character('c');
```

The constructors for the Boolean wrapper take either a `boolean` value `true` or `false`, or a String. If the String's case-insensitive value is "true" the Boolean will be `true`—any other value will equate to `false`. Until Java 5, a Boolean object couldn't be used as an expression in a boolean test—for instance,

```
Boolean b = new Boolean("false");
if (b)      // won't compile, using Java 1.4 or earlier
```

As of Java 5, a Boolean object *can* be used in a boolean test, because the compiler will automatically "unbox" the Boolean to a `boolean`. We'll be focusing on Java 5's autoboxing capabilities in the very next section—so stay tuned!

The valueOf() Methods

The two (well, usually two) static `valueOf()` methods provided in most of the wrapper classes give you another approach to creating wrapper objects. Both methods take a String representation of the appropriate type of primitive as their first argument, the second method (when provided) takes an additional argument, `int radix`, which indicates in what base (for example binary, octal, or hexadecimal) the first argument is represented—for example,

```
Integer i2 = Integer.valueOf("101011", 2);  // converts 101011
                                            // to 43 and
                                            // assigns the value
                                            // 43 to the
                                            // Integer object i2
```

or

```
Float f2 = Float.valueOf("3.14f");   // assigns 3.14 to the
                                     // Float object f2
```

Using Wrapper Conversion Utilities

As we said earlier, a wrapper's second big function is converting stuff. The following methods are the most commonly used, and are the ones you're most likely to see on the test.

xxxValue()

When you need to convert the value of a wrapped numeric to a primitive, use one of the many `xxxValue()` methods. All of the methods in this family are no-arg methods. As you can see by referring to Table 3-3, there are 36 `xxxValue()` methods. Each of the six numeric wrapper classes has six methods, so that any numeric wrapper can be converted to any primitive numeric type—for example,

```
Integer i2 = new Integer(42);  //  make a new wrapper object
byte b = i2.byteValue();       //  convert i2's value to a byte
                               //  primitive
short s = i2.shortValue();     //  another of Integer's xxxValue
                               //  methods
double d = i2.doubleValue();   //  yet another of Integer's
                               //  xxxValue methods
```

or

```
Float f2 = new Float(3.14f);    // make a new wrapper object
short s = f2.shortValue();      // convert f2's value to a short
                                // primitive
System.out.println(s);          // result is 3  (truncated, not
                                // rounded)
```

parseXxx() and valueOf()

The six `parseXxx()` methods (one for each numeric wrapper type) are closely related to the `valueOf()` method that exists in all of the numeric wrapper classes. Both `parseXxx()` and `valueOf()` take a String as an argument, throw a NumberFormatException (a.k.a. NFE) if the String argument is not properly formed, and can convert String objects from different bases (radix), when the underlying primitive type is any of the four integer types. (See Table 3-3.) The difference between the two methods is

- `parseXxx()` returns the named primitive.
- `valueOf()` returns a newly created wrapped object of the type that invoked the method.

Here are some examples of these methods in action:

```
double d4 = Double.parseDouble("3.14");   // convert a String
                                          // to a primitive
System.out.println("d4 = " + d4);         // result is d4 = 3.14

Double d5 = Double.valueOf("3.14");       // create a Double obj
System.out.println(d5 instanceof Double); // result is "true"
```

The next examples involve using the radix argument (in this case binary):

```
long L2 = Long.parseLong("101010", 2);    // binary String to a
                                          // primitive
System.out.println("L2 = " + L2);         // result is: L2 = 42

Long L3 = Long.valueOf("101010", 2);      // binary String to
                                          // Long object
System.out.println("L3 value = " + L3);   // result is:
                                          // L3 value = 42
```

toString()

Class Object, the alpha class, has a `toString()` method. Since we know that all other Java classes inherit from class Object, we also know that all other Java classes have a `toString()` method. The idea of the `toString()` method is to allow you to get some meaningful representation of a given object. For instance, if you have a Collection of various types of objects, you can loop through the Collection and print out some sort of meaningful representation of each object using the `toString()` method, which is guaranteed to be in every class. We'll talk more about `toString()` in the Collections chapter, but for now let's focus on how `toString()` relates to the wrapper classes which, as we know, are marked `final`. All of the wrapper classes have a no-arg, nonstatic, instance version of `toString()`. This method returns a String with the value of the primitive wrapped in the object—for instance,

```
Double d = new Double("3.14");
System.out.println("d = "+ d.toString() ); // result is d = 3.14
```

All of the numeric wrapper classes provide an overloaded, `static toString()` method that takes a primitive numeric of the appropriate type (`Double.toString()` takes a `double`, `Long.toString()` takes a `long`, and so on) and, of course, returns a String:

```
String d = Double.toString(3.14);      // d = "3.14"
```

Finally, Integer and Long provide a third toString() method. It's static, its first argument is the primitive, and its second argument is a radix. The radix tells the method to take the first argument, which is radix 10 (base 10) by default, and convert it to the radix provided, then return the result as a String—for instance,

```
String s = "hex = "+ Long.toString(254,16); // s = "hex = fe"
```

toXxxString() (Binary, Hexadecimal, Octal)

The Integer and Long wrapper classes let you convert numbers in base 10 to other bases. These conversion methods, `toXxxString()`, take an `int` or `long`, and return a String representation of the converted number, for example,

```
String s3 = Integer.toHexString(254);  // convert 254 to hex
System.out.println("254 is " + s3);    // result: "254 is fe"

String s4 = Long.toOctalString(254); // convert 254 to octal
System.out.print("254(oct) ="+ s4);  // result: "254(oct) =376"
```

Studying Table 3-3 is the single best way to prepare for this section of the test. If you can keep the differences between `xxxValue()`, `parseXxx()`, and `valueOf()` straight, you should do well on this part of the exam.

TABLE 3-3 Common Wrapper Conversion Methods

Method s = static n = NFE exception	Boolean	Byte	Character	Double	Float	Integer	Long	Short
byteValue		x		x	x	x	x	x
doubleValue		x		x	x	x	x	x
floatValue		x		x	x	x	x	x
intValue		x		x	x	x	x	x
longValue		x		x	x	x	x	x
shortValue		x		x	x	x	x	x
parseXxx s,n		x		x	x	x	x	x
parseXxx s,n (with radix)		x				x	x	x
valueOf s,n	s	x		x	x	x	x	x
valueOf s,n (with radix)		x				x	x	x
toString	x	x	x	x	x	x	x	x
toString s (primitive)	x	x	x	x	x	x	x	x
toString s (primitive, radix)						x	x	

In summary, the essential method signatures for Wrapper conversion methods are

primitive xxxValue() - to convert a Wrapper to a primitive

primitive parseXxx(String) - to convert a String to a primitive

Wrapper valueOf(String) - to convert a String to a Wrapper

Autoboxing

New to Java 5 is a feature known variously as: autoboxing, auto-unboxing, boxing, and unboxing. We'll stick with the terms boxing and unboxing. Boxing and unboxing make using wrapper classes more convenient. In the old, pre–Java 5 days, if you wanted to make a wrapper, unwrap it, use it, and then rewrap it, you might do something like this:

```
Integer y = new Integer(567);    // make it
int x = y.intValue();            // unwrap it
x++;                             // use it
y = new Integer(x);              // re-wrap it
System.out.println("y = " + y);  // print it
```

Now, with new and improved Java 5 you can say

```
Integer y = new Integer(567);    // make it
y++;                             // unwrap it, increment it,
                                 // rewrap it
System.out.println("y = " + y);  // print it
```

Both examples produce the output:

```
y = 568
```

And yes, you read that correctly. The code appears to be using the postincrement operator on an object reference variable! But it's simply a convenience. Behind the scenes, the compiler does the unboxing and reassignment for you. Earlier we mentioned that wrapper objects are immutable... this example appears to contradict that statement. It sure looks like `y`'s value changed from 567 to 568. What actually happened, is that a second wrapper object was created and its value was set to 568. If only we could access that first wrapper object, we could prove it...

Let's try this:

```
Integer y = 567;                 // make a wrapper
Integer x = y;                   // assign a second ref
                                 // var to THE wrapper

System.out.println(y==x);        // verify that they refer
                                 // to the same object
```

```
y++;                                    // unwrap, use, "rewrap"
System.out.println(x + " " + y);        // print values

System.out.println(y==x);               // verify that they refer
                                        // to different objects
```

Which produces the output:

```
true
567 568
false
```

So, under the covers, when the compiler got to the line `y++;` it had to substitute something like this:

```
int x2 = y.intValue();              // unwrap it
x2++;                               // use it
y = new Integer(x2);                // re-wrap it
```

Just as we suspected, there's gotta be a call to `new` in there somewhere.

Boxing, ==, and equals()

We just used == to do a little exploration of wrappers. Let's take a more thorough look at how wrappers work with ==, !=, and `equals()`. We'll talk a lot more about the `equals()` method in later chapters. For now all we have to know is that the intention of the `equals()` method is to determine whether two instances of a given class are "meaningfully equivalent." This definition is intentionally subjective; it's up to the creator of the class to determine what "equivalent" means for objects of the class in question. The API developers decided that for all the wrapper classes, two objects are equal if they are of the same type and have the same value. It shouldn't be surprising that

```
Integer i1 = 1000;
Integer i2 = 1000;
if(i1 != i2) System.out.println("different objects");
if(i1.equals(i2)) System.out.println("meaningfully equal");
```

Produces the output:

```
different objects
meaningfully equal
```

It's just two wrapper objects that happen to have the same value. Because they have the same `int` value, the `equals()` method considers them to be "meaningfully equivalent", and therefore returns `true`. How about this one:

```
Integer i3 = 10;
Integer i4 = 10;
if(i3 == i4) System.out.println("same object");
if(i3.equals(i4)) System.out.println("meaningfully equal");
```

This example produces the output:

```
same object
meaningfully equal
```

Yikes! The `equals()` method seems to be working, but what happened with == and != ? Why is != telling us that `i1` and `i2` are different objects, when == is saying that `i3` and `i4` are the same object? In order to save memory, two instances of the following wrapper objects (created through boxing), will always be == when their primitive values are the same:

- `Boolean`
- `Byte`
- `Character` from \u0000 to \u007f (7f is 127 in decimal)
- `Short` and `Integer` from -128 to 127

Note: When == is used to compare a primitive to a wrapper, the wrapper will be unwrapped and the comparison will be primitive to primitive.

Where Boxing Can Be Used

As we discussed earlier, it's very common to use wrappers in conjunction with collections. Any time you want your collection to hold objects and primitives, you'll want to use wrappers to make those primitives collection-compatible. The general rule is that boxing and unboxing work wherever you can normally use a primitive or a wrapped object. The following code demonstrates some legal ways to use boxing:

```
class UseBoxing {
  public static void main(String [] args) {
    UseBoxing u = new UseBoxing();
    u.go(5);
  }
  boolean go(Integer i) {       // boxes the int it was passed
    Boolean ifSo = true;        // boxes the literal
    Short s = 300;              // boxes the primitive
```

```
    if(ifSo) {                  // unboxing
      System.out.println(++s);  // unboxes, increments, reboxes
    }
    return !ifSo;               // unboxes, returns the inverse
  }
}
```

exam Watch

Remember, wrapper reference variables can be null. That means that you have to watch out for code that appears to be doing safe primitive operations, but that could throw a `NullPointerException`***:***

```
class Boxing2 {
  static Integer x;
  public static void main(String [] args) {
    doStuff(x);
  }
  static void doStuff(int z) {
    int z2 = 5;
    System.out.println(z2 + z);
} }
```

This code compiles fine, but the JVM throws a `NullPointerException` ***when it attempts to invoke*** `doStuff(x)`***, because*** `x` ***doesn't refer to an*** `Integer` ***object, so there's no value to unbox.***

CERTIFICATION OBJECTIVE

Overloading (Exam Objectives 1.5 and 5.4)

1.5 Given a code example, determine if a method is correctly overriding or overloading another method, and identify legal return values (including covariant returns), for the method.

5.4 Given a scenario, develop code that declares and/or invokes overridden or overloaded methods…

Overloading Made Hard—Method Matching

Although we covered some rules for overloading methods in Chapter 2, in this chapter we've added some new tools to our Java toolkit. In this section we're going to take a look at three factors that can make overloading a little tricky:

- Widening
- Autoboxing
- Var-args

When a class has overloaded methods, one of the compiler's jobs is to determine which method to use whenever it finds an invocation for the overloaded method. Let's look at an example that doesn't use any new Java 5 features:

```
class EasyOver {
  static void go(int x) { System.out.print("int "); }
  static void go(long x) { System.out.print("long "); }
  static void go(double x) { System.out.print("double "); }

  public static void main(String [] args) {
    byte b = 5;
    short s = 5;
    long l = 5;
    float f = 5.0f;

    go(b);
    go(s);
    go(l);
    go(f);
  }
}
```

Which produces the output:

```
int int long double
```

This probably isn't much of a surprise; the calls that use `byte` and the `short` arguments are implicitly widened to match the version of the `go()` method that takes an `int`. Of course, the call with the `long` uses the long version of `go()`, and finally, the call that uses a `float` is matched to the method that takes a `double`.

In every case, when an exact match isn't found, the JVM uses the method with the smallest argument that is wider than the parameter.

You can verify for yourself that if there is only one version of the `go()` method, and it takes a `double`, it will be used to match all four invocations of `go()`.

Overloading with Boxing and Var-args

Now let's take our last example, and add *boxing* into the mix:

```
class AddBoxing {
  static void go(Integer x) { System.out.println("Integer"); }
  static void go(long x) { System.out.println("long"); }

  public static void main(String [] args) {
    int i = 5;
    go(i);           // which go() will be invoked?
  }
}
```

As we've seen earlier, if the only version of the `go()` method was one that took an Integer, then Java 5's boxing capability would allow the invocation of `go()` to succeed. Likewise, if only the `long` version existed, the compiler would use it to handle the `go()` invocation. The question is, given that both methods exist, which one will be used? In other words, does the compiler think that widening a primitive parameter is more desirable than performing an autoboxing operation? The answer is that the compiler will choose widening over boxing, so the output will be

```
long
```

Java 5's designers decided that the most important rule should be that preexisting code should function the way it used to, so since widening capability already existed, a method that is invoked via widening shouldn't lose out to a newly created method that relies on boxing. Based on that rule, try to predict the output of the following:

```
class AddVarargs {
  static void go(int x, int y) { System.out.println("int,int");}
  static void go(byte... x) { System.out.println("byte... "); }
  public static void main(String[] args) {
    byte b = 5;
    go(b,b);         // which go() will be invoked?
  }
}
```

As you probably guessed, the output is

```
int,int
```

Because, once again, even though each invocation will require some sort of conversion, the compiler will choose the older style before it chooses the newer style, keeping existing code more robust. So far we've seen that

- Widening beats boxing
- Widening beats var-args

At this point, inquiring minds want to know, does boxing beat var-args?

```
class BoxOrVararg {
  static void go(Byte x, Byte y)
                { System.out.println("Byte, Byte"); }
  static void go(byte... x) { System.out.println("byte... "); }

  public static void main(String [] args) {
    byte b = 5;
    go(b,b);          // which go() will be invoked?
  }
}
```

As it turns out, the output is

```
Byte, Byte
```

A good way to remember this rule is to notice that the var-args method is "looser" than the other method, in that it could handle invocations with any number of `byte` parameters. A var-args method is more like a catch-all method, in terms of what invocations it can handle, and as we'll see in Chapter 5, it makes most sense for catch-all capabilities to be used as a *last resort*.

Widening Reference Variables

We've seen that it's legal to widen a primitive. Can you widen a reference variable, and if so, what would it mean? Let's think back to our favorite polymorphic assignment:

```
Animal a = new Dog();
```

Along the same lines, an invocation might be:

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

class Dog3 extends Animal {
  public static void main(String[] args) {
    Dog3 d = new Dog3();
    d.go(d);              // is this legal ?
  }
  void go(Animal a) { }
}
```

No problem! The `go()` method needs an Animal, and `Dog3` IS-A Animal. (Remember, the `go()` method thinks it's getting an Animal object, so it will only ask it to do Animal things, which of course anything that inherits from Animal can do.) So, in this case, the compiler widens the `Dog3` reference to an Animal, and the invocation succeeds. The key point here is that reference widening depends on inheritance, in other words the IS-A test. Because of this, it's not legal to widen from one wrapper class to another, because the wrapper classes are peers to one another. For instance, it's NOT valid to say that Short IS-A Integer.

exam Watch

It's tempting to think that you might be able to widen an Integer wrapper to a Long wrapper, but the following will NOT compile:

```
class Dog4 {
  public static void main(String [] args) {
    Dog4 d = new Dog4();
    d.test(new Integer(5));  // can't widen an Integer
                             // to a Long
  }
  void test(Long x) { }
}
```

Remember, none of the wrapper classes will widen from one to another! Bytes won't widen to Shorts, Shorts won't widen to Longs, etc.

Overloading When Combining Widening and Boxing

We've looked at the rules that apply when the compiler can match an invocation to a method by performing a single conversion. Now let's take a look at what happens when more than one conversion is required. In this case the compiler will have to widen and then autobox the parameter for a match to be made:

```
class WidenAndBox {
  static void go(Long x) { System.out.println("Long"); }

  public static void main(String [] args) {
    byte b = 5;
    go(b);           // must widen then box - illegal
  }
}
```

This is just too much for the compiler:

```
WidenAndBox.java:6: go(java.lang.Long) in WidenAndBox cannot be
applied to (byte)
```

Strangely enough, it IS possible for the compiler to perform a boxing operation followed by a widening operation in order to match an invocation to a method. This one might blow your mind:

```
class BoxAndWiden {
  static void go(Object o) {
    Byte b2 = (Byte) o;       // ok - it's a Byte object
    System.out.println(b2);
  }

  public static void main(String [] args) {
    byte b = 5;
    go(b);       // can this byte turn into an Object ?
  }
}
```

This compiles (!), and produces the output:

```
5
```

Wow! Here's what happened under the covers when the compiler, then the JVM, got to the line that invokes the `go()` method:

1. The `byte b` was boxed to a Byte.
2. The Byte reference was widened to an Object (since Byte extends Object).
3. The `go()` method got an Object reference that actually refers to a Byte object.
4. The `go()` method cast the Object reference back to a Byte reference (remember, there was never an object of type Object in this scenario, only an object of type Byte!).
5. The `go()` method printed the Byte's value.

Why didn't the compiler try to use the box-then-widen logic when it tried to deal with the WidenAndBox class? Think about it…if it tried to box first, the byte would have been converted to a Byte. Now we're back to trying to widen a Byte to a Long, and of course, the IS-A test fails.

Overloading in Combination with Var-args

What happens when we attempt to combine var-args with either widening or boxing in a method-matching scenario? Let's take a look:

```
class Vararg {
  static void wide_vararg(long... x)
              { System.out.println("long..."); }
  static void box_vararg(Integer... x)
              { System.out.println("Integer..."); }
  public static void main(String [] args) {
    int i = 5;
    wide_vararg(i,i);    // needs to widen and use var-args
    box_vararg(i,i);     // needs to box and use var-args
  }
}
```

This compiles and produces:

```
long...
Integer...
```

As we can see, you can successfully combine var-args with either widening or boxing. Here's a review of the rules for overloading methods using widening, boxing, and var-args:

- Primitive widening uses the "smallest" method argument possible.
- Used individually, boxing and var-args are compatible with overloading.
- You CANNOT widen from one wrapper type to another. (IS-A fails.)
- You CANNOT widen and then box. (An `int` can't become a Long.)
- You can box and then widen. (An `int` can become an Object, via Integer.)
- You can combine var-args with either widening or boxing.

There are more tricky aspects to overloading, but other than a few rules concerning generics (which we'll cover in Chapter 7), this is all you'll need to know for the exam. Phew!

CERTIFICATION OBJECTIVE

Garbage Collection (Exam Objective 7.4)

7.4 Given a code example, recognize the point at which an object becomes eligible for garbage collection, and determine what is and is not guaranteed by the garbage collection system, and recognize the behaviors of the Object finalize() method.

Overview of Memory Management and Garbage Collection

This is the section you've been waiting for! It's finally time to dig into the wonderful world of memory management and garbage collection.

Memory management is a crucial element in many types of applications. Consider a program that reads in large amounts of data, say from somewhere else on a network, and then writes that data into a database on a hard drive. A typical design would be to read the data into some sort of collection in memory, perform some operations on the data, and then write the data into the database. After the data is written into the database, the collection that stored the data temporarily must be emptied of old data or deleted and recreated before processing the next

batch. This operation might be performed thousands of times, and in languages like C or C++ that do not offer automatic garbage collection, a small flaw in the logic that manually empties or deletes the collection data structures can allow small amounts of memory to be improperly reclaimed or lost. Forever. These small losses are called memory leaks, and over many thousands of iterations they can make enough memory inaccessible that programs will eventually crash. Creating code that performs manual memory management cleanly and thoroughly is a nontrivial and complex task, and while estimates vary, it is arguable that manual memory management can double the development effort for a complex program.

Java's garbage collector provides an automatic solution to memory management. In most cases it frees you from having to add any memory management logic to your application. The downside to automatic garbage collection is that you can't completely control when it runs and when it doesn't.

Overview of Java's Garbage Collector

Let's look at what we mean when we talk about garbage collection in the land of Java. From the 30,000 ft. level, garbage collection is the phrase used to describe automatic memory management in Java. Whenever a software program executes (in Java, C, C++, Lisp, Ruby, and so on), it uses memory in several different ways. We're not going to get into Computer Science 101 here, but it's typical for memory to be used to create a stack, a heap, in Java's case constant pools, and method areas. The heap is that part of memory where Java objects live, and it's the one and only part of memory that is in any way involved in the garbage collection process.

A heap is a heap is a heap. For the exam it's important to know that you can call it the heap, you can call it the garbage collectible heap, or you can call it Johnson, but there is one and only one heap.

So, all of garbage collection revolves around making sure that the heap has as much free space as possible. For the purpose of the exam, what this boils down to is deleting any objects that are no longer reachable by the Java program running. We'll talk more about what reachable means, but let's drill this point in. When the garbage collector runs, its purpose is to find and delete objects that cannot be reached. If you think of a Java program as being in a constant cycle of creating the objects it needs (which occupy space on the heap), and then discarding them when they're no longer needed, creating new objects, discarding them, and so on, the missing piece of the puzzle is the garbage collector. When it runs, it looks for those

discarded objects and deletes them from memory so that the cycle of using memory and releasing it can continue. Ah, the great circle of life.

When Does the Garbage Collector Run?

The garbage collector is under the control of the JVM. The JVM decides when to run the garbage collector. From within your Java program you can ask the JVM to run the garbage collector, but there are no guarantees, under any circumstances, that the JVM will comply. Left to its own devices, the JVM will typically run the garbage collector when it senses that memory is running low. Experience indicates that when your Java program makes a request for garbage collection, the JVM will usually grant your request in short order, but there are no guarantees. Just when you think you can count on it, the JVM will decide to ignore your request.

How Does the Garbage Collector Work?

You just can't be sure. You might hear that the garbage collector uses a mark and sweep algorithm, and for any given Java implementation that might be true, but the Java specification doesn't guarantee any particular implementation. You might hear that the garbage collector uses reference counting; once again maybe yes maybe no. The important concept to understand for the exam is when does an object become eligible for garbage collection? To answer this question fully, we have to jump ahead a little bit and talk about threads. (See Chapter 9 for the real scoop on threads.) In a nutshell, every Java program has from one to many threads. Each thread has its own little execution stack. Normally, you (the programmer) cause at least one thread to run in a Java program, the one with the `main()` method at the bottom of the stack. However, as you'll learn in excruciating detail in Chapter 9, there are many really cool reasons to launch additional threads from your initial thread. In addition to having its own little execution stack, each thread has its own lifecycle. For now, all we need to know is that threads can be alive or dead. With this background information, we can now say with stunning clarity and resolve that *an object is eligible for garbage collection when no live thread can access it.* (Note: Due to the vagaries of the String constant pool, the exam focuses its garbage collection questions on non-String objects, and so our garbage collection discussions apply to only non-String objects too.)

Based on that definition, the garbage collector does some magical, unknown operations, and when it discovers an object that can't be reached by any live thread, it will consider that object as eligible for deletion, and it might even delete it at some point. (You guessed it; it also might not ever delete it.) When we talk about reaching an object, we're really talking about having a reachable reference variable

that refers to the object in question. If our Java program has a reference variable that refers to an object, and that reference variable is available to a live thread, then that object is considered reachable. We'll talk more about how objects can become unreachable in the following section.

Can a Java application run out of memory? Yes. The garbage collection system attempts to remove objects from memory when they are not used. However, if you maintain too many live objects (objects referenced from other live objects), the system can run out of memory. Garbage collection cannot ensure that there is enough memory, only that the memory that is available will be managed as efficiently as possible.

Writing Code That Explicitly Makes Objects Eligible for Collection

In the preceding section, we learned the theories behind Java garbage collection. In this section, we show how to make objects eligible for garbage collection using actual code. We also discuss how to attempt to force garbage collection if it is necessary, and how you can perform additional cleanup on objects before they are removed from memory.

Nulling a Reference

As we discussed earlier, an object becomes eligible for garbage collection when there are no more reachable references to it. Obviously, if there are no reachable references, it doesn't matter what happens to the object. For our purposes it is just floating in space, unused, inaccessible, and no longer needed.

The first way to remove a reference to an object is to set the reference variable that refers to the object to `null`. Examine the following code:

```
1. public class GarbageTruck {
2.   public static void main(String [] args) {
3.     StringBuffer sb = new StringBuffer("hello");
4.     System.out.println(sb);
5.     // The StringBuffer object is not eligible for collection
6.     sb = null;
7.     // Now the StringBuffer object is eligible for collection
8.   }
9. }
```

The StringBuffer object with the value `hello` is assigned to the reference variable `sb` in the third line. To make the object eligible (for GC), we set the reference variable `sb` to `null`, which removes the single reference that existed to the

StringBuffer object. Once line 6 has run, our happy little `hello` StringBuffer object is doomed, eligible for garbage collection.

Reassigning a Reference Variable

We can also decouple a reference variable from an object by setting the reference variable to refer to another object. Examine the following code:

```
class GarbageTruck {
   public static void main(String [] args) {
      StringBuffer s1 = new StringBuffer("hello");
      StringBuffer s2 = new StringBuffer("goodbye");
      System.out.println(s1);
      // At this point the StringBuffer "hello" is not eligible
      s1 = s2; // Redirects s1 to refer to the "goodbye" object
      // Now the StringBuffer "hello" is eligible for collection
   }
}
```

Objects that are created in a method also need to be considered. When a method is invoked, any local variables created exist only for the duration of the method. Once the method has returned, the objects created in the method are eligible for garbage collection. There is an obvious exception, however. If an object is returned from the method, its reference might be assigned to a reference variable in the method that called it; hence, it will not be eligible for collection. Examine the following code:

```
import java.util.Date;
public class GarbageFactory {
   public static void main(String [] args) {
      Date d = getDate();
      doComplicatedStuff();
      System.out.println("d = " + d);
   }

   public static Date getDate() {
      Date d2 = new Date();
      StringBuffer now = new StringBuffer(d2.toString());
      System.out.println(now);
      return d2;
   }
}
```

In the preceding example, we created a method called `getDate()` that returns a Date object. This method creates two objects: a Date and a StringBuffer containing the date information. Since the method returns the Date object, it will not be eligible for collection even after the method has completed. The StringBuffer object, though, will be eligible, even though we didn't explicitly set the `now` variable to `null`.

Isolating a Reference

There is another way in which objects can become eligible for garbage collection, even if they still have valid references! We call this scenario "islands of isolation."

A simple example is a class that has an instance variable that is a reference variable to another instance of the same class. Now imagine that two such instances exist and that they refer to each other. If all other references to these two objects are removed, then even though each object still has a valid reference, there will be no way for any live thread to access either object. When the garbage collector runs, it can *usually* discover any such islands of objects and remove them. As you can imagine, such islands can become quite large, theoretically containing hundreds of objects. Examine the following code:

```
public class Island {
  Island i;
  public static void main(String [] args) {

    Island i2 = new Island();
    Island i3 = new Island();
    Island i4 = new Island();

    i2.i = i3;   // i2 refers to i3
    i3.i = i4;   // i3 refers to i4
    i4.i = i2;   // i4 refers to i2

    i2 = null;
    i3 = null;
    i4 = null;

    // do complicated, memory intensive stuff
  }
}
```

When the code reaches `// do complicated`, the three Island objects (previously known as `i2`, `i3`, and `i4`) have instance variables so that they refer to

each other, but their links to the outside world (`i2`, `i3`, and `i4`) have been nulled. These three objects are eligible for garbage collection.

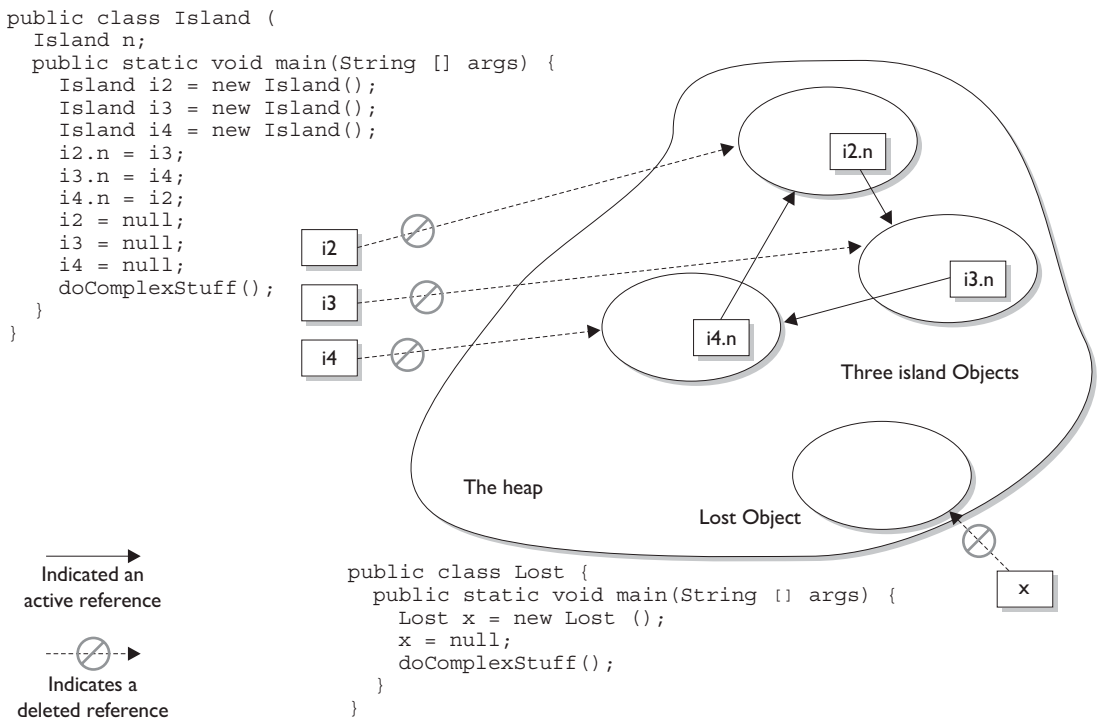
This covers everything you will need to know about making objects eligible for garbage collection. Study Figure 3-7 to reinforce the concepts of objects without references and islands of isolation.

Forcing Garbage Collection

The first thing that should be mentioned here is that, contrary to this section's title, garbage collection cannot be forced. However, Java provides some methods that allow you to request that the JVM perform garbage collection.

Note: As of the Java 6 exam, the topic of using `System.gc()` has been removed from the exam. The garbage collector has evolved to such an advanced state that it's recommended that you never invoke `System.gc()` in your code - leave it to the JVM. We are leaving this section in the book in case you're studying for a version of the exam prior to SCJP 6.

FIGURE 3-7 "Island" objects eligible for garbage collection



In reality, it is possible only to suggest to the JVM that it perform garbage collection. However, there are no guarantees the JVM will actually remove all of the unused objects from memory (even if garbage collection is run). It is essential that you understand this concept for the exam.

The garbage collection routines that Java provides are members of the Runtime class. The Runtime class is a special class that has a single object (a Singleton) for each main program. The Runtime object provides a mechanism for communicating directly with the virtual machine. To get the Runtime instance, you can use the method Runtime.getRuntime(), which returns the Singleton. Once you have the Singleton you can invoke the garbage collector using the `gc()` method. Alternatively, you can call the same method on the System class, which has static methods that can do the work of obtaining the Singleton for you. The simplest way to ask for garbage collection (remember—just a request) is

```
System.gc();
```

Theoretically, after calling `System.gc()`, you will have as much free memory as possible. We say theoretically because this routine does not always work that way. First, your JVM may not have implemented this routine; the language specification allows this routine to do nothing at all. Second, another thread (again, see the Chapter 9) might grab lots of memory right after you run the garbage collector.

This is not to say that `System.gc()` is a useless method—it's much better than nothing. You just can't rely on `System.gc()` to free up enough memory so that you don't have to worry about running out of memory. The Certification Exam is interested in guaranteed behavior, not probable behavior.

Now that we are somewhat familiar with how this works, let's do a little experiment to see if we can see the effects of garbage collection. The following program lets us know how much total memory the JVM has available to it and how much free memory it has. It then creates 10,000 Date objects. After this, it tells us how much memory is left and then calls the garbage collector (which, if it decides to run, should halt the program until all unused objects are removed). The final free memory result should indicate whether it has run. Let's look at the program:

```
1.  import java.util.Date;
2.  public class CheckGC {
3.     public static void main(String [] args) {
4.        Runtime rt = Runtime.getRuntime();
5.        System.out.println("Total JVM memory: "
                              + rt.totalMemory());
```

```
 6.         System.out.println("Before Memory = "
                               + rt.freeMemory());
 7.         Date d = null;
 8.         for(int i = 0;i<10000;i++) {
 9.            d = new Date();
10.            d = null;
11.         }
12.         System.out.println("After Memory = "
                               + rt.freeMemory());
13.         rt.gc();   // an alternate to System.gc()
14.         System.out.println("After GC Memory = "
                               + rt.freeMemory());
15.      }
16.   }
```

Now, let's run the program and check the results:

```
Total JVM memory: 1048568
Before Memory = 703008
After Memory = 458048
After GC Memory = 818272
```

As we can see, the JVM actually did decide to garbage collect (i.e., delete) the eligible objects. In the preceding example, we suggested to the JVM to perform garbage collection with 458,048 bytes of memory remaining, and it honored our request. This program has only one user thread running, so there was nothing else going on when we called `rt.gc()`. Keep in mind that the behavior when `gc()` is called may be different for different JVMs, so there is no guarantee that the unused objects will be removed from memory. About the only thing you can guarantee is that if you are running very low on memory, the garbage collector will run before it throws an OutOfMemoryException.

EXERCISE 3-2

Garbage Collection Experiment

Try changing the CheckGC program by putting lines 13 and 14 inside a loop. You might see that not all memory is released on any given run of the GC.

Cleaning Up Before Garbage Collection—the finalize() Method

Java provides you a mechanism to run some code just before your object is deleted by the garbage collector. This code is located in a method named `finalize()` that all classes inherit from class Object. On the surface this sounds like a great idea; maybe your object opened up some resources, and you'd like to close them before your object is deleted. The problem is that, as you may have gathered by now, you can't count on the garbage collector to ever delete an object. So, any code that you put into your class's overridden `finalize()` method is not guaranteed to run. The `finalize()` method for any given object might run, but you can't count on it, so don't put any essential code into your `finalize()` method. In fact, we recommend that in general you don't override `finalize()` at all.

Tricky Little finalize() Gotcha's

There are a couple of concepts concerning `finalize()` that you need to remember.

- For any given object, `finalize()` will be called only once (at most) by the garbage collector.
- Calling `finalize()` can actually result in saving an object from deletion.

Let's look into these statements a little further. First of all, remember that any code that you can put into a normal method you can put into `finalize()`. For example, in the `finalize()` method you could write code that passes a reference to the object in question back to another object, effectively *uneligiblizing* the object for garbage collection. If at some point later on this same object becomes eligible for garbage collection again, the garbage collector can still process this object and delete it. The garbage collector, however, will remember that, for this object, `finalize()` already ran, and it will not run `finalize()` again.

CERTIFICATION SUMMARY

This was a monster chapter! Don't worry if you find that you have to review some of these topics as you get into later chapters. This chapter has a lot of foundation stuff that will come into play later.

We started the chapter by reviewing the stack and the heap; remember local variables live on the stack, and instance variables live with their objects on the heap.

We reviewed legal literals for primitives and Strings, then we discussed the basics of assigning values to primitives and reference variables, and the rules for casting primitives.

Next we discussed the concept of scope, or "How long will this variable live?" Remember the four basic scopes, in order of lessening life span: static, instance, local, block.

We covered the implications of using uninitialized variables, and the importance of the fact that local variables MUST be assigned a value explicitly. We talked about some of the tricky aspects of assigning one reference variable to another, and some of the finer points of passing variables into methods, including a discussion of "shadowing."

The next topic was creating arrays, where we talked about declaring, constructing, and initializing one-, and multi-dimensional arrays. We talked about anonymous arrays, and arrays of references.

Next we reviewed static and instance initialization blocks, what they look like, and when they are called.

Phew!

We continued the chapter with a discussion of the wrapper classes; used to create immutable objects that hold a primitive, and also used to provide conversion capabilities for primitives: remember valueOf(), xxxValue(), and parseXxx().

Closely related to wrappers, we talked about a big new feature in Java 5, autoboxing. Boxing is a way to automate the use of wrappers, and we covered some of its trickier aspects such as how wrappers work with == and the equals() method.

Having added boxing to our toolbox, it was time to take a closer look at method overloading and how boxing and var-args, in conjunction with widening conversions, make overloading more complicated.

Finally, we dove into garbage collection, Java's automatic memory management feature. We learned that the heap is where objects live and where all the cool garbage collection activity takes place. We learned that in the end, the JVM will perform garbage collection whenever it wants to. You (the programmer) can request a garbage collection run, but you can't force it. We talked about garbage collection only applying to objects that are eligible, and that eligible means "inaccessible from any live thread." Finally, we discussed the rarely useful finalize() method, and what you'll have to know about it for the exam. All in all, one fascinating chapter.

TWO-MINUTE DRILL

Here are some of the key points from this chapter.

Stack and Heap

- ❑ Local variables (method variables) live on the stack.
- ❑ Objects and their instance variables live on the heap.

Literals and Primitive Casting (Objective 1.3)

- ❑ Integer literals can be decimal, octal (e.g. `013`), or hexadecimal (e.g. `0x3d`).
- ❑ Literals for `long`s end in `L` or `l`.
- ❑ Float literals end in `F` or `f`, `double` literals end in a digit or `D` or `d`.
- ❑ The `boolean` literals are `true` and `false`.
- ❑ Literals for `char`s are a single character inside single quotes: `'d'`.

Scope (Objectives 1.3 and 7.6)

- ❑ Scope refers to the lifetime of a variable.
- ❑ There are four basic scopes:
  - ❑ Static variables live basically as long as their class lives.
  - ❑ Instance variables live as long as their object lives.
  - ❑ Local variables live as long as their method is on the stack; however, if their method invokes another method, they are temporarily unavailable.
  - ❑ Block variables (e.g., in a `for` or an `if`) live until the block completes.

Basic Assignments (Objectives 1.3 and 7.6)

- ❑ Literal integers are implicitly ints.
- ❑ Integer expressions always result in an `int`-sized result, never smaller.
- ❑ Floating-point numbers are implicitly doubles (64 bits).
- ❑ Narrowing a primitive truncates the *high order* bits.
- ❑ Compound assignments (e.g. `+=`), perform an automatic cast.
- ❑ A reference variable holds the bits that are used to refer to an object.
- ❑ Reference variables can refer to subclasses of the declared type but not to superclasses.

- When creating a new object, e.g., `Button b = new Button();`, three things happen:
  - Make a reference variable named `b`, of type Button
  - Create a new Button object
  - Assign the Button object to the reference variable `b`

Using a Variable or Array Element That Is Uninitialized and Unassigned (Objectives 1.3 and 7.6)

- When an array of objects is instantiated, objects within the array are not instantiated automatically, but all the references get the default value of null.
- When an array of primitives is instantiated, elements get default values.
- Instance variables are always initialized with a default value.
- Local/automatic/method variables are never given a default value. If you attempt to use one before initializing it, you'll get a compiler error.

Passing Variables into Methods (Objective 7.3)

- Methods can take primitives and/or object references as arguments.
- Method arguments are always copies.
- Method arguments are never actual objects (they can be references to objects).
- A primitive argument is an unattached copy of the original primitive.
- A reference argument is another copy of a reference to the original object.
- Shadowing occurs when two variables with different scopes share the same name. This leads to hard-to-find bugs, and hard-to-answer exam questions.

Array Declaration, Construction, and Initialization (Obj. 1.3)

- Arrays can hold primitives or objects, but the array itself is always an object.
- When you declare an array, the brackets can be left or right of the name.
- It is never legal to include the size of an array in the declaration.
- You must include the size of an array when you construct it (using `new`) unless you are creating an anonymous array.
- Elements in an array of objects are not automatically created, although primitive array elements are given default values.
- You'll get a NullPointerException if you try to use an array element in an object array, if that element does not refer to a real object.

- ❑ Arrays are indexed beginning with zero.
- ❑ An ArrayIndexOutOfBoundsException occurs if you use a bad index value.
- ❑ Arrays have a `length` variable whose value is the number of array elements.
- ❑ The last index you can access is always one less than the length of the array.
- ❑ Multidimensional arrays are just arrays of arrays.
- ❑ The dimensions in a multidimensional array can have different lengths.
- ❑ An array of primitives can accept any value that can be promoted implicitly to the array's declared type;. e.g., a `byte` variable can go in an `int` array.
- ❑ 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.
- ❑ If you assign an array to a previously declared array reference, the array you're assigning must be the same dimension as the reference you're assigning it to.
- ❑ You can assign an array of one type to a previously declared array reference of one of its supertypes. For example, a Honda array can be assigned to an array declared as type Car (assuming Honda extends Car).

Initialization Blocks (Objectives 1.3 and 7.6)

- ❑ Static initialization blocks run once, when the class is first loaded.
- ❑ Instance initialization blocks run every time a new instance is created. They run after all super-constructors and before the constructor's code has run.
- ❑ If multiple init blocks exist in a class, they follow the rules stated above, AND they run in the order in which they appear in the source file.

Using Wrappers (Objective 3.1)

- ❑ The wrapper classes correlate to the primitive types.
- ❑ Wrappers have two main functions:
  - ❑ To wrap primitives so that they can be handled like objects
  - ❑ To provide utility methods for primitives (usually conversions)
- ❑ The three most important method families are
  - ❑ `xxxValue()` Takes no arguments, returns a primitive
  - ❑ `parseXxx()` Takes a String, returns a primitive, throws NFE
  - ❑ `valueOf()` Takes a String, returns a wrapped object, throws NFE

- ❑ Wrapper constructors can take a String or a primitive, except for Character, which can only take a `char`.
- ❑ Radix refers to bases (typically) other than 10; octal is radix = 8, hex = 16.

Boxing (Objective 3.1)

- ❑ As of Java 5, boxing allows you to convert primitives to wrappers or to convert wrappers to primitives automatically.
- ❑ Using == with wrappers created through boxing is tricky; those with the same small values (typically lower than 127), will be ==, larger values will not be ==.

Advanced Overloading (Objectives 1.5 and 5.4)

- ❑ Primitive widening uses the "smallest" method argument possible.
- ❑ Used individually, boxing and var-args are compatible with overloading.
- ❑ You CANNOT widen from one wrapper type to another. (IS-A fails.)
- ❑ You CANNOT widen and then box. (An `int` can't become a Long.)
- ❑ You can box and then widen. (An `int` can become an Object, via an Integer.)
- ❑ You can combine var-args with either widening or boxing.

Garbage Collection (Objective 7.4)

- ❑ In Java, garbage collection (GC) provides automated memory management.
- ❑ The purpose of GC is to delete objects that can't be reached.
- ❑ Only the JVM decides when to run the GC, you can only suggest it.
- ❑ You can't know the GC algorithm for sure.
- ❑ Objects must be considered eligible before they can be garbage collected.
- ❑ An object is eligible when no live thread can reach it.
- ❑ To reach an object, you must have a live, reachable reference to that object.
- ❑ Java applications can run out of memory.
- ❑ Islands of objects can be GCed, even though they refer to each other.
- ❑ Request garbage collection with `System.gc();` (only before the SCJP 6).
- ❑ Class Object has a `finalize()` method.
- ❑ The `finalize()` method is guaranteed to run once and only once before the garbage collector deletes an object.
- ❑ The garbage collector makes no guarantees, `finalize()` may never run.
- ❑ You can uneligibilize an object for GC from within `finalize()`.

SELF TEST

1. Given:

```
class CardBoard {
  Short story = 200;
  CardBoard go(CardBoard cb) {
    cb = null;
    return cb;
  }
  public static void main(String[] args) {
    CardBoard c1 = new CardBoard();
    CardBoard c2 = new CardBoard();
    CardBoard c3 = c1.go(c2);
    c1 = null;
    // do Stuff
} }
```

When `// doStuff` is reached, how many objects are eligible for GC?

A. 0
B. 1
C. 2
D. Compilation fails
E. It is not possible to know
F. An exception is thrown at runtime

2. Given:

```
class Alien {
  String invade(short ships) { return "a few"; }
  String invade(short... ships) { return "many"; }
}
class Defender {
  public static void main(String [] args) {
    System.out.println(new Alien().invade(7));
  }
}
```

What is the result?

A. `many`
B. `a few`
C. Compilation fails
D. The output is not predictable
E. An exception is thrown at runtime

3. Given:

```
1. class Dims {
2.   public static void main(String[] args) {
3.     int[][] a = {{1,2,}, {3,4}};
4.     int[] b = (int[]) a[1];
5.     Object o1 = a;
6.     int[][] a2 = (int[][]) o1;
7.     int[] b2 = (int[]) o1;
8.     System.out.println(b[1]);
9. } }
```

What is the result?

A. 2

B. 4

C. An exception is thrown at runtime

D. Compilation fails due to an error on line 4

E. Compilation fails due to an error on line 5

F. Compilation fails due to an error on line 6

G. Compilation fails due to an error on line 7

4. Given:

```
class Mixer {
  Mixer() { }
  Mixer(Mixer m) { m1 = m; }
  Mixer m1;
  public static void main(String[] args) {
    Mixer m2 = new Mixer();
    Mixer m3 = new Mixer(m2);  m3.go();
    Mixer m4 = m3.m1;          m4.go();
    Mixer m5 = m2.m1;          m5.go();
  }
  void go() { System.out.print("hi "); }
}
```

What is the result?

A. `hi`

B. `hi hi`

C. `hi hi hi`

D. Compilation fails
E. `hi`, followed by an exception
F. `hi hi`, followed by an exception

5. Given:

```
class Fizz {
  int x = 5;
  public static void main(String[] args) {
    final Fizz f1 = new Fizz();
    Fizz f2 = new Fizz();
    Fizz f3 = FizzSwitch(f1,f2);
    System.out.println((f1 == f3) + " " + (f1.x == f3.x));
  }
  static Fizz FizzSwitch(Fizz x, Fizz y) {
    final Fizz z = x;
    z.x = 6;
    return z;
} }
```

What is the result?

A. `true true`
B. `false true`
C. `true false`
D. `false false`
E. Compilation fails
F. An exception is thrown at runtime

6. 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

7. Given:

```
3. public class Bridge {
4.   public enum Suits {
5.     CLUBS(20), DIAMONDS(20), HEARTS(30), SPADES(30),
6.     NOTRUMP(40) { public int getValue(int bid) {
                       return ((bid-1)*30)+40; } };
7.     Suits(int points) {  this.points = points;  }
8.     private int points;
9.     public int getValue(int bid) { return points * bid; }
10.  }
11.  public static void main(String[] args) {
12.    System.out.println(Suits.NOTRUMP.getValue(3));
13.    System.out.println(Suits.SPADES + " " + Suits.SPADES.points);
14.    System.out.println(Suits.values());
15.  }
16. }
```

Which are true? (Choose all that apply.)

A. The output could contain `30`

B. The output could contain `@bf73fa`

C. The output could contain `DIAMONDS`

D. Compilation fails due to an error on line 6

E. Compilation fails due to an error on line 7

F. Compilation fails due to an error on line 8

G. Compilation fails due to an error on line 9

H. Compilation fails due to an error within lines 12 to 14

8. Given:

```
3. public class Ouch {
4.   static int ouch = 7;
5.   public static void main(String[] args) {
6.     new Ouch().go(ouch);
7.     System.out.print(" " + ouch);
8.   }
9.   void go(int ouch) {
10.    ouch++;
11.    for(int ouch = 3; ouch < 6; ouch++)
12.      ;
13.    System.out.print(" " + ouch);
14.  }
15. }
```

What is the result?

A. `5 7`

B. `5 8`

C. `8 7`

D. `8 8`

E. Compilation fails

F. An exception is thrown at runtime

9. Given:

```
3. public class Bertha {
4.   static String s = "";
5.   public static void main(String[] args) {
6.     int x = 4; Boolean y = true; short[] sa = {1,2,3};
7.     doStuff(x, y);
8.     doStuff(x);
9.     doStuff(sa, sa);
10.    System.out.println(s);
11.  }
12.  static void doStuff(Object o)        { s += "1"; }
13.  static void doStuff(Object... o)     { s += "2"; }
14.  static void doStuff(Integer... i)    { s += "3"; }
15.  static void doStuff(Long L)          { s += "4"; }
16. }
```

What is the result?

A. `212`

B. `232`

C. `234`

D. `312`

E. `332`

F. `334`

G. Compilation fails

10. Given:

```
3. class Dozens {
4.   int[] dz = {1,2,3,4,5,6,7,8,9,10,11,12};
5. }
6. public class Eggs {
7.   public static void main(String[] args) {
8.     Dozens [] da = new Dozens[3];
9.     da[0] = new Dozens();
10.    Dozens d = new Dozens();
11.    da[1] = d;
12.    d = null;
13.    da[1] = null;
14.    // do stuff
15.  }
16. }
```

Which two are true about the objects created within `main()`, and eligible for garbage collection when line 14 is reached?

A. Three objects were created

B. Four objects were created

C. Five objects were created

D. Zero objects are eligible for GC

E. One object is eligible for GC

F. Two objects are eligible for GC

G. Three objects are eligible for GC

11. Given:

```
 3. class Beta { }
 4. class Alpha {
 5.   static Beta b1;
 6.   Beta b2;
 7. }
 8. public class Tester {
 9.   public static void main(String[] args) {
10.     Beta b1 = new Beta();     Beta b2 = new Beta();
11.     Alpha a1 = new Alpha();   Alpha a2 = new Alpha();
12.     a1.b1 = b1;
13.     a1.b2 = b1;
14.     a2.b2 = b2;
15.     a1 = null;  b1 = null;  b2 = null;
16.     // do stuff
17.   }
18. }
```

When line 16 is reached, how many objects will be eligible for garbage collection?

A. 0

B. 1

C. 2

D. 3

E. 4

F. 5

12. Given:

```
 3. class Box {
 4.   int size;
 5.   Box(int s) { size = s; }
 6. }
 7. public class Laser {
 8.   public static void main(String[] args) {
 9.     Box b1 = new Box(5);
10.     Box[] ba = go(b1, new Box(6));
11.     ba[0] = b1;
12.     for(Box b : ba) System.out.print(b.size + " ");
13.   }
14.   static Box[] go(Box b1, Box b2) {
15.      b1.size = 4;
16.      Box[] ma = {b2, b1};
17.      return ma;
18.   }
19. }
```

What is the result?

A. `4 4`

B. `5 4`

C. `6 4`

D. `4 5`

E. `5 5`

F. Compilation fails

13. Given:

```
3. public class Dark {
4.   int x = 3;
5.   public static void main(String[] args) {
6.     new Dark().go1();
7.   }
8.   void go1() {
9.     int x;
10.    go2(++x);
11.  }
12.  void go2(int y) {
13.    int x = ++y;
14.    System.out.println(x);
15.  }
16. }
```

What is the result?

A. 2

B. 3

C. 4

D. 5

E. Compilation fails

F. An exception is thrown at runtime

SELF TEST ANSWERS

1. Given:

```
class CardBoard {
  Short story = 200;
  CardBoard go(CardBoard cb) {
    cb = null;
    return cb;
  }
  public static void main(String[] args) {
    CardBoard c1 = new CardBoard();
    CardBoard c2 = new CardBoard();
    CardBoard c3 = c1.go(c2);
    c1 = null;
    // do Stuff
} }
```

When `// doStuff` is reached, how many objects are eligible for GC?

A. 0
B. 1
C. 2
D. Compilation fails
E. It is not possible to know
F. An exception is thrown at runtime

Answer:

☑ **C** is correct. Only one `CardBoard object (c1)` is eligible, but it has an associated Short wrapper object that is also eligible.

☒ **A, B, D, E,** and **F** are incorrect based on the above. (Objective 7.4)

2. Given:

```
class Alien {
  String invade(short ships) { return "a few"; }
  String invade(short... ships) { return "many"; }
}
class Defender {
  public static void main(String [] args) {
    System.out.println(new Alien().invade(7));
} }
```

What is the result?

A. `many`
B. `a few`
C. Compilation fails
D. The output is not predictable
E. An exception is thrown at runtime

Answer:

☑ **C** is correct, compilation fails. The var-args declaration is fine, but `invade` takes a `short`, so the argument 7 needs to be cast to a `short`. With the cast, the answer is **B**, '`a few`'.

☒ **A, B, D,** and **E** are incorrect based on the above. (Objective 1.3)

3. Given:

```
1. class Dims {
2.   public static void main(String[] args) {
3.     int[][] a = {{1,2,}, {3,4}};
4.     int[] b = (int[]) a[1];
5.     Object o1 = a;
6.     int[][] a2 = (int[][]) o1;
7.     int[] b2 = (int[]) o1;
8.     System.out.println(b[1]);
9. } }
```

What is the result?

A. `2`
B. `4`
C. An exception is thrown at runtime
D. Compilation fails due to an error on line 4
E. Compilation fails due to an error on line 5
F. Compilation fails due to an error on line 6
G. Compilation fails due to an error on line 7

Answer:

☑ **C** is correct. A `ClassCastException` is thrown at line 7 because `o1` refers to an `int[][]` not an `int[]`. If line 7 was removed, the output would be 4.

☒ **A, B, D, E, F,** and **G** are incorrect based on the above. (Objective 1.3)

4. Given:

```
class Mixer {
  Mixer() { }
  Mixer(Mixer m) { m1 = m; }
  Mixer m1;
  public static void main(String[] args) {
    Mixer m2 = new Mixer();
    Mixer m3 = new Mixer(m2);  m3.go();
    Mixer m4 = m3.m1;          m4.go();
    Mixer m5 = m2.m1;          m5.go();
  }
  void go() { System.out.print("hi "); }
}
```

What is the result?

A. `hi`

B. `hi hi`

C. `hi hi hi`

D. Compilation fails

E. `hi`, followed by an exception

F. `hi hi`, followed by an exception

Answer:

☑ **F** is correct. The `m2` object's `m1` instance variable is never initialized, so when `m5` tries to use it a `NullPointerException` is thrown.

☒ **A, B, C, D,** and **E** are incorrect based on the above. (Objective 7.3)

5. Given:

```
class Fizz {
  int x = 5;
  public static void main(String[] args) {
    final Fizz f1 = new Fizz();
    Fizz f2 = new Fizz();
    Fizz f3 = FizzSwitch(f1,f2);
    System.out.println((f1 == f3) + " " + (f1.x == f3.x));
  }
  static Fizz FizzSwitch(Fizz x, Fizz y) {
    final Fizz z = x;
    z.x = 6;
    return z;
} }
```

What is the result?

A. `true true`
B. `false true`
C. `true false`
D. `false false`
E. Compilation fails
F. An exception is thrown at runtime

Answer:

☑ **A** is correct. The references `f1`, `z`, and `f3` all refer to the same instance of `Fizz`. The `final` modifier assures that a reference variable cannot be referred to a different object, but `final` doesn't keep the object's state from changing.

☒ **B, C, D, E,** and **F** are incorrect based on the above. (Objective 7.3)

6. 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

Answer:

☑ **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! (Objective 1.3)

7. Given:

```
3. public class Bridge {
4.   public enum Suits {
5.     CLUBS(20), DIAMONDS(20), HEARTS(30), SPADES(30),
6.     NOTRUMP(40) { public int getValue(int bid) {
                       return ((bid-1)*30)+40; } };
7.     Suits(int points) {  this.points = points;  }
8.     private int points;
9.     public int getValue(int bid) { return points * bid; }
10.  }
11.  public static void main(String[] args) {
12.    System.out.println(Suits.NOTRUMP.getBidValue(3));
13.    System.out.println(Suits.SPADES + " " + Suits.SPADES.points);
14.    System.out.println(Suits.values());
15.  }
16. }
```

Which are true? (Choose all that apply.)

A. The output could contain `30`

B. The output could contain `@bf73fa`

C. The output could contain `DIAMONDS`

D. Compilation fails due to an error on line 6

E. Compilation fails due to an error on line 7

F. Compilation fails due to an error on line 8

G. Compilation fails due to an error on line 9

H. Compilation fails due to an error within lines 12 to 14

Answer:

☑ **A** and **B** are correct. The code compiles and runs without exception. The `values()` method returns an array reference, not the contents of the enum, so DIAMONDS is never printed.

☒ **C, D, E, F, G,** and **H** are incorrect based on the above. (Objective 1.3)

8. Given:

```
3. public class Ouch {
4.   static int ouch = 7;
5.   public static void main(String[] args) {
6.     new Ouch().go(ouch);
7.     System.out.print(" " + ouch);
8.   }
9.   void go(int ouch) {
10.    ouch++;
11.    for(int ouch = 3; ouch < 6; ouch++)
12.      ;
13.    System.out.print(" " + ouch);
14.  }
15. }
```

What is the result?

A. `5 7`

B. `5 8`

C. `8 7`

D. `8 8`

E. Compilation fails

F. An exception is thrown at runtime

Answer:

☑ **E** is correct. The parameter declared on line 9 is valid (although ugly), but the variable name `ouch` cannot be declared again on line 11 in the same scope as the declaration on line 9.

☒ **A, B, C, D,** and **F** are incorrect based on the above. (Objective 1.3)

9. Given:

```
3. public class Bertha {
4.   static String s = "";
5.   public static void main(String[] args) {
6.     int x = 4; Boolean y = true; short[] sa = {1,2,3};
7.     doStuff(x, y);
8.     doStuff(x);
9.     doStuff(sa, sa);
10.    System.out.println(s);
11.  }
12.  static void doStuff(Object o)        { s += "1"; }
13.  static void doStuff(Object... o)     { s += "2"; }
14.  static void doStuff(Integer... i)    { s += "3"; }
15.  static void doStuff(Long L)          { s += "4"; }
16. }
```

What is the result?

A. `212`

B. `232`

C. `234`

D. `312`

E. `332`

F. `334`

G. Compilation fails

Answer:

☑ **A** is correct. It's legal to autobox and then widen. The first call to `doStuff()` boxes the int to an Integer then passes two objects. The second call cannot widen and then box (making the `Long` method unusable), so it boxes the int to an Integer. As always, a var-args method will be chosen only if no non-var-arg method is possible. The third call is passing two objects—they are of type 'short array.'

☒ **B, C, D, E, F,** and **G** are incorrect based on the above. (Objective 3.1)

10. Given:

```
3. class Dozens {
4.   int[] dz = {1,2,3,4,5,6,7,8,9,10,11,12};
5. }
6. public class Eggs {
7.   public static void main(String[] args) {
```

```
 8.     Dozens [] da = new Dozens[3];
 9.     da[0] = new Dozens();
10.     Dozens d = new Dozens();
11.     da[1] = d;
12.     d = null;
13.     da[1] = null;
14.     // do stuff
15.   }
16. }
```

Which two are true about the objects created within `main()`, and eligible for garbage collection when line 14 is reached?

A. Three objects were created

B. Four objects were created

C. Five objects were created

D. Zero objects are eligible for GC

E. One object is eligible for GC

F. Two objects are eligible for GC

G. Three objects are eligible for GC

Answer:

☑ **C** and **F** are correct. `da` refers to an object of type "Dozens array," and each Dozens object that is created comes with its own "int array" object. When line 14 is reached, only the second Dozens object (and its "int array" object) are not reachable.

☒ **A, B, D, E,** and **G** are incorrect based on the above. (Objective 7.4)

11. Given:

```
 3. class Beta { }
 4. class Alpha {
 5.   static Beta b1;
 6.   Beta b2;
 7. }
 8. public class Tester {
 9.   public static void main(String[] args) {
10.     Beta b1 = new Beta();     Beta b2 = new Beta();
11.     Alpha a1 = new Alpha();   Alpha a2 = new Alpha();
12.     a1.b1 = b1;
13.     a1.b2 = b1;
14.     a2.b2 = b2;
15.     a1 = null;  b1 = null;  b2 = null;
```

```
16.      // do stuff
17.    }
18. }
```

When line 16 is reached, how many objects will be eligible for garbage collection?

A. 0

B. 1

C. 2

D. 3

E. 4

F. 5

Answer:

☑ **B** is correct. It should be clear that there is still a reference to the object referred to by `a2`, and that there is still a reference to the object referred to by `a2.b2`. What might be less clear is that you can still access the other Beta object through the static variable `a2.b1`—because it's static.

☒ **A, C, D, E,** and **F** are incorrect based on the above. (Objective 7.4)

12. Given:

```
3. class Box {
4.   int size;
5.   Box(int s) { size = s; }
6. }
7. public class Laser {
8.   public static void main(String[] args) {
9.     Box b1 = new Box(5);
10.    Box[] ba = go(b1, new Box(6));
11.    ba[0] = b1;
12.    for(Box b : ba) System.out.print(b.size + " ");
13.  }
14.  static Box[] go(Box b1, Box b2) {
15.     b1.size = 4;
16.     Box[] ma = {b2, b1};
17.     return ma;
18.  }
19. }
```

What is the result?

A. `4 4`

B. `5 4`

C. `6 4`

D. `4 5`

E. `5 5`

F. Compilation fails

Answer:

☑ **A** is correct. Although `main()`'s `b1` is a different reference variable than `go()`'s `b1`, they refer to the same Box object.

☒ **B, C, D, E,** and **F** are incorrect based on the above. (Objective 7.3)

13. Given:

```
3. public class Dark {
4.   int x = 3;
5.   public static void main(String[] args) {
6.     new Dark().go1();
7.   }
8.   void go1() {
9.     int x;
10.    go2(++x);
11.  }
12.  void go2(int y) {
13.    int x = ++y;
14.    System.out.println(x);
15.  }
16. }
```

What is the result?

A. 2

B. 3

C. 4

D. 5

E. Compilation fails

F. An exception is thrown at runtime

Answer:

☑ **E** is correct. In `go1()` the local variable `x` is not initialized.

☒ **A, B, C, D,** and **F** are incorrect based on the above. (Objective 1.3)

4

Operators

CERTIFICATION OBJECTIVES

- Using Operators

✓ Two-Minute Drill

Q&A Self Test

If you've got variables, you're going to modify them. You'll increment them, add them together, and compare one to another (in about a dozen different ways). In this chapter, you'll learn how to do all that in Java. For an added bonus, you'll learn how to do things that you'll probably never use in the real world, but that will almost certainly be on the exam.

CERTIFICATION OBJECTIVE

Java Operators (Exam Objective 7.6)

*7.6 Write code that correctly applies the appropriate operators including assignment operators (limited to: =, +=, -=), arithmetic operators (limited to: +, -, *, /, %, ++, --), relational operators (limited to: <, <=, >, >=, ==, !=), the instanceof operator, logical operators (limited to: &, |, ^, !, &&, ||), and the conditional operator (? :), to produce a desired result. Write code that determines the equality of two objects or two primitives.*

Java operators produce new values from one or more operands (just so we're all clear, remember the operands are the things on the right or left side of the operator). The result of most operations is either a `boolean` or numeric value. Because you know by now that Java is not C++, you won't be surprised that Java operators aren't typically overloaded. There are, however, a few exceptional operators that come overloaded:

- The + operator can be used to add two numeric primitives together, or to perform a concatenation operation if either operand is a String.
- The &, |, and ^ operators can all be used in two different ways, although as of this version of the exam, their bit-twiddling capabilities won't be tested.

Stay awake. Operators are often the section of the exam where candidates see their lowest scores. Additionally, operators and assignments are a part of many questions in other topics…it would be a shame to nail a really tricky threads question, only to blow it on a pre-increment statement.

Assignment Operators

We covered most of the functionality of the assignment operator, "=", in Chapter 3. To summarize:

- When assigning a value to a primitive, *size* matters. Be sure you know when implicit casting will occur, when explicit casting is necessary, and when truncation might occur.
- Remember that a reference variable isn't an object; it's a way to *get* to an object. (We know all you C++ programmers are just dying for us to say "it's a pointer", but we're not going to.)
- When assigning a value to a reference variable, *type* matters. Remember the rules for supertypes, subtypes, and arrays.

Next we'll cover a few more details about the assignment operators that are on the exam, and when we get to Chapter 7, we'll take a look at how the assignment operator "=" works with Strings (which are immutable).

exam Watch

Don't spend time preparing for topics that are no longer on the exam! In a nutshell, the Java 5 exam differs from the 1.4 exam by moving away from bits, and towards the API. Many 1.4 topics related to operators have been removed from the exam, so in this chapter you WON'T see

- bit shifting operators
- bitwise operators
- two's complement
- divide by zero stuff

It's not that these aren't important topics, it's just that they're not on the exam anymore, and we're really focused on the exam.

Compound Assignment Operators

There are actually 11 or so compound assignment operators, but only the four most commonly used (`+=, -=, *=, and /=`), are on the exam (despite what the objectives say). The compound assignment operators let lazy typists shave a few keystrokes off their workload. Here are several example assignments, first without using a compound operator,

```
y = y - 6;
x = x + 2 * 5;
```

Now, with compound operators:

```
y -= 6;
x += 2 * 5;
```

The last two assignments give the same result as the first two.

exam Watch

Earlier versions of the exam put big emphasis on operator precedence (like: What's the result of: `x = y++ + ++x/z;`). Other than a very basic knowledge of precedence (such as: * and / are higher precedence than + and -), you won't need to study operator precedence, except that when using a compound operator, the expression on the right side of the = will always be evaluated first. For example, you might expect

```
x *= 2 + 5;
```

to be evaluated like this:

```
x = (x * 2) + 5;   // incorrect precedence
```

since multiplication has higher precedence than addition. But instead, the expression on the right is always placed inside parentheses. it is evaluated like this:

```
x = x * (2 + 5);
```

Relational Operators

The exam covers six relational operators (<, <=, >, >=, ==, and !=). Relational operators always result in a boolean (`true` or `false`) value. This `boolean` value is most often used in an `if` test, as follows,

```
int x = 8;
if (x < 9) {
  // do something
}
```

but the resulting value can also be assigned directly to a `boolean` primitive:

```
class CompareTest {
  public static void main(String [] args) {
    boolean b = 100 > 99;
    System.out.println("The value of b is " + b);
  }
}
```

Java has four relational operators that can be used to compare any combination of integers, floating-point numbers, or characters:

- **>** greater than
- **>=** greater than or equal to
- **<** less than
- **<=** less than or equal to

Let's look at some legal comparisons:

```
class GuessAnimal {
  public static void main(String[] args) {
    String animal = "unknown";
    int weight = 700;
    char sex = 'm';
    double colorWaveLength = 1.630;
    if (weight >= 500) { animal = "elephant"; }
    if (colorWaveLength > 1.621) { animal = "gray " + animal; }
    if (sex <= 'f') { animal = "female " + animal; }
    System.out.println("The animal is a " + animal);
  }
}
```

In the preceding code, we are using a comparison between characters. It's also legal to compare a character primitive with any number (though it isn't great programming style). Running the preceding class will output the following:

```
The animal is a gray elephant
```

We mentioned that characters can be used in comparison operators. When comparing a character with a character, or a character with a number, Java will use the Unicode value of the character as the numerical value, for comparison.

"Equality" Operators

Java also has two relational operators (sometimes called "equality operators") that compare two similar "things" and return a `boolean` the represents what's true about the two "things" being equal. These operators are

- == equals (also known as "equal to")
- != not equals (also known as "not equal to")

Each individual comparison can involve two numbers (including `char`), two `boolean` values, or two object reference variables. You can't compare incompatible types, however. What would it mean to ask if a `boolean` is equal to a `char`? Or if a `Button` is equal to a `String` array? (Exactly, nonsense, which is why you can't do it.) There are four different types of things that can be tested:

- numbers
- characters
- boolean primitives
- Object reference variables

So what does == look at? The value in the variable—in other words, the bit pattern.

Equality for Primitives

Most programmers are familiar with comparing primitive values. The following code shows some equality tests on primitive variables:

```
class ComparePrimitives {
  public static void main(String[] args) {
    System.out.println("char 'a' == 'a'? " + ('a' == 'a'));
    System.out.println("char 'a' == 'b'? " + ('a' == 'b'));
    System.out.println("5 != 6? " + (5 != 6));
    System.out.println("5.0 == 5L? " + (5.0 == 5L));
    System.out.println("true == false? " + (true == false));
  }
}
```

This program produces the following output:

```
char 'a' == 'a'? true
char 'a' == 'b'? false
5 != 6? true
5.0 == 5L? true
true == false? false
```

As we can see, usually if a floating-point number is compared with an integer and the values are the same, the == operator returns `true` as expected.

Equality for Reference Variables

As we saw earlier, two reference variables can refer to the same object, as the following code snippet demonstrates:

```
JButton a = new JButton("Exit");
JButton b = a;
```

After running this code, both variable `a` and variable `b` will refer to the same object (a `JButton` with the label `Exit`). Reference variables can be tested to see if they refer to the same object by using the == operator. Remember, the == operator is looking at the bits in the variable, so for reference variables this means that if the

exam Watch

Don't mistake = for == in a boolean expression. The following is legal:

```
11. boolean b = false;
12. if (b = true) { System.out.println("b is true");
13. } else { System.out.println("b is false");  }
```

Look carefully! You might be tempted to think the output is `b is false` ***but look at the boolean test in line 12. The boolean variable*** `b` ***is not being compared to*** `true`***, it's being set to*** `true`***, so the*** `println` ***executes and we get*** `b is true`***. The result of any assignment expression is the value of the variable following the assignment. This substitution of = for == works only with boolean variables, since the*** `if` ***test can be done only on boolean expressions. Thus, this does not compile:***

```
7. int x = 1;
8. if (x = 0) { }
```

Because `x` ***is an integer (and not a*** `boolean`***), the result of*** `(x = 0)` ***is 0 (the result of the assignment). Primitive*** `ints` ***cannot be used where a boolean value is expected, so the code in line 8 won't work unless changed from an assignment (=) to an equality test (==) as follows:***

```
8. if (x == 0) { }
```

bits in both reference variables are identical, then both refer to the same object. Look at the following code:

```
import javax.swing.JButton;
class CompareReference {
  public static void main(String[] args) {
    JButton a = new JButton("Exit");
    JButton b = new JButton("Exit");
    JButton c = a;
    System.out.println("Is reference a == b? " + (a == b));
    System.out.println("Is reference a == c? " + (a == c));
  }
}
```

This code creates three reference variables. The first two, `a` and `b`, are separate JButton objects that happen to have the same label. The third reference variable, `c`, is initialized to refer to the same object that a refers to. When this program runs, the following output is produced:

```
Is reference a == b? false
Is reference a == c? true
```

This shows us that `a` and `c` reference the same instance of a `JButton`. The `==` operator will not test whether two objects are "meaningfully equivalent," a concept we'll cover in much more detail in Chapter 7, when we look at the `equals()` *method* (as opposed to the equals *operator* we're looking at here).

Equality for Enums

Once you've declared an enum, it's not expandable. At runtime, there's no way to make additional enum constants. Of course, you can have as many variables as you'd like refer to a given enum constant, so it's important to be able to compare two enum reference variables to see if they're "equal", i.e. do they refer to the same enum constant. You can use either the `==` operator or the `equals()` method to determine if two variables are referring to the same enum constant:

```
class EnumEqual {
  enum Color {RED, BLUE}                    // ; is optional
  public static void main(String[] args) {
    Color c1 = Color.RED;  Color c2 = Color.RED;
    if(c1 == c2) { System.out.println("=="); }
    if(c1.equals(c2)) { System.out.println("dot equals"); }
} }
```

(We know } } is ugly, we're prepping you). This produces the output:

```
==
dot equals
```

instanceof Comparison

The `instanceof` operator is used for object reference variables only, and you can use it to check whether an object is of a particular type. By type, we mean class or interface type—in other words, if the object referred to by the variable on the left side of the operator passes the IS-A test for the class or interface type on the right side (Chapter 2 covered IS-A relationships in detail). The following simple example

```
public static void main(String[] args) {
  String s = new String("foo");
  if (s instanceof String) {
    System.out.print("s is a String");
  }
}
```

prints this: `s is a String`

Even if the object being tested is not an actual instantiation of the class type on the right side of the operator, `instanceof` will still return `true` if the object being compared is *assignment compatible* with the type on the right.

The following example demonstrates a common use for `instanceof`: testing an object to see if it's an instance of one of its subtypes, before attempting a "downcast":

```
class A { }
class B extends A {
  public static void main (String [] args) {
    A myA = new B();
    m2(myA);
  }
  public static void m2(A a) {
    if (a instanceof B)
      ((B)a).doBstuff();     // downcasting an A reference
                             // to a B reference
  }
  public static void doBstuff() {
    System.out.println("'a' refers to a B");
  }
}
```

The preceding code compiles and produces the output:

```
'a' refers to a B
```

In examples like this, the use of the `instanceof` operator protects the program from attempting an illegal downcast.

You can test an object reference against its own class type, or any of its superclasses. This means that *any* object reference will evaluate to `true` if you use the `instanceof` operator against type `Object`, as follows,

```
B b = new B();
if (b instanceof Object) {
   System.out.print("b is definitely an Object");
}
```

which prints this: `b is definitely an Object`

exam Watch

Look for instanceof questions that test whether an object is an instance of an interface, when the object's class implements the interface indirectly. An indirect implementation occurs when one of an object's superclasses implements an interface, but the actual class of the instance does not—for example,

```
interface Foo { }
class A implements Foo { }
class B extends A { }
...
A a = new A();
B b = new B();
```

the following are true:

```
a instanceof Foo
b instanceof A
b instanceof Foo  // implemented indirectly
```

An object is said to be of a particular interface type (meaning it will pass the `instanceof` test) if any of the object's superclasses implement the interface.

In addition, it is legal to test whether the `null` reference is an instance of a class. This will always result in `false`, of course. For example:

```
class InstanceTest {
   public static void main(String [] args) {
      String a = null;
      boolean b = null instanceof String;
      boolean c = a instanceof String;
      System.out.println(b + " " + c);
   }
}
```

prints this: `false false`

instanceof Compiler Error

You can't use the `instanceof` operator to test across two different class hierarchies. For instance, the following will NOT compile:

```
class Cat { }
class Dog {
  public static void main(String [] args) {
    Dog d = new Dog();
    System.out.println(d instanceof Cat);
  }
}
```

Compilation fails—there's no way `d` could ever refer to a `Cat` or a subtype of `Cat`.

exam Watch

Remember that arrays are objects, even if the array is an array of primitives. Watch for questions that look something like this:

```
int [] nums = new int[3];
if (nums instanceof Object) { } // result is true
```

An array is always an instance of Object. Any array.

Table 4-1 summarizes the use of the `instanceof` operator given the following:

```
interface Face { }
class Bar implements Face{ }
class Foo extends Bar { }
```

TABLE 4-1 Operands and Results Using *instanceof* Operator.

First Operand (Reference Being Tested)	instanceof Operand (Type We're Comparing the Reference Against)	Result
null	Any class or interface type	false
Foo instance	Foo, Bar, Face, Object	true
Bar instance	Bar, Face, Object	true
Bar instance	Foo	false
Foo [ ]	Foo, Bar, Face	false
Foo [ ]	Object	true
Foo [1]	Foo, Bar, Face, Object	true

Arithmetic Operators

We're sure you're familiar with the basic arithmetic operators.

- `+` addition
- `-` subtraction
- `*` multiplication
- `/` division

These can be used in the standard way:

```
int x = 5 * 3;
int y = x - 4;
System.out.println("x - 4 is " +  y);  // Prints 11
```

The Remainder (%) Operator

One operator you might not be as familiar with is the remainder operator, %. The remainder operator divides the left operand by the right operand, and the result is the remainder, as the following code demonstrates:

```
class MathTest {
  public static void main (String [] args) {
    int x = 15;
    int y = x % 4;
    System.out.println("The result of 15 % 4 is the "
      + "remainder of 15 divided by 4. The remainder is " + y);
  }
}
```

Running class MathTest prints the following:

```
The result of 15 % 4 is the remainder of 15 divided by 4. The
remainder is 3
```

(Remember: Expressions are evaluated from left to right by default. You can change this sequence, or *precedence*, by adding parentheses. Also remember that the * , / , and % operators have a higher precedence than the + and - operators.)

String Concatenation Operator

The plus sign can also be used to concatenate two strings together, as we saw earlier (and as we'll definitely see again):

```
String animal = "Grey " + "elephant";
```

String concatenation gets interesting when you combine numbers with Strings. Check out the following:

```
String a = "String";
int b = 3;
int c = 7;
System.out.println(a + b + c);
```

Will the + operator act as a plus sign when adding the `int` variables `b + c`? Or will the + operator treat `3` and `7` as characters, and concatenate them individually? Will the result be `String10` or `String37`? OK, you've had long enough to think about it. The `int` values were simply treated as characters and glued on to the right side of the String, giving the result:

```
String37
```

So we could read the previous code as

> "Start with `String a`, `String`, and add the character `3` (the value of `b`) to it, to produce a new string `String3`, and then add the character `7` (the value of `c`) to that, to produce a new string `String37`, then print it out."

However, if you put parentheses around the two `int` variables, as follows,

```
System.out.println(a + (b + c));
```

you'll get this: `String10`

Using parentheses causes the `(b + c)` to evaluate first, so the rightmost + operator functions as the addition operator, given that both operands are `int` values. The key point here is that within the parentheses, the left-hand operand is not a `String`. If it were, then the + operator would perform `String` concatenation. The previous code can be read as

> "Add the values of `b + c` together, then take the sum and convert it to a `String` and concatenate it with the `String` from variable a."

The rule to remember is this:

> *If either operand is a* String*, the + operator becomes a* String *concatenation operator. If both operands are numbers, the + operator is the addition operator.*

You'll find that sometimes you might have trouble deciding whether, say, the left-hand operator is a `String` or not. On the exam, don't expect it to always be obvious. (Actually, now that we think about it, don't expect it ever to be obvious.) Look at the following code:

```
System.out.println(x.foo() + 7);
```

You can't know how the + operator is being used until you find out what the `foo()` method returns! If `foo()` returns a `String`, then 7 is concatenated to the returned

`String`. But if `foo()` returns a number, then the + operator is used to add 7 to the return value of `foo()`.

Finally, you need to know that it's legal to mush together the compound additive operator (+=) and Strings, like so:

```
String s = "123";
s += "45";
s += 67;
System.out.println(s);
```

Since both times the += operator was used and the left operand was a String, both operations were concatenations, resulting in

```
1234567
```

exam Watch

If you don't understand how String concatenation works, especially within a `print` ***statement, you could actually fail the exam even if you know the rest of the answer to the questions! Because so many questions ask, "What is the result?", you need to know not only the result of the code running, but also how that result is printed. Although there will be at least a few questions directly testing your String knowledge, String concatenation shows up in other questions on virtually every objective. Experiment! For example, you might see a line such as***

```
int b = 2;
System.out.println("" + b + 3);
```

which prints `23`

but if the print statement changes to

```
System.out.println(b + 3);
```

then the result becomes `5`

Increment and Decrement Operators

Java has two operators that will increment or decrement a variable by exactly one. These operators are composed of either two plus signs (++) or two minus signs (--):

- ++ increment (prefix and postfix)
- -- decrement (prefix and postfix)

The operator is placed either before (prefix) or after (postfix) a variable to change its value. Whether the operator comes before or after the operand can change the outcome of an expression. Examine the following:

```
1. class MathTest {
2.   static int players = 0;
3.     public static void main (String [] args) {
4.       System.out.println("players online: " + players++);
5.       System.out.println("The value of players is "
                              + players);
6.       System.out.println("The value of players is now "
                              + ++players);
7.    }
8. }
```

Notice that in the fourth line of the program the increment operator is *after* the variable `players`. That means we're using the postfix increment operator, which causes `players` to be incremented by one but only *after* the value of `players` is used in the expression. When we run this program, it outputs the following:

```
%java MathTest
players online: 0
The value of players is 1
The value of players is now 2
```

Notice that when the variable is written to the screen, at first it says the value is 0. Because we used the postfix increment operator, the increment doesn't happen until after the `players` variable is used in the `print` statement. Get it? The "post" in postfix means *after*. Line 5 doesn't increment `players`; it just outputs its value to the screen, so the newly incremented value displayed is 1. Line 6 applies the prefix increment operator to `players`, which means the increment happens *before* the value of the variable is used, so the output is 2.

Expect to see questions mixing the increment and decrement operators with other operators, as in the following example:

```
int x = 2;  int y = 3;
if ((y == x++) | (x < ++y)) {
  System.out.println("x = " + x + " y = " + y);
 }
```

The preceding code prints: `x = 3 y = 4`

You can read the code as follows: "If 3 is equal to 2 OR 3 < 4"

The first expression compares `x` and `y`, and the result is `false`, because the increment on `x` doesn't happen until *after* the == test is made. Next, we increment `x`, so now `x` is 3. Then we check to see if `x` is less than `y`, but we increment `y` *before* comparing it with `x`! So the second logical test is `(3 < 4)`. The result is `true`, so the `print` statement runs.

As with String concatenation, the increment and decrement operators are used throughout the exam, even on questions that aren't trying to test your knowledge of how those operators work. You might see them in questions on `for` loops, exceptions, even threads. Be ready.

exam Watch

Look out for questions that use the increment or decrement operators on a final variable. Because final variables can't be changed, the increment and decrement operators can't be used with them, and any attempt to do so will result in a compiler error. The following code won't compile:

```
final int x = 5;
int y = x++;
```

and produces the error:

```
Test.java:4: cannot assign a value to final variable x
int y = x++;
        ^
```

You can expect a violation like this to be buried deep in a complex piece of code. If you spot it, you know the code won't compile and you can move on without working through the rest of the code.

This question might seem to be testing you on some complex arithmetic operator trivia, when in fact it's testing you on your knowledge of the `final` modifier.

Conditional Operator

The conditional operator is a *ternary* operator (it has *three* operands) and is used to evaluate `boolean` expressions, much like an `if` statement except instead of executing a block of code if the test is `true`, a conditional operator will assign a value to a variable. In other words, the goal of the conditional operator is to decide which of two values to assign to a variable. This operator is constructed using a ? (question mark) and a : (colon). The parentheses are optional. Its structure is:

`x =` (boolean expression) `?` value to assign if `true` `:` value to assign if `false`

Let's take a look at a conditional operator in code:

```
class Salary {
  public static void main(String [] args) {
    int numOfPets = 3;
    String status = (numOfPets<4) ? "Pet limit not exceeded"
                    : "too many pets";
    System.out.println("This pet status is " + status);
  }
}
```

You can read the preceding code as

> Set `numOfPets` equal to 3. Next we're going to assign a `String` to the status variable. If `numOfPets` is less than 4, assign "`Pet limit not exceeded`" to the `status` variable; otherwise, assign "`too many pets`" to the `status` variable.

A conditional operator starts with a `boolean` operation, followed by two possible values for the variable to the left of the assignment (=) operator. The first value (the one to the left of the colon) is assigned if the conditional (`boolean`) test is `true`, and the second value is assigned if the conditional test is `false`. You can even nest conditional operators into one statement:

```
class AssignmentOps {
  public static void main(String [] args) {
   int sizeOfYard = 10;
   int numOfPets = 3;
   String status = (numOfPets<4)?"Pet count OK"
       :(sizeOfYard > 8)? "Pet limit on the edge"
         :"too many pets";
    System.out.println("Pet status is " + status);
```

```
    }
}
```

Don't expect many questions using conditional operators, but remember that conditional operators are sometimes confused with assertion statements, so be certain you can tell the difference. Chapter 5 covers assertions in detail.

Logical Operators

The exam objectives specify six "logical" operators (&, |, ^, !, &&, and ||). Some Sun documentation uses other terminology for these operators, but for our purposes the "logical operators" are the six listed above, and in the exam objectives.

Bitwise Operators (Not on the Exam!)

Okay, this is going to be confusing. Of the six logical operators listed above, three of them (&, |, and ^) can also be used as "bitwise" operators. Bitwise operators were included in previous versions of the exam, but they're not on the Java 5 exam. Here are several legal statements that use bitwise operators:

```
byte b1 = 6 & 8;
byte b2 = 7 | 9;
byte b3 = 5 ^ 4;
System.out.println(b1 + " " + b2 + " " + b3);
```

Bitwise operators compare two variables bit by bit, and return a variable whose bits have been set based on whether the two variables being compared had respective bits that were either both "on" (&), one or the other "on" (|), or exactly one "on" (^). By the way, when we run the preceding code, we get

```
0 15 1
```

Having said all this about bitwise operators, the key thing to remember is this:

BITWISE OPERATORS ARE NOT ON THE EXAM!

So why did we bring them up? If you get hold of an old exam preparation book, or if you find some mock exams that haven't been properly updated, you're bound to find questions that perform bitwise operations. Unless you're a glutton for punishment, you can skip this kind of mock question.

Short-Circuit Logical Operators

There are five logical operators on the exam that are used to evaluate statements that contain more than one `boolean` expression. The most commonly used of the five are the two *short-circuit* logical operators. They are

- **`&&`** short-circuit AND
- `||` short-circuit OR

They are used to link little `boolean` expressions together to form bigger `boolean` expressions. The && and || operators evaluate only `boolean` values. For an AND (&&) expression to be `true`, both operands must be `true`—for example,

```
if ((2 < 3) && (3 < 4)) { }
```

The preceding expression evaluates to `true` because *both* operand one (2 < 3) and operand two (3 < 4) evaluate to `true`.

The short-circuit feature of the && operator is so named because it doesn't waste its time on pointless evaluations. A short-circuit && evaluates the left side of the operation first (operand one), and if it resolves to `false`, the && operator doesn't bother looking at the right side of the expression (operand two) since the && operator already *knows* that the complete expression can't possibly be `true`.

```
class Logical {
  public static void main(String [] args) {
    boolean b = true && false;
    System.out.println("boolean b = " + b);
  }
}
```

When we run the preceding code, we get

```
%java Logical
boolean b = false
```

The || operator is similar to the && operator, except that it evaluates to `true` if EITHER of the operands is true. If the first operand in an OR operation is `true`, the result will be `true`, so the short-circuit || doesn't waste time looking at the right side of the equation. If the first operand is `false`, however, the short-circuit || has to evaluate the second operand to see if the result of the OR operation will be

`true` or `false`. Pay close attention to the following example; you'll see quite a few questions like this on the exam:

```
 1. class TestOR {
 2.   public static void main(String[] args) {
 3.     if ((isItSmall(3)) || (isItSmall(7))) {
 4.       System.out.println("Result is true");
 5.     }
 6.     if ((isItSmall(6)) || (isItSmall(9))) {
 7.       System.out.println("Result is true");
 8.     }
 9.  }
10.
11.   public static boolean isItSmall(int i) {
12.     if (i < 5) {
13.       System.out.println("i < 5");
14.       return true;
15.     } else {
16.       System.out.println("i >= 5");
17.       return false;
18.     }
19.   }
20. }
```

What is the result?

```
% java TestOR
i < 5
Result is true
i >= 5
i >= 5
```

Here's what happened when the `main()` method ran:

1. When we hit line 3, the first operand in the `||` expression (in other words, the *left* side of the `||` operation) is evaluated.

2. The `isItSmall(3)` method is invoked, prints `"i < 5"`, and returns `true`.

3. Because the *first* operand in the `||` expression on line 3 is `true`, the `||` operator doesn't bother evaluating the second operand. So we never see the `"i >= 5"` that would have printed had the *second* operand been evaluated (which would have invoked `isItSmall(7)`).

4. Line 6 is evaluated, beginning with the *first* operand in the || expression.
5. The `isItSmall(6)` method is called, prints "`i >= 5`", and returns `false`.
6. Because the *first* operand in the || expression on line 6 is `false`, the || operator can't skip the *second* operand; there's still a chance the expression can be `true`, if the *second* operand evaluates to `true`.
7. The `isItSmall(9)` method is invoked and prints "`i >= 5`".
8. The `isItSmall(9)` method returns `false`, so the expression on line 6 is `false`, and thus line 7 never executes.

exam Watch

The || and && operators work only with boolean operands. The exam may try to fool you by using integers with these operators:

```
if (5 && 6) { }
```

It looks as though we're trying to do a bitwise AND on the bits representing the integers 5 and 6, but the code won't even compile.

Logical Operators (Not Short-Circuit)

There are two *non-short-circuit* logical operators.

- `&` non-short-circuit AND
- `|` non-short-circuit OR

These operators are used in logical expressions just like the && and || operators are used, but because they aren't the short-circuit operators, they evaluate both sides of the expression, always! They're inefficient. For example, even if the *first* operand (left side) in an & expression is `false`, the *second* operand will still be evaluated—even though it's now impossible for the result to be `true`! And the | is just as inefficient: if the *first* operand is `true`, the Java Virtual Machine (JVM) still plows ahead and evaluates the *second* operand even when it knows the expression will be `true` regardless.

You'll find a lot of questions on the exam that use both the short-circuit and non-short-circuit logical operators. You'll have to know exactly which operands are evaluated and which are not, since the result will vary depending on whether the second operand in the expression is evaluated:

```
int z = 5;
if(++z > 5 || ++z > 6) z++;   // z = 7 after this code
```

versus:

```
int z = 5;
if(++z > 5 | ++z > 6) z++;   // z = 8 after this code
```

Logical Operators ^ and !

The last two logical operators on the exam are

- ^ exclusive-OR (XOR)
- ! boolean invert

The ^ (exclusive-OR) operator evaluates only `boolean` values. The ^ operator is related to the non-short-circuit operators we just reviewed, in that it always evaluates *both* the left and right operands in an expression. For an exclusive-OR (^) expression to be `true`, EXACTLY one operand must be `true`—for example,

```
System.out.println("xor " + ((2<3) ^ (4>3)));
```

produces the output: `xor false`

The preceding expression evaluates to **`false`** because BOTH operand one `(2 < 3)` and operand two `(4 > 3)` evaluate to `true`.

The ! (boolean invert) operator returns the opposite of a boolean's current value:

```
if(!(7 == 5)) { System.out.println("not equal"); }
```

can be read "if it's not true that 7 == 5," and the statement produces this output:

```
not equal
```

Here's another example using booleans:

```
boolean t = true;
boolean f = false;
System.out.println("! " + (t & !f) + " " + f);
```

produces the output:

```
! true false
```

In the preceding example, notice that the & test succeeded (printing `true`), and that the value of the `boolean` variable `f` did not change, so it printed `false`.

CERTIFICATION SUMMARY

If you've studied this chapter diligently, you should have a firm grasp on Java operators, and you should understand what equality means when tested with the == operator. Let's review the highlights of what you've learned in this chapter.

The logical operators (&& , ||, &, |, and ^) can be used only to evaluate two `boolean` expressions. The difference between && and & is that the && operator won't bother testing the right operand if the left evaluates to `false`, because the result of the && expression can never be `true`. The difference between || and | is that the || operator won't bother testing the right operand if the left evaluates to `true`, because the result is already known to be `true` at that point.

The == operator can be used to compare values of primitives, but it can also be used to determine whether two reference variables refer to the same object.

The `instanceof` operator is used to determine if the object referred to by a reference variable passes the IS-A test for a specified type.

The + operator is overloaded to perform `String` concatenation tasks, and can also concatenate `String`s and primitives, but be careful—concatenation can be tricky.

The conditional operator (a.k.a. the "ternary operator") has an unusual, three-operand syntax—don't mistake it for a complex assert statement.

The **++** and **--** operators will be used throughout the exam, and you must pay attention to whether they are prefixed or postfixed to the variable being updated.

Be prepared for a lot of exam questions involving the topics from this chapter. Even within questions testing your knowledge of another objective, the code will frequently use operators, assignments, object and primitive passing, and so on.

TWO-MINUTE DRILL

Here are some of the key points from each section in this chapter.

Relational Operators (Objective 7.6)

- Relational operators always result in a `boolean` value (`true` or `false`).
- There are six relational operators: >, >=, <, <=, ==, and !=. The last two (== and !=) are sometimes referred to as *equality operators*.
- When comparing characters, Java uses the Unicode value of the character as the numerical value.
- Equality operators
  - There are two equality operators: == and !=.
  - Four types of things can be tested: numbers, characters, booleans, and reference variables.
- When comparing reference variables, == returns `true` only if both references refer to the same object.

instanceof Operator (Objective 7.6)

- `instanceof` is for reference variables only, and checks for whether the object is of a particular type.
- The `instanceof` operator can be used only to test objects (or `null`) against class types that are in the same class hierarchy.
- For interfaces, an object passes the `instanceof` test if any of its superclasses implement the interface on the right side of the `instanceof` operator.

Arithmetic Operators (Objective 7.6)

- There are four primary math operators: add, subtract, multiply, and divide.
- The remainder operator (%), returns the remainder of a division.
- Expressions are evaluated from left to right, unless you add parentheses, or unless some operators in the expression have higher precedence than others.
- The *, /, and % operators have higher precedence than + and -.

String Concatenation Operator (Objective 7.6)

- If either operand is a `String`, the + operator concatenates the operands.
- If both operands are numeric, the + operator adds the operands.

Increment/Decrement Operators (Objective 7.6)

- Prefix operators (`++` and `--`) run before the value is used in the expression.
- Postfix operators (`++` and `--`) run after the value is used in the expression.
- In any expression, both operands are fully evaluated *before* the operator is applied.
- Variables marked `final` cannot be incremented or decremented.

Ternary (Conditional Operator) (Objective 7.6)

- Returns one of two values based on whether a `boolean` expression is `true` or `false`.
  - Returns the value after the `?` if the expression is `true`.
  - Returns the value after the `:` if the expression is `false`.

Logical Operators (Objective 7.6)

- The exam covers six "logical" operators: &, |, ^, !, &&, and ||.
- Logical operators work with two expressions (except for !) that must resolve to `boolean` values.
- The && and & operators return `true` only if both operands are `true`.
- The || and | operators return `true` if either or both operands are `true`.
- The && and || operators are known as short-circuit operators.
- The && operator does not evaluate the right operand if the left operand is `false`.
- The || does not evaluate the right operand if the left operand is `true`.
- The & and | operators always evaluate both operands.
- The ^ operator (called the "logical XOR"), returns `true` if exactly one operand is `true`.
- The ! operator (called the "inversion" operator), returns the opposite value of the `boolean` operand it precedes.

SELF TEST

1. Given:

```
class Hexy {
  public static void main(String[] args) {
    Integer i = 42;
    String s = (i<40)?"life":(i>50)?"universe":"everything";
    System.out.println(s);
  }
}
```

What is the result?

A. `null`
B. `life`
C. `universe`
D. `everything`
E. Compilation fails
F. An exception is thrown at runtime

2. Given:

```
1. class Comp2 {
2.   public static void main(String[] args) {
3.     float f1 = 2.3f;
4.     float[][] f2 = {{42.0f}, {1.7f, 2.3f}, {2.6f, 2.7f}};
5.     float[] f3 = {2.7f};
6.     Long x = 42L;
7.     // insert code here
8.       System.out.println("true");
9.   }
10. }
```

And the following five code fragments:

```
F1.  if(f1 == f2)
F2.  if(f1 == f2[2][1])
F3.  if(x == f2[0][0])
F4.  if(f1 == f2[1,1])
F5.  if(f3 == f2[2])
```

What is true?

A. One of them will compile, only one will be `true`
B. Two of them will compile, only one will be `true`
C. Two of them will compile, two will be `true`
D. Three of them will compile, only one will be `true`
E. Three of them will compile, exactly two will be `true`
F. Three of them will compile, exactly three will be `true`

3. Given:

```
class Fork {
  public static void main(String[] args) {
    if(args.length == 1 | args[1].equals("test")) {
      System.out.println("test case");
    } else {
      System.out.println("production " + args[0]);
    }
  }
}
```

And the command-line invocation:

```
java Fork live2
```

What is the result?

A. `test case`
B. `production live2`
C. `test case live2`
D. Compilation fails
E. An exception is thrown at runtime

4. Given:

```
class Feline {
  public static void main(String[] args) {
    Long x = 42L;
    Long y = 44L;
    System.out.print(" " + 7 + 2 + " ");
    System.out.print(foo() + x + 5 + " ");
    System.out.println(x + y + foo());
  }
  static String foo() { return "foo"; }
}
```

What is the result?

A. `9 foo47 86foo`

B. `9 foo47 4244foo`

C. `9 foo425 86foo`

D. `9 foo425 4244foo`

E. `72 foo47 86foo`

F. `72 foo47 4244foo`

G. `72 foo425 86foo`

H. `72 foo425 4244foo`

I. Compilation fails

5. Place the fragments into the code to produce the output 33. Note, you must use each fragment exactly once.

CODE:

```
class Incr {
  public static void main(String[] args) {
    Integer x = 7;
    int y = 2;

    x    ___  ___;
    ___  ___  ___;
    ___  ___  ___;
    ___  ___  ___;

    System.out.println(x);
  }
}
```

FRAGMENTS:

y	y	y	y
y	x	x	
-=	*=	*=	*=

6. Given:

```
3. public class Twisty {
4.   { index = 1; }
5.   int index;
6.   public static void main(String[] args) {
7.     new Twisty().go();
8.   }
9.   void go() {
10.    int [][] dd = {{9,8,7}, {6,5,4}, {3,2,1,0}};
11.    System.out.println(dd[index++][index++]);
12.  }
13. }
```

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

A. 1

B. 2

C. 4

D. 6

E. 8

F. Compilation fails

G. An exception is thrown at runtime

7. Given:

```
3. public class McGee {
4.   public static void main(String[] args) {
5.     Days d1 = Days.TH;
6.     Days d2 = Days.M;
7.     for(Days d: Days.values()) {
8.       if(d.equals(Days.F)) break;
9.       d2 = d;
10.    }
11.    System.out.println((d1 == d2)?"same old" : "newly new");
12.  }
13.  enum Days {M, T, W, TH, F, SA, SU};
14. }
```

What is the result?

A. `same old`

B. `newly new`

C. Compilation fails due to multiple errors
D. Compilation fails due only to an error on line 7
E. Compilation fails due only to an error on line 8
F. Compilation fails due only to an error on line 11
G. Compilation fails due only to an error on line 13

8. Given:

```
 4. public class SpecialOps {
 5.   public static void main(String[] args) {
 6.     String s = "";
 7.     Boolean b1 = true;
 8.     boolean b2 = false;
 9.     if((b2 = false) | (21%5) > 2) s += "x";
10.     if(b1 || (b2 == true))        s += "y";
11.     if(b2 == true)                s += "z";
12.     System.out.println(s);
13.   }
14. }
```

Which are true? (Choose all that apply.)

A. Compilation fails
B. x will be included in the output
C. y will be included in the output
D. z will be included in the output
E. An exception is thrown at runtime

9. Given:

```
 3. public class Spock {
 4.   public static void main(String[] args) {
 5.     int mask = 0;
 6.     int count = 0;
 7.     if( ((5<7) || (++count < 10)) | mask++ < 10 )   mask = mask + 1;
 8.     if(  (6 > 8) ^ false)                           mask = mask + 10;
 9.     if( !(mask > 1) && ++count > 1)                 mask = mask + 100;
10.     System.out.println(mask + " " + count);
11.   }
12. }
```

Which two are true about the value of mask and the value of count at line 10? (Choose two.)

A. `mask` is 0

B. `mask` is 1

C. `mask` is 2

D. `mask` is 10

E. `mask` is greater than 10

F. `count` is 0

G. `count` is greater than 0

10. Given:

```
 3. interface Vessel { }
 4. interface Toy { }
 5. class Boat implements Vessel { }
 6. class Speedboat extends Boat implements Toy { }
 7. public class Tree {
 8.   public static void main(String[] args) {
 9.     String s = "0";
10.     Boat b = new Boat();
11.     Boat b2 = new Speedboat();
12.     Speedboat s2 = new Speedboat();
13.     if((b instanceof Vessel) && (b2 instanceof Toy))  s += "1";
14.     if((s2 instanceof Vessel) && (s2 instanceof Toy)) s += "2";
15.     System.out.println(s);
16.   }
17. }
```

What is the result?

A. `0`

B. `01`

C. `02`

D. `012`

E. Compilation fails

F. An exception is thrown at runtime

SELF TEST ANSWERS

1. Given:

```
class Hexy {
  public static void main(String[] args) {
    Integer i = 42;
    String s = (i<40)?"life":(i>50)?"universe":"everything";
    System.out.println(s);
  }
}
```

What is the result?

A. `null`
B. `life`
C. `universe`
D. `everything`
E. Compilation fails
F. An exception is thrown at runtime

Answer:

☑ **D** is correct. This is a ternary nested in a ternary with a little unboxing thrown in. Both of the ternary expressions are `false`.

☒ **A, B, C, E,** and **F** are incorrect based on the above. (Objective 7.6)

2. Given:

```
1. class Comp2 {
2.   public static void main(String[] args) {
3.     float f1 = 2.3f;
4.     float[][] f2 = {{42.0f}, {1.7f, 2.3f}, {2.6f, 2.7f}};
5.     float[] f3 = {2.7f};
6.     Long x = 42L;
7.     // insert code here
8.       System.out.println("true");
9.   }
10. }
```

And the following five code fragments:

```
F1.  if(f1 == f2)
F2.  if(f1 == f2[2][1])
F3.  if(x == f2[0][0])
F4.  if(f1 == f2[1,1])
F5.  if(f3 == f2[2])
```

What is true?

A. One of them will compile, only one will be `true`
B. Two of them will compile, only one will be `true`
C. Two of them will compile, two will be `true`
D. Three of them will compile, only one will be `true`
E. Three of them will compile, exactly two will be `true`
F. Three of them will compile, exactly three will be `true`

Answer:

☑ **D** is correct. Fragments `F2`, `F3`, and `F5` will compile, and only `F3` is `true`.
☒ **A, B, C, E,** and **F** are incorrect. `F1` is incorrect because you can't compare a primitive to an array. `F4` is incorrect syntax to access an element of a two-dimensional array. (Objective 7.6)

3. Given:

```
class Fork {
  public static void main(String[] args) {
    if(args.length == 1 | args[1].equals("test")) {
      System.out.println("test case");
    } else {
      System.out.println("production " + args[0]);
    }
  }
}
```

And the command-line invocation:

```
java Fork live2
```

What is the result?

A. `test case`
B. `production live2`

C. `test case live2`
D. Compilation fails
E. An exception is thrown at runtime

Answer:

☑ **E** is correct. Because the short circuit `(||)` is not used, both operands are evaluated. Since `args[1]` is past the args array bounds, an `ArrayIndexOutOfBoundsException` is thrown.

☒ **A, B, C,** and **D** are incorrect based on the above. (Objective 7.6)

4. Given:

```
class Feline {
  public static void main(String[] args) {
    Long x = 42L;
    Long y = 44L;
    System.out.print(" " + 7 + 2 + " ");
    System.out.print(foo() + x + 5 + " ");
    System.out.println(x + y + foo());
  }
  static String foo() { return "foo"; }
}
```

What is the result?

A. `9 foo47 86foo`
B. `9 foo47 4244foo`
C. `9 foo425 86foo`
D. `9 foo425 4244foo`
E. `72 foo47 86foo`
F. `72 foo47 4244foo`
G. `72 foo425 86foo`
H. `72 foo425 4244foo`
I. Compilation fails

Answer:

☑ **G** is correct. Concatenation runs from left to right, and if either operand is a `String`, the operands are concatenated. If both operands are numbers they are added together. Unboxing works in conjunction with concatenation.

☒ **A, B, C, D, E, F, H,** and **I** are incorrect based on the above. (Objective 7.6)

5. Place the fragments into the code to produce the output 33. Note, you must use each fragment exactly once.

CODE:

```
class Incr {
  public static void main(String[] args) {
    Integer x = 7;
    int y = 2;

    x    ___  ___;
    ___  ___  ___;
    ___  ___  ___;
    ___  ___  ___;

    System.out.println(x);
  }
}
```

FRAGMENTS:

y	y	y	y
y	x	x	
-=	*=	*=	*=

Answer:

```
class Incr {
  public static void main(String[] args) {
    Integer x = 7;
    int y = 2;

    x *= x;
    y *= y;
    y *= y;
    x -= y;

    System.out.println(x);
  }
}
```

Yeah, we know it's kind of puzzle-y, but you might encounter something like it on the real exam. (Objective 7.6)

6. Given:

```
3. public class Twisty {
4.   { index = 1; }
5.   int index;
6.   public static void main(String[] args) {
7.     new Twisty().go();
8.   }
9.   void go() {
10.    int [][] dd = {{9,8,7}, {6,5,4}, {3,2,1,0}};
11.    System.out.println(dd[index++][index++]);
12.  }
13. }
```

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

A. 1

B. 2

C. 4

D. 6

E. 8

F. Compilation fails

G. An exception is thrown at runtime

Answer:

☑ **C** is correct. Multidimensional arrays' dimensions can be inconsistent, the code uses an initialization block, and the increment operators are both post-increment operators.

☒ **A, B, D, E, F,** and **G** are incorrect based on the above. (Objective 1.3)

7. Given:

```
3. public class McGee {
4.   public static void main(String[] args) {
5.     Days d1 = Days.TH;
6.     Days d2 = Days.M;
7.     for(Days d: Days.values()) {
8.       if(d.equals(Days.F)) break;
9.       d2 = d;
10.    }
11.    System.out.println((d1 == d2)?"same old" : "newly new");
```

```
12.   }
13.   enum Days {M, T, W, TH, F, SA, SU};
14. }
```

What is the result?

A. `same old`

B. `newly new`

C. Compilation fails due to multiple errors

D. Compilation fails due only to an error on line 7

E. Compilation fails due only to an error on line 8

F. Compilation fails due only to an error on line 11

G. Compilation fails due only to an error on line 13

Answer:

☑ **A** is correct. All of this syntax is correct. The for-each iterates through the enum using the `values()` method to return an array. Enums can be compared using either `equals()` or ==. Enums can be used in a ternary operator's Boolean test.

☒ **B, C, D, E, F,** and **G** are incorrect based on the above. (Objective 7.6)

8. Given:

```
 4. public class SpecialOps {
 5.   public static void main(String[] args) {
 6.     String s = "";
 7.     Boolean b1 = true;
 8.     Boolean b2 = false;
 9.     if((b2 = false) | (21%5) > 2) s += "x";
10.     if(b1 || (b2 = true))         s += "y";
11.     if(b2 == true)                s += "z";
12.     System.out.println(s);
13.   }
14. }
```

Which are true? (Choose all that apply.)

A. Compilation fails

B. x will be included in the output

C. y will be included in the output

D. z will be included in the output

E. An exception is thrown at runtime

Answer:

☑ **C** is correct. First of all, boxing takes care of the Boolean. Line 9 uses the modulus operator, which returns the remainder of the division, which in this case is 1. Also, line 9 sets `b2` to false, and it doesn't test `b2`'s value. Line 10 sets `b2` to true, and it doesn't test its value; however, the short circuit operator keeps the expression `b2 = true` from being executed.

☒ **A, B, D,** and **E** are incorrect based on the above. (Objective 7.6)

9. Given:

```
 3. public class Spock {
 4.   public static void main(String[] args) {
 5.     int mask = 0;
 6.     int count = 0;
 7.     if( ((5<7) || (++count < 10)) | mask++ < 10 )    mask = mask + 1;
 8.     if(  (6 > 8) ^ false)                            mask = mask + 10;
 9.     if( !(mask > 1) && ++count > 1)                  mask = mask + 100;
10.     System.out.println(mask + " " + count);
11.   }
12. }
```

Which two answers are true about the value of mask and the value of count at line 10? (Choose two.)

A. `mask` is 0

B. `mask` is 1

C. `mask` is 2

D. `mask` is 10

E. `mask` is greater than 10

F. `count` is 0

G. `count` is greater than 0

Answer:

☑ **C** and **F** are correct. At line 7 the `||` keeps `count` from being incremented, but the `|` allows `mask` to be incremented. At line 8 the ^ returns true only if exactly one operand is true. At line 9 `mask` is 2 and the `&&` keeps `count` from being incremented.

☒ **A, B, D, E,** and **G** are incorrect based on the above. (Objective 7.6)

10. Given:

```
 3. interface Vessel { }
 4. interface Toy { }
 5. class Boat implements Vessel { }
 6. class Speedboat extends Boat implements Toy { }
 7. public class Tree {
 8.   public static void main(String[] args) {
 9.     String s = "0";
10.     Boat b = new Boat();
11.     Boat b2 = new Speedboat();
12.     Speedboat s2 = new Speedboat();
13.     if((b instanceof Vessel) && (b2 instanceof Toy))  s += "1";
14.     if((s2 instanceof Vessel) && (s2 instanceof Toy)) s += "2";
15.     System.out.println(s);
16.   }
17. }
```

What is the result?

A. `0`

B. `01`

C. `02`

D. `012`

E. Compilation fails

F. An exception is thrown at runtime

Answer:

☑ **D** is correct. First, remember that `instanceof` can look up through multiple levels of an inheritance tree. Also remember that `instanceof` is commonly used before attempting a downcast, so in this case, after line 15, it would be possible to say `Speedboat s3 = (Speedboat)b2;`.

☒ **A, B, C, E,** and **F** are incorrect based on the above. (Objective 7.6)

5

Flow Control, Exceptions, and Assertions

CERTIFICATION OBJECTIVES

- Use if and switch Statements
- Develop for, do, and while Loops
  Use break and continue Statements
- Develop Code with Assertions
- Use try, catch, and finally Statements
- State the Effects of Exceptions
- Recognize Common Exceptions

✓ Two-Minute Drill

Q&A Self Test

Can you imagine trying to write code using a language that didn't give you a way to execute statements conditionally? Flow control is a key part of most any useful programming language, and Java offers several ways to do it. Some, like `if` statements and `for` loops, are common to most languages. But Java also throws in a couple of flow control features you might not have used before—exceptions and assertions.

The `if` statement and the `switch` statement are types of conditional/decision controls that allow your program to behave differently at a "fork in the road," depending on the result of a logical test. Java also provides three different looping constructs—`for`, `while`, and `do`—so you can execute the same code over and over again depending on some condition being true. Exceptions give you a clean, simple way to organize code that deals with problems that might crop up at runtime. Finally, the assertion mechanism, added to the language with version 1.4, gives you a way to do testing and debugging checks on conditions you expect to smoke out while developing, when you don't necessarily need or want the runtime overhead associated with exception handling.

With these tools, you can build a robust program that can handle any logical situation with grace. Expect to see a wide range of questions on the exam that include flow control as part of the question code, even on questions that aren't testing your knowledge of flow control.

CERTIFICATION OBJECTIVE

if and switch Statements (Exam Objective 2.1)

2.1 Develop code that implements an if or switch statement; and identify legal argument types for these statements.

The `if` and `switch` statements are commonly referred to as decision statements. When you use decision statements in your program, you're asking the program to evaluate a given expression to determine which course of action to take. We'll look at the `if` statement first.

if-else Branching

The basic format of an `if` statement is as follows:

```
if (booleanExpression) {
  System.out.println("Inside if statement");
}
```

The expression in parentheses must evaluate to (a boolean) `true` or `false`. Typically you're testing something to see if it's true, and then running a code block (one or more statements) if it is true, and (optionally) another block of code if it isn't. The following code demonstrates a legal `if-else` statement:

```
if (x > 3) {
  System.out.println("x is greater than 3");
} else {
  System.out.println("x is not greater than 3");
}
```

The `else` block is optional, so you can also use the following:

```
if (x > 3) {
  y = 2;
}
z += 8;
a = y + x;
```

The preceding code will assign 2 to y if the test succeeds (meaning x really is greater than 3), but the other two lines will execute regardless. Even the curly braces are optional if you have only one statement to execute within the body of the conditional block. The following code example is legal (although not recommended for readability):

```
if (x > 3)    // bad practice, but seen on the exam
  y = 2;
z += 8;
a = y + x;
```

Sun considers it good practice to enclose blocks within curly braces, even if there's only one statement in the block. Be careful with code like the above, because you might think it should read as,

"If x is greater than 3, then set y to 2, z to z + 8, and a to y + x."

But the last two lines are going to execute no matter what! They aren't part of the conditional flow. You might find it even more misleading if the code were indented as follows:

```
if (x > 3)
  y = 2;
  z += 8;
  a = y + x;
```

You might have a need to nest `if-else` statements (although, again, it's not recommended for readability, so nested `if` tests should be kept to a minimum). You can set up an `if-else` statement to test for multiple conditions. The following example uses two conditions so that if the first test fails, we want to perform a second test before deciding what to do:

```
if (price < 300) {
  buyProduct();
} else {
  if (price < 400) {
    getApproval();
  }
  else {
    dontBuyProduct();
  }
}
```

This brings up the other if-else construct, the `if, else if, else`. The preceding code could (and should) be rewritten:

```
if (price < 300) {
  buyProduct();
} else if (price < 400) {
    getApproval();
} else {
    dontBuyProduct();
}
```

There are a couple of rules for using `else` and `else if`:

- You can have zero or one `else` for a given `if`, and it must come after any `else ifs`.
- You can have zero to many `else ifs` for a given `if` and they must come before the (optional) `else`.
- Once an `else if` succeeds, none of the remaining `else ifs` or `elses` will be tested.

The following example shows code that is horribly formatted for the real world. As you've probably guessed, it's fairly likely that you'll encounter formatting like this on the exam. In any case, the code demonstrates the use of multiple `else ifs`:

```
int x = 1;
if ( x == 3 ) { }
else if (x < 4) {System.out.println("<4"); }
else if (x < 2) {System.out.println("<2"); }
else { System.out.println("else"); }
```

It produces the output:

```
<4
```

(Notice that even though the second `else if` is true, it is never reached.) Sometimes you can have a problem figuring out which `if` your `else` should pair with, as follows:

```
if (exam.done())
if (exam.getScore() < 0.61)
System.out.println("Try again.");
// Which if does this belong to?
else System.out.println("Java master!");
```

We intentionally left out the indenting in this piece of code so it doesn't give clues as to which `if` statement the `else` belongs to. Did you figure it out? Java law decrees that an `else` clause belongs to the innermost `if` statement to which it might possibly belong (in other words, the closest preceding `if` that doesn't have an `else`). In the case of the preceding example, the `else` belongs to the second `if` statement in the listing. With proper indenting, it would look like this:

```
if (exam.done())
  if (exam.getScore() < 0.61)
    System.out.println("Try again.");
  // Which if does this belong to?
  else
    System.out.println("Java master!");
```

Following our coding conventions by using curly braces, it would be even easier to read:

```
if (exam.done()) {
  if (exam.getScore() < 0.61) {
    System.out.println("Try again.");
  // Which if does this belong to?
  } else {
    System.out.println("Java master!");
  }
}
```

Don't get your hopes up about the exam questions being all nice and indented properly. Some exam takers even have a slogan for the way questions are presented on the exam: anything that can be made more confusing, will be.

Be prepared for questions that not only fail to indent nicely, but intentionally indent in a misleading way: Pay close attention for misdirection like the following:

```
if (exam.done())
  if (exam.getScore() < 0.61)
    System.out.println("Try again.");
else
  System.out.println("Java master!"); // Hmmmmm… now where does
                                       // it belong?
```

Of course, the preceding code is exactly the same as the previous two examples, except for the way it looks.

Legal Expressions for if Statements

The expression in an `if` statement must be a `boolean` expression. Any expression that resolves to a `boolean` is fine, and some of the expressions can be complex. Assume `doStuff()` returns `true`,

```
int y = 5;
int x = 2;
```

```
if (((x > 3) && (y < 2)) | doStuff()) {
   System.out.println("true");
}
```

which prints

```
true
```

You can read the preceding code as, "If both (x > 3) and (y < 2) are `true`, or if the result of `doStuff()` is `true`, then print `true`." So basically, if just `doStuff()` alone is `true`, we'll still get `true`. If `doStuff()` is `false`, though, then both (x > 3) and (y < 2) will have to be `true` in order to print `true`. The preceding code is even more complex if you leave off one set of parentheses as follows,

```
int y = 5;
int x = 2;
if ((x > 3) && (y < 2) | doStuff()) {
  System.out.println("true");
}
```

which now prints…nothing! Because the preceding code (with one less set of parentheses) evaluates as though you were saying, "If (x > 3) is `true`, and either (y < 2) or the result of `doStuff()` is `true`, then print `true`." So if (x > 3) is not `true`, no point in looking at the rest of the expression." Because of the short-circuit `&&`, the expression is evaluated as though there were parentheses around `(y < 2) | doStuff()`. In other words, it is evaluated as a single expression before the `&&` and a single expression after the `&&`.

Remember that the only legal expression in an `if` test is a boolean. In some languages, 0 == false, and 1 == true. Not so in Java! The following code shows `if` statements that might look tempting, but are illegal, followed by legal substitutions:

```
int trueInt = 1;
int falseInt = 0;
if (trueInt)              // illegal
if (trueInt == true)      // illegal
if (1)                    // illegal
if (falseInt == false)    // illegal
if (trueInt == 1)         // legal
if (falseInt == 0)        // legal
```

exam Watch

One common mistake programmers make (and that can be difficult to spot), is assigning a boolean variable when you meant to test a boolean variable. Look out for code like the following:

```
boolean boo = false;
if (boo = true) { }
```

You might think one of three things:

1. ***The code compiles and runs fine, and the*** `if` ***test fails because boo is*** `false`.
2. ***The code won't compile because you're using an assignment (=) rather than an equality test (==).***
3. ***The code compiles and runs fine and the*** `if` ***test succeeds because boo is SET to*** `true` ***(rather than TESTED for*** `true`***) in the*** `if` ***argument!***

Well, number 3 is correct. Pointless, but correct. Given that the result of any assignment is the value of the variable after the assignment, the expression (`boo = true` ***) has a result of*** `true`***. Hence, the*** `if` ***test succeeds. But the only variables that can be assigned (rather than tested against something else) are a*** `boolean` ***or a*** `Boolean`***; all other assignments will result in something non-boolean, so they're not legal, as in the following:***

```
int x = 3;
if (x = 5) { }  // Won't compile because x is not a boolean!
```

Because `if` ***tests require boolean expressions, you need to be really solid on both logical operators and*** `if` ***test syntax and semantics.***

switch Statements

A way to simulate the use of multiple `if` statements is with the `switch` statement. Take a look at the following `if-else` code, and notice how confusing it can be to have nested `if` tests, even just a few levels deep:

```
int x = 3;
if(x == 1) {
```

```
  System.out.println("x equals 1");
}
else if(x == 2) {
      System.out.println("x equals 2");
   }
   else if(x == 3) {
         System.out.println("x equals 3");
      }
      else {
         System.out.println("No idea what x is");
      }
```

Now let's see the same functionality represented in a `switch` construct:

```
int x = 3;
switch (x) {
   case 1:
      System.out.println("x is equal to 1");
      break;
   case 2:
      System.out.println("x is equal to 2");
      break;
   case 3:
      System.out.println("x is equal to 3");
      break;
   default:
      System.out.println("Still no idea what x is");
}
```

Note: The reason this `switch` statement emulates the nested `if`s listed earlier is because of the `break` statements that were placed inside of the `switch`. In general, `break` statements are optional, and as we will see in a few pages, their inclusion or exclusion causes huge changes in how a `switch` statement will execute.

Legal Expressions for switch and case

The general form of the `switch` statement is:

```
switch (expression) {
  case constant1: code block
  case constant2: code block
  default: code block
}
```

A switch's expression must evaluate to a char, byte, short, int, or, as of Java 6, an enum. That means if you're not using an enum, only variables and values that can be automatically promoted (in other words, implicitly cast) to an int are acceptable. You won't be able to compile if you use anything else, including the remaining numeric types of long, float, and double.

A case constant must evaluate to the same type as the switch expression can use, with one additional—and big—constraint: the case constant must be a compile time constant! Since the case argument has to be resolved at compile time, that means you can use only a constant or final variable that is assigned a literal value. It is not enough to be final, it must be a compile time *constant*. For example:

```
final int a = 1;
final int b;
b = 2;
int x = 0;
switch (x) {
  case a:     // ok
  case b:     // compiler error
```

Also, the switch can only check for equality. This means that the other relational operators such as greater than are rendered unusable in a case. The following is an example of a valid expression using a method invocation in a switch statement. Note that for this code to be legal, the method being invoked on the object reference must return a value compatible with an int.

```
String s = "xyz";
switch (s.length()) {
  case 1:
    System.out.println("length is one");
    break;
  case 2:
    System.out.println("length is two");
    break;
  case 3:
    System.out.println("length is three");
    break;
  default:
    System.out.println("no match");
}
```

One other rule you might not expect involves the question, "What happens if I `switch` on a variable smaller than an `int`?" Look at the following `switch`:

```
byte g = 2;
switch(g) {
  case 23:
  case 128:
}
```

This code won't compile. Although the `switch` argument is legal—a byte is implicitly cast to an `int`—the second `case` argument (128) is too large for a `byte`, and the compiler knows it! Attempting to compile the preceding example gives you an error something like

```
Test.java:6: possible loss of precision
found   : int
required: byte
    case 128:
         ^
```

It's also illegal to have more than one `case` label using the same value. For example, the following block of code won't compile because it uses two cases with the same value of 80:

```
int temp = 90;
switch(temp) {
  case 80 :  System.out.println("80");
  case 80 :  System.out.println("80");   // won't compile!
  case 90 :  System.out.println("90");
  default :  System.out.println("default");
}
```

It *is* legal to leverage the power of boxing in a `switch` expression. For instance, the following is legal:

```
switch(new Integer(4)) {
  case 4: System.out.println("boxing is OK");
}
```

exam Watch

Look for any violation of the rules for `switch` ***and*** `case` ***arguments. For example, you might find illegal examples like the following snippets:***

```
switch(x) {
  case 0 {
    y = 7;
  }
}

switch(x) {
  0: { }
  1: { }
}
```

In the first example, the `case` ***uses a curly brace and omits the colon. The second example omits the keyword*** `case`***.***

Break and Fall-Through in switch Blocks

We're finally ready to discuss the `break` statement, and more details about flow control within a `switch` statement. The most important thing to remember about the flow of execution through a `switch` statement is this:

> `case` **constants are evaluated from the top down, and the first** `case` **constant that matches the** `switch`**'s expression is the execution *entry point*.**

In other words, once a `case` constant is matched, the JVM will execute the associated code block, and ALL subsequent code blocks (barring a `break` statement) too! The following example uses an `enum` in a `case` statement.

```
enum Color {red, green, blue}
class SwitchEnum {
  public static void main(String [] args) {
    Color c = Color.green;
    switch(c) {
```

```
        case red: System.out.print("red ");
        case green: System.out.print("green ");
        case blue: System.out.print("blue ");
        default: System.out.println("done");
      }
    }
  }
```

In this example `case green:` matched, so the JVM executed that code block and all subsequent code blocks to produce the output:

```
green blue done
```

Again, when the program encounters the keyword `break` during the execution of a `switch` statement, execution will immediately move out of the `switch` block to the next statement after the `switch`. If `break` is omitted, the program just keeps executing the remaining `case` blocks until either a `break` is found or the `switch` statement ends. Examine the following code:

```
int x = 1;
switch(x) {
  case 1:  System.out.println("x is one");
  case 2:  System.out.println("x is two");
  case 3:  System.out.println("x is three");
}
System.out.println("out of the switch");
```

The code will print the following:

```
x is one
x is two
x is three
out of the switch
```

This combination occurs because the code didn't hit a `break` statement; execution just kept dropping down through each `case` until the end. This dropping down is actually called "fall-through," because of the way execution falls from one `case` to the next. Remember, the matching `case` is simply your entry point into the `switch` block! In other words, you must *not* think of it as, "Find the matching `case`, execute just that code, and get out." That's *not* how it works. If you do want that "just the matching code" behavior, you'll insert a `break` into each `case` as follows:

```
int x = 1;
switch(x) {
  case 1:  {
    System.out.println("x is one");  break;
  }
  case 2:  {
    System.out.println("x is two");  break;
  }
  case 3:  {
    System.out.println("x is two");  break;
  }
}
System.out.println("out of the switch");
```

Running the preceding code, now that we've added the `break` statements, will print

```
x is one
out of the switch
```

and that's it. We entered into the `switch` block at `case 1`. Because it matched the `switch()` argument, we got the `println` statement, then hit the `break` and jumped to the end of the `switch`.

An interesting example of this fall-through logic is shown in the following code:

```
int x = someNumberBetweenOneAndTen;

switch (x) {
  case 2:
  case 4:
  case 6:
  case 8:
  case 10: {
    System.out.println("x is an even number");  break;
  }
}
```

This `switch` statement will print `x is an even number` or nothing, depending on whether the number is between one and ten and is odd or even. For example, if x is 4, execution will begin at `case 4`, but then fall down through `6`, `8`, and `10`, where it prints and then breaks. The `break` at `case 10`, by the way, is not needed; we're already at the end of the `switch` anyway.

Note: Because fall-through is less than intuitive, Sun recommends that you add a comment like: `// fall through` when you use fall-through logic.

The Default Case

What if, using the preceding code, you wanted to print "x is an odd number" if none of the cases (the even numbers) matched? You couldn't put it after the `switch` statement, or even as the last `case` in the `switch`, because in both of those situations it would always print `x is an odd number`. To get this behavior, you'll use the `default` keyword. (By the way, if you've wondered why there is a `default` keyword even though we don't use a modifier for default access control, now you'll see that the `default` keyword is used for a completely different purpose.) The only change we need to make is to add the default case to the preceding code:

```
int x = someNumberBetweenOneAndTen;

switch (x) {
  case 2:
  case 4:
  case 6:
  case 8:
  case 10: {
    System.out.println("x is an even number");
    break;
  }
  default: System.out.println("x is an odd number");
}
```

exam Watch

The `default` case doesn't have to come at the end of the `switch`. Look for it in strange places such as the following:

```
int x = 2;
switch (x) {
  case 2:  System.out.println("2");
  default: System.out.println("default");
  case 3: System.out.println("3");
  case 4: System.out.println("4");
}
```

exam Watch

Running the preceding code prints

```
2
default
3
4
```

And if we modify it so that the only match is the default case:

```
int x = 7;
switch (x) {
  case 2:  System.out.println("2");
  default: System.out.println("default");
  case 3: System.out.println("3");
  case 4: System.out.println("4");
}
```

Running the preceding code prints

```
default
3
4
```

The rule to remember is that `default` ***works just like any other*** `case` ***for fall-through!***

EXERCISE 5-1

Creating a switch-case Statement

Try creating a `switch` statement using a `char` value as the case. Include a default behavior if none of the `char` values match.

- Make sure a `char` variable is declared before the `switch` statement.
- Each `case` statement should be followed by a `break`.
- The `default` case can be located at the end, middle, or top.

CERTIFICATION OBJECTIVE

Loops and Iterators (Exam Objective 2.2)

2.2 Develop code that implements all forms of loops and iterators, including the use of for, the enhanced for loop (for-each), do, while, labels, break, and continue; and explain the values taken by loop counter variables during and after loop execution.

Java loops come in three flavors: `while`, `do`, and `for` (and as of Java 6, the `for` loop has two variations). All three let you repeat a block of code as long as some condition is true, or for a specific number of iterations. You're probably familiar with loops from other languages, so even if you're somewhat new to Java, these won't be a problem to learn.

Using while Loops

The `while` loop is good for scenarios where you don't know how many times a block or statement should repeat, but you want to continue looping as long as some condition is true. A `while` statement looks like this:

```
while (expression) {
  // do stuff
}
```

or

```
int x = 2;
while(x == 2) {
  System.out.println(x);
  ++x;
}
```

In this case, as in all loops, the expression (test) must evaluate to a `boolean` result. The body of the `while` loop will only execute if the expression (sometimes called the "condition") results in a value of `true`. Once inside the loop, the loop body will repeat until the condition is no longer met because it evaluates to `false`. In the previous example, program control will enter the loop body because x is equal to 2. However, x is incremented in the loop, so when the condition is checked again it will evaluate to `false` and exit the loop.

Any variables used in the expression of a `while` loop must be declared before the expression is evaluated. In other words, you can't say

```
while (int x = 2) { }   // not legal
```

Then again, why would you? Instead of testing the variable, you'd be declaring and initializing it, so it would always have the exact same value. Not much of a test condition!

The key point to remember about a `while` loop is that it might not ever run. If the test expression is `false` the first time the `while` expression is checked, the loop body will be skipped and the program will begin executing at the first statement *after* the `while` loop. Look at the following example:

```
int x = 8;
while (x > 8) {
  System.out.println("in the loop");
  x = 10;
}
System.out.println("past the loop");
```

Running this code produces

```
past the loop
```

Because the expression `(x > 8)` evaluates to `false`, none of the code within the `while` loop ever executes.

Using do Loops

The `do` loop is similar to the `while` loop, except that the expression is not evaluated until after the `do` loop's code is executed. Therefore the code in a `do` loop is guaranteed to execute at least once. The following shows a `do` loop in action:

```
do {
   System.out.println("Inside loop");
} while(false);
```

The `System.out.println()` statement will print once, even though the expression evaluates to `false`. Remember, the `do` loop will always run the code in the loop body at least once. Be sure to note the use of the semicolon at the end of the `while` expression.

exam Watch

As with `if` tests, look for `while` loops (and the `while` test in a `do` loop) with an expression that does not resolve to a `boolean`. Take a look at the following examples of legal and illegal `while` expressions:

```
int x = 1;
while (x) { }        // Won't compile; x is not a boolean
while (x = 5) { }    // Won't compile; resolves to 5
                     //(as the result of assignment)
while (x == 5) { }   // Legal, equality test
while (true) { }     // Legal
```

Using for Loops

As of Java 6, the `for` loop took on a second structure. We'll call the old style of `for` loop the "basic `for` loop", and we'll call the new style of `for` loop the "enhanced `for` loop" (even though the Sun objective 2.2 refers to it as the `for-each`). Depending on what documentation you use (Sun's included), you'll see both terms, along with `for-in`. The terms `for-in`, `for-each`, and "enhanced `for`" all refer to the same Java construct.

The basic `for` loop is more flexible than the enhanced `for` loop, but the enhanced `for` loop was designed to make iterating through arrays and collections easier to code.

The Basic for Loop

The `for` loop is especially useful for flow control when you already know how many times you need to execute the statements in the loop's block. The `for` loop declaration has three main parts, besides the body of the loop:

- Declaration and initialization of variables
- The `boolean` expression (conditional test)
- The iteration expression

The three `for` declaration parts are separated by semicolons. The following two examples demonstrate the `for` loop. The first example shows the parts of a `for` loop in a pseudocode form, and the second shows a typical example of a `for` loop.

```
for (/*Initialization*/ ; /*Condition*/ ;  /* Iteration */) {
  /* loop body */
}

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

The Basic for Loop: Declaration and Initialization

The first part of the `for` statement lets you declare and initialize zero, one, or multiple variables of the same type inside the parentheses after the `for` keyword. If you declare more than one variable of the same type, then you'll need to separate them with commas as follows:

```
for (int x = 10, y = 3; y > 3; y++) { }
```

The declaration and initialization happens before anything else in a `for` loop. And whereas the other two parts—the boolean test and the iteration expression—will run with each iteration of the loop, the declaration and initialization happens just once, at the very beginning. You also must know that the scope of variables declared in the `for` loop ends with the `for` loop! The following demonstrates this:

```
for (int x = 1; x < 2; x++) {
  System.out.println(x);  // Legal
}
System.out.println(x);  // Not Legal! x is now out of scope
                        // and can't be accessed.
```

If you try to compile this, you'll get something like this:

```
Test.java:19: cannot resolve symbol
symbol  : variable x
location: class Test
  System.out.println(x);
                     ^
```

Basic for Loop: Conditional (boolean) Expression

The next section that executes is the conditional expression, which (like all other conditional tests) must evaluate to a `boolean` value. You can have only one

logical expression, but it can be very complex. Look out for code that uses logical expressions like this:

```
for (int x = 0; ((((x < 10) && (y-- > 2)) | x == 3)); x++) { }
```

The preceding code is legal, but the following is not:

```
for (int x = 0; (x > 5), (y < 2); x++) { } // too many
                                           //expressions
```

The compiler will let you know the problem:

```
TestLong.java:20: ';' expected
for (int x = 0; (x > 5), (y < 2); x++) { }
                       ^
```

The rule to remember is this: *You can have only one test expression.*

In other words, you can't use multiple tests separated by commas, even though the other two parts of a `for` statement can have multiple parts.

Basic for Loop: Iteration Expression

After each execution of the body of the `for` loop, the iteration expression is executed. This is where you get to say what you want to happen with each iteration of the loop. Remember that it always happens after the loop body runs! Look at the following:

```
for (int x = 0; x < 1; x++) {
  // body code that doesn't change the value of x
}
```

The preceding loop executes just once. The first time into the loop x is set to 0, then x is tested to see if it's less than 1 (which it is), and then the body of the loop executes. After the body of the loop runs, the iteration expression runs, incrementing x by 1. Next, the conditional test is checked, and since the result is now `false`, execution jumps to below the `for` loop and continues on.

Keep in mind that barring a forced exit, evaluating the iteration expression and then evaluating the conditional expression are always the last two things that happen in a `for` loop!

Examples of forced exits include a `break`, a return, a `System.exit()`, or an exception, which will all cause a loop to terminate abruptly, without running the iteration expression. Look at the following code:

```
static boolean doStuff() {
  for (int x = 0; x < 3; x++) {
    System.out.println("in for loop");
    return true;
  }
  return true;
}
```

Running this code produces

```
in for loop
```

The statement only prints once, because a return causes execution to leave not just the current iteration of a loop, but the entire method. So the iteration expression never runs in that case. Table 5-1 lists the causes and results of abrupt loop termination.

TABLE 5-1 Causes of Early Loop Termination

Code in Loop	What Happens
`break`	Execution jumps immediately to the 1st statement after the `for` loop.
`return`	Execution jumps immediately back to the calling method.
`System.exit()`	All program execution stops; the VM shuts down.

Basic for Loop: for Loop Issues

None of the three sections of the `for` declaration are required! The following example is perfectly legal (although not necessarily good practice):

```
for( ; ; ) {
  System.out.println("Inside an endless loop");
}
```

In the preceding example, all the declaration parts are left out so the `for` loop will act like an endless loop. For the exam, it's important to know that with the absence

of the initialization and increment sections, the loop will act like a `while` loop. The following example demonstrates how this is accomplished:

```
int i = 0;

for (;i<10;) {
  i++;
  //do some other work
}
```

The next example demonstrates a `for` loop with multiple variables in play. A comma separates the variables, and they must be of the same type. Remember that the variables declared in the `for` statement are all local to the `for` loop, and can't be used outside the scope of the loop.

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

exam Watch

Variable scope plays a large role in the exam. You need to know that a variable declared in the `for` loop can't be used beyond the `for` loop. But a variable only initialized in the `for` statement (but declared earlier) can be used beyond the loop. For example, the following is legal,

```
int x = 3;
for (x = 12; x < 20; x++) { }
System.out.println(x);
```

while this is not

```
for (int x = 3; x < 20; x++) { } System.out.println(x);
```

The last thing to note is that all three sections of the `for` loop are independent of each other. The three expressions in the `for` statement don't need to operate on the same variables, although they typically do. But even the iterator expression, which

many mistakenly call the "increment expression," doesn't need to increment or set anything; you can put in virtually any arbitrary code statements that you want to happen with each iteration of the loop. Look at the following:

```
int b = 3;
for (int a = 1;  b != 1; System.out.println("iterate")) {
  b = b - a;
}
```

The preceding code prints

```
iterate
iterate
```

exam Watch

Many questions in the new (Java 6) exam list "Compilation fails" and "An exception occurs at runtime" as possible answers. This makes it more difficult because you can't simply work through the behavior of the code. You must first make sure the code isn't violating any fundamental rules that will lead to compiler error, and then look for possible exceptions. Only after you've satisfied those two, should you dig into the logic and flow of the code in the question.

The Enhanced for Loop (for Arrays)

The enhanced `for` loop, new to Java 6, is a specialized `for` loop that simplifies looping through an array or a collection. In this chapter we're going to focus on using the enhanced `for` to loop through arrays. In Chapter 7 we'll revisit the enhanced `for` as we discuss collections—where the enhanced `for` really comes into its own.

Instead of having *three* components, the enhanced `for` has *two*. Let's loop through an array the basic (old) way, and then using the enhanced `for`:

```
int [] a = {1,2,3,4};
for(int x = 0; x < a.length; x++)    // basic for loop
  System.out.print(a[x]);
for(int n : a)                       // enhanced for loop
  System.out.print(n);
```

Which produces this output:

```
12341234
```

More formally, let's describe the enhanced `for` as follows:

```
for(declaration : expression)
```

The two pieces of the `for` statement are

- **`declaration`** The *newly declared* block variable, of a type compatible with the elements of the array you are accessing. This variable will be available within the `for` block, and its value will be the same as the current array element.
- **`expression`** This must evaluate to the array you want to loop through. This could be an array variable or a method call that returns an array. The array can be any type: primitives, objects, even arrays of arrays.

Using the above definitions, let's look at some legal and illegal enhanced `for` declarations:

```
int x;
long x2;
Long [] La = {4L, 5L, 6L};
long [] la = {7L, 8L, 9L};
int [][] twoDee = {{1,2,3}, {4,5,6}, {7,8,9}};
String [] sNums = {"one", "two", "three"};
Animal [] animals = {new Dog(), new Cat()};

// legal 'for' declarations
for(long y : la ) ;         // loop thru an array of longs
for(long lp : La) ;         // autoboxing the Long objects
                            // into longs
for(int[] n : twoDee) ;     // loop thru the array of arrays
for(int n2 : twoDee[2]) ;   // loop thru the 3rd sub-array
for(String s : sNums) ;     // loop thru the array of Strings
for(Object o : sNums) ;     // set an Object reference to
                            // each String
for(Animal a : animals) ;   // set an Animal reference to each
                            // element
```

```
// ILLEGAL 'for' declarations
for(x2 : la) ;              // x2 is already declared
for(int x2 : twoDee) ;      // can't stuff an array into an int
for(int x3 : la) ;          // can't stuff a long into an int
for(Dog d : animals) ;      // you might get a Cat!
```

The enhanced `for` loop assumes that, barring an early exit from the loop, you'll always loop through every element of the array. The following discussions of `break` and `continue` apply to both the basic and enhanced `for` loops.

Using break and continue

The `break` and `continue` keywords are used to stop either the entire loop (`break`) or just the current iteration (`continue`). Typically if you're using `break` or `continue`, you'll do an `if` test within the loop, and if some condition becomes `true` (or `false` depending on the program), you want to get out immediately. The difference between them is whether or not you continue with a new iteration or jump to the first statement below the loop and continue from there.

exam Watch

Remember, `continue` ***statements must be inside a loop; otherwise, you'll get a compiler error.*** `break` ***statements must be used inside either a loop or*** `switch` ***statement. (Note: this does not apply to labeled*** `break` ***statements.).***

The `break` statement causes the program to stop execution of the innermost loop and start processing the next line of code after the block.

The `continue` statement causes only the current iteration of the innermost loop to cease and the next iteration of the same loop to start if the condition of the loop is met. When using a `continue` statement with a `for` loop, you need to consider the effects that `continue` has on the loop iteration. Examine the following code:

```
for (int i = 0; i < 10; i++) {
  System.out.println("Inside loop");
  continue;
}
```

The question is, is this an endless loop? The answer is no. When the `continue` statement is hit, the iteration expression still runs! It runs just as though the current

iteration ended "in the natural way." So in the preceding example, `i` will still increment before the condition `(i < 10)` is checked again. Most of the time, a `continue` is used within an `if` test as follows:

```
for (int i = 0; i < 10; i++) {
  System.out.println("Inside loop");
  if (foo.doStuff() == 5) {
    continue;
  }
  // more loop code, that won't be reached when the above if
  // test is true
}
```

Unlabeled Statements

Both the `break` statement and the `continue` statement can be unlabeled or labeled. Although it's far more common to use `break` and `continue` unlabeled, the exam expects you to know how labeled `break` and `continue` statements work. As stated before, a `break` statement (unlabeled) will exit out of the innermost looping construct and proceed with the next line of code beyond the loop block. The following example demonstrates a `break` statement:

```
boolean problem = true;
while (true) {
  if (problem) {
    System.out.println("There was a problem");
    break;
  }
}
// next line of code
```

In the previous example, the `break` statement is unlabeled. The following is an example of an unlabeled `continue` statement:

```
while (!EOF) {
  //read a field from a file
  if (wrongField) {
    continue;    // move to the next field in the file
  }
  // otherwise do other stuff with the field
}
```

In this example, a file is being read one field at a time. When an error is encountered, the program moves to the next field in the file and uses the `continue` statement to go back into the loop (if it is not at the end of the file) and keeps reading the various fields. If the `break` command were used instead, the code would stop reading the file once the error occurred and move on to the next line of code after the loop. The `continue` statement gives you a way to say, "This particular iteration of the loop needs to stop, but not the whole loop itself. I just don't want the rest of the code in this iteration to finish, so do the iteration expression and then start over with the test, and don't worry about what was below the `continue` statement."

Labeled Statements

Although many statements in a Java program can be labeled, it's most common to use labels with loop statements like `for` or `while`, in conjunction with `break` and `continue` statements. A label statement must be placed just before the statement being labeled, and it consists of a valid identifier that ends with a colon (:).

You need to understand the difference between labeled and unlabeled `break` and `continue`. The labeled varieties are needed only in situations where you have a nested loop, and need to indicate which of the nested loops you want to break from, or from which of the nested loops you want to continue with the next iteration. A `break` statement will exit out of the labeled loop, as opposed to the innermost loop, if the `break` keyword is combined with a label. An example of what a label looks like is in the following code:

```
foo:
  for (int x = 3; x < 20; x++) {
    while(y > 7) {
      y--;
    }
  }
```

The label must adhere to the rules for a valid variable name and should adhere to the Java naming convention. The syntax for the use of a label name in conjunction with a `break` statement is the `break` keyword, then the label name, followed by a semicolon. A more complete example of the use of a labeled `break` statement is as follows:

```
boolean isTrue = true;
outer:
  for(int i=0; i<5; i++) {
    while (isTrue) {
```

```
        System.out.println("Hello");
        break outer;
      } // end of inner while loop
      System.out.println("Outer loop."); // Won't print
    } // end of outer for loop
  System.out.println("Good-Bye");
```

Running this code produces

```
Hello
Good-Bye
```

In this example the word `Hello` will be printed one time. Then, the labeled `break` statement will be executed, and the flow will exit out of the loop labeled `outer`. The next line of code will then print out `Good-Bye`. Let's see what will happen if the `continue` statement is used instead of the `break` statement. The following code example is similar to the preceding one, with the exception of substituting `continue` for `break`:

```
outer:
  for (int i=0; i<5; i++) {
    for (int j=0; j<5; j++) {
      System.out.println("Hello");
      continue outer;
    } // end of inner loop
    System.out.println("outer"); // Never prints
  }
System.out.println("Good-Bye");
```

Running this code produces

```
Hello
Hello
Hello
Hello
Hello
Good-Bye
```

In this example, `Hello` will be printed five times. After the `continue` statement is executed, the flow continues with the next iteration of the loop identified with the label. Finally, when the condition in the outer loop evaluates to `false`, this loop will finish and `Good-Bye` will be printed.

EXERCISE 5-2

Creating a Labeled while Loop

Try creating a labeled `while` loop. Make the label `outer` and provide a condition to check whether a variable `age` is less than or equal to 21. Within the loop, increment `age` by one. Every time the program goes through the loop, check whether `age` is 16. If it is, print the message "get your driver's license" and continue to the outer loop. If not, print "Another year."

- The `outer` label should appear just before the `while` loop begins.
- Make sure `age` is declared outside of the `while` loop.

exam Watch

Labeled `continue` ***and*** `break` ***statements must be inside the loop that has the same label name; otherwise, the code will not compile.***

CERTIFICATION OBJECTIVE

Handling Exceptions (Exam Objectives 2.4 and 2.5)

2.4 Develop code that makes use of exceptions and exception handling clauses (try, catch, finally), and declares methods and overriding methods that throw exceptions.

2.5 Recognize the effect of an exception arising at a specific point in a code fragment. Note that the exception may be a runtime exception, a checked exception, or an error.

An old maxim in software development says that 80 percent of the work is used 20 percent of the time. The 80 percent refers to the effort required to check and handle errors. In many languages, writing program code that checks for and deals with errors is tedious and bloats the application source into confusing spaghetti.

Still, error detection and handling may be the most important ingredient of any robust application. Java arms developers with an elegant mechanism for handling errors that produces efficient and organized error-handling code: exception handling.

Exception handling allows developers to detect errors easily without writing special code to test return values. Even better, it lets us keep exception-*handling* code cleanly separated from the exception-*generating* code. It also lets us use the same exception-handling code to deal with a range of possible exceptions.

The exam has three objectives covering exception handling. We'll cover the first two in this section, and in the next section we'll cover those aspects of exception handling that are new to the exam as of Java 6.

Catching an Exception Using try and catch

Before we begin, let's introduce some terminology. The term "exception" means "exceptional condition" and is an occurrence that alters the normal program flow. A bunch of things can lead to exceptions, including hardware failures, resource exhaustion, and good old bugs. When an exceptional event occurs in Java, an exception is said to be "thrown." The code that's responsible for doing something about the exception is called an "exception handler," and it "catches" the thrown exception.

Exception handling works by transferring the execution of a program to an appropriate exception handler when an exception occurs. For example, if you call a method that opens a file but the file cannot be opened, execution of that method will stop, and code that you wrote to deal with this situation will be run. Therefore, we need a way to tell the JVM what code to execute when a certain exception happens. To do this, we use the `try` and `catch` keywords. The `try` is used to define a block of code in which exceptions may occur. This block of code is called a guarded region (which really means "risky code goes here"). One or more `catch` clauses match a specific exception (or group of exceptions—more on that later) to a block of code that handles it. Here's how it looks in pseudocode:

```
1. try {
2.   // This is the first line of the "guarded region"
3.   // that is governed by the try keyword.
4.   // Put code here that might cause some kind of exception.
5.   // We may have many code lines here or just one.
6. }
7. catch(MyFirstException) {
8.   // Put code here that handles this exception.
```

```
 9.   // This is the next line of the exception handler.
10.   // This is the last line of the exception handler.
11. }
12. catch(MySecondException) {
13.   // Put code here that handles this exception
14. }
15.
16. // Some other unguarded (normal, non-risky) code begins here
```

In this pseudocode example, lines 2 through 5 constitute the guarded region that is governed by the `try` clause. Line 7 is an exception handler for an exception of type MyFirstException. Line 12 is an exception handler for an exception of type MySecondException. Notice that the `catch` blocks immediately follow the `try` block. This is a requirement; if you have one or more `catch` blocks, they must immediately follow the `try` block. Additionally, the `catch` blocks must all follow each other, without any other statements or blocks in between. Also, the order in which the `catch` blocks appear matters, as we'll see a little later.

Execution of the guarded region starts at line 2. If the program executes all the way past line 5 with no exceptions being thrown, execution will transfer to line 15 and continue downward. However, if at any time in lines 2 through 5 (the `try` block) an exception is thrown of type MyFirstException, execution will immediately transfer to line 7. Lines 8 through 10 will then be executed so that the entire `catch` block runs, and then execution will transfer to line 15 and continue.

Note that if an exception occurred on, say, line 3 of the `try` block, the rest of the lines in the `try` block (4 and 5) would never be executed. Once control jumps to the `catch` block, it never returns to complete the balance of the `try` block. This is exactly what you want, though. Imagine your code looks something like this pseudocode:

```
try {
  getTheFileFromOverNetwork
  readFromTheFileAndPopulateTable
}
catch(CantGetFileFromNetwork) {
  displayNetworkErrorMessage
}
```

The preceding pseudocode demonstrates how you typically work with exceptions. Code that's dependent on a risky operation (as populating a table with file data is dependent on getting the file from the network) is grouped into a `try` block in such

a way that if, say, the first operation fails, you won't continue trying to run other code that's also guaranteed to fail. In the pseudocode example, you won't be able to read from the file if you can't get the file off the network in the first place.

One of the benefits of using exception handling is that code to handle any particular exception that may occur in the governed region needs to be written only once. Returning to our earlier code example, there may be three different places in our `try` block that can generate a MyFirstException, but wherever it occurs it will be handled by the same `catch` block (on line 7). We'll discuss more benefits of exception handling near the end of this chapter.

Using finally

Although `try` and `catch` provide a terrific mechanism for trapping and handling exceptions, we are left with the problem of how to clean up after ourselves if an exception occurs. Because execution transfers out of the `try` block as soon as an exception is thrown, we can't put our cleanup code at the bottom of the `try` block and expect it to be executed if an exception occurs. Almost as bad an idea would be placing our cleanup code in each of the `catch` blocks—let's see why.

Exception handlers are a poor place to clean up after the code in the `try` block because each handler then requires its own copy of the cleanup code. If, for example, you allocated a network socket or opened a file somewhere in the guarded region, each exception handler would have to close the file or release the socket. That would make it too easy to forget to do cleanup, and also lead to a lot of redundant code. To address this problem, Java offers the `finally` block.

A `finally` block encloses code that is always executed at some point after the `try` block, whether an exception was thrown or not. Even if there is a `return` statement in the `try` block, the `finally` block executes right after the `return` statement is encountered, and before the `return` executes!

This is the right place to close your files, release your network sockets, and perform any other cleanup your code requires. If the `try` block executes with no exceptions, the `finally` block is executed immediately after the `try` block completes. If there was an exception thrown, the `finally` block executes immediately after the proper `catch` block completes. Let's look at another pseudocode example:

```
1: try {
2:   // This is the first line of the "guarded region".
3: }
4: catch(MyFirstException) {
```

```
 5:   // Put code here that handles this exception
 6: }
 7: catch(MySecondException) {
 8:   // Put code here that handles this exception
 9: }
10: finally {
11:   // Put code here to release any resource we
12:   // allocated in the try clause.
13: }
14:
15: // More code here
```

As before, execution starts at the first line of the `try` block, line 2. If there are no exceptions thrown in the `try` block, execution transfers to line 11, the first line of the `finally` block. On the other hand, if a `MySecondException` is thrown while the code in the `try` block is executing, execution transfers to the first line of that exception handler, line 8 in the `catch` clause. After all the code in the `catch` clause is executed, the program moves to line 11, the first line of the `finally` clause. Repeat after me: `finally` always runs! OK, we'll have to refine that a little, but for now, start burning in the idea that `finally` always runs. If an exception is thrown, `finally` runs. If an exception is not thrown, `finally` runs. If the exception is caught, `finally` runs. If the exception is not caught, `finally` runs. Later we'll look at the few scenarios in which `finally` might not run or complete.

Remember, `finally` clauses are not required. If you don't write one, your code will compile and run just fine. In fact, if you have no resources to clean up after your `try` block completes, you probably don't need a `finally` clause. Also, because the compiler doesn't even require `catch` clauses, sometimes you'll run across code that has a `try` block immediately followed by a `finally` block. Such code is useful when the exception is going to be passed back to the calling method, as explained in the next section. Using a `finally` block allows the cleanup code to execute even when there isn't a `catch` clause.

The following legal code demonstrates a `try` with a `finally` but no `catch`:

```
try {
  // do stuff
} finally {
  //clean up
}
```

The following legal code demonstrates a `try`, `catch`, and `finally`:

```
try {
  // do stuff
} catch (SomeException ex) {
  // do exception handling
} finally {
  // clean up
}
```

The following ILLEGAL code demonstrates a `try` without a `catch` or `finally`:

```
try {
  // do stuff
}
// need a catch or finally here
System.out.println("out of try block");
```

The following ILLEGAL code demonstrates a misplaced `catch` block:

```
try {
  // do stuff
}
// can't have code between try/catch
System.out.println("out of try block");
catch(Exception ex) { }
```

exam Watch

It is illegal to use a `try` clause without either a `catch` clause or a `finally` clause. A `try` clause by itself will result in a compiler error. Any `catch` clauses must immediately follow the `try` block. Any `finally` clause must immediately follow the last `catch` clause (or it must immediately follow the `try` block if there is no `catch`). It is legal to omit either the `catch` clause or the `finally` clause, but not both.

exam Watch

You can't sneak any code in between the `try`***,*** `catch`***, or*** `finally` ***blocks. The following won't compile:***

```
try {
  // do stuff
}
System.out.print("below the try");  //Illegal!
catch(Exception ex) { }
```

Propagating Uncaught Exceptions

Why aren't `catch` clauses required? What happens to an exception that's thrown in a `try` block when there is no `catch` clause waiting for it? Actually, there's no requirement that you code a `catch` clause for every possible exception that could be thrown from the corresponding `try` block. In fact, it's doubtful that you could accomplish such a feat! If a method doesn't provide a `catch` clause for a particular exception, that method is said to be "ducking" the exception (or "passing the buck").

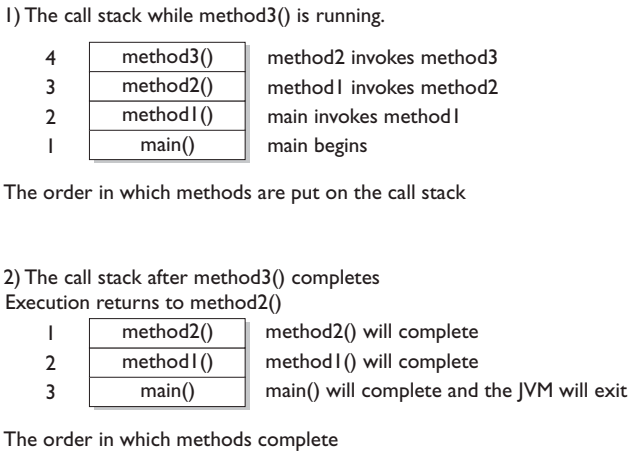
So what happens to a ducked exception? Before we discuss that, we need to briefly review the concept of the call stack. Most languages have the concept of a method stack or a call stack. Simply put, the call stack is the chain of methods that your program executes to get to the current method. If your program starts in method `main()` and `main()` calls method `a()`, which calls method `b()`, which in turn calls method `c()`, the call stack consists of the following:

```
c
b
a
main
```

We will represent the stack as growing upward (although it can also be visualized as growing downward). As you can see, the last method called is at the top of the stack, while the first calling method is at the bottom. The method at the very top of the stack trace would be the method you were currently executing. If we move back down the call stack, we're moving from the current method to the previously called method. Figure 5-1 illustrates a way to think about how the call stack in Java works.

FIGURE 5-1

The Java method call stack



Now let's examine what happens to ducked exceptions. Imagine a building, say, five stories high, and at each floor there is a deck or balcony. Now imagine that on each deck, one person is standing holding a baseball mitt. Exceptions are like balls dropped from person to person, starting from the roof. An exception is first thrown from the top of the stack (in other words, the person on the roof), and if it isn't caught by the same person who threw it (the person on the roof), it drops down the call stack to the previous method, which is the person standing on the deck one floor down. If not caught there, by the person one floor down, the exception/ball again drops down to the previous method (person on the next floor down), and so on until it is caught or until it reaches the very bottom of the call stack. This is called exception propagation.

If an exception reaches the bottom of the call stack, it's like reaching the bottom of a very long drop; the ball explodes, and so does your program. An exception that's never caught will cause your application to stop running. A description (if one is available) of the exception will be displayed, and the call stack will be "dumped." This helps you debug your application by telling you what exception was thrown, from what method it was thrown, and what the stack looked like at the time.

exam Watch

You can keep throwing an exception down through the methods on the stack. But what about when you get to the `main()` method at the bottom? You can throw the exception out of `main()` as well. This results in the Java Virtual Machine (JVM) halting, and the stack trace will be printed to the output.

The following code throws an exception,

```
class TestEx {
  public static void main (String [] args) {
    doStuff();
  }
  static void doStuff() {
    doMoreStuff();
  }
  static void doMoreStuff() {
    int x = 5/0;  // Can't divide by zero!
                  // ArithmeticException is thrown here
  }
}
```

which prints out a stack trace something like,

```
 %java TestEx
Exception in thread "main" java.lang.ArithmeticException: /
by zero
at TestEx.doMoreStuff(TestEx.java:10)
at TestEx.doStuff(TestEx.java:7)
at TestEx.main(TestEx.java:3)
```

EXERCISE 5-3

Propagating and Catching an Exception

In this exercise you're going to create two methods that deal with exceptions. One of the methods is the `main()` method, which will call another method. If an exception is thrown in the other method, `main()` must deal with it. A `finally` statement will be included to indicate that the program has completed. The method that `main()`

will call will be named `reverse`, and it will reverse the order of the characters in a String. If the String contains no characters, `reverse` will propagate an exception up to the `main()` method.

- Create a class called `Propagate` and a `main()` method, which will remain empty for now.
- Create a method called `reverse`. It takes an argument of a `String` and returns a `String`.
- In `reverse`, check if the `String` has a length of 0 by using the `String.length()` method. If the length is 0, the `reverse` method will throw an exception.
- Now include the code to reverse the order of the `String`. Because this isn't the main topic of this chapter, the reversal code has been provided, but feel free to try it on your own.

```
String reverseStr = "";
for(int i=s.length()-1;i>=0;--i) {
  reverseStr += s.charAt(i);
}
return reverseStr;
```

- Now in the `main()` method you will attempt to call this method and deal with any potential exceptions. Additionally, you will include a `finally` statement that displays when `main()` has finished.

Defining Exceptions

We have been discussing exceptions as a concept. We know that they are thrown when a problem of some type happens, and we know what effect they have on the flow of our program. In this section we will develop the concepts further and use exceptions in functional Java code. Earlier we said that an exception is an occurrence that alters the normal program flow. But because this is Java, anything that's not a primitive must be…an object. Exceptions are no, well, exception to this rule. Every exception is an instance of a class that has class `Exception` in its inheritance hierarchy. In other words, exceptions are always some subclass of `java.lang.Exception`.

When an exception is thrown, an object of a particular `Exception` subtype is instantiated and handed to the exception handler as an argument to the `catch` clause. An actual `catch` clause looks like this:

```
try {
   // some code here
}
catch (ArrayIndexOutOfBoundsException e) {
   e.printStackTrace();
}
```

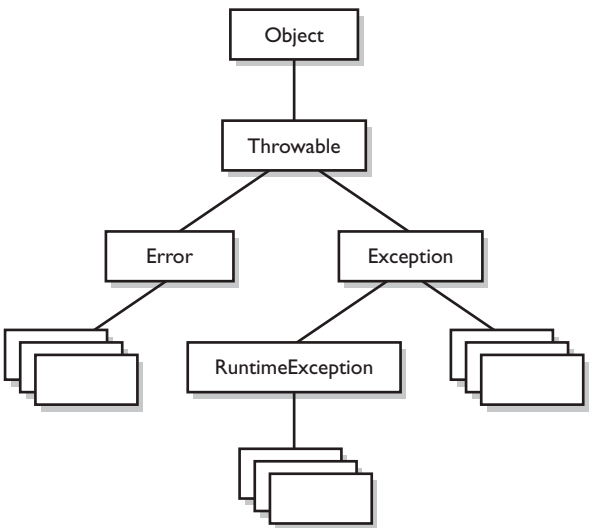
In this example, `e` is an instance of the `ArrayIndexOutOfBoundsException` class. As with any other object, you can call its methods.

Exception Hierarchy

All exception classes are subtypes of class `Exception`. This class derives from the class `Throwable` (which derives from the class `Object`). Figure 5-2 shows the hierarchy for the exception classes.

FIGURE 5-2

Exception class hierarchy



As you can see, there are two subclasses that derive from `Throwable`: `Exception` and `Error`. Classes that derive from `Error` represent unusual situations that are not caused by program errors, and indicate things that would not normally happen

during program execution, such as the JVM running out of memory. Generally, your application won't be able to recover from an `Error`, so you're not required to handle them. If your code does not handle them (and it usually won't), it will still compile with no trouble. Although often thought of as exceptional conditions, Errors are technically not exceptions because they do not derive from class `Exception`.

In general, an exception represents something that happens not as a result of a programming error, but rather because some resource is not available or some other condition required for correct execution is not present. For example, if your application is supposed to communicate with another application or computer that is not answering, this is an exception that is not caused by a bug. Figure 5-2 also shows a subtype of `Exception` called `RuntimeException`. These exceptions are a special case because they sometimes do indicate program errors. They can also represent rare, difficult-to-handle exceptional conditions. Runtime exceptions are discussed in greater detail later in this chapter.

Java provides many exception classes, most of which have quite descriptive names. There are two ways to get information about an exception. The first is from the type of the exception itself. The next is from information that you can get from the exception object. Class `Throwable` (at the top of the inheritance tree for exceptions) provides its descendants with some methods that are useful in exception handlers. One of these is `printStackTrace()`. As expected, if you call an exception object's `printStackTrace()` method, as in the earlier example, a stack trace from where the exception occurred will be printed.

We discussed that a call stack builds upward with the most recently called method at the top. You will notice that the `printStackTrace()` method prints the most recently entered method first and continues down, printing the name of each method as it works its way down the call stack (this is called unwinding the stack) from the top.

exam Watch

For the exam, it is not necessary to know any of the methods contained in the `Throwable` classes, including `Exception` and `Error`. You are expected to know that `Exception`, `Error`, `RuntimeException`, and `Throwable` types can all be thrown using the `throw` keyword, and can all be caught (although you rarely will catch anything other than `Exception` subtypes).

Handling an Entire Class Hierarchy of Exceptions

We've discussed that the `catch` keyword allows you to specify a particular type of exception to catch. You can actually catch more than one type of exception in a single `catch` clause. If the exception class that you specify in the `catch` clause has no subclasses, then only the specified class of exception will be caught. However, if the class specified in the `catch` clause does have subclasses, any exception object that subclasses the specified class will be caught as well.

For example, class `IndexOutOfBoundsException` has two subclasses, `ArrayIndexOutOfBoundsException` and `StringIndexOutOfBoundsException`. You may want to write one exception handler that deals with exceptions produced by either type of boundary error, but you might not be concerned with which exception you actually have. In this case, you could write a `catch` clause like the following:

```
try {
  // Some code here that can throw a boundary exception
}
catch (IndexOutOfBoundsException e) {
  e.printStackTrace();
}
```

If any code in the `try` block throws `ArrayIndexOutOfBoundsException` or `StringIndexOutOfBoundsException`, the exception will be caught and handled. This can be convenient, but it should be used sparingly. By specifying an exception class's superclass in your `catch` clause, you're discarding valuable information about the exception. You can, of course, find out exactly what exception class you have, but if you're going to do that, you're better off writing a separate `catch` clause for each exception type of interest.

on the Job

Resist the temptation to write a single catchall exception handler such as the following:

```
try {
  // some code
}
catch (Exception e) {
  e.printStackTrace();
}
```

This code will catch every exception generated. Of course, no single exception handler can properly handle every exception, and programming in this way defeats the design objective. Exception handlers that trap many errors at once will probably reduce the reliability of your program because it's likely that an exception will be caught that the handler does not know how to handle.

Exception Matching

If you have an exception hierarchy composed of a superclass exception and a number of subtypes, and you're interested in handling one of the subtypes in a special way but want to handle all the rest together, you need write only two `catch` clauses.

When an exception is thrown, Java will try to find (by looking at the available `catch` clauses from the top down) a `catch` clause for the exception type. If it doesn't find one, it will search for a handler for a supertype of the exception. If it does not find a `catch` clause that matches a supertype for the exception, then the exception is propagated down the call stack. This process is called exception matching. Let's look at an example:

```
 1: import java.io.*;
 2: public class ReadData {
 3:   public static void main(String args[]) {
 4:     try {
 5:       RandomAccessFile raf =
 6:         new RandomAccessFile("myfile.txt", "r");
 7:       byte b[] = new byte[1000];
 8:       raf.readFully(b, 0, 1000);
 9:     }
10:     catch(FileNotFoundException e) {
11:       System.err.println("File not found");
12:       System.err.println(e.getMessage());
13:       e.printStackTrace();
14:     }
15:     catch(IOException e) {
16:       System.err.println("IO Error");
17:       System.err.println(e.toString());
18:       e.printStackTrace();
19:     }
20:   }
21: }
```

This short program attempts to open a file and to read some data from it. Opening and reading files can generate many exceptions, most of which are some type of `IOException`. Imagine that in this program we're interested in knowing only whether the exact exception is a `FileNotFoundException`. Otherwise, we don't care exactly what the problem is.

`FileNotFoundException` is a subclass of `IOException`. Therefore, we could handle it in the `catch` clause that catches all subtypes of `IOException`, but then we would have to test the exception to determine whether it was a `FileNotFoundException`. Instead, we coded a special exception handler for the `FileNotFoundException` and a separate exception handler for all other `IOException` subtypes.

If this code generates a `FileNotFoundException`, it will be handled by the `catch` clause that begins at line 10. If it generates another `IOException`—perhaps `EOFException`, which is a subclass of `IOException`—it will be handled by the `catch` clause that begins at line 15. If some other exception is generated, such as a runtime exception of some type, neither `catch` clause will be executed and the exception will be propagated down the call stack.

Notice that the `catch` clause for the `FileNotFoundException` was placed above the handler for the `IOException`. This is really important! If we do it the opposite way, the program will not compile. The handlers for the most specific exceptions must always be placed above those for more general exceptions. The following will not compile:

```
try {
  // do risky IO things
} catch (IOException e) {
  // handle general IOExceptions
} catch (FileNotFoundException ex) {
  // handle just FileNotFoundException
}
```

You'll get a compiler error something like this:

```
TestEx.java:15: exception java.io.FileNotFoundException has
 already been caught
} catch (FileNotFoundException ex) {
  ^
```

If you think back to the people with baseball mitts (in the section "Propagating Uncaught Exceptions"), imagine that the most general mitts are the largest, and can thus catch many different kinds of balls. An `IOException` mitt is large enough

and flexible enough to catch any type of `IOException`. So if the person on the fifth floor (say, Fred) has a big 'ol `IOException` mitt, he can't help but catch a `FileNotFoundException` ball with it. And if the guy (say, Jimmy) on the second floor is holding a `FileNotFoundException` mitt, that `FileNotFoundException` ball will never get to him, since it will always be stopped by Fred on the fifth floor, standing there with his big-enough-for-any-`IOException` mitt.

So what do you do with exceptions that are siblings in the class hierarchy? If one Exception class is not a subtype or supertype of the other, then the order in which the `catch` clauses are placed doesn't matter.

Exception Declaration and the Public Interface

So, how do we know that some method throws an exception that we have to catch? Just as a method must specify what type and how many arguments it accepts and what is returned, the exceptions that a method can throw must be *declared* (unless the exceptions are subclasses of `RuntimeException`). The list of thrown exceptions is part of a method's public interface. The `throws` keyword is used as follows to list the exceptions that a method can throw:

```
void myFunction() throws MyException1, MyException2 {
  // code for the method here
}
```

This method has a `void` return type, accepts no arguments, and declares that it can throw one of two types of exceptions: either type `MyException1` or type `MyException2`. (Just because the method declares that it throws an exception doesn't mean it always will. It just tells the world that it might.)

Suppose your method doesn't directly throw an exception, but calls a method that does. You can choose not to handle the exception yourself and instead just declare it, as though it were your method that actually throws the exception. If you do declare the exception that your method might get from another method, and you don't provide a `try/catch` for it, then the method will propagate back to the method that called your method, and either be caught there or continue on to be handled by a method further down the stack.

Any method that might throw an exception (unless it's a subclass of `RuntimeException`) must declare the exception. That includes methods that aren't actually throwing it directly, but are "ducking" and letting the exception pass down to the next method in the stack. If you "duck" an exception, it is just as if you were the one actually throwing the exception. `RuntimeException` subclasses are

exempt, so the compiler won't check to see if you've declared them. But all non-`RuntimeExceptions` are considered "checked" exceptions, because the compiler checks to be certain you've acknowledged that "bad things could happen here."

Remember this:

Each method must either handle all checked exceptions by supplying a `catch` *clause or list each unhandled checked exception as a thrown exception.*

This rule is referred to as Java's "handle or declare" requirement. (Sometimes called "catch or declare.")

exam Watch

Look for code that invokes a method declaring an exception, where the calling method doesn't handle or declare the checked exception. The following code (which uses the `throw` ***keyword to throw an exception manually—more on this next) has two big problems that the compiler will prevent:***

```
void doStuff() {
  doMore();
}
void doMore() {
  throw new IOException();
}
```

First, the `doMore()` ***method throws a checked exception, but does not declare it! But suppose we fix the*** `doMore()` ***method as follows:***

```
void doMore() throws IOException { … }
```

The `doStuff()` ***method is still in trouble because it, too, must declare the*** `IOException`***, unless it handles it by providing a*** `try/catch`***, with a*** `catch` ***clause that can take an*** `IOException`***.***

Again, some exceptions are exempt from this rule. An object of type `RuntimeException` may be thrown from any method without being specified as part of the method's public interface (and a handler need not be present). And even if a method does declare a `RuntimeException`, the calling method is under no obligation to handle or declare it. `RuntimeException`, `Error`, and all of their subtypes are unchecked exceptions and unchecked exceptions do not have to be specified or handled. Here is an example:

```
import java.io.*;
class Test {
  public int myMethod1() throws EOFException {
    return myMethod2();
  }
  public int myMethod2() throws EOFException {
    // code that actually could throw the exception goes here
    return 1;
  }
}
```

Let's look at `myMethod1()`. Because `EOFException` subclasses `IOException` and `IOException` subclasses `Exception`, it is a checked exception and must be declared as an exception that may be thrown by this method. But where will the exception actually come from? The public interface for method `myMethod2()` called here declares that an exception of this type can be thrown. Whether that method actually throws the exception itself or calls another method that throws it is unimportant to us; we simply know that we have to either catch the exception or declare that we throw it. The method `myMethod1()` does not catch the exception, so it declares that it throws it. Now let's look at another legal example, `myMethod3()`.

```
public void myMethod3() {
  // code that could throw a NullPointerException goes here
}
```

According to the comment, this method can throw a `NullPointerException`. Because `RuntimeException` is the superclass of `NullPointerException`, it is an unchecked exception and need not be declared. We can see that `myMethod3()` does not declare any exceptions.

Runtime exceptions are referred to as *unchecked* exceptions. All other exceptions are *checked* exceptions, and they don't derive from `java.lang.RuntimeException`. A checked exception must be caught somewhere in your code. If you invoke a method that throws a checked exception but you don't catch the checked exception

somewhere, your code will not compile. That's why they're called checked exceptions; the compiler checks to make sure that they're handled or declared. A number of the methods in the Java 2 Standard Edition libraries throw checked exceptions, so you will often write exception handlers to cope with exceptions generated by methods you didn't write.

You can also throw an exception yourself, and that exception can be either an existing exception from the Java API or one of your own. To create your own exception, you simply subclass `Exception` (or one of its subclasses) as follows:

```
class MyException extends Exception { }
```

And if you throw the exception, the compiler will guarantee that you declare it as follows:

```
class TestEx {
  void doStuff() {
    throw new MyException();  // Throw a checked exception
  }
}
```

The preceding code upsets the compiler:

```
TestEx.java:6: unreported exception MyException; must be caught
or
declared to be thrown
  throw new MyException();
  ^
```

exam Watch

When an object of a subtype of `Exception` ***is thrown, it must be handled or declared. These objects are called checked exceptions, and include all exceptions except those that are subtypes of*** `RuntimeException`***, which are unchecked exceptions. Be ready to spot methods that don't follow the "handle or declare" rule, such as***

```
class MyException extends Exception {
  void someMethod () {
    doStuff();
  }
```

exam Watch

```
void doStuff() throws MyException {
  try {
    throw new MyException();
  }
  catch(MyException me) {
    throw me;
  }
 }
}
```

You need to recognize that this code won't compile. If you try, you'll get

```
MyException.java:3: unreported exception MyException; must
be caught or declared to be thrown
doStuff();
       ^
```

Notice that `someMethod()` ***fails to either handle or declare the exception that*** *can* ***be thrown by*** `doStuff()`**.**

You need to know how an `Error` compares with checked and unchecked exceptions. Objects of type `Error` are not `Exception` objects, although they do represent exceptional conditions. Both `Exception` and `Error` share a common superclass, `Throwable`, thus both can be thrown using the `throw` keyword. When an `Error` or a subclass of `Error` is thrown, it's unchecked. You are not required to catch `Error` objects or `Error` subtypes. You can also throw an `Error` yourself (although other than `AssertionError` you probably won't ever want to), and you can catch one, but again, you probably won't. What, for example, would you actually do if you got an `OutOfMemoryError`? It's not like you can tell the garbage collector to run; you can bet the JVM fought desperately to save itself (and reclaimed all the memory it could) by the time you got the error. In other words, don't expect the JVM at that point to say, "Run the garbage collector? Oh, thanks so much for telling me. That just never occurred to me. Sure, I'll get right on it." Even better, what would you do if a `VirtualMachineError` arose? Your program is toast by the time you'd catch the Error, so there's really no point in trying to catch one of these babies. Just remember, though, that you can! The following compiles just fine:

```
class TestEx {
  public static void main (String [] args) {
    badMethod();
  }
  static void badMethod() {  // No need to declare an Error
    doStuff();
  }
  static void doStuff() {  //No need to declare an Error
    try {
      throw new Error();
    }
    catch(Error me) {
      throw me; // We catch it, but then rethrow it
    }
  }
}
```

If we were throwing a checked exception rather than `Error`, then the `doStuff()` method would need to declare the exception. But remember, since `Error` is not a subtype of `Exception`, it doesn't need to be declared. You're free to declare it if you like, but the compiler just doesn't care one way or another when or how the `Error` is thrown, or by whom.

Because Java has checked exceptions, it's commonly said that Java forces developers to handle exceptions. Yes, Java forces us to write exception handlers for each exception that can occur during normal operation, but it's up to us to make the exception handlers actually do something useful. We know software managers who melt down when they see a programmer write:

```
try {
  callBadMethod();
} catch (Exception ex) { }
```

Notice anything missing? Don't "eat" the exception by catching it without actually handling it. You won't even be able to tell that the exception occurred, because you'll never see the stack trace.

Rethrowing the Same Exception

Just as you can throw a new exception from a `catch` clause, you can also throw the same exception you just caught. Here's a `catch` clause that does this:

```
catch(IOException e) {
  // Do things, then if you decide you can't handle it…
  throw e;
}
```

All other `catch` clauses associated with the same `try` are ignored, if a `finally` block exists, it runs, and the exception is thrown back to the calling method (the next method down the call stack). If you throw a checked exception from a `catch` clause, you must also declare that exception! In other words, you must handle *and* declare, as opposed to handle *or* declare. The following example is illegal:

```
public void doStuff() {
  try {
    // risky IO things
  } catch(IOException ex) {
    // can't handle it
     throw ex;  // Can't throw it unless you declare it
  }
}
```

In the preceding code, the `doStuff()` method is clearly able to throw a checked exception—in this case an `IOException`—so the compiler says, "Well, that's just peachy that you have a `try/catch` in there, but it's not good enough. If you might rethrow the `IOException` you catch, then you must declare it!"

EXERCISE 5-4

Creating an Exception

In this exercise we attempt to create a custom exception. We won't put in any new methods (it will have only those inherited from `Exception`), and because it extends `Exception`, the compiler considers it a checked exception. The goal of the program is to determine whether a command-line argument, representing a particular food (as a string), is considered bad or OK.

- Let's first create our exception. We will call it `BadFoodException`. This exception will be thrown when a bad food is encountered.
- Create an enclosing class called `MyException` and a `main()` method, which will remain empty for now.

- Create a method called `checkFood()`. It takes a `String` argument and throws our exception if it doesn't like the food it was given. Otherwise, it tells us it likes the food. You can add any foods you aren't particularly fond of to the list.
- Now in the `main()` method, you'll get the command-line argument out of the `String` array, and then pass that `String` on to the `checkFood()` method. Because it's a checked exception, the `checkFood()` method must declare it, and the `main()` method must handle it (using a `try/catch`). Do not have `main()` declare the exception, because if `main()` ducks the exception, who else is back there to catch it?
- As nifty as exception handling is, it's still up to the developer to make proper use of it. Exception handling makes organizing our code and signaling problems easy, but the exception handlers still have to be written. You'll find that even the most complex situations can be handled, and your code will be reusable, readable, and maintainable.

CERTIFICATION OBJECTIVE

Common Exceptions and Errors (Exam Objective 2.6)

2.6 Recognize situations that will result in any of the following being thrown: ArrayIndexOutOfBoundsException, ClassCastException, IllegalArgumentException, IllegalStateException, NullPointerException, NumberFormatException, AssertionError, ExceptionInInitializerError, StackOverflowError, or NoClassDefFoundError. Understand which of these are thrown by the virtual machine and recognize situations in which others should be thrown programmatically.

Exception handling is another area that the exam creation team decided to expand for the SCJP 5 exam. This section discusses the aspects of exceptions that were added for this new version. The intention of Objective 2.6 is to make sure that you are familiar with some of the most common exceptions and errors you'll encounter as a Java programmer.

exam Watch

The questions from this section are likely to be along the lines of, "Here's some code that just did something bad, which exception will be thrown?"

Throughout the exam, questions will present some code and ask you to determine whether the code will run, or whether an exception will be thrown. Since these questions are so common, understanding the causes for these exceptions is critical to your success.

This is another one of those objectives that will turn up all through the real exam (does "An exception is thrown at runtime" ring a bell?), so make sure this section gets a lot of your attention.

Where Exceptions Come From

Jump back a page and take a look at the last sentence of Objective 2.6. It's important to understand what causes exceptions and errors, and where they come from. For the purposes of exam preparation, let's define two broad categories of exceptions and errors:

- **JVM exceptions** Those exceptions or errors that are either exclusively or most logically thrown by the JVM.
- **Programmatic exceptions** Those exceptions that are thrown explicitly by application and/or API programmers.

JVM Thrown Exceptions

Let's start with a very common exception, the `NullPointerException`. As we saw in Chapter 3, this exception occurs when you attempt to access an object using a reference variable with a current value of `null`. There's no way that the compiler can hope to find these problems before runtime. Let's look at the following:

```
class NPE {
  static String s;
  public static void main(String [] args) {
    System.out.println(s.length());
  }
}
```

Surely, the compiler can find the problem with that tiny little program! Nope, you're on your own. The code will compile just fine, and the JVM will throw a `NullPointerException` when it tries to invoke the `length()` method.

Earlier in this chapter we discussed the call stack. As you recall, we used the convention that `main()` would be at the bottom of the call stack, and that as `main()` invokes another method, and that method invokes another, and so on, the stack grows upward. Of course the stack resides in memory, and even if your OS gives you a gigabyte of RAM for your program, it's still a finite amount. It's possible to grow the stack so large that the OS runs out of space to store the call stack. When this happens you get (wait for it...), a `StackOverflowError`. The most common way for this to occur is to create a recursive method. A recursive method is one that invokes itself in the method body. While that may sound weird, it's a very common and useful technique for such things as searching and sorting algorithms. Take a look at this code:

```
void go() {    // recursion gone bad
   go();
}
```

As you can see, if you ever make the mistake of invoking the `go()` method, your program will fall into a black hole; `go()` invoking `go()` invoking `go()`, until, no matter how much memory you have, you'll get a `StackOverflowError`. Again, only the JVM knows when this moment occurs, and the JVM will be the source of this error.

Programmatically Thrown Exceptions

Now let's look at programmatically thrown exceptions. Remember we defined "programmatically" as meaning something like this:

Created by an application and/or API developer.

For instance, many classes in the Java API have methods that take `String` arguments, and convert these `String`s into numeric primitives. A good example of these classes are the so-called "wrapper classes" that we studied in Chapter 3.

At some point long ago, some programmer wrote the `java.lang.Integer` class, and created methods like `parseInt()` and `valueOf()`. That programmer wisely decided that if one of these methods was passed a `String` that could not be converted into a number, the method should throw a `NumberFormatException`. The partially implemented code might look something like this:

```
int parseInt(String s) throws NumberFormatException {
  boolean parseSuccess = false;
  int result = 0;
  // do complicated parsing
  if (!parseSuccess)   // if the parsing failed
    throw new NumberFormatException();
  return result;
}
```

Other examples of programmatic exceptions include an `AssertionError` (okay, it's not an exception, but it IS thrown programmatically), and throwing an `IllegalArgumentException`. In fact, our mythical API developer could have used `IllegalArgumentException` for her `parseInt()` method. But it turns out that `NumberFormatException` extends `IllegalArgumentException`, and is a little more precise, so in this case, using `NumberFormatException` supports the notion we discussed earlier: that when you have an exception hierarchy, you should use the most precise exception that you can.

Of course, as we discussed earlier, you can also make up your very own special, custom exceptions, and throw them whenever you want to. These homemade exceptions also fall into the category of "programmatically thrown exceptions."

A Summary of the Exam's Exceptions and Errors

Objective 2.6 lists ten specific exceptions and errors. In this section we discussed the `StackOverflowError`. The other nine exceptions and errors listed in the objective are covered elsewhere in this book. Table 5-2 summarizes this list and provides chapter references to the exceptions and errors we did not discuss here.

TABLE 5-2 Descriptions and Sources of Common Exceptions.

Exception (Chapter Location)	Description	Typically Thrown
ArrayIndexOutOfBoundsException (Chapter 3, "Assignments")	Thrown when attempting to access an array with an invalid index value (either negative or beyond the length of the array).	By the JVM
ClassCastException (Chapter 2, "Object Orientation")	Thrown when attempting to cast a reference variable to a type that fails the IS-A test.	By the JVM
IllegalArgumentException (This chapter)	Thrown when a method receives an argument formatted differently than the method expects.	Programmatically
IllegalStateException (Chapter 6, "Formatting")	Thrown when the state of the environment doesn't match the operation being attempted, e.g., using a Scanner that's been closed.	Programmatically
NullPointerException (Chapter 3, "Assignments")	Thrown when attempting to access an object with a reference variable whose current value is `null`.	By the JVM
NumberFormatException (Chapter 3, "Assignments")	Thrown when a method that converts a String to a number receives a String that it cannot convert.	Programmatically
AssertionError (This chapter)	Thrown when a statement's boolean test returns `false`.	Programmatically
ExceptionInInitializerError (Chapter 3, "Assignments")	Thrown when attempting to initialize a static variable or an initialization block.	By the JVM
StackOverflowError (This chapter)	Typically thrown when a method recurses too deeply. (Each invocation is added to the stack.)	By the JVM
NoClassDefFoundError (Chapter 10, "Development")	Thrown when the JVM can't find a class it needs, because of a command-line error, a classpath issue, or a missing `.class` file.	By the JVM

CERTIFICATION OBJECTIVE

Working with the Assertion Mechanism (Exam Objective 2.3)

2.3 Develop code that makes use of assertions, and distinguish appropriate from inappropriate uses of 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 && y) {
  // do something
} else if (x < 2 || y) {
  // 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");
}
```

Added to the Java language beginning with version 1.4, 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.

Starting with exam 310-035 (version 1.4 of the Sun Certified Java Programmer exam) and continuing through to the current exam 310-065 (SCJP 6), 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 must be < 0
            // This code should never be reached!
    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 let your code stay cleaner and tighter, but 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 (that 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 while the really simple version simply 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 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, 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 is used only with the simple `assert` statement, where 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;
}
```

exam Watch

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), then always assume the word "expression" refers to expression1, the boolean test. For example, consider the following question:

exam Watch

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 not be correct, since expression2 can have a result of any value, not just a `boolean`.

Enabling Assertions

If you want to use assertions, you have to think first about how to compile with assertions in your code, and then about how to run with assertions enabled. Both require version 1.4 or greater, and that brings us to the first issue: how to compile with assertions in your code.

Identifier vs. Keyword

Prior to version 1.4, you might very well have written code like this:

```
int assert = getInitialValue();
if (assert == getActualResult()) {
  // do something
}
```

Notice that in the preceding code, `assert` is used as an identifier. That's not a problem prior to 1.4. But you cannot use a keyword/reserved word as an identifier, and beginning with version 1.4, `assert` is a keyword. The bottom line is this:

You can use `assert` as a keyword or as an identifier, but not both.

on the Job

If for some reason you're using a Java 1.4 compiler, and if you're using `assert` as a keyword (in other words, you're actually trying to `assert` something in your code), then you must explicitly enable assertion-awareness at compile time, as follows:

```
javac -source 1.4 com/geeksanonymous/TestClass.java
```

You can read that as "compile the class TestClass, in the directory `com/geeksanonymous`***, and do it in the 1.4 way, where*** `assert` ***is a keyword."***

Use Version 6 of java and javac

As far as the exam is concerned, you'll ALWAYS be using version 6 of the Java compiler (`javac`), and version 6 of the Java application launcher (`java`). You might see questions about older versions of source code, but those questions will always be in the context of compiling and launching old code with the current versions of `javac` and `java`.

Compiling Assertion-Aware Code

The Java 6 compiler will use the `assert` keyword by default. Unless you tell it otherwise, the compiler will generate an error message if it finds the word `assert` used as an identifier. However, you can tell the compiler that you're giving it an old piece of code to compile, and that it should pretend to be an old compiler! (More about compiler commands in Chapter 10.) Let's say you've got to make a quick fix to an old piece of 1.3 code that uses `assert` as an identifier. At the command line you can type

```
javac -source 1.3 OldCode.java
```

The compiler will issue warnings when it discovers the word `assert` used as an identifier, but the code will compile and execute. Suppose you tell the compiler that your code is version 1.4 or later, for instance:

```
javac -source 1.4 NotQuiteSoOldCode.java
```

In this case, the compiler will issue errors when it discovers the word `assert` used as an identifier.

If you want to tell the compiler to use Java 6 rules you can do one of three things: omit the `-source` option, which is the default, or add one of two source options:

`-source 1.6` or `-source 6`.

If you want to use assert as an identifier in your code, you MUST compile using the `-source 1.3` option. Table 5-3 summarizes how the Java 6 compiler will react to `assert` as either an identifier or a keyword.

TABLE 5-3 Using Java 6 to Compile Code That Uses `assert` as an Identifier or a Keyword

Command Line	If assert Is an Identifier	If assert Is a Keyword
`javac -source 1.3 TestAsserts.java`	Code compiles with warnings.	Compilation fails.
`javac -source 1.4 TestAsserts.java`	Compilation fails.	Code compiles.
`javac -source 1.5 TestAsserts.java`	Compilation fails.	Code compiles.
`javac -source 5 TestAsserts.java`	Compilation fails.	Code compiles.
`javac -source 1.6 TestAsserts.java`	Compilation fails.	Code compiles.
`javac -source 6 TestAsserts.java`	Compilation fails.	Code compiles.
`javac TestAsserts.java`	Compilation fails.	Code compiles.

Running with Assertions

Here's where it gets cool. Once you've written your assertion-aware code (in other words, code that uses `assert` as a keyword, to actually perform assertions at runtime), you can choose to enable or disable your assertions at runtime! Remember, assertions are disabled by default.

Enabling Assertions at Runtime

You enable assertions at runtime with

```
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 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 there wasn't much use of that term 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
                  |_twelvesteps
                              |_StepOne
                              |_StepTwo
```

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 5-4 lists examples of command-line switches for enabling and disabling assertions.

TABLE 5-4 Assertion Command-Line Switches

Command-Line Example	What It Means
`java -ea` `java -enableassertions`	Enable assertions.
`java -da` `java -disableassertions`	Disable assertions (the default behavior of Java 6).
`java -ea:com.foo.Bar`	Enable assertions in class `com.foo.Bar`.
`java -ea:com.foo...`	Enable assertions in package `com.foo` and any of its subpackages.
`java -ea -dsa`	Enable assertions in general, but disable assertions in system classes.
`java -ea -da:com.foo...`	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 Sun'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? Sun. The exam uses Sun'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
}
```

exam Watch

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 Sun. 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 either 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, 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! When a hot tempered Q.A. 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 SUMMARY

This chapter covered a lot of ground, all of which involves ways of controlling your program flow, based on a conditional test. First you learned about `if` and `switch` statements. The `if` statement evaluates one or more expressions to a `boolean` result. If the result is `true`, the program will execute the code in the block that is encompassed by the `if`. If an `else` statement is used and the `if` expression evaluates to `false`, then the code following the `else` will be performed. If no `else` block is defined, then none of the code associated with the `if` statement will execute.

You also learned that the `switch` statement can be used to replace multiple `if-else` statements. The `switch` statement can evaluate integer primitive types that can be implicitly cast to an `int` (those types are `byte`, `short`, `int`, and `char`), or it can evaluate `enum`s.

At runtime, the JVM will try to find a match between the expression in the `switch` statement and a constant in a corresponding `case` statement. If a match is found, execution will begin at the matching case, and continue on from there, executing code in all the remaining `case` statements until a `break` statement is found or the end of the `switch` statement occurs. If there is no match, then the `default` case will execute, if there is one.

You've learned about the three looping constructs available in the Java language. These constructs are the `for` loop (including the basic `for` and the enhanced `for` which is new to Java 6), the `while` loop, and the `do` loop. In general, the `for` loop is used when you know how many times you need to go through the loop. The `while` loop is used when you do not know how many times you want to go through, whereas the `do` loop is used when you need to go through at least once. In the `for` loop and the `while` loop, the expression will have to evaluate to `true` to get inside the block and will check after every iteration of the loop. The `do` loop does not check the condition until after it has gone through the loop once. The major benefit of the `for` loop is the ability to initialize one or more variables and increment or decrement those variables in the `for` loop definition.

The `break` and `continue` statements can be used in either a labeled or unlabeled fashion. When unlabeled, the `break` statement will force the program to stop processing the innermost looping construct and start with the line of code following the loop. Using an unlabeled `continue` command will cause the program to stop execution of the current iteration of the innermost loop and proceed with the next iteration. When a `break` or a `continue` statement is used in a labeled manner, it will perform in the same way, with one exception: the statement will not apply to the innermost loop; instead, it will apply to the loop with the label. The `break` statement is used most often in conjunction with the `switch` statement. When there is a match between the `switch` expression and the case constant, the code following the case constant will be performed. To stop execution, a `break` is needed.

You've seen how Java provides an elegant mechanism in exception handling. Exception handling allows you to isolate your error-correction code into separate blocks so that the main code doesn't become cluttered by error-checking code. Another elegant feature allows you to handle similar errors with a single error-handling block, without code duplication. Also, the error handling can be deferred to methods further back on the call stack.

You learned that Java's `try` keyword is used to specify a guarded region—a block of code in which problems might be detected. An exception handler is the code that is executed when an exception occurs. The handler is defined by using Java's `catch` keyword. All `catch` clauses must immediately follow the related `try` block. Java also provides the `finally` keyword. This is used to define a block of code that is always executed, either immediately after a `catch` clause completes or immediately

after the associated `try` block in the case that no exception was thrown (or there was a `try` but no `catch`). Use `finally` blocks to release system resources and to perform any cleanup required by the code in the `try` block. A `finally` block is not required, but if there is one it must immediately follow the last `catch`. (If there is no `catch` block, the `finally` block must immediately follow the `try` block.) It's guaranteed to be called except when the `try` or `catch` issues a `System.exit()`.

An exception object is an instance of class `Exception` or one of its subclasses. The `catch` clause takes, as a parameter, an instance of an object of a type derived from the `Exception` class. Java requires that each method either catches any checked exception it can throw or else declares that it throws the exception. The exception declaration is part of the method's public interface. To declare that an exception may be thrown, the `throws` keyword is used in a method definition, along with a list of all checked exceptions that might be thrown.

Runtime exceptions are of type `RuntimeException` (or one of its subclasses). These exceptions are a special case because they do not need to be handled or declared, and thus are known as "unchecked" exceptions. Errors are of type `java.lang.Error` or its subclasses, and like runtime exceptions, they do not need to be handled or declared. Checked exceptions include any exception types that are not of type `RuntimeException` or Error. If your code fails to either handle a checked exception or declare that it is thrown, your code won't compile. But with unchecked exceptions or objects of type `Error`, it doesn't matter to the compiler whether you declare them or handle them, do nothing about them, or do some combination of declaring and handling. In other words, you're free to declare them and handle them, but the compiler won't care one way or the other. It's not good practice to handle an `Error`, though, because you can rarely recover from one.

Exceptions can be generated by the JVM, or by a programmer.

Assertions, added to the language in version 1.4, 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. If you have older Java code that uses the word `assert` as an identifier, then you won't be able to use assertions, and you must recompile your older code using the `-source 1.3` flag. Remember that as of Java 6, assertions are compiled as a keyword by default, but must be enabled explicitly at runtime.

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!

TWO-MINUTE DRILL

Here are some of the key points from each certification objective in this chapter. You might want to loop through them several times.

Writing Code Using if and switch Statements (Obj. 2.1)

- The only legal expression in an `if` statement is a `boolean` expression, in other words an expression that resolves to a `boolean` or a `Boolean` variable.
- Watch out for `boolean` assignments (`=`) that can be mistaken for `boolean` equality (`==`) tests:

```
boolean x = false;
if (x = true) { } // an assignment, so x will always be true!
```

- Curly braces are optional for `if` blocks that have only one conditional statement. But watch out for misleading indentations.
- `switch` statements can evaluate only to `enums` or the `byte`, `short`, `int`, and `char` data types. You can't say,

```
long s = 30;
switch(s) { }
```

- The `case` constant must be a literal or `final` variable, or a constant expression, including an `enum`. You cannot have a case that includes a non-final variable, or a range of values.
- If the condition in a `switch` statement matches a `case` constant, execution will run through all code in the `switch` following the matching `case` statement until a `break` statement or the end of the `switch` statement is encountered. In other words, the matching `case` is just the entry point into the `case` block, but unless there's a `break` statement, the matching `case` is not the only `case` code that runs.
- The `default` keyword should be used in a `switch` statement if you want to run some code when none of the `case` values match the conditional value.
- The `default` block can be located anywhere in the `switch` block, so if no `case` matches, the `default` block will be entered, and if the `default` does not contain a `break`, then code will continue to execute (fall-through) to the end of the `switch` or until the `break` statement is encountered.

Writing Code Using Loops (Objective 2.2)

- A basic `for` statement has three parts: declaration and/or initialization, boolean evaluation, and the iteration expression.
- If a variable is incremented or evaluated within a basic `for` loop, it must be declared before the loop, or within the `for` loop declaration.
- A variable declared (not just initialized) within the basic `for` loop declaration cannot be accessed outside the `for` loop (in other words, code below the `for` loop won't be able to use the variable).
- You can initialize more than one variable of the same type in the first part of the basic `for` loop declaration; each initialization must be separated by a comma.
- An enhanced `for` statement (new as of Java 6), has two parts, the *declaration* and the *expression*. It is used only to loop through arrays or collections.
- With an enhanced `for`, the *expression* is the array or collection through which you want to loop.
- With an enhanced `for`, the *declaration* is the block variable, whose type is compatible with the elements of the array or collection, and that variable contains the value of the element for the given iteration.
- You cannot use a number (old C-style language construct) or anything that does not evaluate to a `boolean` value as a condition for an `if` statement or looping construct. You can't, for example, say `if(x)`, unless `x` is a `boolean` variable.
- The `do` loop will enter the body of the loop at least once, even if the test condition is not met.

Using break and continue (Objective 2.2)

- An unlabeled `break` statement will cause the current iteration of the innermost looping construct to stop and the line of code following the loop to run.
- An unlabeled `continue` statement will cause: the current iteration of the innermost loop to stop, the condition of that loop to be checked, and if the condition is met, the loop to run again.
- If the `break` statement or the `continue` statement is labeled, it will cause similar action to occur on the labeled loop, not the innermost loop.

Handling Exceptions (Objectives 2.4, 2.5, and 2.6)

- ❑ Exceptions come in two flavors: checked and unchecked.
- ❑ Checked exceptions include all subtypes of `Exception`, excluding classes that extend `RuntimeException`.
- ❑ Checked exceptions are subject to the handle or declare rule; any method that might throw a checked exception (including methods that invoke methods that can throw a checked exception) must either declare the exception using `throws`, or handle the exception with an appropriate try/catch.
- ❑ Subtypes of `Error` or `RuntimeException` are unchecked, so the compiler doesn't enforce the handle or declare rule. You're free to handle them, or to declare them, but the compiler doesn't care one way or the other.
- ❑ If you use an optional `finally` block, it will always be invoked, regardless of whether an exception in the corresponding `try` is thrown or not, and regardless of whether a thrown exception is caught or not.
- ❑ The only exception to the `finally`-will-always-be-called rule is that a `finally` will not be invoked if the JVM shuts down. That could happen if code from the `try` or `catch` blocks calls `System.exit()`.
- ❑ Just because `finally` is invoked does not mean it will complete. Code in the `finally` block could itself raise an exception or issue a `System.exit()`.
- ❑ Uncaught exceptions propagate back through the call stack, starting from the method where the exception is thrown and ending with either the first method that has a corresponding catch for that exception type or a JVM shutdown (which happens if the exception gets to `main()`, and `main()` is "ducking" the exception by declaring it).
- ❑ You can create your own exceptions, normally by extending `Exception` or one of its subtypes. Your exception will then be considered a checked exception, and the compiler will enforce the handle or declare rule for that exception.
- ❑ All `catch` blocks must be ordered from most specific to most general. If you have a `catch` clause for both `IOException` and `Exception`, you must put the `catch` for `IOException` first in your code. Otherwise, the `IOException` would be caught by `catch(Exception e)`, because a `catch` argument can catch the specified exception or any of its subtypes! The compiler will stop you from defining `catch` clauses that can never be reached.
- ❑ Some exceptions are created by programmers, some by the JVM.

Working with the Assertion Mechanism (Objective 2.3)

- Assertions give you a way to test your assumptions during development and debugging.
- Assertions are typically enabled during testing but disabled during deployment.
- You can use `assert` as a keyword (as of version 1.4) or an identifier, but not both together. To compile older code that uses `assert` as an identifier (for example, a method name), use the `-source 1.3` command-line flag to `javac`.
- 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.

SELF TEST

1. Given two files:

```
1. class One {
2.   public static void main(String[] args) {
3.     int assert = 0;
4.   }
5. }
```

```
1. class Two {
2.   public static void main(String[] args) {
3.     assert(false);
4.   }
5. }
```

And the four command-line invocations:

```
javac -source 1.3 One.java
javac -source 1.4 One.java
javac -source 1.3 Two.java
javac -source 1.4 Two.java
```

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

A. Only one compilation will succeed
B. Exactly two compilations will succeed
C. Exactly three compilations will succeed
D. All four compilations will succeed
E. No compiler warnings will be produced
F. At least one compiler warning will be produced

2. Given:

```
class Plane {
  static String s = "-";
  public static void main(String[] args) {
    new Plane().s1();
    System.out.println(s);
  }
  void s1() {
    try { s2(); }
    catch (Exception e) { s += "c"; }
  }
  void s2() throws Exception  {
```

```
    s3();  s += "2";
    s3();  s += "2b";
  }
  void s3() throws Exception {
    throw new Exception();
  }
}
```

What is the result?

A. `-`
B. `-c`
C. `-c2`
D. `-2c`
E. `-c22b`
F. `-2c2b`
G. `-2c2bc`
H. Compilation fails

3. Given:

```
try { int x = Integer.parseInt("two"); }
```

Which could be used to create an appropriate `catch` block? (Choose all that apply.)

A. `ClassCastException`
B. `IllegalStateException`
C. `NumberFormatException`
D. `IllegalArgumentException`
E. `ExceptionInInitializerError`
F. `ArrayIndexOutOfBoundsException`

4. 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

5. Given:

```
1. class Loopy {
2.   public static void main(String[] args) {
3.     int[] x = {7,6,5,4,3,2,1};
4.     // insert code here
5.       System.out.print(y + " ");
6.     }
7.   }
8. }
```

Which, inserted independently at line 4, compiles? (Choose all that apply.)

A. `for(int y : x) {`

B. `for(x : int y) {`

C. `int y = 0; for(y : x) {`

D. `for(int y=0, z=0; z<x.length; z++) { y = x[z];`

E. `for(int y=0, int z=0; z<x.length; z++) { y = x[z];`

F. `int y = 0; for(int z=0; z<x.length; z++) { y = x[z];`

6. Given:

```
class Emu {
  static String s = "-";
  public static void main(String[] args) {
    try {
      throw new Exception();
    } catch (Exception e) {
        try {
          try { throw new Exception();
          } catch (Exception ex) { s += "ic "; }
          throw new Exception(); }
        catch (Exception x) { s += "mc "; }
        finally { s += "mf "; }
    } finally { s += "of "; }
    System.out.println(s);
} }
```

What is the result?

A. `-ic of`

B. `-mf of`

C. `-mc mf`

D. `-ic mf of`

E. `-ic mc mf of`

F. `-ic mc of mf`

G. Compilation fails

7. Given:

```
 3. class SubException extends Exception { }
 4. class SubSubException extends SubException { }
 5.
 6. public class CC { void doStuff() throws SubException { } }
 7.
 8. class CC2 extends CC { void doStuff() throws SubSubException { } }
 9.
10. class CC3 extends CC { void doStuff() throws Exception { } }
11.
12. class CC4 extends CC { void doStuff(int x) throws Exception { } }
13.
14. class CC5 extends CC {  void doStuff()  { } }
```

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

A. Compilation succeeds

B. Compilation fails due to an error on line 8

C. Compilation fails due to an error on line 10

D. Compilation fails due to an error on line 12

E. Compilation fails due to an error on line 14

8. Given:

```
 3. public class Ebb {
 4.   static int x = 7;
 5.   public static void main(String[] args) {
 6.     String s = "";
 7.     for(int y = 0; y < 3; y++) {
 8.       x++;
 9.       switch(x) {
10.         case 8: s += "8 ";
11.         case 9: s += "9 ";
12.         case 10: { s+= "10 "; break; }
13.         default: s += "d ";
14.         case 13: s+= "13 ";
```

```
15.        }
16.      }
17.      System.out.println(s);
18.    }
19.    static { x++; }
20. }
```

What is the result?

A. `9 10 d`

B. `8 9 10 d`

C. `9 10 10 d`

D. `9 10 10 d 13`

E. `8 9 10 10 d 13`

F. `8 9 10 9 10 10 d 13`

G. Compilation fails

9. Given:

```
 3. class Infinity { }
 4. public class Beyond extends Infinity {
 5.   static Integer i;
 6.   public static void main(String[] args) {
 7.     int sw = (int)(Math.random() * 3);
 8.     switch(sw) {
 9.       case 0: {  for(int x = 10; x > 5; x++)
10.                   if(x > 10000000) x = 10;
11.                 break; }
12.       case 1: {  int y = 7 * i;  break;  }
13.       case 2: {  Infinity inf = new Beyond();
14.                  Beyond b = (Beyond)inf;  }
15.     }
16.   }
17. }
```

And given that line 7 will assign the value 0, 1, or 2 to `sw`, which are true? (Choose all that apply.)

A. Compilation fails

B. A ClassCastException might be thrown

C. A StackOverflowError might be thrown

D. A NullPointerException might be thrown

E. An IllegalStateException might be thrown

F. The program might hang without ever completing

G. The program will always complete without exception

10. Given:

```
 3. public class Circles {
 4.   public static void main(String[] args) {
 5.     int[] ia = {1,3,5,7,9};
 6.     for(int x : ia) {
 7.       for(int j = 0; j < 3; j++) {
 8.         if(x > 4 && x < 8) continue;
 9.         System.out.print(" " + x);
10.         if(j == 1) break;
11.         continue;
12.       }
13.       continue;
14.     }
15.   }
16. }
```

What is the result?

A. `1 3 9`

B. `5 5 7 7`

C. `1 3 3 9 9`

D. `1 1 3 3 9 9`

E. `1 1 1 3 3 3 9 9 9`

F. Compilation fails

11. Given:

```
 3. public class OverAndOver {
 4.   static String s = "";
 5.   public static void main(String[] args) {
 6.     try {
 7.       s += "1";
 8.       throw new Exception();
 9.     } catch (Exception e) { s += "2";
10.     } finally { s += "3"; doStuff(); s += "4";
11.     }
```

```
12.     System.out.println(s);
13.   }
14.   static void doStuff() { int x = 0; int y = 7/x; }
15. }
```

What is the result?

A. `12`

B. `13`

C. `123`

D. `1234`

E. Compilation fails

F. `123` followed by an exception

G. `1234` followed by an exception

H. An exception is thrown with no other output

12. Given:

```
 3. public class Wind {
 4.   public static void main(String[] args) {
 5.     foreach:
 6.     for(int j=0; j<5; j++) {
 7.       for(int k=0; k< 3; k++) {
 8.         System.out.print(" " + j);
 9.         if(j==3 && k==1) break foreach;
10.         if(j==0 || j==2) break;
11.       }
12.     }
13.   }
14. }
```

What is the result?

A. `0 1 2 3`

B. `1 1 1 3 3`

C. `0 1 1 1 2 3 3`

D. `1 1 1 3 3 4 4 4`

E. `0 1 1 1 2 3 3 4 4 4`

F. Compilation fails

13. Given:

```
 3. public class Gotcha {
 4.   public static void main(String[] args) {
 5.     // insert code here
 6.
 7.   }
 8.   void go() {
 9.     go();
10.   }
11. }
```

And given the following three code fragments:

```
I.   new Gotcha().go();
II.  try { new Gotcha().go(); }
     catch (Error e) { System.out.println("ouch"); }

III. try { new Gotcha().go(); }
     catch (Exception e) { System.out.println("ouch"); }
```

When fragments I - III are added, independently, at line 5, which are true? (Choose all that apply.)

A. Some will not compile
B. They will all compile
C. All will complete normally
D. None will complete normally
E. Only one will complete normally
F. Two of them will complete normally

14. 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

15. Given:

```
1. public class Frisbee {
2.   // insert code here
3.     int x = 0;
4.     System.out.println(7/x);
5.   }
6. }
```

And given the following four code fragments:

```
I.   public static void main(String[] args) {
II.  public static void main(String[] args) throws Exception {
III. public static void main(String[] args) throws IOException {
IV.  public static void main(String[] args) throws RuntimeException {
```

If the four fragments are inserted independently at line 4, which are true? (Choose all that apply.)

A. All four will compile and execute without exception

B. All four will compile and execute and throw an exception

C. Some, but not all, will compile and execute without exception

D. Some, but not all, will compile and execute and throw an exception

E. When considering fragments II, III, and IV, of those that will compile, adding a try/catch block around line 6 will cause compilation to fail

16. Given:

```
 2. class MyException extends Exception { }
 3. class Tire {
 4.   void doStuff() {  }
 5. }
 6. public class Retread extends Tire {
 7.   public static void main(String[] args) {
 8.     new Retread().doStuff();
 9.   }
10.   // insert code here
11.     System.out.println(7/0);
12.   }
13. }
```

And given the following four code fragments:

```
I.   void doStuff() {
II.  void doStuff() throws MyException {
III. void doStuff() throws RuntimeException {
IV.  void doStuff() throws ArithmeticException {
```

When fragments I - IV are added, independently, at line 10, which are true? (Choose all that apply.)

A. None will compile

B. They will all compile

C. Some, but not all, will compile

D. All of those that compile will throw an exception at runtime

E. None of those that compile will throw an exception at runtime

F. Only some of those that compile will throw an exception at runtime

SELF TEST ANSWERS

1. Given two files:

```
1. class One {
2.   public static void main(String[] args) {
3.     int assert = 0;
4.   }
5. }
1. class Two {
2.   public static void main(String[] args) {
3.     assert(false);
4.   }
5. }
```

And the four command-line invocations:

```
javac -source 1.3 One.java
javac -source 1.4 One.java
javac -source 1.3 Two.java
javac -source 1.4 Two.java
```

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

A. Only one compilation will succeed
B. Exactly two compilations will succeed
C. Exactly three compilations will succeed
D. All four compilations will succeed
E. No compiler warnings will be produced
F. At least one compiler warning will be produced

Answer:

☑ **B** and **F** are correct. Class One will compile (and issue a warning) using the `1.3` flag, and class Two will compile using the `1.4` flag.

☒ **A, C, D,** and **E** are incorrect based on the above. (Objective 2.3)

2. Given:

```
class Plane {
  static String s = "-";
  public static void main(String[] args) {
    new Plane().s1();
```

```
    System.out.println(s);
  }
  void s1() {
    try { s2(); }
    catch (Exception e) { s += "c"; }
  }
  void s2() throws Exception  {
    s3();  s += "2";
    s3();  s += "2b";
  }
  void s3() throws Exception {
    throw new Exception();
} }
```

What is the result?

A. `-`
B. `-c`
C. `-c2`
D. `-2c`
E. `-c22b`
F. `-2c2b`
G. `-2c2bc`
H. Compilation fails

Answer:

☑ **B** is correct. Once `s3()` throws the exception to `s2()`, `s2()` throws it to `s1()`, and no more of `s2()`'s code will be executed.

☒ **A, C, D, E, F, G,** and **H** are incorrect based on the above. (Objective 2.5)

3. Given:

```
try { int x = Integer.parseInt("two"); }
```

Which could be used to create an appropriate `catch` block? (Choose all that apply.)

A. `ClassCastException`
B. `IllegalStateException`
C. `NumberFormatException`
D. `IllegalArgumentException`

E. `ExceptionInInitializerError`

F. `ArrayIndexOutOfBoundsException`

Answer:

☑ **C** and **D** are correct. `Integer.parseInt` can throw a `NumberFormatException`, and `IllegalArgumentException` is its superclass (i.e., a broader exception).

☒ **A, B, E,** and **F** are not in `NumberFormatException`'s class hierarchy. (Objective 2.6)

4. 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

Answer:

☑ **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, Sun discourages you from doing so. (Objective 2.3)

5. Given:

```
1. class Loopy {
2.   public static void main(String[] args) {
3.     int[] x = {7,6,5,4,3,2,1};
4.     // insert code here
5.       System.out.print(y + " ");
6.     }
7. } }
```

Which, inserted independently at line 4, compiles? (Choose all that apply.)

A. `for(int y : x) {`

B. `for(x : int y) {`

C. `int y = 0; for(y : x) {`

D. `for(int y=0, z=0; z<x.length; z++) { y = x[z];`

E. `for(int y=0, int z=0; z<x.length; z++) { y = x[z];`

F. `int y = 0; for(int z=0; z<x.length; z++) { y = x[z];`

Answer:

☑ **A, D,** and **F** are correct. **A** is an example of the enhanced `for` loop. **D** and **F** are examples of the basic `for` loop.

☒ **B** is incorrect because its operands are swapped. **C** is incorrect because the enhanced `for` must declare its first operand. **E** is incorrect syntax to declare two variables in a `for` statement. (Objective 2.2)

6. Given:

```
class Emu {
  static String s = "-";
  public static void main(String[] args) {
    try {
      throw new Exception();
    } catch (Exception e) {
        try {
          try { throw new Exception();
          } catch (Exception ex) { s += "ic "; }
          throw new Exception(); }
        catch (Exception x) { s += "mc "; }
        finally { s += "mf "; }
    } finally { s += "of "; }
    System.out.println(s);
} }
```

What is the result?

A. `-ic of`

B. `-mf of`

C. `-mc mf`

D. `-ic mf of`

E. `-ic mc mf of`

F. `-ic mc of mf`

G. Compilation fails

Answer:

☑ **E** is correct. There is no problem nesting `try / catch` blocks. As is normal, when an exception is thrown, the code in the `catch` block runs, then the code in the `finally` block runs.

☒ **A, B, C, D,** and **F** are incorrect based on the above. (Objective 2.5)

7. Given:

```
 3. class SubException extends Exception { }
 4. class SubSubException extends SubException { }
 5.
 6. public class CC { void doStuff() throws SubException { } }
 7.
 8. class CC2 extends CC { void doStuff() throws SubSubException { } }
 9.
10. class CC3 extends CC { void doStuff() throws Exception { } }
11.
12. class CC4 extends CC { void doStuff(int x) throws Exception { } }
13.
14. class CC5 extends CC {  void doStuff()  { } }
```

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

A. Compilation succeeds

B. Compilation fails due to an error on line 8

C. Compilation fails due to an error on line 10

D. Compilation fails due to an error on line 12

E. Compilation fails due to an error on line 14

Answer:

☑ **C** is correct. An overriding method cannot throw a broader exception than the method it's overriding. Class CC4's method is an overload, not an override.

☒ **A, B, D,** and **E** are incorrect based on the above. (Objectives 1.5, 2.4)

8. Given:

```
3. public class Ebb {
4.   static int x = 7;
5.   public static void main(String[] args) {
6.     String s = "";
```

```
 7.      for(int y = 0; y < 3; y++) {
 8.        x++;
 9.        switch(x) {
10.          case 8: s += "8 ";
11.          case 9: s += "9 ";
12.          case 10: { s+= "10 "; break; }
13.          default: s += "d ";
14.          case 13: s+= "13 ";
15.        }
16.      }
17.      System.out.println(s);
18.    }
19.    static { x++; }
20. }
```

What is the result?

A. `9 10 d`

B. `8 9 10 d`

C. `9 10 10 d`

D. `9 10 10 d 13`

E. `8 9 10 10 d 13`

F. `8 9 10 9 10 10 d 13`

G. Compilation fails

Answer:

☑ **D** is correct. Did you catch the static initializer block? Remember that switches work on "fall-thru" logic, and that fall-thru logic also applies to the default case, which is used when no other case matches.

☒ **A, B, C, E, F,** and **G** are incorrect based on the above. (Objective 2.1)

9. Given:

```
3. class Infinity { }
4. public class Beyond extends Infinity {
5.   static Integer i;
6.   public static void main(String[] args) {
7.     int sw = (int)(Math.random() * 3);
8.     switch(sw) {
9.       case 0: {  for(int x = 10; x > 5; x++)
```

```
10.                   if(x > 10000000) x = 10;
11.                 break; }
12.       case 1: {  int y = 7 * i;  break;  }
13.       case 2: {  Infinity inf = new Beyond();
14.                  Beyond b = (Beyond)inf;  }
15.     }
16.   }
17. }
```

And given that line 7 will assign the value 0, 1, or 2 to `sw`, which are true? (Choose all that apply.)

A. Compilation fails

B. A ClassCastException might be thrown

C. A StackOverflowError might be thrown

D. A NullPointerException might be thrown

E. An IllegalStateException might be thrown

F. The program might hang without ever completing

G. The program will always complete without exception

Answer:

☑ **D** and **F** are correct. Because `i` was not initialized, case 1 will throw an NPE. Case 0 will initiate an endless loop, not a stack overflow. Case 2's downcast will *not* cause an exception.

☒ **A, B, C, E,** and **G** are incorrect based on the above. (Objective 2.6)

10. Given:

```
3. public class Circles {
4.   public static void main(String[] args) {
5.     int[] ia = {1,3,5,7,9};
6.     for(int x : ia) {
7.       for(int j = 0; j < 3; j++) {
8.         if(x > 4 && x < 8) continue;
9.         System.out.print(" " + x);
10.        if(j == 1) break;
11.        continue;
12.      }
13.      continue;
14.    }
15.  }
16. }
```

What is the result?

A. `1 3 9`

B. `5 5 7 7`

C. `1 3 3 9 9`

D. `1 1 3 3 9 9`

E. `1 1 1 3 3 3 9 9 9`

F. Compilation fails

Answer:

☑ **D** is correct. The basic rule for unlabeled continue statements is that the current iteration stops early and execution jumps to the next iteration. The last two continue statements are redundant!

☒ **A, B, C, E,** and **F** are incorrect based on the above. (Objective 2.2)

11. Given:

```
 3. public class OverAndOver {
 4.   static String s = "";
 5.   public static void main(String[] args) {
 6.     try {
 7.       s += "1";
 8.       throw new Exception();
 9.     } catch (Exception e) { s += "2";
10.     } finally { s += "3"; doStuff(); s += "4";
11.     }
12.     System.out.println(s);
13.   }
14.   static void doStuff() { int x = 0; int y = 7/x; }
15. }
```

What is the result?

A. `12`

B. `13`

C. `123`

D. `1234`

E. Compilation fails

F. `123` followed by an exception

G. `1234` followed by an exception

H. An exception is thrown with no other output

Answer:

☑ **H** is correct. It's true that the value of `String s` is `123` at the time that the divide-by-zero exception is thrown, but `finally()` is *not* guaranteed to complete, and in this case `finally()` never completes, so the System.out.println (S.O.P.) never executes.

☒ **A, B, C, D, E, F,** and **G** are incorrect based on the above. (Objective 2.5)

12. Given:

```
 3. public class Wind {
 4.   public static void main(String[] args) {
 5.     foreach:
 6.     for(int j=0; j<5; j++) {
 7.       for(int k=0; k< 3; k++) {
 8.         System.out.print(" " + j);
 9.         if(j==3 && k==1) break foreach;
10.         if(j==0 || j==2) break;
11.       }
12.     }
13.   }
14. }
```

What is the result?

A. `0 1 2 3`

B. `1 1 1 3 3`

C. `0 1 1 1 2 3 3`

D. `1 1 1 3 3 4 4 4`

E. `0 1 1 1 2 3 3 4 4 4`

F. Compilation fails

Answer:

☑ **C** is correct. A `break` breaks out of the current innermost loop and continues. A labeled `break` breaks out of and terminates the current loops.

☒ **A, B, D, E,** and **F** are incorrect based on the above. (Objective 2.2)

13. Given:

```
 3. public class Gotcha {
 4.   public static void main(String[] args) {
 5.     // insert code here
 6.
 7.   }
 8.   void go() {
 9.     go();
10.   }
11. }
```

And given the following three code fragments:

```
I.   new Gotcha().go();
II.  try { new Gotcha().go(); }
     catch (Error e) { System.out.println("ouch"); }

III. try { new Gotcha().go(); }
     catch (Exception e) { System.out.println("ouch"); }
```

When fragments I - III are added, independently, at line 5, which are true? (Choose all that apply.)

A. Some will not compile

B. They will all compile

C. All will complete normally

D. None will complete normally

E. Only one will complete normally

F. Two of them will complete normally

Answer:

☑ **B** and **E** are correct. First off, `go()` is a badly designed recursive method, guaranteed to cause a StackOverflowError. Since `Exception` is not a superclass of `Error`, catching an `Exception` will not help handle an `Error`, so fragment III will not complete normally. Only fragment II will catch the `Error`.

☒ **A, C, D,** and **F** are incorrect based on the above. (Objective 2.5)

14. 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

Answer:

☑ **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. (Objective 2.3)

15. Given:

```
1. public class Frisbee {
2.   // insert code here
3.     int x = 0;
4.     System.out.println(7/x);
5.   }
6. }
```

And given the following four code fragments:

```
I.   public static void main(String[] args) {
II.  public static void main(String[] args) throws Exception {
III. public static void main(String[] args) throws IOException {
IV.  public static void main(String[] args) throws RuntimeException {
```

If the four fragments are inserted independently at line 4, which are true? (Choose all that apply.)

A. All four will compile and execute without exception

B. All four will compile and execute and throw an exception

C. Some, but not all, will compile and execute without exception

D. Some, but not all, will compile and execute and throw an exception

E. When considering fragments II, III, and IV, of those that will compile, adding a try/catch block around line 6 will cause compilation to fail

Answer:

☑ **D** is correct. This is kind of sneaky, but remember that we're trying to toughen you up for the real exam. If you're going to throw an IOException, you have to import the java.io package or declare the exception with a fully qualified name.

☒ **E** is incorrect because it's okay to both handle and declare an exception. **A, B,** and **C** are incorrect based on the above. (Objective 2.4)

16. Given:

```
 2. class MyException extends Exception { }
 3. class Tire {
 4.   void doStuff() {  }
 5. }
 6. public class Retread extends Tire {
 7.   public static void main(String[] args) {
 8.     new Retread().doStuff();
 9.   }
10.   // insert code here
11.     System.out.println(7/0);
12.   }
13. }
```

And given the following four code fragments:

```
I.   void doStuff() {
II.  void doStuff() throws MyException {
III. void doStuff() throws RuntimeException {
IV.  void doStuff() throws ArithmeticException {
```

When fragments I - IV are added, independently, at line 10, which are true? (Choose all that apply.)

A. None will compile

B. They will all compile

C. Some, but not all, will compile

D. All of those that compile will throw an exception at runtime

E. None of those that compile will throw an exception at runtime

F. Only some of those that compile will throw an exception at runtime

Answer:

☑ **C** and **D** are correct. An overriding method cannot throw checked exceptions that are broader than those thrown by the overridden method. However an overriding method *can* throw RuntimeExceptions not thrown by the overridden method.

☒ **A, B, E,** and **F** are incorrect based on the above. (Objective 2.4)

This page intentionally left blank

6

Strings, I/O, Formatting, and Parsing

CERTIFICATION OBJECTIVES

- Using String, StringBuilder, and StringBuffer
- File I/O using the java.io package
- Serialization using the java.io package
- Working with Dates, Numbers, and Currencies
- Using Regular Expressions

✓ Two-Minute Drill

Q&A Self Test

This chapter focuses on the various API-related topics that were added to the exam for Java 5 and remain in the Java 6 exam. J2SE comes with an enormous API, and a lot of your work as a Java programmer will revolve around using this API. The exam team chose to focus on APIs for I/O, formatting, and parsing. Each of these topics could fill an entire book. Fortunately, you won't have to become a total I/O or regex 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 String, I/O, formatting, and parsing objectives on the exam.

CERTIFICATION OBJECTIVE

String, StringBuilder, and StringBuffer (Exam Objective 3.1)

3.1 Discuss the differences between the String, StringBuilder, and StringBuffer classes.

Everything you needed to know about Strings in the SCJP 1.4 exam, you'll need to know for the SCJP 5 and SCJP 6 exams. plus, Sun added the *StringBuilder* class to the API, to provide faster, nonsynchronized StringBuffer capability. The StringBuilder class has exactly the same methods as the old StringBuffer class, but StringBuilder is faster because its methods aren't synchronized. Both classes give you String-like objects that handle some of the String class's shortcomings (like immutability).

The String Class

This section covers the String class, and the key concept to understand is that once a String object is created, it can never be changed—so what is happening when a String object seems to be changing? Let's find out.

Strings Are Immutable Objects

We'll start with a little background information about strings. You may not need this for the test, but a little context will help. Handling "strings" of characters is a fundamental aspect of most programming languages. In Java, each character in a

string is a 16-bit Unicode character. Because Unicode characters are 16 bits (not the skimpy 7 or 8 bits that ASCII provides), a rich, international set of characters is easily represented in Unicode.

In Java, strings are objects. Just like other objects, you can create an instance of a String with the new keyword, as follows:

```
String s = new String();
```

This line of code creates a new object of class String, and assigns it to the reference variable s. So far, String objects seem just like other objects. Now, let's give the String a value:

```
s = "abcdef";
```

As you might expect, the String class has about a zillion constructors, so you can use a more efficient shortcut:

```
String s = new String("abcdef");
```

And just because you'll use strings all the time, you can even say this:

```
String s = "abcdef";
```

There are some subtle differences between these options that we'll discuss later, but what they have in common is that they all create a new String object, with a value of "abcdef", and assign it to a reference variable s. Now let's say that you want a second reference to the String object referred to by s:

```
String s2 = s;   //  refer s2 to the same String as s
```

So far so good. String objects seem to be behaving just like other objects, so what's all the fuss about?…Immutability! (What the heck is immutability?) Once you have assigned a String a value, that value can never change— it's immutable, frozen solid, won't budge, fini, done. (We'll talk about why later, don't let us forget.) The good news is that while the String object is immutable, its reference variable is not, so to continue with our previous example:

```
s = s.concat(" more stuff");  // the concat() method 'appends'
                              // a literal to the end
```

Now wait just a minute, didn't we just say that Strings were immutable? So what's all this "appending to the end of the string" talk? Excellent question: let's look at what really happened…

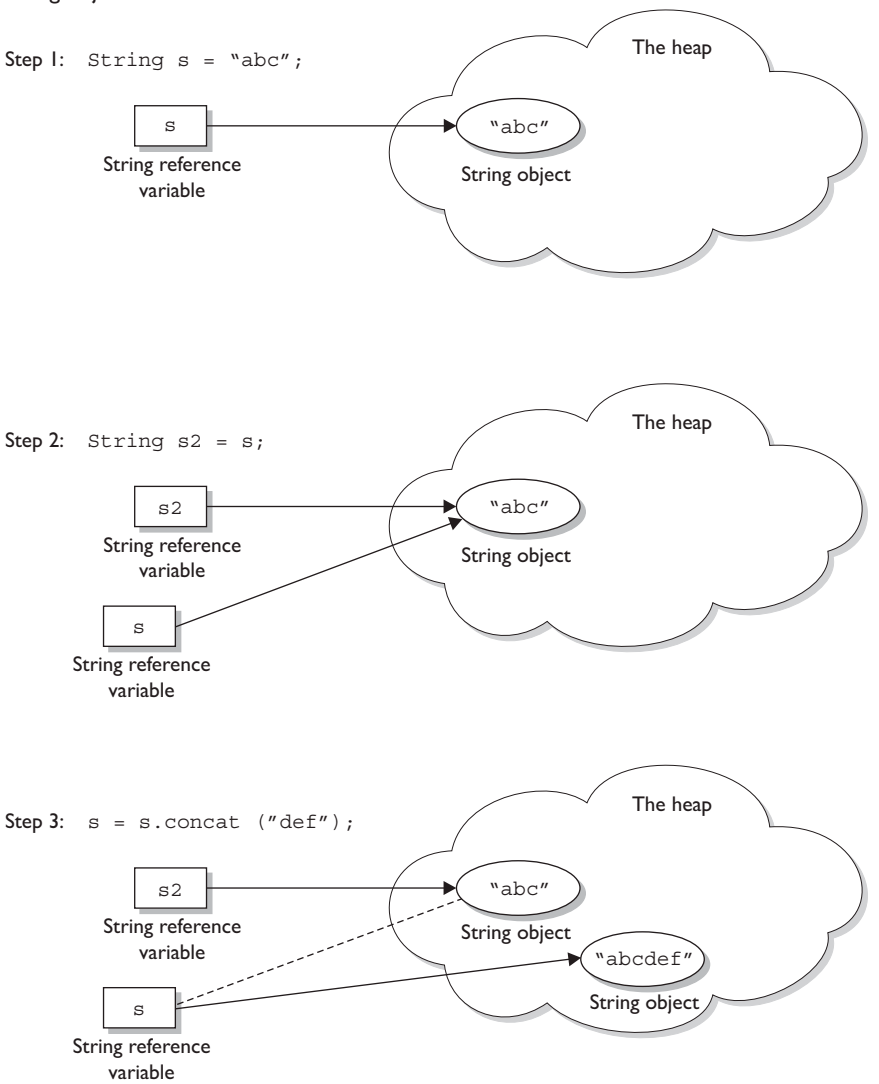
The VM took the value of String s (which was `"abcdef"`), and tacked `" more stuff"` onto the end, giving us the value `"abcdef more stuff"`. Since Strings are immutable, the VM couldn't stuff this new value into the old String referenced by s, so it created a new String object, gave it the value `"abcdef more stuff"`, and made s refer to it. At this point in our example, we have two String objects: the first one we created, with the value `"abcdef"`, and the second one with the value `"abcdef more stuff"`. Technically there are now three String objects, because the literal argument to concat, `" more stuff"`, is itself a new String object. But we have references only to `"abcdef"` (referenced by s2) and `"abcdef more stuff"` (referenced by s).

What if we didn't have the foresight or luck to create a second reference variable for the `"abcdef"` String before we called `s = s.concat(" more stuff");`? In that case, the original, unchanged String containing `"abcdef"` would still exist in memory, but it would be considered "lost." No code in our program has any way to reference it—it is lost to us. Note, however, that the original `"abcdef"` String didn't change (it can't, remember, it's immutable); only the reference variable s was changed, so that it would refer to a different String. Figure 6-1 shows what happens on the heap when you reassign a reference variable. Note that the dashed line indicates a deleted reference.

To review our first example:

```
String s = "abcdef";   // create a new String object, with
                       // value "abcdef", refer s to it
String s2 = s;         // create a 2nd reference variable
                       // referring to the same String

// create a new String object, with value "abcdef more stuff",
// refer s to it. (Change s's reference from the old String
// to the new String.) ( Remember s2 is still referring to
// the original "abcdef" String.)

s = s.concat(" more stuff");
```

FIGURE 6-1 String objects and their reference variables

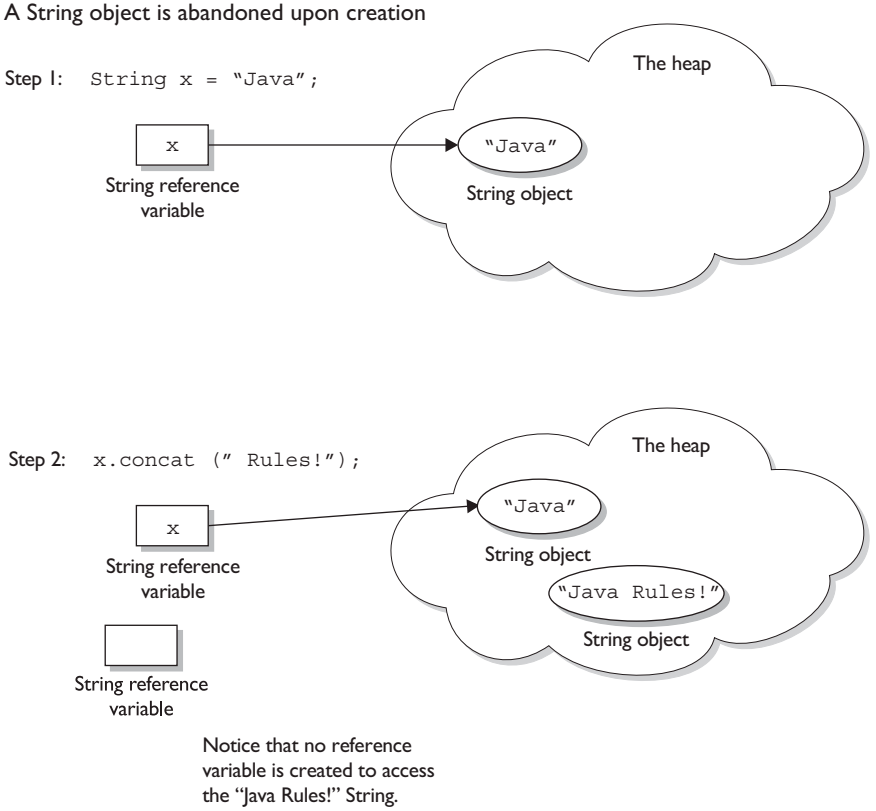


Let's look at another example:

```
String x = "Java";
x.concat(" Rules!");
System.out.println("x = " + x);  // the output is "x = Java"
```

The first line is straightforward: create a new String object, give it the value `"Java"`, and refer `x` to it. Next the VM creates a second String object with the value `"Java Rules!"` but nothing refers to it. The second String object is instantly lost; you can't get to it. The reference variable `x` still refers to the original String with the value `"Java"`. Figure 6-2 shows creating a String without assigning a reference to it.

FIGURE 6-2 A String object is abandoned upon creation



Let's expand this current example. We started with

```
String x = "Java";
x.concat(" Rules!");
System.out.println("x = " + x);   // the output is: x = Java
```

Now let's add

```
x.toUpperCase();
System.out.println("x = " + x);   // the output is still:
                                  // x = Java
```

(We actually did just create a new String object with the value `"JAVA"`, but it was lost, and `x` still refers to the original, unchanged String `"Java"`.) How about adding

```
x.replace('a', 'X');
System.out.println("x = " + x);   // the output is still:
                                  // x = Java
```

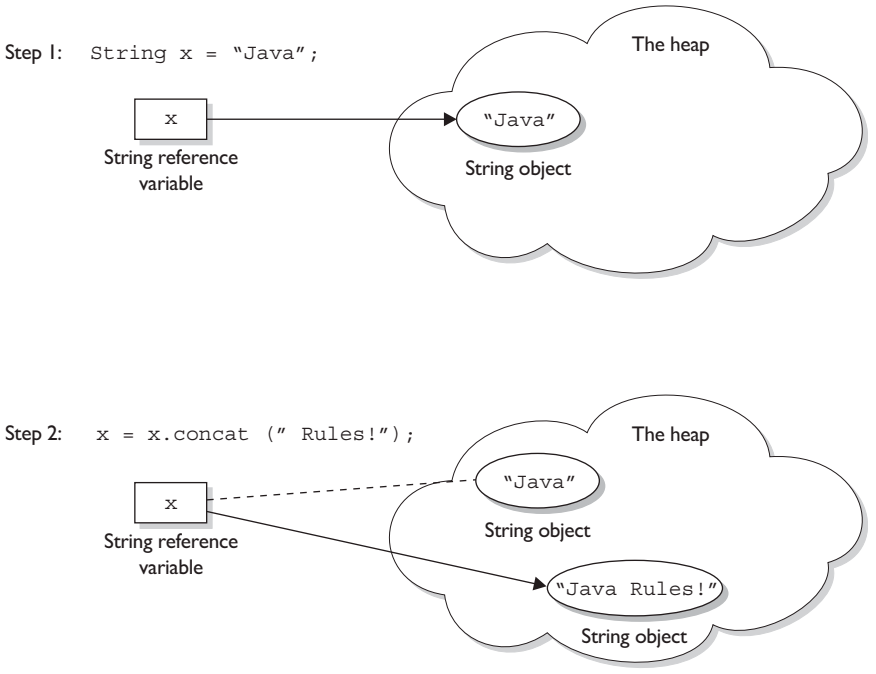
Can you determine what happened? The VM created yet another new String object, with the value `"JXvX"`, (replacing the `a`'s with `X`'s), but once again this new String was lost, leaving `x` to refer to the original unchanged and unchangeable String object, with the value `"Java"`. In all of these cases we called various String methods to create a new String by altering an existing String, but we never assigned the newly created String to a reference variable.

But we can put a small spin on the previous example:

```
String x = "Java";
x = x.concat(" Rules!");           // Now we're assigning the
                                   // new String to x
System.out.println("x = " + x);    // the output will be:
                                   // x = Java Rules!
```

This time, when the VM runs the second line, a new String object is created with the value of `"Java Rules!"`, and `x` is set to reference it. But wait, there's more—now the original String object, `"Java"`, has been lost, and no one is referring to it. So in both examples we created two String objects and only one reference variable, so one of the two String objects was left out in the cold. See Figure 6-3 for a graphic depiction of this sad story. The dashed line indicates a deleted reference.

FIGURE 6-3 An old String object being abandoned



Let's take this example a little further:

```
String x = "Java";
x = x.concat(" Rules!");
System.out.println("x = " + x);     // the output is:
                                    // x = Java Rules!

x.toLowerCase();                    // no assignment, create a
                                    // new, abandoned String

System.out.println("x = " + x);     // no assignment, the output
                                    // is still: x = Java Rules!
```

```
x = x.toLowerCase();                // create a new String,
                                    // assigned to x
System.out.println("x = " + x);     // the assignment causes the
                                    // output: x = java rules!
```

The preceding discussion contains the keys to understanding Java String immutability. If you really, really get the examples and diagrams, backward and forward, you should get 80 percent of the String questions on the exam correct.

We will cover more details about Strings next, but make no mistake—in terms of bang for your buck, what we've already covered is by far the most important part of understanding how String objects work in Java.

We'll finish this section by presenting an example of the kind of devilish String question you might expect to see on the exam. Take the time to work it out on paper (as a hint, try to keep track of how many objects and reference variables there are, and which ones refer to which).

```
String s1 = "spring ";
String s2 = s1 + "summer ";
s1.concat("fall ");
s2.concat(s1);
s1 += "winter ";
System.out.println(s1 + " " + s2);
```

What is the output? For extra credit, how many String objects and how many reference variables were created prior to the `println` statement?

Answer: The result of this code fragment is `spring winter  spring summer`. There are two reference variables, `s1` and `s2`. There were a total of eight String objects created as follows: "spring", "summer " (lost), "spring summer", "fall" (lost), "spring fall" (lost), "spring summer spring" (lost), "winter" (lost), "spring winter" (at this point "spring" is lost). Only two of the eight String objects are not lost in this process.

Important Facts About Strings and Memory

In this section we'll discuss how Java handles String objects in memory, and some of the reasons behind these behaviors.

One of the key goals of any good programming language is to make efficient use of memory. As applications grow, it's very common for String literals to occupy large amounts of a program's memory, and there is often a lot of redundancy within the

universe of String literals for a program. To make Java more memory efficient, the JVM sets aside a special area of memory called the "String constant pool." When the compiler encounters a String literal, it checks the pool to see if an identical String already exists. If a match is found, the reference to the new literal is directed to the existing String, and no new String literal object is created. (The existing String simply has an additional reference.) Now we can start to see why making String objects immutable is such a good idea. If several reference variables refer to the same String without even knowing it, it would be very bad if any of them could change the String's value.

You might say, "Well that's all well and good, but what if someone overrides the String class functionality; couldn't that cause problems in the pool?" That's one of the main reasons that the String class is marked `final`. Nobody can override the behaviors of any of the String methods, so you can rest assured that the String objects you are counting on to be immutable will, in fact, be immutable.

Creating New Strings

Earlier we promised to talk more about the subtle differences between the various methods of creating a String. Let's look at a couple of examples of how a String might be created, and let's further assume that no other String objects exist in the pool:

```
String s = "abc";     //  creates one String object and one
                      //  reference variable
```

In this simple case, "abc" will go in the pool and `s` will refer to it.

```
String s = new String("abc");  // creates two objects,
                               // and one reference variable
```

In this case, because we used the `new` keyword, Java will create a new String object in normal (nonpool) memory, and `s` will refer to it. In addition, the literal "`abc`" will be placed in the pool.

Important Methods in the String Class

The following methods are some of the more commonly used methods in the String class, and also the ones that you're most likely to encounter on the exam.

- **charAt()** Returns the character located at the specified index
- **concat()** Appends one String to the end of another ("+" also works)
- **equalsIgnoreCase()** Determines the equality of two Strings, ignoring case
- **length()** Returns the number of characters in a String
- **replace()** Replaces occurrences of a character with a new character
- **substring()** Returns a part of a String
- **toLowerCase()** Returns a String with uppercase characters converted
- **toString()** Returns the value of a String
- **toUpperCase()** Returns a String with lowercase characters converted
- **trim()** Removes whitespace from the ends of a String

Let's look at these methods in more detail.

public char charAt(int index) This method returns the character located at the String's specified index. Remember, String indexes are zero-based—for example,

```
String x = "airplane";
System.out.println( x.charAt(2) );       //  output is 'r'
```

public String concat(String s) This method returns a String with the value of the String passed in to the method appended to the end of the String used to invoke the method—for example,

```
String x = "taxi";
System.out.println( x.concat(" cab") ); // output is "taxi cab"
```

The overloaded + and += operators perform functions similar to the `concat()` method—for example,

```
String x = "library";
System.out.println( x + " card");    // output is "library card"

String x = "Atlantic";
x+= " ocean";
System.out.println( x );         // output is "Atlantic ocean"
```

In the preceding "Atlantic ocean" example, notice that the value of x really did change! Remember that the += operator is an assignment operator, so line 2 is really creating a new String, "Atlantic ocean", and assigning it to the x variable. After line 2 executes, the original String x was referring to, "Atlantic", is abandoned.

public boolean equalsIgnoreCase(String s) This method returns a boolean value (true or false) depending on whether the value of the String in the argument is the same as the value of the String used to invoke the method. This method will return true even when characters in the String objects being compared have differing cases—for example,

```
String x = "Exit";
System.out.println( x.equalsIgnoreCase("EXIT"));   // is "true"
System.out.println( x.equalsIgnoreCase("tixe"));   // is "false"
```

public int length() This method returns the length of the String used to invoke the method—for example,

```
String x = "01234567";
System.out.println( x.length() );     // returns "8"
```

public String replace(char old, char new) This method returns a String whose value is that of the String used to invoke the method, updated so that any occurrence of the char in the first argument is replaced by the char in the second argument—for example,

```
String x = "oxoxoxox";
System.out.println( x.replace('x', 'X') );    // output is
                                              // "oXoXoXoX"
```

public String substring(int begin)
public String substring(int begin, int end) The substring() method is used to return a part (or substring) of the String used to invoke the method. The first argument represents the starting location (zero-based) of the substring. If the call has only one argument, the substring returned will include the characters to the end of the original String. If the call has two arguments, the substring returned will end with the character located in the *n*th position of the original String where *n* is the

exam watch

Arrays have an attribute (not a method), called `length`***. You may encounter questions in the exam that attempt to use the length() method on an array, or that attempt to use the length attribute on a String. Both cause compiler errors—for example,***

```
String x = "test";
System.out.println( x.length );     // compiler error
```

or

```
String[] x = new String[3];
System.out.println( x.length() );   // compiler error
```

second argument. Unfortunately, the ending argument is not zero-based, so if the second argument is 7, the last character in the returned String will be in the original String's 7 position, which is index 6 (ouch). Let's look at some examples:

```
String x = "0123456789";          // as if by magic, the value
                                  // of each char
                                  // is the same as its index!
System.out.println( x.substring(5) );     // output is  "56789"
System.out.println( x.substring(5, 8));   // output is "567"
```

The first example should be easy: start at index 5 and return the rest of the String. The second example should be read as follows: start at index 5 and return the characters up to and including the 8th position (index 7).

public String toLowerCase() This method returns a String whose value is the String used to invoke the method, but with any uppercase characters converted to lowercase—for example,

```
String x = "A New Moon";
System.out.println( x.toLowerCase() );    // output is
                                          // "a new moon"
```

public String toString() This method returns the value of the String used to invoke the method. What? Why would you need such a seemingly "do nothing" method? All objects in Java must have a `toString()` method, which typically returns a String that in some meaningful way describes the object in question. In the case of a String object, what more meaningful way than the String's value? For the sake of consistency, here's an example:

```
String x = "big surprise";
System.out.println( x.toString() );      // output -
                                         // reader's exercise
```

public String toUpperCase() This method returns a String whose value is the String used to invoke the method, but with any lowercase characters converted to uppercase—for example,

```
String x = "A New Moon";
System.out.println( x.toUpperCase() );     // output is
                                           // "A NEW MOON"
```

public String trim() This method returns a String whose value is the String used to invoke the method, but with any leading or trailing blank spaces removed—for example,

```
String x = "       hi        ";
System.out.println( x + "x" );              // result is
                                            // "     hi    x"
System.out.println( x.trim() + "x");        // result is "hix"
```

The StringBuffer and StringBuilder Classes

The java.lang.StringBuffer and java.lang.StringBuilder classes should be used when you have to make a lot of modifications to strings of characters. As we discussed in the previous section, String objects are immutable, so if you choose to do a lot of manipulations with String objects, you will end up with a lot of abandoned String objects in the String pool. (Even in these days of gigabytes of RAM, it's not a good idea to waste precious memory on discarded String pool objects.) On the other hand, objects of type StringBuffer and StringBuilder can be modified over and over again without leaving behind a great effluence of discarded String objects.

on the job

A common use for StringBuffers and StringBuilders is file I/O when large, ever-changing streams of input are being handled by the program. In these cases, large blocks of characters are handled as units, and StringBuffer objects are the ideal way to handle a block of data, pass it on, and then reuse the same memory to handle the next block of data.

StringBuffer vs. StringBuilder

The StringBuilder class was added in Java 5. It has exactly the same API as the StringBuffer class, except StringBuilder is not thread safe. In other words, its methods are not synchronized. (More about thread safety in Chapter 9.) Sun recommends that you use StringBuilder instead of StringBuffer whenever possible because StringBuilder will run faster (and perhaps jump higher). So apart from synchronization, anything we say about StringBuilder's methods holds true for StringBuffer's methods, and vice versa. The exam might use these classes in the creation of thread-safe applications, and we'll discuss how *that* works in Chapter 9.

Using StringBuilder and StringBuffer

In the previous section, we saw how the exam might test your understanding of String immutability with code fragments like this:

```
String x = "abc";
x.concat("def");
System.out.println("x = " + x);      //  output is "x = abc"
```

Because no new assignment was made, the new String object created with the `concat()` method was abandoned instantly. We also saw examples like this:

```
String x = "abc";
x = x.concat("def");
System.out.println("x = " + x);     // output is "x = abcdef"
```

We got a nice new String out of the deal, but the downside is that the old String "abc" has been lost in the String pool, thus wasting memory. If we were using a StringBuffer instead of a String, the code would look like this:

```
StringBuffer sb = new StringBuffer("abc");
sb.append("def");
System.out.println("sb = " + sb);      // output is "sb = abcdef"
```

All of the StringBuffer methods we will discuss operate on the value of the StringBuffer object invoking the method. So a call to `sb.append("def");` is actually appending "def" to itself (StringBuffer `sb`). In fact, these method calls can be chained to each other—for example,

```
StringBuilder sb = new StringBuilder("abc");
sb.append("def").reverse().insert(3, "---");
System.out.println( sb );              // output is  "fed---cba"
```

Notice that in each of the previous two examples, there was a single call to new, concordantly in each example we weren't creating any extra objects. Each example needed only a single StringXxx object to execute.

Important Methods in the StringBuffer and StringBuilder Classes

The following method returns a StringXxx object with the argument's value appended to the value of the object that invoked the method.

public synchronized StringBuffer append(String s) As we've seen earlier, this method will update the value of the object that invoked the method, whether or not the return is assigned to a variable. This method will take many different arguments, including boolean, char, double, float, int, long, and others, but the most likely use on the exam will be a String argument—for example,

```
StringBuffer sb = new StringBuffer("set ");
sb.append("point");
System.out.println(sb);       // output is "set point"
StringBuffer sb2 = new StringBuffer("pi = ");
sb2.append(3.14159f);
System.out.println(sb2);      // output is  "pi = 3.14159"
```

public StringBuilder delete(int start, int end) This method returns a StringBuilder object and updates the value of the StringBuilder object that invoked the method call. In both cases, a substring is removed from the original object. The starting index of the substring to be removed is defined by the first argument (which is zero-based), and the ending index of the substring to be removed is defined by the second argument (but it is one-based)! Study the following example carefully:

```
StringBuilder sb = new StringBuilder("0123456789");
System.out.println(sb.delete(4,6));      // output is "01236789"
```

exam Watch

The exam will probably test your knowledge of the difference between String and StringBuffer objects. Because StringBuffer objects are changeable, the following code fragment will behave differently than a similar code fragment that uses String objects:

```
StringBuffer sb = new StringBuffer("abc");
sb.append("def");
System.out.println( sb );
```

In this case, the output will be: `"abcdef"`

public StringBuilder insert(int offset, String s) This method returns a StringBuilder object and updates the value of the StringBuilder object that invoked the method call. In both cases, the String passed in to the second argument is inserted into the original StringBuilder starting at the offset location represented by the first argument (the offset is zero-based). Again, other types of data can be passed in through the second argument (`boolean`, `char`, `double`, `float`, `int`, `long`, and so on), but the String argument is the one you're most likely to see:

```
StringBuilder sb = new StringBuilder("01234567");
sb.insert(4, "---");
System.out.println( sb );          //   output is  "0123---4567"
```

public synchronized StringBuffer reverse() This method returns a StringBuffer object and updates the value of the StringBuffer object that invoked the method call. In both cases, the characters in the StringBuffer are reversed, the first character becoming the last, the second becoming the second to the last, and so on:

```
StringBuffer s = new StringBuffer("A man a plan a canal Panama");
sb.reverse();
System.out.println(sb); // output: "amanaP lanac a nalp a nam A"
```

public String toString() This method returns the value of the StringBuffer object that invoked the method call as a String:

```
StringBuffer sb = new StringBuffer("test string");
System.out.println( sb.toString() );  // output is "test string"
```

That's it for StringBuffers and StringBuilders. If you take only one thing away from this section, it's that unlike Strings, StringBuffer objects and StringBuilder objects can be changed.

exam Watch

Many of the exam questions covering this chapter's topics use a tricky (and not very readable) bit of Java syntax known as "chained methods." A statement with chained methods has this general form:

```
result = method1().method2().method3();
```

In theory, any number of methods can be chained in this fashion, although typically you won't see more than three. Here's how to decipher these "handy Java shortcuts" when you encounter them:

1. Determine what the leftmost method call will return (let's call it x).
2. Use x as the object invoking the second (from the left) method. If there are only two chained methods, the result of the second method call is the expression's result.
3. If there is a third method, the result of the second method call is used to invoke the third method, whose result is the expression's result—for example,

```
String x = "abc";
String y = x.concat("def").toUpperCase().replace('C','x');
//chained methods
System.out.println("y = " + y); // result is "y = ABxDEF"
```

Let's look at what happened. The literal `def` ***was concatenated to*** `abc`***, creating a temporary, intermediate String (soon to be lost), with the value*** `abcdef`***. The*** `toUpperCase()` ***method created a new (soon to be lost) temporary String with the value*** `ABCDEF`***. The*** `replace()` ***method created a final String with the value*** `ABxDEF`***, and referred*** `y` ***to it.***

CERTIFICATION OBJECTIVE

File Navigation and I/O (Exam Objective 3.2)

3.2 Given a scenario involving navigating file systems, reading from files, or writing to files, develop the correct solution using the following classes (sometimes in combination), from java.io: BufferedReader, BufferedWriter, File, FileReader, FileWriter, PrintWriter, and Console.

I/O has had a strange history with the SCJP certification. It was included in all the versions of the exam up to and including 1.2, then removed from the 1.4 exam, and then re-introduced for Java 5 and extended for Java 6.

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 5 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 class File 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 very flexible and powerful.

- **Console** This new, Java 6 convenience class provides methods to read input from the console and write formatted output to the console.

exam Watch

Stream classes are used to read and write bytes, and Readers and Writers are used to read and write characters. Since all of the file I/O on the exam is related to characters, if you see API class names containing the word "Stream", for instance DataOutputStream, then the question is probably about serialization, or something unrelated to the actual I/O objective.

Creating Files Using Class File

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 Java 6 exam focuses 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
                   ("fileWrite1.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 that 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:

- **`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.

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());
```

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 bytes 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 `size`, and we looped through the `in` array printing out each character we read, 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 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.

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 very specific purpose (creating high cohesion), 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 pouring 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 we've artificially reduced the API. In terms of studying for exam Objective 3.2, we can imagine that the entire java.io package consisted of the classes listed in exam Objective 3.2, and summarized in Table 6-1, our mini I/O API.

TABLE 6-1 java.io Mini API

java.io Class	Extends From	Key Constructor(s) Arguments	Key Methods
File	Object	File, String String String, String	createNewFile() delete() exists() isDirectory() isFile() list() mkdir() renameTo()
FileWriter	Writer	File String	close() flush() write()
BufferedWriter	Writer	Writer	close() flush() newLine() write()
PrintWriter	Writer	File (as of Java 5) String (as of Java 5) OutputStream Writer	close() flush() format()*, printf()* print(), println() write()
FileReader	Reader	File String	read()
BufferedReader	Reader	Reader	read() readLine()
			*Discussed later

Now let's say that 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 6-1 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, Table 6-1.

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 6-1 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:

1. It can be constructed using a File.
2. It extends Writer.

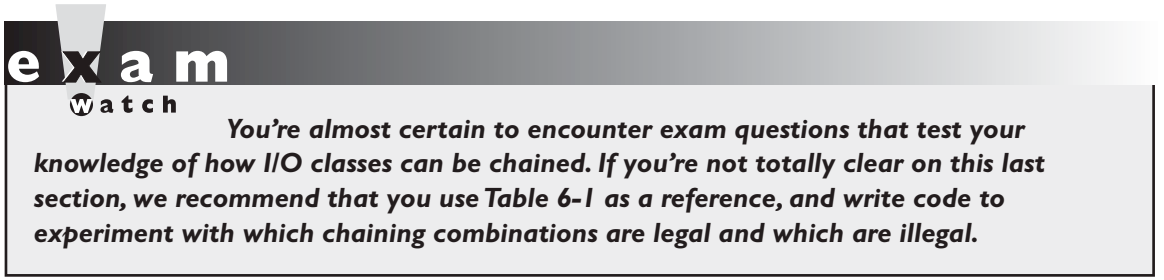
Given all of this information, we can put together the following code (remember, this is a Java 1.4 example):

```
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 that you use Table 6-1 as a reference, and write 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 File object, capital D, (even though it's of type File.) We'll call an actual directory on a computer a directory, 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 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 that we already have a subdirectory called `existingDir` in which resides an existing file `existingDirFile.txt`, which contains several lines of text. When you 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 also 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 that 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, which 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 vary :)

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, and serialization, which is covered in the next section).

The java.io.Console Class

New to Java 6 is 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) {
    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. Later in this chapter, when you're studying regex and formatting, 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

Serialization (Exam Objective 3.3)

3.3 Develop code that serializes and/or de-serializes objects using the following APIs from java.io: DataInputStream, DataOutputStream, FileInputStream, FileOutputStream, ObjectInputStream, ObjectOutputStream, and Serializable.

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 `int`s 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 that 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 set of 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—like 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(); }
  }
}
```

Let's take a look at the preceding code.

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."

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.

exam 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 if 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 vs. 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 deserialized. When an instance of a serializable class is deserialized, 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 deserialized, 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 deserialized we do NOT want any of the normal initialization to happen. We don't want the constructor to run, and we don't want the explicitly declared values to be assigned. We want only the values saved as part of the serialized state of the object to be reassigned.

Of course if you have variables marked `transient`, they will not be restored to their original state (unless you implement `readObject()`), but will instead be given the default value for that data type. In other words, even if you say

```
class Bar implements Serializable {
   transient int x = 42;
}
```

when the Bar instance is deserialized, the variable `x` will be set to a value of `0`. Object references marked `transient` will always be reset to `null`, regardless of whether they were initialized at the time of declaration in the class.

So, that's what happens when the object is deserialized, and the class of the serialized object directly extends `Object`, or has ONLY serializable classes in its inheritance tree. It gets a little trickier when the serializable class has one or more non-serializable superclasses. Getting back to our non-serializable Animal class with a serializable Dog subclass example:

```
class Animal {
   public String name;
}
class Dog extends Animal implements Serializable {
   // the rest of the Dog code
}
```

Because Animal is NOT serializable, any state maintained in the Animal class, even though the state variable is inherited by the Dog, isn't going to be restored with the Dog when it's deserialized! The reason is, the (unserialized) Animal part of the Dog is going to be reinitialized just as it would be if you were making a new Dog (as opposed to deserializing one). That means all the things that happen to an object

during construction, will happen—but only to the Animal parts of a Dog. In other words, the instance variables from the Dog's class will be serialized and deserialized correctly, but the inherited variables from the non-serializable Animal superclass will come back with their default/initially assigned values rather than the values they had at the time of serialization.

If you are a serializable class, but your superclass is NOT serializable, then any instance variables you INHERIT from that superclass will be reset to the values they were given during the original construction of the object. This is because the non-serializable class constructor WILL run!

In fact, every constructor ABOVE the first non-serializable class constructor will also run, no matter what, because once the first super constructor is invoked, (during deserialization), it of course invokes its super constructor and so on up the inheritance tree.

For the exam, you'll need to be able to recognize which variables will and will not be restored with the appropriate values when an object is deserialized, so be sure to study the following code example and the output:

```
import java.io.*;
class SuperNotSerial {
  public static void main(String [] args) {

    Dog d = new Dog(35, "Fido");
    System.out.println("before: " + d.name + " "
                       + d.weight);
    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:  " + d.name + " "
                       + d.weight);
  }
}
class Dog extends Animal implements Serializable {
  String name;
```

```
  Dog(int w, String n) {
    weight = w;          // inherited
    name = n;            // not inherited
  }
}
class Animal {           // not serializable !
  int weight = 42;
}
```

which produces the output:

```
before: Fido 35
after:  Fido 42
```

The key here is that because Animal is not serializable, when the Dog was deserialized, the Animal constructor ran and reset the Dog's inherited weight variable.

exam Watch

If you serialize a collection or an array, every element must be serializable! A single non-serializable element will cause serialization to fail. Note also that while the collection interfaces are not serializable, the concrete collection classes in the Java API are.

Serialization Is Not for Statics

Finally, you might notice that we've talked ONLY about instance variables, not static variables. Should static variables be saved as part of the object's state? Isn't the state of a static variable at the time an object was serialized important? Yes and no. It might be important, but it isn't part of the instance's state at all. Remember, you should think of static variables purely as CLASS variables. They have nothing to do with individual instances. But serialization applies only to OBJECTS. And what happens if you deserialize three different Dog instances, all of which were serialized at different times, and all of which were saved when the value of a static variable in class Dog was different. Which instance would "win"? Which instance's static value would be used to replace the one currently in the one and only Dog class that's currently loaded? See the problem?

Static variables are NEVER saved as part of the object's state…because they do not belong to the object!

exam Watch

What about DataInputStream and DataOutputStream? They're in the objectives! It turns out that while the exam was being created, it was decided that those two classes wouldn't be on the exam after all, but someone forgot to remove them from the objectives! So you get a break. That's one less thing you'll have to worry about.

on the Job

As simple as serialization code is to write, versioning problems can occur in the real world. If you save a Dog object using one version of the class, but attempt to deserialize it using a newer, different version of the class, deserialization might fail. See the Java API for details about versioning issues and solutions.

CERTIFICATION OBJECTIVE

Dates, Numbers, and Currency (Exam Objective 3.4)

3.4 Use standard J2SE APIs in the java.text package to correctly format or parse dates, numbers and currency values for a specific locale; and, given a scenario, determine the appropriate methods to use if you want to use the default locale or a specific locale. Describe the purpose and use of the java.util.Locale class.

The Java API provides an extensive (perhaps a little *too* extensive) set of classes to help you work with dates, numbers, and currency. 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 new Date and DateFormat objects, converting Strings to Dates and back again, performing Calendaring functions, printing properly formatted currency values, 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").

Working with Dates, Numbers, and Currencies

If you want to work with dates from around the world (and who doesn't?), you'll need to be familiar with at least four classes from the java.text and java.util packages. In fact, we'll admit it right up front, you might encounter questions on the exam that use classes that aren't specifically mentioned in the Sun objective. Here are the four date related classes you'll need to understand:

- **`java.util.Date`** Most of this class's methods have been deprecated, but you can use this class to bridge between the `Calendar` and `DateFormat` class. An instance of `Date` represents a mutable date and time, to a millisecond.
- **`java.util.Calendar`** This class provides a huge variety of methods that help you convert and manipulate dates and times. For instance, if you want to add a month to a given date, or find out what day of the week January 1, 3000 falls on, the methods in the `Calendar` class will save your bacon.
- **`java.text.DateFormat`** This class is used to format dates not only providing various styles such as "01/01/70" or "January 1, 1970," but also to format dates for numerous locales around the world.
- **`java.text.NumberFormat`** This class is used to format numbers and currencies for locales around the world.
- **`java.util.Locale`** This class is used in conjunction with `DateFormat` and `NumberFormat` to format dates, numbers and currency for specific locales. With the help of the `Locale` class you'll be able to convert a date like "10/10/2005" to "Segunda-feira, 10 de Outubro de 2005" in no time. If you want to manipulate dates without producing formatted output, you can use the Locale class directly with the Calendar class.

Orchestrating Date- and Number-Related Classes

When you work with dates and numbers, you'll often use several classes together. It's important to understand how the classes we described above relate to each other, and when to use which classes in combination. For instance, you'll need to know that if you want to do date formatting for a specific locale, you need to create your Locale object before your DateFormat object, because you'll need your Locale object as an argument to your DateFormat factory method. Table 6-2 provides a quick overview of common date- and number-related use cases and solutions using these classes. Table 6-2 will undoubtedly bring up specific questions about individual classes, and we will dive into specifics for each class next. Once you've gone through the class level discussions, you should find that Table 6-2 provides a good summary.

TABLE 6-2 Common Use Cases When Working with Dates and Numbers

Use Case	Steps
Get the current date and time.	1. Create a Date: `Date d = new Date();` 2. Get its value: `String s = d.toString();`
Get an object that lets you perform date and time calculations in your locale.	1. Create a Calendar: `Calendar c = Calendar.getInstance();` 2. Use `c.add(...)` and `c.roll(...)` to perform date and time manipulations.

Get an object that lets you perform date and time calculations in a different locale.	1. Create a Locale:
	`Locale loc = new Locale(language);` or
	`Locale loc = new Locale(language, country);`
	2. Create a Calendar for that locale:
	`Calendar c = Calendar.getInstance(loc);`
	3. Use `c.add(...)` and `c.roll(...)` to perform date and time manipulations.

Get an object that lets you perform date and time calculations, and then format it for output in different locales with different date styles.	1. Create a Calendar:
	`Calendar c = Calendar.getInstance();`
	2. Create a Locale for each location:
	`Locale loc = new Locale(...);`
	3. Convert your Calendar to a Date:
	`Date d = c.getTime();`
	4. Create a DateFormat for each Locale:
	`DateFormat df = DateFormat.getDateInstance (style, loc);`
	5. Use the format() method to create formatted dates:
	`String s = df.format(d);`

Get an object that lets you format numbers or currencies across many different locales.	1. Create a Locale for each location:
	`Locale loc = new Locale(...);`
	2. Create a NumberFormat:
	`NumberFormat nf = NumberFormat.getInstance(loc);`
	`-or- NumberFormat nf = NumberFormat. getCurrencyInstance(loc);`
	3. Use the format() method to create formatted output:
	`String s = nf.format(someNumber);`

The Date Class

The Date class has a checkered past. Its API design didn't do a good job of handling internationalization and localization situations. In its current state, most of its methods have been deprecated, and for most purposes you'll want to use the Calendar class instead of the Date class. The Date class is on the exam for several reasons: you might find it used in legacy code, it's really easy if all you want is a quick and dirty way to get the current date and time, it's good when you want a universal time that is not affected by time zones, and finally, you'll use it as a temporary bridge to format a Calendar object using the DateFormat class.

As we mentioned briefly above, an instance of the Date class represents a single date and time. Internally, the date and time is stored as a primitive `long`. Specifically, the `long` holds the number of milliseconds (you know, 1000 of these per second), between the date being represented and January 1, 1970.

Have you ever tried to grasp how big really big numbers are? Let's use the `Date` class to find out how long it took for a trillion milliseconds to pass, starting at January 1, 1970:

```
import java.util.*;
class TestDates {
  public static void main(String[] args) {
    Date d1 = new Date(1000000000000L);  // a trillion!
    System.out.println("1st date " + d1.toString());
  }
}
```

On our JVM, which has a US locale, the output is

```
1st date Sat Sep 08 19:46:40 MDT 2001
```

Okay, for future reference remember that there are a trillion milliseconds for every 31 and 2/3 years.

Although most of Date's methods have been deprecated, it's still acceptable to use the `getTime` and `setTime` methods, although as we'll soon see, it's a bit painful. Let's add an hour to our Date instance, `d1`, from the previous example:

```
import java.util.*;
class TestDates {
  public static void main(String[] args) {
    Date d1 = new Date(1000000000000L);  // a trillion!
```

```
      System.out.println("1st date " + d1.toString());
      d1.setTime(d1.getTime() + 3600000); // 3600000 millis / hour
      System.out.println("new time " + d1.toString());
    }
}
```

which produces (again, on our JVM):

```
1st date Sat Sep 08 19:46:40 MDT 2001
new time Sat Sep 08 20:46:40 MDT 2001
```

Notice that both `setTime()` and `getTime()` used the handy millisecond scale... if you want to manipulate dates using the Date class, that's your only choice. While that wasn't too painful, imagine how much fun it would be to add, say, a year to a given date.

We'll revisit the Date class later on, but for now the only other thing you need to know is that if you want to create an instance of Date to represent "now," you use Date's no-argument constructor:

```
Date now = new Date();
```

(We're guessing that if you call `now.getTime()`, you'll get a number somewhere between one trillion and two trillion.)

The Calendar Class

We've just seen that manipulating dates using the Date class is tricky. The Calendar class is designed to make date manipulation easy! (Well, easier.) While the Calendar class has about a million fields and methods, once you get the hang of a few of them the rest tend to work in a similar fashion.

When you first try to use the `Calendar` class you might notice that it's an abstract class. You can't say

```
Calendar c = new Calendar();   // illegal, Calendar is abstract
```

In order to create a `Calendar` instance, you have to use one of the overloaded `getInstance()` static factory methods:

```
Calendar cal = Calendar.getInstance();
```

When you get a Calendar reference like `cal`, from above, your Calendar reference variable is actually referring to an instance of a concrete subclass of Calendar. You can't know for sure what subclass you'll get (`java.util.GregorianCalendar` is what you'll almost certainly get), but it won't matter to you. You'll be using Calendar's API. (As Java continues to spread around the world, in order to maintain cohesion, you might find additional, locale-specific subclasses of `Calendar`.)

Okay, so now we've got an instance of Calendar, let's go back to our earlier example, and find out what day of the week our trillionth millisecond falls on, and then let's add a month to that date:

```
import java.util.*;
class Dates2 {
  public static void main(String[] args) {
    Date d1 = new Date(1000000000000L);
    System.out.println("1st date " + d1.toString());

    Calendar c = Calendar.getInstance();
    c.setTime(d1);                          // #1

    if(Calendar.SUNDAY == c.getFirstDayOfWeek())   // #2
      System.out.println("Sunday is the first day of the week");
    System.out.println("trillionth milli day of week is "
                + c.get(Calendar.DAY_OF_WEEK));    // #3

    c.add(Calendar.MONTH, 1);                      // #4
    Date d2 = c.getTime();                         // #5
    System.out.println("new date " + d2.toString() );
  }
}
```

This produces something like

```
1st date Sat Sep 08 19:46:40 MDT 2001
Sunday is the first day of the week
trillionth milli day of week is 7
new date Mon Oct 08 19:46:40 MDT 2001
```

Let's take a look at this program, focusing on the five highlighted lines:

1. We assign the Date `d1` to the Calendar instance `c`.

2. We use Calendar's `SUNDAY` field to determine whether, for our JVM, `SUNDAY` is considered to be the first day of the week. (In some locales, `MONDAY` is the first day of the week.) The Calendar class provides similar fields for days of the week, months, the day of the month, the day of the year, and so on.
3. We use the `DAY_OF_WEEK` field to find out the day of the week that the trillionth millisecond falls on.
4. So far we've used setter and getter methods that should be intuitive to figure out. Now we're going to use Calendar's `add()` method. This very powerful method lets you add or subtract units of time appropriate for whichever Calendar field you specify. For instance:

```
c.add(Calendar.HOUR, -4);     // subtract 4 hours from c's value
c.add(Calendar.YEAR, 2);           // add 2 years to c's value
c.add(Calendar.DAY_OF_WEEK, -2);  // subtract two days from
                                   // c's  value
```

5. Convert `c`'s value back to an instance of Date.

The other Calendar method you should know for the exam is the `roll()` method. The `roll()` method acts like the `add()` method, except that when a part of a Date gets incremented or decremented, larger parts of the Date will not get incremented or decremented. Hmmm…for instance:

```
// assume c is October 8, 2001
c.roll(Calendar.MONTH, 9);      // notice the year in the output
Date d4 = c.getTime();
System.out.println("new date " + d4.toString() );
```

The output would be something like this

```
new date Fri Jul 08 19:46:40 MDT 2001
```

Notice that the year did not change, even though we added 9 months to an October date. In a similar fashion, invoking `roll()` with `HOUR` won't change the date, the month, or the year.

For the exam, you won't have to memorize the Calendar class's fields. If you need them to help answer a question, they will be provided as part of the question.

The DateFormat Class

Having learned how to create dates and manipulate them, let's find out how to format them. So that we're all on the same page, here's an example of how a date can be formatted in different ways:

```
import java.text.*;
import java.util.*;
class Dates3 {
  public static void main(String[] args) {
    Date d1 = new Date(1000000000000L);

    DateFormat[] dfa = new DateFormat[6];
    dfa[0] = DateFormat.getInstance();
    dfa[1] = DateFormat.getDateInstance();
    dfa[2] = DateFormat.getDateInstance(DateFormat.SHORT);
    dfa[3] = DateFormat.getDateInstance(DateFormat.MEDIUM);
    dfa[4] = DateFormat.getDateInstance(DateFormat.LONG);
    dfa[5] = DateFormat.getDateInstance(DateFormat.FULL);

    for(DateFormat df : dfa)
      System.out.println(df.format(d1));
  }
}
```

which on our JVM produces

```
9/8/01 7:46 PM
Sep 8, 2001
9/8/01
Sep 8, 2001
September 8, 2001
Saturday, September 8, 2001
```

Examining this code we see a couple of things right away. First off, it looks like `DateFormat` is another abstract class, so we can't use `new` to create instances of `DateFormat`. In this case we used two factory methods, `getInstance()` and `getDateInstance()`. Notice that `getDateInstance()` is overloaded; when we discuss locales, we'll look at the other version of `getDateInstance()` that you'll need to understand for the exam.

Next, we used static fields from the DateFormat class to customize our various instances of DateFormat. Each of these static fields represents a formatting *style*. In

this case it looks like the no-arg version of `getDateInstance()` gives us the same style as the `MEDIUM` version of the method, but that's not a hard and fast rule. (More on this when we discuss locales.) Finally, we used the `format()` method to create Strings representing the properly formatted versions of the Date we're working with.

The last method you should be familiar with is the `parse()` method. The `parse()` method takes a String formatted in the style of the DateFormat instance being used, and converts the String into a Date object. As you might imagine, this is a risky operation because the `parse()` method could easily receive a badly formatted String. Because of this, `parse()` can throw a `ParseException`. The following code creates a `Date` instance, uses `DateFormat.format()` to convert it into a String, and then uses `DateFormat.parse()` to change it back into a Date:

```
Date d1 = new Date(1000000000000L);
System.out.println("d1 = " + d1.toString());

DateFormat df = DateFormat.getDateInstance(
                                    DateFormat.SHORT);
String s = df.format(d1);
System.out.println(s);

try {
  Date d2 = df.parse(s);
  System.out.println("parsed = " + d2.toString());
} catch (ParseException pe) {
  System.out.println("parse exc"); }
```

which on our JVM produces

```
d1 = Sat Sep 08 19:46:40 MDT 2001
9/8/01
parsed = Sat Sep 08 00:00:00 MDT 2001
```

Note: If we'd wanted to retain the time along with the date we could have used the `getDateTimeInstance()` method, but it's not on the exam.

on the job

The API for DateFormat.parse() explains that by default, the `parse()` ***method is lenient when parsing dates. Our experience is that*** `parse()` ***isn't very lenient about the formatting of Strings it will successfully parse into dates; take care when you use this method!***

The Locale Class

Earlier we said that a big part of why this objective exists is to test your ability to do some basic internationalization tasks. Your wait is over; the `Locale` class is your ticket to worldwide domination. Both the DateFormat class and the NumberFormat class (which we'll cover next) can use an instance of Locale to customize formatted output to be specific to a locale. 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 (`"tlh"`), 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 won't 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 locPT = new Locale("it");         // Italian
Locale locBR = 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 Calendar object, sets its date, then converts it to a Date. After that we'll take that Date object and print it out using locales from around the world:

```
Calendar c = Calendar.getInstance();
c.set(2010, 11, 14);                  // December 14, 2010
                                      // (month is 0-based
Date d2 = c.getTime();

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

DateFormat dfUS = DateFormat.getInstance();
System.out.println("US       " + dfUS.format(d2));

DateFormat dfUSfull = DateFormat.getDateInstance(
                                      DateFormat.FULL);
System.out.println("US full  " + dfUSfull.format(d2));

DateFormat dfIT = DateFormat.getDateInstance(
                                      DateFormat.FULL, locIT);
System.out.println("Italy    " + dfIT.format(d2));

DateFormat dfPT = DateFormat.getDateInstance(
                                      DateFormat.FULL, locPT);
System.out.println("Portugal " + dfPT.format(d2));

DateFormat dfBR = DateFormat.getDateInstance(
                                      DateFormat.FULL, locBR);
System.out.println("Brazil   " + dfBR.format(d2));

DateFormat dfIN = DateFormat.getDateInstance(
                                      DateFormat.FULL, locIN);
System.out.println("India    " + dfIN.format(d2));

DateFormat dfJA = DateFormat.getDateInstance(
                                      DateFormat.FULL, locJA);
System.out.println("Japan    " + dfJA.format(d2));
```

This, on our JVM, produces

```
US       12/14/10 3:32 PM
US full  Sunday, December 14, 2010
Italy    domenica 14 dicembre 2010
```

```
Portugal Domingo, 14 de Dezembro de 2010
Brazil   Domingo, 14 de Dezembro de 2010
India    ??????, ?? ??????, ????
Japan    2010?12?14?
```

Oops! Our machine isn't configured to support locales for India or Japan, but you can see how a single Date object can be formatted to work for many locales.

exam Watch

Remember that both DateFormat and NumberFormat objects can have their locales set only at the time of instantiation. Watch for code that attempts to change the locale of an existing instance—no such methods exist!

There are a couple more methods in Locale (`getDisplayCountry()` and `getDisplayLanguage()`) that you'll have 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:

```
Calendar c = Calendar.getInstance();
c.set(2010, 11, 14);
Date d2 = c.getTime();

Locale locBR = new Locale("pt", "BR");  // Brazil
Locale locDK = new Locale("da", "DK");  // Denmark
Locale locIT = new Locale("it", "IT");  // Italy

System.out.println("def " + locBR.getDisplayCountry());
System.out.println("loc " + locBR.getDisplayCountry(locBR));

System.out.println("def " + locDK.getDisplayLanguage());
System.out.println("loc " + locDK.getDisplayLanguage(locDK));
System.out.println("D>I " + locDK.getDisplayLanguage(locIT));
```

This, on our JVM, produces

```
def Brazil
loc Brasil
def Danish
loc dansk
D>I danese
```

Given that our JVM's locale (the default for us) is `US`, the default for the country Brazil is "`Brazil`", and the default for the Danish language is "`Danish`". In Brazil, the country is called "`Brasil`", and in Denmark the language is called "`dansk`". Finally, just for fun, we discovered that in Italy, the Danish language is called "`danese`".

The NumberFormat Class

We'll wrap up this objective by discussing the NumberFormat class. Like the DateFormat class, NumberFormat is abstract, so you'll typically use some version of either `getInstance()` or `getCurrencyInstance()` to create a NumberFormat object. Not surprisingly, you use this class to format numbers or currency values:

```
float f1 = 123.4567f;
Locale locFR = new Locale("fr");          // France
NumberFormat[] nfa = new NumberFormat[4];

nfa[0] = NumberFormat.getInstance();
nfa[1] = NumberFormat.getInstance(locFR);
nfa[2] = NumberFormat.getCurrencyInstance();
nfa[3] = NumberFormat.getCurrencyInstance(locFR);

for(NumberFormat nf : nfa)
  System.out.println(nf.format(f1));
```

This, on our JVM, produces

```
123.457
123,457
$123.46
123,46 ?
```

Don't be worried if, like us, you're not set up to display the symbols for francs, pounds, rupees, yen, baht, or drachmas. You won't be expected to

know the symbols used for currency: if you need one, it will be specified in the question. You might encounter methods other than the format method on the exam. Here's a little code that uses `getMaximumFractionDigits()`, `setMaximumFractionDigits()`, `parse()`, and `setParseIntegerOnly()`:

```
float f1 = 123.45678f;
NumberFormat nf = NumberFormat.getInstance();
System.out.print(nf.getMaximumFractionDigits() + " ");
System.out.print(nf.format(f1) + "  ");

nf.setMaximumFractionDigits(5);
System.out.println(nf.format(f1) + "  ");

try {
  System.out.println(nf.parse("1234.567"));
  nf.setParseIntegerOnly(true);
  System.out.println(nf.parse("1234.567"));
} catch (ParseException pe) {
  System.out.println("parse exc");
}
```

This, on our JVM, produces

```
3  123.457  123.45678
1234.567
1234
```

Notice that in this case, the initial number of fractional digits for the default NumberFormat is three: and that the `format()` method rounds `f1`'s value, it doesn't truncate it. After changing `nf`'s fractional digits, the entire value of `f1` is displayed. Next, notice that the `parse()` method must run in a try/catch block and that the `setParseIntegerOnly()` method takes a `boolean` and in this case, causes subsequent calls to `parse()` to return only the integer part of Strings formatted as floating-point numbers.

As we've seen, several of the classes covered in this objective are `abstract`. In addition, for all of these classes, key functionality for every instance is established at the time of creation. Table 6-3 summarizes the constructors or methods used to create instances of all the classes we've discussed in this section.

TABLE 6-3 Instance Creation for Key java.text and java.util Classes

Class	Key Instance Creation Options
`util.Date`	`new Date();` `new Date(long millisecondsSince010170);`
`util.Calendar`	`Calendar.getInstance();` `Calendar.getInstance(Locale);`

`util.Locale`	`Locale.getDefault();`
	`new Locale(String language);`
	`new Locale(String language, String country);`

`text.DateFormat`	`DateFormat.getInstance();`
	`DateFormat.getDateInstance();`
	`DateFormat.getDateInstance(style);`
	`DateFormat.getDateInstance(style, Locale);`

`text.NumberFormat`	`NumberFormat.getInstance()`
	`NumberFormat.getInstance(Locale)`
	`NumberFormat.getNumberInstance()`
	`NumberFormat.getNumberInstance(Locale)`
	`NumberFormat.getCurrencyInstance()`
	`NumberFormat.getCurrencyInstance(Locale)`

CERTIFICATION OBJECTIVE

Parsing, Tokenizing, and Formatting (Exam Objective 3.5)

*3.5 Write code that uses standard J2SE APIs in the java.util and java.util.regex packages to format or parse strings or streams. For strings, write code that uses the Pattern and Matcher classes and the String.split method. Recognize and use regular expression patterns for matching (limited to: .(dot), *(star), +(plus), ?, \d, \s, \w, [ ], ( )). The use of *, + , and ? will be limited to greedy quantifiers, and the parenthesis operator will only be used as a grouping mechanism, not for capturing content during matching. For streams, write code using the Formatter and Scanner classes and the PrintWriter.format/printf methods. Recognize and use formatting parameters (limited to: %b, %c, %d, %f, %s) in format strings.*

We're going to start with yet another disclaimer: This small section isn't going to morph you from regex newbie to regex guru. In this section we'll cover three basic ideas:

- **Finding stuff** You've got big heaps of text to look through. Maybe you're doing some screen scraping, maybe you're reading from a file. In any case, you need easy ways to find textual needles in textual haystacks. We'll use the java.util.regex.Pattern, java.util.regex.Matcher, and java.util.Scanner classes to help us find stuff.
- **Tokenizing stuff** You've got a delimited file that you want to get useful data out of. You want to transform a piece of a text file that looks like: "1500.00,343.77,123.4" into some individual float variables. We'll show you the basics of using the `String.split()` method and the java.util.Scanner class, to tokenize your data.
- **Formatting stuff** You've got a report to create and you need to take a float variable with a value of 32500.000f and transform it into a String with a value of "$32,500.00". We'll introduce you to the java.util.Formatter class and to the `printf()` and `format()` methods.

A Search Tutorial

Whether you're looking for stuff or tokenizing stuff, a lot of the concepts are the same, so let's start with some basics. No matter what language you're using, sooner or later you'll probably be faced with the need to search through large amounts of textual data, looking for some specific stuff.

Regular expressions (regex for short) are a kind of language within a language, designed to help programmers with these searching tasks. Every language that provides regex capabilities uses one or more regex *engines*. Regex engines search through textual data using instructions that are coded into *expressions*. A regex expression is like a very short program or script. When you invoke a regex engine, you'll pass it the chunk of textual data you want it to process (in Java this is usually a String or a stream), and you pass it the expression you want it to use to search through the data.

It's fair to think of regex as a language, and we will refer to it that way throughout this section. The regex language is used to create expressions, and as we work through this section, whenever we talk about expressions or expression syntax, we're talking about syntax for the regex "language." Oh, one more disclaimer…we know that you regex mavens out there can come up with better expressions than what

we're about to present. Keep in mind that for the most part we're creating these expressions using only a portion of the total regex instruction set, thanks.

Simple Searches

For our first example, we'd like to search through the following *source* String

```
abaaaba
```

for all occurrences (or *matches*) of the *expression*

```
ab
```

In all of these discussions we'll assume that our data sources use zero-based indexes, so if we apply an index to our source string we get

```
source: abaaaba
index:  0123456
```

We can see that we have two occurrences of the expression `ab`: one starting at position 0 and the second starting at position 4. If we sent the previous source data and expression to a regex engine, it would reply by telling us that it found matches at positions 0 and 4:

```
import java.util.regex.*;
class RegexSmall {
  public static void main(String [] args) {
    Pattern p = Pattern.compile("ab");      // the expression
    Matcher m = p.matcher("abaaaba");       // the source
    while(m.find()) {
      System.out.print(m.start() + " ");
    }
  }
}
```

This produces

```
0 4
```

We're not going to explain this code right now. In a few pages we're going to show you a lot more regex code, but first we want to go over some more regex syntax. Once you understand a little more regex, the code samples will make a lot more sense. Here's a more complicated example of a source and an expression:

```
source: abababa
index:  0123456
expression: aba
```

How many occurrences do we get in this case? Well, there is clearly an occurrence starting at position 0, and another starting at position 4. But how about starting at position 2? In general in the world of regex, the `aba` string that starts at position 2 will not be considered a valid occurrence. The first general regex search rule is

> In general, a regex search runs from left to right, and once a source's character has been used in a match, it cannot be reused.

So in our previous example, the first match used positions 0, 1, and 2 to match the expression. (Another common term for this is that the first three characters of the source were *consumed*.) Because the character in position 2 was consumed in the first match, it couldn't be used again. So the engine moved on, and didn't find another occurrence of `aba` until it reached position 4. This is the typical way that a regex matching engine works. However, in a few pages, we'll look at an exception to the first rule we stated above.

So we've matched a couple of exact strings, but what would we do if we wanted to find something a little more dynamic? For instance, what if we wanted to find all of the occurrences of hex numbers or phone numbers or ZIP codes?

Searches Using Metacharacters

As luck would have it, regex has a powerful mechanism for dealing with the cases we described above. At the heart of this mechanism is the idea of a *metacharacter*. As an easy example, let's say that we want to search through some source data looking for all occurrences of numeric digits. In regex, the following expression is used to look for numeric digits:

```
\d
```

If we change the previous program to apply the expression `\d` to the following source string

```
source: a12c3e456f
index:  0123456789
```

regex will tell us that it found digits at positions 1, 2, 4, 6, 7, and 8. (If you want to try this at home, you'll need to "escape" the `compile` method's "`\d`" argument by making it "`\\d`", more on this a little later.)

Regex provides a rich set of metacharacters that you can find described in the API documentation for java.util.regex.Pattern. We won't discuss them all here, but we will describe the ones you'll need for the exam:

`\d` A digit
`\s` A whitespace character
`\w` A word character (letters, digits, or "_" (underscore))

So for example, given

```
source: "a 1 56 _Z"
index:   012345678
pattern: \w
```

regex will return positions 0, 2, 4, 5, 7, and 8. The only characters in this source that don't match the definition of a word character are the whitespaces. (Note: In this example we enclosed the source data in quotes to clearly indicate that there was no whitespace at either end.)

You can also specify sets of characters to search for using square brackets and ranges of characters to search for using square brackets and a dash:

`[abc]` Searches only for a's, b's or c's
`[a-f]` Searches only for a, b, c, d, e, or f characters

In addition, you can search across several ranges at once. The following expression is looking for occurrences of the letters `a - f` or `A - F`, it's NOT looking for an `fA` combination:

`[a-fA-F]` Searches for the first six letters of the alphabet, both cases.

So for instance,

```
source: "cafeBABE"
index:   01234567
pattern: [a-cA-C]
```

returns positions 0, 1, 4, 5, 6.

on the Job

In addition to the capabilities described for the exam, you can also apply the following attributes to sets and ranges within square brackets: "^" to negate the characters specified, nested brackets to create a union of sets, and "&&" to specify the intersection of sets. While these constructs are not on the exam, they are quite useful, and good examples can be found in the API for the java.util.regex.Pattern class.

Searches Using Quantifiers

Let's say that we want to create a regex pattern to search for hexadecimal literals. As a first step, let's solve the problem for one-digit hexadecimal numbers:

```
0[xX][0-9a-fA-F]
```

The preceding expression could be stated: "Find a set of characters in which the first character is a "0", the second character is either an "x" or an "X", and the third character is either a digit from "0" to "9", a letter from "a" to "f" or an uppercase letter from "A" to "F" ". Using the preceding expression, and the following data,

```
source: "12 0x 0x12 0Xf 0xg"
index:   012345678901234567
```

regex would return 6 and 11. (Note: 0x and 0xg are not valid hex numbers.)

As a second step, let's think about an easier problem. What if we just wanted regex to find occurrences of integers? Integers can be one or more digits long, so it would be great if we could say "one or more" in an expression. There is a set of regex constructs called quantifiers that let us specify concepts such as "one or more." In fact, the quantifier that represents "one or more" is the "+" character. We'll see the others shortly.

The other issue this raises is that when we're searching for something whose length is variable, getting only a starting position as a return value has limited value. So, in addition to returning starting positions, another bit of information that a regex engine can return is the entire match or *group* that it finds. We're going to change the way we talk about what regex returns by specifying each return on its own line, remembering that now for each return we're going to get back the starting position AND then the group:

```
source: "1 a12 234b"
pattern: \d+
```

You can read this expression as saying: "Find one or more digits in a row." This expression produces the regex output

```
0 1
3 12
6 234
```

You can read this as "At position 0 there's an integer with a value of 1, then at position 3 there's an integer with a value of 12, then at position 6 there's an integer with a value of 234." Returning now to our hexadecimal problem, the last thing we need to know is how to specify the use of a quantifier for only part of an expression. In this case we must have exactly one occurrence of `0x` or `0X` but we can have from one to many occurrences of the hex "digits" that follow. The following expression adds parentheses to limit the "+" quantifier to only the hex digits:

```
0[xX]([0-9a-fA-F])+
```

The parentheses and "+" augment the previous find-the-hex expression by saying in effect: "Once we've found our 0x or 0X, you can find from one to many occurrences of hex digits." Notice that we put the "+" quantifier at the end of the expression. It's useful to think of quantifiers as always quantifying the part of the expression that precedes them.

The other two quantifiers we're going to look at are

* Zero or more occurrences
? Zero or one occurrence

Let's say you have a text file containing a comma-delimited list of all the file names in a directory that contains several very important projects. (BTW, this isn't how we'd arrange our directories :) You want to create a list of all the files whose names start with `proj1`. You might discover .txt files, .java files, .pdf files, who knows? What kind of regex expression could we create to find these various `proj1` files? First let's take a look at what a part of this text might look like:

```
..."proj3.txt,proj1sched.pdf,proj1,proj2,proj1.java"...
```

To solve this problem we're going to use the regex ^ (carat) operator, which we mentioned earlier. The regex ^ operator isn't on the exam, but it will help us create a fairly clean solution to our problem. The ^ is the negation symbol in regex. For instance, if you want to find anything but a's, b's, or c's in a file you could say

```
[^abc]
```

So, armed with the ^ operator and the * (zero or more) quantifier we can create the following:

```
proj1([^,])*
```

If we apply this expression to just the portion of the text file we listed above, regex returns

```
10 proj1sched.pdf
25 proj1
37 proj1.java
```

The key part of this expression is the "give me zero or more characters that aren't a comma."

The last quantifier example we'll look at is the ? (zero or one) quantifier. Let's say that our job this time is to search a text file and find anything that might be a local, 7-digit phone number. We're going to say, arbitrarily, that if we find either seven digits in a row, or three digits followed by a dash or a space followed by 4 digits, that we have a candidate. Here are examples of "valid" phone numbers:

```
1234567
123 4567
123-4567
```

The key to creating this expression is to see that we need "zero or one instance of either a space or a dash" in the middle of our digits:

```
\d\d\d([-\s])?\d\d\d\d
```

The Predefined Dot

In addition to the \s, \d, and \w metacharacters that we discussed, you also have to understand the "." (dot) metacharacter. When you see this character in a regex expression, it means "any character can serve here." For instance, the following source and pattern

```
source: "ac abc a c"
pattern: a.c
```

will produce the output

```
3 abc
7 a c
```

The "." was able to match both the "b" and the " " in the source data.

Greedy Quantifiers

When you use the *, +, and ? quantifiers, you can fine tune them a bit to produce behavior that's known as "greedy", "reluctant", or "possessive". Although you need to understand only the greedy quantifier for the exam, we're also going to discuss the reluctant quantifier to serve as a basis for comparison. First the syntax:

`?` is greedy, `??` is reluctant, for zero or once
`*` is greedy, `*?` is reluctant, for zero or more
`+` is greedy, `+?` is reluctant, for one or more

What happens when we have the following source and pattern?

```
source:  yyxxxyxx
pattern:  .*xx
```

First off, we're doing something a bit different here by looking for characters that prefix the static (xx) portion of the expression. We think we're saying something

like: "Find sets of characters that ends with xx". Before we tell what happens, we at least want you to consider that there are two plausible results…can you find them? Remember we said earlier that in general, regex engines worked from left to right, and consumed characters as they went. So, working from left to right, we might predict that the engine would search the first 4 characters (0–3), find xx starting in position 2, and have its first match. Then it would proceed and find the second xx starting in position 6. This would lead us to a result like this:

```
0 yyxx
4 xyxx
```

A plausible second argument is that since we asked for a set of characters that ends with xx we might get a result like this:

```
0 yyxxxyxx
```

The way to think about this is to consider the name *greedy*. In order for the second answer to be correct, the regex engine would have to look (greedily) at the *entire* source data before it could determine that there was an xx at the end. So in fact, the second result is the correct result because in the original example we used the greedy quantifier *. The result that finds two different sets can be generated by using the reluctant quantifier *?. Let's review:

```
source:  yyxxxyxx
pattern:  .*xx
```

is using the greedy quantifier * and produces

```
0 yyxxxyxx
```

If we change the pattern to

```
source:  yyxxxyxx
pattern:  .*?xx
```

we're now using the reluctant qualifier *?, and we get the following:

```
0 yyxx
4 xyxx
```

The greedy quantifier does in fact read the entire source data, and then it works backward (from the right) until it finds the rightmost match. At that point, it includes everything from earlier in the source data up to and including the data that is part of the rightmost match.

There are a lot more aspects to regex quantifiers than we've discussed here, but we've covered more than enough for the exam. Sun has several tutorials that will help you learn more about quantifiers, and turn you into the go-to person at your job.

When Metacharacters and Strings Collide

So far we've been talking about regex from a theoretical perspective. Before we can put regex to work we have to discuss one more gotcha. When it's time to implement regex in our code, it will be quite common that our source data and/or our expressions will be stored in Strings. The problem is that metacharacters and Strings don't mix too well. For instance. let's say we just want to do a simple regex pattern that looks for digits. We might try something like

```
String pattern = "\d";    // compiler error!
```

This line of code won't compile! The compiler sees the `\` and thinks, "Ok, here comes an escape sequence, maybe it'll be a new line!" But no, next comes the `d` and the compiler says "I've never heard of the `\d` escape sequence." The way to satisfy the compiler is to add another backslash in front of the `\d`

```
String pattern = "\\d";    // a compilable metacharacter
```

The first backslash tells the compiler that whatever comes next should be taken literally, not as an escape sequence. How about the dot (.) metacharacter? If we want a dot in our expression to be used as a metacharacter, then no problem, but what if we're reading some source data that happens to use dots as delimiters? Here's another way to look at our options:

```
String p = ".";   // regex sees this as the "." metacharacter
String p = "\.";  // the compiler sees this as an illegal
                  // Java escape sequence
```

```
String p = "\\."; // the compiler is happy, and regex sees a
                  // dot, not a metacharacter
```

A similar problem can occur when you hand metacharacters to a Java program via command-line arguments. If we want to pass the `\d` metacharacter into our Java program, our JVM does the right thing if we say

```
% java DoRegex "\d"
```

But your JVM might not. If you have problems running the following examples, you might try adding a backslash (i.e., `\\d`) to your command-line metacharacters. Don't worry, you won't see any command-line metacharacters on the exam!

The Java language defines several escape sequences, including

`\n` ***= linefeed (which you might see on the exam)***
`\b` ***= backspace***
`\t` ***= tab***

And others, which you can find in the Java Language Specification. Other than perhaps seeing a `\n` ***inside a String, you won't have to worry about Java's escape sequences on the exam.***

At this point we've learned enough of the regex language to start using it in our Java programs. We'll start by looking at using regex expressions to find stuff, and then we'll move to the closely related topic of tokenizing stuff.

Locating Data via Pattern Matching

Once you know a little regex, using the java.util.regex.Pattern (Pattern) and java.util.regex.Matcher (Matcher) classes is pretty straightforward. The Pattern class is used to hold a representation of a regex expression, so that it can be used and reused by instances of the Matcher class. The Matcher class is used to invoke the regex engine with the intention of performing match operations. The following program shows Pattern and Matcher in action, and it's not a bad way for you to do your own regex experiments. Note, you might want to modify the following class by adding some functionality from the Console class. That way you'll get some practice with the Console class, and it'll be easier to run multiple regex experiments.

```
import java.util.regex.*;
class Regex {
  public static void main(String [] args) {
    Pattern p = Pattern.compile(args[0]);
    Matcher m = p.matcher(args[1]);
    System.out.println("Pattern is " + m.pattern());
    while(m.find()) {
      System.out.println(m.start() + " " + m.group());
    }
  }
}
```

This program uses the first command-line argument (args[0]) to represent the regex expression you want to use, and it uses the second argument (args[1]) to represent the source data you want to search. Here's a test run:

```
% java Regex "\d\w" "ab4 56_7ab"
```

Produces the output

```
Pattern is \d\w
4 56
7 7a
```

(Remember, if you want this expression to be represented in a String, you'd use `\\d\\w`). Because you'll often have special characters or whitespace as part of your arguments, you'll probably want to get in the habit of always enclosing your argument in quotes. Let's take a look at this code in more detail. First off, notice that we aren't using `new` to create a Pattern; if you check the API, you'll find no constructors are listed. You'll use the overloaded, static `compile()` method (that takes `String expression`) to create an instance of Pattern. For the exam, all you'll need to know to create a `Matcher`, is to use the `Pattern.matcher()` method (that takes `String sourceData`).

The important method in this program is the `find()` method. This is the method that actually cranks up the regex engine and does some searching. The `find()` method returns `true` if it gets a match, and remembers the start position of the match. If `find()` returns `true`, you can call the `start()` method to get the starting position of the match, and you can call the `group()` method to get the string that represents the actual bit of source data that was matched.

exam Watch

To provide the most flexibility, `Matcher.find()`***, when coupled with the greedy quantifiers ? or *, allow for (somewhat unintuitively) the idea of a*** zero-length match. ***As an experiment, modify the previous*** `Regex.java` ***class and add an invocation of*** `m.end()` ***to the S.O.P. in the while loop. With that modification in place, the invocation***

```
java Regex "a?" "aba"
```

should produce something very similar to this:

```
Pattern is a?
0 1 a
1 1
2 3 a
3 3
```

The lines of output `1 1` ***and*** `3 3` ***are examples of zero-length matches. Zero-length matches can occur in several places:***

- ***After the last character of source data (the*** `3 3` ***example)***
- ***In between characters after a match has been found (the*** `1 1` ***example)***
- ***At the beginning of source data (try*** `java Regex "a?" "baba"`***)***
- ***At the beginning of zero-length source data***

on the Job

A common reason to use regex is to perform search and replace operations. Although replace operations are not on the exam you should know that the Matcher class provides several methods that perform search and replace operations. See the `appendReplacement()`***,*** `appendTail()`***, and*** `replaceAll()` ***methods in the Matcher API for more details.***

The Matcher class allows you to look at subsets of your source data by using a concept called *regions*. In real life, regions can greatly improve performance, but you won't need to know anything about them for the exam.

Searching Using the Scanner Class Although the java.util.Scanner class is primarily intended for tokenizing data (which we'll cover next), it can also be used to find stuff, just like the Pattern and Matcher classes. While Scanner doesn't provide location information or search and replace functionality, you can use it to apply regex expressions to source data to tell you how many instances of an expression exist in a given piece of source data. The following program uses the first command-line argument as a regex expression, then asks for input using `System.in`. It outputs a message every time a match is found:

```
import java.util.*;
class ScanIn {
  public static void main(String[] args) {
    System.out.print("input: ");
    System.out.flush();
    try {
      Scanner s = new Scanner(System.in);
      String token;
      do {
        token = s.findInLine(args[0]);
        System.out.println("found " + token);
      } while (token != null);
    } catch (Exception e) { System.out.println("scan exc"); }
  }
}
```

The invocation and input

```
java ScanIn "\d\d"
input: 1b2c335f456
```

produce the following:

```
found 33
found 45
found null
```

Tokenizing

Tokenizing is the process of taking big pieces of source data, breaking them into little pieces, and storing the little pieces in variables. Probably the most common tokenizing situation is reading a delimited file in order to get the contents of the file

moved into useful places like objects, arrays or collections. We'll look at two classes in the API that provide tokenizing capabilities: String (using the `split()` method) and Scanner, which has many methods that are useful for tokenizing.

Tokens and Delimiters

When we talk about tokenizing, we're talking about data that starts out composed of two things: tokens and delimiters. Tokens are the actual pieces of data, and delimiters are the expressions that *separate* the tokens from each other. When most people think of delimiters, they think of single characters, like commas or backslashes or maybe a single whitespace. These are indeed very common delimiters, but strictly speaking, delimiters can be much more dynamic. In fact, as we hinted at a few sentences ago, delimiters can be anything that qualifies as a regex expression. Let's take a single piece of source data and tokenize it using a couple of different delimiters:

```
source: "ab,cd5b,6x,z4"
```

If we say that our delimiter is a comma, then our four tokens would be

```
ab
cd5b
6x
z4
```

If we use the same source, but declare our delimiter to be `\d`, we get three tokens:

```
ab,cd
b,
x,z
```

In general, when we tokenize source data, the delimiters themselves are discarded, and all that we are left with are the tokens. So in the second example, we defined digits to be delimiters, so the 5, 6, and 4 do not appear in the final tokens.

Tokenizing with String.split()

The String class's `split()` method takes a regex expression as its argument, and returns a String array populated with the tokens produced by the split (or tokenizing) process. This is a handy way to tokenize relatively small pieces of data. The following program uses `args[0]` to hold a source string, and `args[1]` to hold the regex pattern to use as a delimiter:

```
import java.util.*;
class SplitTest {
  public static void main(String[] args) {
    String[] tokens = args[0].split(args[1]);
    System.out.println("count " + tokens.length);
    for(String s : tokens)
      System.out.println(">" + s + "<");
  }
}
```

Everything happens all at once when the `split()` method is invoked. The source string is split into pieces, and the pieces are all loaded into the `tokens` String array. All the code after that is just there to verify what the split operation generated. The following invocation

```
% java SplitTest "ab5 ccc 45 @" "\d"
```

produces

```
count 4
>ab<
> ccc <
><
> @<
```

(Note: Remember that to represent "\" in a string you may need to use the escape sequence "\\". Because of this, and depending on your OS, your second argument might have to be "\\d" or even "\\\\d".)

We put the tokens inside "> <" characters to show whitespace. Notice that every digit was used as a delimiter, and that contiguous digits created an empty token.

One drawback to using the `String.split()` method is that often you'll want to look at tokens as they are produced, and possibly quit a tokenization operation early when you've created the tokens you need. For instance, you might be searching a large file for a phone number. If the phone number occurs early in the file, you'd like to quit the tokenization process as soon as you've got your number. The Scanner class provides a rich API for doing just such on-the-fly tokenization operations.

exam Watch

Because `System.out.println()` ***is so heavily used on the exam, you might see examples of escape sequences tucked in with questions on most any topic, including regex. Remember that if you need to create a String that contains a double quote*** `"` ***or a backslash*** `\` ***you need to add an escape character first:***

```
System.out.println("\" \\");
```

This prints

```
" \
```

So, what if you need to search for periods (.) in your source data? If you just put a period in the regex expression, you get the "any character" behavior. So, what if you try `\.` ***? Now the Java compiler thinks you're trying to create an escape sequence that doesn't exist. The correct syntax is***

```
String s = "ab.cde.fg";
String[] tokens = s.split("\\.");
```

Tokenizing with Scanner

The java.util.Scanner class is the Cadillac of tokenizing. When you need to do some serious tokenizing, look no further than Scanner—this beauty has it all. In addition to the basic tokenizing capabilities provided by `String.split()`, the Scanner class offers the following features:

- Scanners can be constructed using files, streams, or Strings as a source.
- Tokenizing is performed within a loop so that you can exit the process at any point.
- Tokens can be converted to their appropriate primitive types automatically.

Let's look at a program that demonstrates several of Scanner's methods and capabilities. Scanner's default delimiter is whitespace, which this program uses.

The program makes two Scanner objects: `s1` is iterated over with the more generic `next()` method, which returns every token as a String, while `s2` is analyzed with several of the specialized `nextXxx()` methods (where `Xxx` is a primitive type):

```
import java.util.Scanner;
class ScanNext {
  public static void main(String [] args) {
    boolean b2, b;
    int i;
    String s, hits = " ";
    Scanner s1 = new Scanner(args[0]);
    Scanner s2 = new Scanner(args[0]);
    while(b = s1.hasNext()) {
      s = s1.next();  hits += "s";
    }
    while(b = s2.hasNext()) {
      if (s2.hasNextInt()) {
        i = s2.nextInt();  hits += "i";
      } else if (s2.hasNextBoolean()) {
        b2 = s2.nextBoolean();  hits += "b";
      } else {
        s2.next();  hits += "s2";
      }
    }
    System.out.println("hits " + hits);
  }
}
```

If this program is invoked with

```
% java ScanNext "1 true 34 hi"
```

it produces

```
hits  ssssibis2
```

Of course we're not doing anything with the tokens once we've got them, but you can see that `s2`'s tokens are converted to their respective primitives. A key point here is that the methods named `hasNextXxx()` test the value of the next token

but do not actually get the token, nor do they move to the next token in the source data. The `nextXxx()` methods all perform two functions: they get the next token, and then they move to the next token.

The Scanner class has `nextXxx()` (for instance `nextLong()`) and `hasNextXxx()` (for instance `hasNextDouble()`) methods for every primitive type except `char`. In addition, the Scanner class has a `useDelimiter()` method that allows you to set the delimiter to be any valid regex expression.

Formatting with printf() and format()

What fun would accounts receivable reports be if the decimal points didn't line up? Where would you be if you couldn't put negative numbers inside of parentheses? Burning questions like these caused the exam creation team to include formatting as a part of the exam. The `format()` and `printf()` methods were added to java.io.PrintStream in Java 5. These two methods behave exactly the same way, so anything we say about one of these methods applies to both of them. (The rumor is that Sun added `printf()` just to make old C programmers happy.)

Behind the scenes, the `format()` method uses the `java.util.Formatter` class to do the heavy formatting work. You can use the `Formatter` class directly if you choose, but for the exam all you have to know is the basic syntax of the arguments you pass to the `format()` method. The documentation for these formatting arguments can be found in the Formatter API. We're going to take the "nickel tour" of the formatting String syntax, which will be more than enough to allow you to do a lot of basic formatting work, AND ace all the formatting questions on the exam.

Let's start by paraphrasing the API documentation for format strings (for more complete, way-past-what-you-need-for-the-exam coverage, check out the java.util.Formatter API):

```
printf("format string", argument(s));
```

The format string can contain both normal string literal information that isn't associated with any arguments, and argument-specific formatting data. The clue to determining whether you're looking at formatting data, is that formatting data will always start with a percent sign (`%`). Let's look at an example, and don't panic, we'll cover everything that comes after the `%` next:

```
System.out.printf("%2$d  +  %1$d", 123, 456);
```

This produces

```
456 + 123
```

Let's look at what just happened. Inside the double quotes there is a format string, then a +, and then a second format string. Notice that we mixed literals in with the format strings. Now let's dive in a little deeper and look at the construction of format strings:

```
%[arg_index$][flags][width][.precision]conversion char
```

The values within `[ ]` are optional. In other words, the only required elements of a format string are the `%` and a conversion character. In the example above the only optional values we used were for argument indexing. The `2$` represents the second argument, and the `1$` represents the first argument. (Notice that there's no problem switching the order of arguments.) The `d` after the arguments is a conversion character (more or less the type of the argument). Here's a rundown of the format string elements you'll need to know for the exam:

arg_index An integer followed directly by a `$`, this indicates which argument should be printed in this position.

flags While many flags are available, for the exam you'll need to know:

- "-" Left justify this argument
- "+" Include a sign (+ or -) with this argument
- "0" Pad this argument with zeroes
- "," Use locale-specific grouping separators (i.e., the comma in 123,456)
- "(" Enclose negative numbers in parentheses

width This value indicates the minimum number of characters to print. (If you want nice even columns, you'll use this value extensively.)

precision For the exam you'll only need this when formatting a floating-point number, and in the case of floating point numbers, precision indicates the number of digits to print after the decimal point.

conversion The type of argument you'll be formatting. You'll need to know:

- `b` boolean
- `c` char
- `d` integer
- `f` floating point
- `s` string

Let's see some of these formatting strings in action:

```
int i1 = -123;
int i2 = 12345;
System.out.printf(">%1$(7d< \n", i1);
System.out.printf(">%0,7d< \n", i2);
System.out.format(">%+-7d< \n", i2);
System.out.printf(">%2$b + %1$5d< \n", i1, false);
```

This produces:

```
>  (123)<
>012,345<
>+12345 <
>false +  -123<
```

(We added the > and < literals to help show how minimum widths, and zero padding and alignments work.) Finally, it's important to remember that if you have a mismatch between the type specified in your conversion character and your argument, you'll get a runtime exception:

```
System.out.format("%d", 12.3);
```

This produces something like

```
Exception in thread "main" java.util.IllegalFormatConversionEx-
ception: d != java.lang.Double
```

CERTIFICATION SUMMARY

Strings The most important thing to remember about Strings is that String objects are immutable, but references to Strings are not! You can make a new String by using an existing String as a starting point, but if you don't assign a reference variable to the new String it will be lost to your program—you will have no way to access your new String. Review the important methods in the String class.

The StringBuilder class was added in Java 5. It has exactly the same methods as the old StringBuffer class, except StringBuilder's methods aren't thread-safe. Because StringBuilder's methods are not thread safe, they tend to run faster than StringBuffer methods, so choose StringBuilder whenever threading is not an issue. Both StringBuffer and StringBuilder objects can have their value changed over and over without having to create new objects. If you're doing a lot of string manipulation, these objects will be more efficient than immutable String objects, which are, more or less, "use once, remain in memory forever." Remember, these methods ALWAYS change the invoking object's value, even with no explicit assignment.

File I/O Remember that objects of type File can represent either files or directories, but that until you call `createNewFile()` or `mkdir()` you haven't actually created anything on your hard drive. Classes in the java.io package are designed to be chained together. It will be rare that you'll use a FileReader or a FileWriter without "wrapping" them with a BufferedReader or BufferedWriter object, which gives you access to more powerful, higher-level methods. As of Java 5, the PrintWriter class has been enhanced with advanced `append()`, `format()`, and `printf()` methods, and when you couple that with new constructors that allow you to create PrintWriters directly from a String name or a File object, you may use BufferWriters a lot less. The Console class allows you to read non-echoed input (returned in a char[]), and is instantiated using System.console().

Serialization Serialization lets you save, ship, and restore everything you need to know about a *live* object. And when your object points to other objects, they get saved too. The java.io.ObjectOutputStream and java.io.ObjectInputStream classes are used to serialize and deserialize objects. Typically you wrap them around instances of FileOutputStream and FileInputStream, respectively.

The key method you invoke to serialize an object is `writeObject()`, and to deserialize an object invoke `readObject()`. In order to serialize an object, it must implement the Serializable interface. Mark instance variables `transient` if you don't want their state to be part of the serialization process. You can augment

the serialization process for your class by implementing `writeObject()` and `readObject()`. If you do that, an embedded call to `defaultReadObject()` and `defaultWriteObject()` will handle the normal serialization tasks, and you can augment those invocations with manual *reading from* and *writing to* the stream.

If a superclass implements Serializable then all of its subclasses do too. If a superclass doesn't implement Serializable, then when a subclass object is deserialized the unserializable superclass's constructor runs—be careful! Finally, remember that serialization is about instances, so static variables aren't serialized.

Dates, Numbers, and Currency Remember that the Sun objective is a bit misleading, and that you'll have to understand the basics of five related classes: java.util.Date, java.util.Calendar, java.util.Locale, java.text.DateFormat, and java.text.NumberFormat. A Date is the number of milliseconds since Jan 1, 1970, stored in a `long`. Most of Date's methods have been deprecated, so use the Calendar class for your date-manipulation tasks. Remember that in order to create instances of Calendar, DateFormat, and NumberFormat, you have to use static factory methods like `getInstance()`. The Locale class is used with DateFormat and NumberFormat to generate a variety of output styles that are language and/or country specific.

Parsing, Tokenizing, and Formatting To find specific pieces of data in large data sources, Java provides several mechanisms that use the concepts of regular expressions (Regex). regex expressions can be used with the java.util.regex package's Pattern and Matcher classes, as well as with java.util.Scanner and with the `String.split()` method. When creating regex patterns, you can use literal characters for matching or you can use metacharacters, that allow you to match on concepts like "find digits" or "find whitespace". Regex provides *quantifiers* that allow you to say things like "find one or more of these things in a row." You won't have to understand the Matcher methods that facilitate replacing strings in data.

Tokenizing is splitting delimited data into pieces. Delimiters are usually as simple as a comma, but they can be as complex as any other regex pattern. The java.util.Scanner class provides full tokenizing capabilities using regex, and allows you to tokenize in a loop so that you can stop the tokenizing process at any point. `String.split()` allows full regex patterns for tokenizing, but tokenizing is done in one step, hence large data sources can take a long time to process.

Formatting data for output can be handled by using the Formatter class, or more commonly by using the new PrintStream methods `format()` and `printf()`. Remember `format()` and `printf()` behave identically. To use these methods, you create a format string that is associated with every piece of data you want to format.

TWO-MINUTE DRILL

Here are some of the key points from the certification objectives in this chapter.

Using String, StringBuffer, and StringBuilder (Objective 3.1)

- ❑ String objects are immutable, and String reference variables are not.
- ❑ If you create a new String without assigning it, it will be lost to your program.
- ❑ If you redirect a String reference to a new String, the old String can be lost.
- ❑ String methods use zero-based indexes, except for the second argument of `substring()`.
- ❑ The String class is `final`—its methods can't be overridden.
- ❑ When the JVM finds a String literal, it is added to the String literal pool.
- ❑ Strings have a method: `length()`; arrays have an attribute named `length`.
- ❑ The StringBuffer's API is the same as the new StringBuilder's API, except that StringBuilder's methods are not synchronized for thread safety.
- ❑ StringBuilder methods should run faster than StringBuffer methods.
- ❑ All of the following bullets apply to both StringBuffer and StringBuilder:
  - ❑ They are mutable—they can change without creating a new object.
  - ❑ StringBuffer methods act on the invoking object, and objects can change without an explicit assignment in the statement.
  - ❑ StringBuffer `equals()` is not overridden; it doesn't compare values.
- ❑ Remember that chained methods are evaluated from left to right.
- ❑ String methods to remember: `charAt()`, `concat()`, `equalsIgnoreCase()`, `length()`, `replace()`, `substring()`, `toLowerCase()`, `toString()`, `toUpperCase()`, and `trim()`.
- ❑ StringBuffer methods to remember: `append()`, `delete()`, `insert()`, `reverse()`, and `toString()`.

File I/O (Objective 3.2)

- ❑ The classes you need to understand in java.io are File, FileReader, BufferedReader, FileWriter, BufferedWriter, PrintWriter, and Console.
- ❑ A new File object doesn't mean there's a new file on your hard drive.
- ❑ File objects can represent either a file or a directory.

- The File class lets you manage (add, rename, and delete) files and directories.
- The methods `createNewFile()` and `mkdir()` add entries to your file system.
- FileWriter and FileReader are low-level I/O classes. You can use them to write and read files, but they should usually be wrapped.
- Classes in java.io are designed to be "chained" or "wrapped." (This is a common use of the decorator design pattern.)
- It's very common to "wrap" a BufferedReader around a FileReader or a BufferedWriter around a FileWriter, to get access to higher-level (more convenient) methods.
- PrintWriters can be used to wrap other Writers, but as of Java 5 they can be built directly from Files or Strings.
- Java 5 PrintWriters have new `append()`, `format()`, and `printf()` methods.
- Console objects can read non-echoed input and are instantiated using System.console().

Serialization (Objective 3.3)

- The classes you need to understand are all in the java.io package; they include: ObjectOutputStream and ObjectInputStream primarily, and FileOutputStream and FileInputStream because you will use them to create the low-level streams that the ObjectXxxStream classes will use.
- A class must implement Serializable before its objects can be serialized.
- The `ObjectOutputStream.writeObject()` method serializes objects, and the `ObjectInputStream.readObject()` method deserializes objects.
- If you mark an instance variable `transient`, it will not be serialized even thought the rest of the object's state will be.
- You can supplement a class's automatic serialization process by implementing the `writeObject()` and `readObject()` methods. If you do this, embedding calls to `defaultWriteObject()` and `defaultReadObject()`, respectively, will handle the part of serialization that happens normally.
- If a superclass implements Serializable, then its subclasses do automatically.
- If a superclass doesn't implement Serializable, then when a subclass object is deserialized, the superclass constructor will be invoked, along with its superconstructor(s).
- DataInputStream and DataOutputStream aren't actually on the exam, in spite of what the Sun objectives say.

Dates, Numbers, and Currency (Objective 3.4)

- The classes you need to understand are java.util.Date, java.util.Calendar, java.text.DateFormat, java.text.NumberFormat, and java.util.Locale.
- Most of the Date class's methods have been deprecated.
- A Date is stored as a `long`, the number of milliseconds since January 1, 1970.
- Date objects are go-betweens the Calendar and Locale classes.
- The Calendar provides a powerful set of methods to manipulate dates, performing tasks such as getting days of the week, or adding some number of months or years (or other increments) to a date.
- Create Calendar instances using static factory methods (`getInstance()`).
- The Calendar methods you should understand are `add()`, which allows you to add or subtract various pieces (minutes, days, years, and so on) of dates, and `roll()`, which works like `add()` but doesn't increment a date's bigger pieces. (For example: adding 10 months to an October date changes the month to August, but doesn't increment the Calendar's year value.)
- DateFormat instances are created using static factory methods (`getInstance()` and `getDateInstance()`).
- There are several format "styles" available in the DateFormat class.
- DateFormat styles can be applied against various Locales to create a wide array of outputs for any given date.
- The `DateFormat.format()` method is used to create Strings containing properly formatted dates.
- The Locale class is used in conjunction with DateFormat and NumberFormat.
- Both DateFormat and NumberFormat objects can be constructed with a specific, immutable Locale.
- For the exam you should understand creating Locales using language, or a combination of language and country.

Parsing, Tokenizing, and Formatting (Objective 3.5)

- regex is short for regular expressions, which are the patterns used to search for data within large data sources.
- regex is a sub-language that exists in Java and other languages (such as Perl).
- regex lets you to create search patterns using literal characters or metacharacters. Metacharacters allow you to search for slightly more abstract data like "digits" or "whitespace".

- Study the `\d`, `\s`, `\w`, and . metacharacters
- regex provides for *quantifiers* which allow you to specify concepts like: "look for one or more digits in a row."
- Study the `?`, `*`, and `+` greedy quantifiers.
- Remember that metacharacters and Strings don't mix well unless you remember to "escape" them properly. For instance `String s = "\\d";`
- The Pattern and Matcher classes have Java's most powerful regex capabilities.
- You should understand the Pattern `compile()` method and the Matcher `matches()`, `pattern()`, `find()`, `start()`, and `group()` methods.
- You WON'T need to understand Matcher's replacement-oriented methods.
- You can use java.util.Scanner to do simple regex searches, but it is primarily intended for tokenizing.
- Tokenizing is the process of splitting delimited data into small pieces.
- In tokenizing, the data you want is called tokens, and the strings that separate the tokens are called delimiters.
- Tokenizing can be done with the Scanner class, or with `String.split()`.
- Delimiters are single characters like commas, or complex regex expressions.
- The Scanner class allows you to tokenize data from within a loop, which allows you to stop whenever you want to.
- The Scanner class allows you to tokenize Strings or streams or files.
- The `String.split()` method tokenizes the entire source data all at once, so large amounts of data can be quite slow to process.
- New to Java 5 are two methods used to format data for output. These methods are `format()` and `printf()`. These methods are found in the PrintStream class, an instance of which is the `out` in `System.out`.
- The `format()` and `printf()` methods have identical functionality.
- Formatting data with `printf()` (or `format()`) is accomplished using *formatting strings* that are associated with primitive or string arguments.
- The `format()` method allows you to mix literals in with your format strings.
- The format string values you should know are
  - Flags: `-`, `+`, `0`, "," , and `(`
  - Conversions: `b`, `c`, `d`, `f`, and `s`
- If your conversion character doesn't match your argument type, an exception will be thrown.

SELF TEST

1. Given:

```
import java.util.regex.*;
class Regex2 {
  public static void main(String[] args) {
    Pattern p = Pattern.compile(args[0]);
    Matcher m = p.matcher(args[1]);
    boolean b = false;
    while(b = m.find()) {
      System.out.print(m.start() + m.group());
    }
  }
}
```

And the command line:

```
java Regex2 "\d*" ab34ef
```

What is the result?

A. `234`

B. `334`

C. `2334`

D. `0123456`

E. `01234456`

F. `12334567`

G. Compilation fails

2. Given:

```
import java.io.*;
class Player {
  Player() { System.out.print("p"); }
}
class CardPlayer extends Player implements Serializable {
  CardPlayer() { System.out.print("c"); }
```

```
  public static void main(String[] args) {
    CardPlayer c1 = new CardPlayer();
    try {
      FileOutputStream fos = new FileOutputStream("play.txt");
      ObjectOutputStream os = new ObjectOutputStream(fos);
      os.writeObject(c1);
      os.close();
      FileInputStream fis = new FileInputStream("play.txt");
      ObjectInputStream is = new ObjectInputStream(fis);
      CardPlayer c2 = (CardPlayer) is.readObject();
      is.close();
    } catch (Exception x ) { }
  }
}
```

What is the result?

A. `pc`

B. `pcc`

C. `pcp`

D. `pcpc`

E. Compilation fails

F. An exception is thrown at runtime

3. Given:

```
class TKO {
  public static void main(String[] args) {
    String s = "-";
    Integer x = 343;
    long L343 = 343L;
    if(x.equals(L343))  s += ".e1 ";
    if(x.equals(343))  s += ".e2 ";
    Short s1 = (short)((new Short((short)343)) / (new Short((short)49)));
    if(s1 == 7)        s += "=s ";
    if(s1 < new Integer(7+1))  s += "fly ";
    System.out.println(s);
  }
}
```

Which of the following will be included in the output String `s`? (Choose all that apply.)

A. `.e1`
B. `.e2`
C. `=s`
D. `fly`
E. None of the above
F. Compilation fails
G. An exception is thrown at runtime

4. Given:

```
import java.io.*;

class Keyboard { }
public class Computer implements Serializable {
  private Keyboard k = new Keyboard();
  public static void main(String[] args) {
    Computer c = new Computer();
    c.storeIt(c);
  }
  void storeIt(Computer c) {
    try {
      ObjectOutputStream os = new ObjectOutputStream(
         new FileOutputStream("myFile"));
      os.writeObject(c);
      os.close();
      System.out.println("done");
    } catch (Exception x) {System.out.println("exc"); }
  }
}
```

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

A. `exc`
B. `done`
C. Compilation fails
D. Exactly one object is serialized
E. Exactly two objects are serialized

5. Using the fewest **fragments** possible (and filling the fewest slots possible), complete the **code** below so that the class builds a directory named "`dir3`" and creates a file named "`file3`" inside "`dir3`". Note you can use each fragment either zero or one times.

Code:

```
import java.io._____________

class Maker {
  public static void main(String[] args) {

    __________  __________  __________

    __________  __________  __________

    __________  __________  __________

    __________  __________  __________

    __________  __________  __________

    __________  __________  __________

    __________  __________  __________
  }
}
```

Fragments:

```
File;       FileDescriptor;      FileWriter;     Directory;
try {       .createNewDir();     File dir        File
{ }         (Exception x)        ("dir3");       file
file        .createNewFile();    = new File      = new File
dir         (dir, "file3");      (dir, file);    .createFile();
} catch     ("dir3", "file3");   .mkdir();       File file
```

6. Given that 1119280000000L is roughly the number of milliseconds from Jan 1, 1970, to June 20, 2005, and that you want to print that date in German, using the `LONG` style such that "`June`" will be displayed as "`Juni`", complete the code using the fragments below. Note: you can use each fragment either zero or one times, and you might not need to fill all of the slots.

Code:

```
import java._____________

import java._____________

class DateTwo {
  public static void main(String[] args) {
    Date d = new Date(1119280000000L);

    DateFormat df = _________________________________

                    _________________ , _________________ );

    System.out.println(_________________
  }
}
```

Fragments:

```
io.*;      new DateFormat(                Locale.LONG
nio.*;     DateFormat.getInstance(        Locale.GERMANY
util.*;    DateFormat.getDateInstance(    DateFormat.LONG
text.*;    util.regex;                    DateFormat.GERMANY
date.*;    df.format(d));                 d.format(df));
```

7. Given:

```
import java.io.*;
class Directories {
  static String [] dirs = {"dir1", "dir2"};
  public static void main(String [] args) {
    for (String d : dirs) {

      // insert code 1 here

      File file = new File(path, args[0]);

      // insert code 2 here
    }
  }
}
```

and that the invocation

```
java Directories file2.txt
```

is issued from a directory that has two subdirectories, "dir1" and "dir2", and that "dir1" has a file "file1.txt" and "dir2" has a file "file2.txt", and the output is "false true"; which set(s) of code fragments must be inserted? (Choose all that apply.)

A.
```
String path = d;

System.out.print(file.exists() + " ");
```

B.
```
String path = d;

System.out.print(file.isFile() + " ");
```

C.
```
String path = File.separator + d;

System.out.print(file.exists() + " ");
```

D.
```
String path = File.separator + d;

System.out.print(file.isFile() + " ");
```

8. Given:

```
import java.io.*;

public class TestSer {
  public static void main(String[] args) {
    SpecialSerial s = new SpecialSerial();
    try {
      ObjectOutputStream os = new ObjectOutputStream(
         new FileOutputStream("myFile"));
      os.writeObject(s);  os.close();
      System.out.print(++s.z + " ");

      ObjectInputStream is = new ObjectInputStream(
         new FileInputStream("myFile"));
      SpecialSerial s2 = (SpecialSerial)is.readObject();
      is.close();
```

```
      System.out.println(s2.y + " " + s2.z);
    } catch (Exception x) {System.out.println("exc"); }
  }
}
class SpecialSerial implements Serializable {
  transient int y = 7;
  static int z = 9;
}
```

Which are true? (Choose all that apply.)

A. Compilation fails

B. The output is `10 0 9`

C. The output is `10 0 10`

D. The output is `10 7 9`

E. The output is `10 7 10`

F. In order to alter the standard deserialization process you would implement the `readObject()` method in `SpecialSerial`

G. In order to alter the standard deserialization process you would implement the `defaultReadObject()` method in `SpecialSerial`

9. Given:

```
 3. public class Theory {
 4.   public static void main(String[] args) {
 5.     String s1 = "abc";
 6.     String s2 = s1;
 7.     s1 += "d";
 8.     System.out.println(s1 + " " + s2 + " " + (s1==s2));
 9.
10.     StringBuffer sb1 = new StringBuffer("abc");
11.     StringBuffer sb2 = sb1;
12.     sb1.append("d");
13.     System.out.println(sb1 + " " + sb2 + " " + (sb1==sb2));
14.   }
15. }
```

Which are true? (Choose all that apply.)

A. Compilation fails

B. The first line of output is `abc abc true`

C. The first line of output is `abc abc false`

D. The first line of output is `abcd abc false`

E. The second line of output is `abcd abc false`

F. The second line of output is `abcd abcd true`

G. The second line of output is `abcd abcd false`

10. Given:

```
 3. import java.io.*;
 4. public class ReadingFor {
 5.   public static void main(String[] args) {
 6.     String s;
 7.     try {
 8.       FileReader fr = new FileReader("myfile.txt");
 9.       BufferedReader br = new BufferedReader(fr);
10.       while((s = br.readLine()) != null)
11.         System.out.println(s);
12.       br.flush();
13.     } catch (IOException e) { System.out.println("io error"); }
16.   }
17. }
```

And given that myfile.txt contains the following two lines of data:

```
ab
cd
```

What is the result?

A. `ab`

B. `abcd`

C.
```
ab
cd
```

D.
```
a
b
c
d
```

E. Compilation fails

11. Given:

```
 3. import java.io.*;
 4. public class Talker {
 5.   public static void main(String[] args) {
 6.     Console c = System.console();
 7.     String u = c.readLine("%s", "username: ");
 8.     System.out.println("hello " + u);
 9.     String pw;
10.     if(c != null && (pw = c.readPassword("%s", "password: ")) != null)
11.       // check for valid password
12.   }
13. }
```

If line 6 creates a valid Console object, and if the user enters *fred* as a username and *1234* as a password, what is the result? (Choose all that apply.)

A. `username:`
`password:`

B. `username: fred`
`password:`

C. `username: fred`
`password: 1234`

D. Compilation fails

E. An exception is thrown at runtime

12. Given:

```
 3. import java.io.*;
 4. class Vehicle { }
 5. class Wheels { }
 6. class Car extends Vehicle implements Serializable {  }
 7. class Ford extends Car { }
 8. class Dodge extends Car {
 9.   Wheels w = new Wheels();
10. }
```

Instances of which class(es) can be serialized? (Choose all that apply.)

A. Car

B. Ford

C. Dodge
D. Wheels
E. Vehicle

13. Given:

```
 3. import java.text.*;
 4. public class Slice {
 5.   public static void main(String[] args) {
 6.     String s = "987.123456";
 7.     double d  = 987.123456d;
 8.     NumberFormat nf = NumberFormat.getInstance();
 9.     nf.setMaximumFractionDigits(5);
10.     System.out.println(nf.format(d) + " ");
11.     try {
12.       System.out.println(nf.parse(s));
13.     } catch (Exception e) { System.out.println("got exc"); }
14.   }
15. }
```

Which are true? (Choose all that apply.)

A. The output is `987.12345 987.12345`
B. The output is `987.12346 987.12345`
C. The output is `987.12345 987.123456`
D. The output is `987.12346 987.123456`
E. The try/catch block is unnecessary
F. The code compiles and runs without exception
G. The invocation of `parse()` must be placed within a try/catch block

14. Given:

```
 3. import java.util.regex.*;
 4. public class Archie {
 5.   public static void main(String[] args) {
 6.     Pattern p = Pattern.compile(args[0]);
 7.     Matcher m = p.matcher(args[1]);
 8.     int count = 0;
 9.     while(m.find())
10.       count++;
```

```
11.      System.out.print(count);
12.    }
13. }
```

And given the command line invocation:

```
java Archie "\d+" ab2c4d67
```

What is the result?

A. 0

B. 3

C. 4

D. 8

E. 9

F. Compilation fails

15. Given:

```
 3. import java.util.*;
 4. public class Looking {
 5.   public static void main(String[] args) {
 6.     String input = "1 2 a 3 45 6";
 7.     Scanner sc = new Scanner(input);
 8.     int x = 0;
 9.     do {
10.       x = sc.nextInt();
11.       System.out.print(x + " ");
12.     } while (x!=0);
13.   }
14. }
```

What is the result?

A. `1 2`

B. `1 2 3 45 6`

C. `1 2 3 4 5 6`

D. `1 2 a 3 45 6`

E. Compilation fails

F. `1 2` followed by an exception

SELF TEST ANSWERS

1. Given:

```
import java.util.regex.*;
class Regex2 {
  public static void main(String[] args) {
    Pattern p = Pattern.compile(args[0]);
    Matcher m = p.matcher(args[1]);
    boolean b = false;
    while(b = m.find()) {
      System.out.print(m.start() + m.group());
    }
  }
}
```

And the command line:

```
java Regex2 "\d*" ab34ef
```

What is the result?

A. `234`

B. `334`

C. `2334`

D. `0123456`

E. `01234456`

F. `12334567`

G. Compilation fails

Answer:

☑ **E** is correct. The `\d` is looking for digits. The `*` is a quantifier that looks for `0` to many occurrences of the pattern that precedes it. Because we specified `*`, the `group()` method returns empty Strings until consecutive digits are found, so the only time `group()` returns a value is when it returns 34 when the matcher finds digits starting in position 2. The `start()` method returns the starting position of the previous match because, again, we said find `0` to many occurrences.

☒ **A, B, C, D, F,** and **G** are incorrect based on the above. (Objective 3.5)

2. Given:

```
import java.io.*;
class Player {
  Player() { System.out.print("p"); }
}
class CardPlayer extends Player implements Serializable {
  CardPlayer() { System.out.print("c"); }
  public static void main(String[] args) {
    CardPlayer c1 = new CardPlayer();
    try {
      FileOutputStream fos = new FileOutputStream("play.txt");
      ObjectOutputStream os = new ObjectOutputStream(fos);
      os.writeObject(c1);
      os.close();
      FileInputStream fis = new FileInputStream("play.txt");
      ObjectInputStream is = new ObjectInputStream(fis);
      CardPlayer c2 = (CardPlayer) is.readObject();
      is.close();
    } catch (Exception x ) { }
  }
}
```

What is the result?

A. `pc`

B. `pcc`

C. `pcp`

D. `pcpc`

E. Compilation fails

F. An exception is thrown at runtime

Answer:

☑ **C** is correct. It's okay for a class to implement Serializable even if its superclass doesn't. However, when you deserialize such an object, the non-serializable superclass must run its constructor. Remember, constructors don't run on deserialized classes that implement Serializable.

☒ **A, B, D, E,** and **F** are incorrect based on the above. (Objective 3.3)

3. Given:

```
class TKO {
  public static void main(String[] args) {
    String s = "-";
    Integer x = 343;
    long L343 = 343L;
    if(x.equals(L343))  s += ".e1 ";
    if(x.equals(343))  s += ".e2 ";
    Short s1 = (short)((new Short((short)343)) / (new Short((short)49)));
    if(s1 == 7)        s += "=s ";
    if(s1 < new Integer(7+1))  s += "fly ";
    System.out.println(s);
} }
```

Which of the following will be included in the output String `s`? (Choose all that apply.)

A. `.e1`

B. `.e2`

C. `=s`

D. `fly`

E. None of the above

F. Compilation fails

G. An exception is thrown at runtime

Answer:

☑ **B, C,** and **D** are correct. Remember, that the `equals()` method for the integer wrappers will only return `true` if the two primitive types and the two values are equal. With **C**, it's okay to unbox and use `==`. For **D**, it's okay to create a wrapper object with an expression, and unbox it for comparison with a primitive.

☒ **A, E, F,** and **G** are incorrect based on the above. (Remember that **A** is using the `equals()` method to try to compare two different types.) (Objective 3.1)

4. Given:

```
import java.io.*;

class Keyboard { }
public class Computer implements Serializable {
```

```
  private Keyboard k = new Keyboard();
  public static void main(String[] args) {
    Computer c = new Computer();
    c.storeIt(c);
  }
  void storeIt(Computer c) {
    try {
      ObjectOutputStream os = new ObjectOutputStream(
         new FileOutputStream("myFile"));
      os.writeObject(c);
      os.close();
      System.out.println("done");
    } catch (Exception x) {System.out.println("exc"); }
  }
}
```

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

A. `exc`

B. `done`

C. Compilation fails

D. Exactly one object is serialized

E. Exactly two objects are serialized

Answer:

☑ **A** is correct. An instance of type Computer Has-a Keyboard. Because Keyboard doesn't implement Serializable, any attempt to serialize an instance of Computer will cause an exception to be thrown.

☒ **B, C, D,** and **E** are incorrect based on the above. If Keyboard did implement Serializable then two objects would have been serialized. (Objective 3.3)

5. Using the fewest fragments possible (and filling the fewest slots possible), complete the code below so that the class builds a directory named "`dir3`" and creates a file named "`file3`" inside "`dir3`". Note you can use each fragment either zero or one times.

Code:

```
import java.io._____________

class Maker {
  public static void main(String[] args) {

    ___________  ___________  ___________

    ___________  ___________  ___________

    ___________  ___________  ___________

    ___________  ___________  ___________

    ___________  ___________  ___________

    ___________  ___________  ___________

    ___________  ___________  ___________
} }
```

Fragments:

```
File;      FileDescriptor;     FileWriter;    Directory;
try {      .createNewDir();    File dir       File
{ }        (Exception x)       ("dir3");      file
file       .createNewFile();   = new File     = new File
dir        (dir, "file3");     (dir, file);   .createFile();
} catch    ("dir3", "file3");  .mkdir();      File file
```

Answer:

```
import java.io.File;
class Maker {
  public static void main(String[] args) {
    try {
      File dir = new File("dir3");
      dir.mkdir();
      File file = new File(dir, "file3");
      file.createNewFile();
    } catch (Exception x) { }
} }
```

Notes: The `new File` statements don't make actual files or directories, just objects. You need the `mkdir()` and `createNewFile()` methods to actually create the directory and the file. (Objective 3.2)

6. Given that 1119280000000L is roughly the number of milliseconds from Jan. 1, 1970, to June 20, 2005, and that you want to print that date in German, using the `LONG` style such that "`June`" will be displayed as "`Juni`", complete the code using the fragments below. Note: you can use each fragment either zero or one times, and you might not need to fill all of the slots.

Code:

```
import java._____________
import java._____________

class DateTwo {
  public static void main(String[] args) {
    Date d = new Date(1119280000000L);
    DateFormat df = _________________________________

                    _________________ , ___________________ );

    System.out.println(_________________
  }
}
```

Fragments:

```
io.*;      new DateFormat(                 Locale.LONG
nio.*;     DateFormat.getInstance(         Locale.GERMANY
util.*;    DateFormat.getDateInstance(     DateFormat.LONG
text.*;    util.regex;                     DateFormat.GERMANY
date.*;    df.format(d));                  d.format(df));
```

Answer:

```
import java.util.*;
import java.text.*;
class DateTwo {
  public static void main(String[] args) {
    Date d = new Date(1119280000000L);
  DateFormat df = DateFormat.getDateInstance(
                    DateFormat.LONG, Locale.GERMANY);
    System.out.println(df.format(d));
  }
}
```

Notes: Remember that you must build `DateFormat` objects using static methods. Also remember that you must specify a Locale for a `DateFormat` object at the time of instantiation. The `getInstance()` method does not take a Locale. (Objective 3.4)

7. Given:

```
import java.io.*;

class Directories {
  static String [] dirs = {"dir1", "dir2"};
  public static void main(String [] args) {
    for (String d : dirs) {

      // insert code 1 here

      File file = new File(path, args[0]);

      // insert code 2 here
    }
  }
}
```

and that the invocation

```
java Directories file2.txt
```

is issued from a directory that has two subdirectories, "`dir1`" and "`dir2`", and that "`dir1`" has a file "`file1.txt`" and "`dir2`" has a file "`file2.txt`", and the output is "`false true`", which set(s) of code fragments must be inserted? (Choose all that apply.)

A.
```
String path = d;

System.out.print(file.exists() + " ");
```

B.
```
String path = d;

System.out.print(file.isFile() + " ");
```

C.
```
String path = File.separator + d;

System.out.print(file.exists() + " ");
```

D.
```
String path = File.separator + d;

System.out.print(file.isFile() + " ");
```

Answer:

☑ **A** and **B** are correct. Because you are invoking the program from the directory whose direct subdirectories are to be searched, you don't start your path with a `File.separator` character. The `exists()` method tests for either files or directories; the `isFile()` method tests only for files. Since we're looking for a file, both methods work.

☒ **C** and **D** are incorrect based on the above. (Objective 3.2)

8. Given:

```
import java.io.*;

public class TestSer {
  public static void main(String[] args) {
    SpecialSerial s = new SpecialSerial();
    try {
      ObjectOutputStream os = new ObjectOutputStream(
         new FileOutputStream("myFile"));
      os.writeObject(s);  os.close();
      System.out.print(++s.z + " ");

      ObjectInputStream is = new ObjectInputStream(
         new FileInputStream("myFile"));
      SpecialSerial s2 = (SpecialSerial)is.readObject();
      is.close();
      System.out.println(s2.y + " " + s2.z);
    } catch (Exception x) {System.out.println("exc"); }
  }
}
class SpecialSerial implements Serializable {
  transient int y = 7;
  static int z = 9;
}
```

Which are true? (Choose all that apply.)

A. Compilation fails

B. The output is `10 0 9`

C. The output is `10 0 10`

D. The output is `10 7 9`

E. The output is `10 7 10`

F. In order to alter the standard deserialization process you would implement the `readObject()` method in `SpecialSerial`

G. In order to alter the standard deserialization process you would implement the `defaultReadObject()` method in `SpecialSerial`

Answer:

☑ **C** and **F** are correct. **C** is correct because `static` and `transient` variables are not serialized when an object is serialized. **F** is a valid statement.

☒ **A, B, D,** and **E** are incorrect based on the above. **G** is incorrect because you don't implement the `defaultReadObject()` method, you call it from within the `readObject()` method, along with any custom read operations your class needs. (Objective 3.3)

9. Given:

```
 3. public class Theory {
 4.   public static void main(String[] args) {
 5.     String s1 = "abc";
 6.     String s2 = s1;
 7.     s1 += "d";
 8.     System.out.println(s1 + " " + s2 + " " + (s1==s2));
 9.
10.     StringBuffer sb1 = new StringBuffer("abc");
11.     StringBuffer sb2 = sb1;
12.     sb1.append("d");
13.     System.out.println(sb1 + " " + sb2 + " " + (sb1==sb2));
14.   }
15. }
```

Which are true? (Choose all that apply.)

A. Compilation fails

B. The first line of output is `abc abc true`

C. The first line of output is `abc abc false`

D. The first line of output is `abcd abc false`

E. The second line of output is `abcd abc false`

F. The second line of output is `abcd abcd true`

G. The second line of output is `abcd abcd false`

Answer:

☑ **D** and **F** are correct. While String objects are immutable, references to Strings are mutable. The code `s1 += "d";` creates a new String object. StringBuffer objects are mutable, so the `append()` is changing the single StringBuffer object to which both StringBuffer references refer.

☒ **A, B, C, E,** and **G** are incorrect based on the above. (Objective 3.1)

10. Given:

```
 3. import java.io.*;
 4. public class ReadingFor {
 5.   public static void main(String[] args) {
 6.     String s;
 7.     try {
 8.       FileReader fr = new FileReader("myfile.txt");
 9.       BufferedReader br = new BufferedReader(fr);
10.       while((s = br.readLine()) != null)
11.         System.out.println(s);
12.       br.flush();
13.     } catch (IOException e) { System.out.println("io error"); }
16.   }
17. }
```

And given that myfile.txt contains the following two lines of data:

```
ab
cd
```

What is the result?

A. `ab`

B. `abcd`

C.
```
ab
cd
```

D.
```
a
b
c
d
```

E. Compilation fails

Answer:

☑ **E** is correct. You need to call `flush()` only when you're writing data. Readers don't have `flush()` methods. If not for the call to `flush()`, answer C would be correct.

☒ **A, B, C,** and **D** are incorrect based on the above. (Objective 3.2)

11. Given:

```
 3. import java.io.*;
 4. public class Talker {
 5.   public static void main(String[] args) {
 6.     Console c = System.console();
 7.     String u = c.readLine("%s", "username: ");
 8.     System.out.println("hello " + u);
 9.     String pw;
10.     if(c != null && (pw = c.readPassword("%s", "password: ")) != null)
11.       // check for valid password
12.   }
13. }
```

If line 6 creates a valid Console object, and if the user enters *fred* as a username and *1234* as a password, what is the result? (Choose all that apply.)

A.
```
username:
password:
```

B.
```
username: fred
password:
```

C. `username: fred`
`password: 1234`

D. Compilation fails

E. An exception is thrown at runtime

Answer:

☑ **D** is correct. The `readPassword()` method returns a `char[]`. If a `char[]` were used, answer B would be correct.

☒ **A, B, C,** and **E** are incorrect based on the above. (Objective 3.2)

12. Given:

```
3. import java.io.*;
4. class Vehicle { }
5. class Wheels { }
6. class Car extends Vehicle implements Serializable {  }
7. class Ford extends Car { }
8. class Dodge extends Car {
9.   Wheels w = new Wheels();
10. }
```

Instances of which class(es) can be serialized? (Choose all that apply.)

A. Car

B. Ford

C. Dodge

D. Wheels

E. Vehicle

Answer:

☑ **A** and **B** are correct. Dodge instances cannot be serialized because they "have" an instance of Wheels, which is not serializable. Vehicle instances cannot be serialized even though the subclass Car can be.

☒ **C, D,** and **E** are incorrect based on the above. (Objective 3.3)

13. Given:

```
 3. import java.text.*;
 4. public class Slice {
 5.   public static void main(String[] args) {
 6.     String s = "987.123456";
 7.     double d  = 987.123456d;
 8.     NumberFormat nf = NumberFormat.getInstance();
 9.     nf.setMaximumFractionDigits(5);
10.     System.out.println(nf.format(d) + " ");
11.     try {
12.       System.out.println(nf.parse(s));
13.     } catch (Exception e) { System.out.println("got exc"); }
14.   }
15. }
```

Which are true? (Choose all that apply.)

A. The output is `987.12345 987.12345`

B. The output is `987.12346 987.12345`

C. The output is `987.12345 987.123456`

D. The output is `987.12346 987.123456`

E. The try/catch block is unnecessary

F. The code compiles and runs without exception

G. The invocation of `parse()` must be placed within a try/catch block

Answer:

☑ **D, F,** and **G** are correct. The `setMaximumFractionDigits()` applies to the formatting but not the parsing. The try/catch block is placed appropriately. This one might scare you into thinking that you'll need to memorize more than you really do. If you can remember that you're formatting the number and parsing the string you should be fine for the exam.

☒ **A, B, C,** and **E** are incorrect based on the above. (Objective 3.4)

14. Given:

```
3. import java.util.regex.*;
4. public class Archie {
5.   public static void main(String[] args) {
6.     Pattern p = Pattern.compile(args[0]);
```

```
 7.     Matcher m = p.matcher(args[1]);
 8.     int count = 0;
 9.     while(m.find())
10.       count++;
11.     System.out.print(count);
12.   }
13. }
```

And given the command line invocation:

```
java Archie "\d+" ab2c4d67
```

What is the result?

A. 0

B. 3

C. 4

D. 8

E. 9

F. Compilation fails

Answer:

☑ **B** is correct. The "\d" metacharacter looks for digits, and the + quantifier says look for "one or more" occurrences. The `find()` method will find three sets of one or more consecutive digits: 2, 4, and 67.

☒ **A, C, D, E,** and **F** are incorrect based on the above. (Objective 3.5)

15. Given:

```
 3. import java.util.*;
 4. public class Looking {
 5.   public static void main(String[] args) {
 6.     String input = "1 2 a 3 45 6";
 7.     Scanner sc = new Scanner(input);
 8.     int x = 0;
 9.     do {
10.       x = sc.nextInt();
11.       System.out.print(x + " ");
12.     } while (x!=0);
13.   }
14. }
```

What is the result?

A. `1 2`

B. `1 2 3 45 6`

C. `1 2 3 4 5 6`

D. `1 2 a 3 45 6`

E. Compilation fails

F. `1 2` followed by an exception

Answer:

☑ **F** is correct. The `nextXxx()` methods are typically invoked after a call to a `hasNextXxx()`, which determines whether the next token is of the correct type.

☒ **A, B, C, D,** and **E** are incorrect based on the above. (Objective 3.5)

7

Generics and Collections

CERTIFICATION OBJECTIVES

- Design Using Collections
- Override equals() and hashCode(), Distinguish == and equals()
- Use GenericVersions of Collections Including Set, List, and Map
- Use Type Parameters, Write Generics methods
- Use java.util to Sort and Search Use Comparable and Comparator

✓ Two-Minute Drill

Q&A Self Test

Generics are possibly the most talked about feature of Java 5. Some people love 'em, some people hate 'em, but they're here to stay. At their simplest, they can help make code easier to write, and more robust. At their most complex, they can be very, very hard to create, and maintain. Luckily, the exam creators stuck to the simple end of generics, covering the most common and useful features, and leaving out most of the especially tricky bits. Coverage of collections in this exam has expanded in two ways from the previous exam: the use of generics in collections, and the ability to sort and search through collections.

CERTIFICATION OBJECTIVE

Overriding hashCode() and equals() (Objective 6.2)

6.2 Distinguish between correct and incorrect overrides of corresponding hashCode and equals methods, and explain the difference between == and the equals method.

You're an object. Get used to it. You have state, you have behavior, you have a job. (Or at least your chances of getting one will go up after passing the exam.) If you exclude primitives, everything in Java is an object. Not just an *object*, but an Object with a capital O. Every exception, every event, every array extends from java.lang.Object. For the exam, you don't need to know every method in Object, but you will need to know about the methods listed in Table 7-1.

Chapter 9 covers `wait()`, `notify()`, and `notifyAll()`. The `finalize()` method was covered in Chapter 3. So in this section we'll look at just the `hashCode()` and `equals()` methods. Oh, that leaves out `toString()`, doesn't it. Okay, we'll cover that right now because it takes two seconds.

The toString() Method Override `toString()` when you want a mere mortal to be able to read something meaningful about the objects of your class. Code can call `toString()` on your object when it wants to read useful details about your object. When you pass an object reference to the `System.out.println()` method, for example, the object's `toString()` method is called, and the return of `toString()` is shown in the following example:

TABLE 7-1 Methods of Class Object Covered on the Exam

Method	Description
`boolean equals (Object obj)`	Decides whether two objects are meaningfully equivalent.
`void finalize()`	Called by garbage collector when the garbage collector sees that the object cannot be referenced.
`int hashCode()`	Returns a hashcode `int` value for an object, so that the object can be used in Collection classes that use hashing, including Hashtable, HashMap, and HashSet.
`final void notify()`	Wakes up a thread that is waiting for this object's lock.
`final void notifyAll()`	Wakes up *all* threads that are waiting for this object's lock.
`final void wait()`	Causes the current thread to wait until another thread calls `notify()` or `notifyAll()` on this object.
`String toString()`	Returns a "text representation" of the object.

```
public class HardToRead {
   public static void main (String [] args) {
     HardToRead h = new HardToRead();
     System.out.println(h);
   }
}
```

Running the HardToRead class gives us the lovely and meaningful,

```
% java HardToRead
HardToRead@a47e0
```

The preceding output is what you get when you don't override the `toString()` method of class Object. It gives you the class name (at least that's meaningful) followed by the `@` symbol, followed by the unsigned hexadecimal representation of the object's hashcode.

Trying to read this output might motivate you to override the `toString()` method in your classes, for example,

```
public class BobTest {
   public static void main (String[] args) {
     Bob f = new Bob("GoBobGo", 19);
```

```
      System.out.println(f);
    }
}
class Bob {
    int shoeSize;
    String nickName;
    Bob(String nickName, int shoeSize) {
      this.shoeSize = shoeSize;
      this.nickName = nickName;
    }
    public String toString() {
       return ("I am a Bob, but you can call me " + nickName +
               ". My shoe size is " + shoeSize);
    }
}
```

This ought to be a bit more readable:

```
% java BobTest
I am a Bob, but you can call me GoBobGo. My shoe size is 19
```

Some people affectionately refer to `toString()` as the "spill-your-guts method," because the most common implementations of `toString()` simply spit out the object's state (in other words, the current values of the important instance variables). That's it for `toString()`. Now we'll tackle `equals()` and `hashCode()`.

Overriding equals()

You learned about the `equals()` method in earlier chapters, where we looked at the wrapper classes. We discussed how comparing two object references using the == operator evaluates to `true` only when both references refer to the same object (because == simply looks at the bits in the variable, and they're either identical or they're not). You saw that the String class and the wrapper classes have overridden the `equals()` method (inherited from class Object), so that you could compare two different objects (of the same type) to see if their contents are meaningfully equivalent. If two different Integer instances both hold the `int` value `5`, as far as you're concerned they are equal. The fact that the value `5` lives in two separate objects doesn't matter.

When you really need to know if two references are identical, use ==. But when you need to know if the objects themselves (not the references) are equal, use the `equals()` method. For each class you write, you must decide if it makes sense to

consider two different instances equal. For some classes, you might decide that two objects can never be equal. For example, imagine a class Car that has instance variables for things like make, model, year, configuration—you certainly don't want your car suddenly to be treated as the very same car as someone with a car that has identical attributes. Your car is your car and you don't want your neighbor Billy driving off in it just because, "hey, it's really the same car; the `equals()` method said so." So no two cars should ever be considered exactly equal. If two references refer to one car, then you know that both are talking about one car, not two cars that have the same attributes. So in the case of a Car you might not ever need, or want, to override the `equals()` method. Of course, you know that isn't the end of the story.

What It Means If You Don't Override equals()

There's a potential limitation lurking here: if you don't override a class's `equals()` method, you won't be able to use those objects as a key in a hashtable and you probably won't get accurate Sets, such that there are no conceptual duplicates.

The `equals()` method in class Object uses only the == operator for comparisons, so unless you override `equals()`, two objects are considered equal only if the two references refer to the same object.

Let's look at what it means to not be able to use an object as a hashtable key. Imagine you have a car, a very specific car (say, John's red Subaru Outback as opposed to Mary's purple Mini) that you want to put in a HashMap (a type of hashtable we'll look at later in this chapter), so that you can search on a particular car and retrieve the corresponding Person object that represents the owner. So you add the car instance as the key to the HashMap (along with a corresponding Person object as the value). But now what happens when you want to do a search? You want to say to the HashMap collection, "Here's the car, now give me the Person object that goes with this car." But now you're in trouble unless you still have a reference to the exact object you used as the key when you added it to the Collection. *In other words, you can't make an identical Car object and use it for the search.*

The bottom line is this: if you want objects of your class to be used as keys for a hashtable (or as elements in any data structure that uses equivalency for searching for—and/or retrieving—an object), then you must override `equals()` so that two different instances can be considered the same. So how would we fix the car? You might override the `equals()` method so that it compares the unique VIN (Vehicle Identification Number) as the basis of comparison. That way, you can use one instance when you add it to a Collection, and essentially re-create an identical instance when you want to do a search based on that object as the key. Of course, overriding the `equals()` method for Car also allows the potential that more than one object representing a single unique car can exist, which might not be safe

in your design. Fortunately, the String and wrapper classes work well as keys in hashtables—they override the `equals()` method. So rather than using the actual car instance as the key into the car/owner pair, you could simply use a String that represents the unique identifier for the car. That way, you'll never have more than one instance representing a specific car, but you can still use the car—or rather, one of the car's attributes—as the search key.

Implementing an equals() Method

Let's say you decide to override `equals()` in your class. It might look like this:

```
public class EqualsTest {
  public static void main (String [] args) {
     Moof one = new Moof(8);
     Moof two = new Moof(8);
     if (one.equals(two)) {
        System.out.println("one and two are equal");
     }
  }
}
class Moof {
  private int moofValue;
  Moof(int val) {
     moofValue = val;
  }
  public int getMoofValue() {
     return moofValue;
  }
  public boolean equals(Object o) {
    if ((o instanceof Moof) && (((Moof)o).getMoofValue()
         == this.moofValue)) {
      return true;
    } else {
       return false;
    }
  }
}
```

Let's look at this code in detail. In the `main()` method of EqualsTest, we create two Moof instances, passing the same value `8` to the Moof constructor. Now look at the Moof class and let's see what it does with that constructor argument—it assigns the value to the `moofValue` instance variable. Now imagine that you've decided two Moof objects are the same if their `moofValue` is identical. So you override the

`equals()` method and compare the two `moofValues`. It is that simple. But let's break down what's happening in the `equals()` method:

```
1. public boolean equals(Object o) {
2.   if ((o instanceof Moof) && (((Moof)o).getMoofValue()
          == this.moofValue)) {
3.      return true;
4.     } else {
5.       return false;
6.     }
7.  }
```

First of all, you must observe all the rules of overriding, and in line 1 we are indeed declaring a valid override of the `equals()` method we inherited from Object.

Line 2 is where all the action is. Logically, we have to do two things in order to make a valid equality comparison.

First, be sure that the object being tested is of the correct type! It comes in polymorphically as type Object, so you need to do an `instanceof` test on it. Having two objects of different class types be considered equal is usually not a good idea, but that's a design issue we won't go into here. Besides, you'd still have to do the `instanceof` test just to be sure that you could cast the object argument to the correct type so that you can access its methods or variables in order to actually do the comparison. Remember, if the object doesn't pass the `instanceof` test, then you'll get a runtime ClassCastException. For example:

```
public boolean equals(Object o) {
   if (((Moof)o).getMoofValue() == this.moofValue){
      // the preceding line compiles, but it's BAD!
      return true;
   } else {
      return false;
   }
}
```

The `(Moof)o` cast will fail if `o` doesn't refer to something that IS-A Moof.

Second, compare the attributes we care about (in this case, just `moofValue`). Only the developer can decide what makes two instances equal. (For best performance, you're going to want to check the fewest number of attributes.)

In case you were a little surprised by the whole `((Moof)o).getMoofValue()` syntax, we're simply casting the object reference, `o`, just-in-time as we try to call a method that's in the Moof class but not in Object. Remember, without the cast, you

can't compile because the compiler would see the object referenced by `o` as simply, well, an Object. And since the Object class doesn't have a `getMoofValue()` method, the compiler would squawk (technical term). But then as we said earlier, even with the cast, the code fails at runtime if the object referenced by `o` isn't something that's castable to a Moof. So don't ever forget to use the `instanceof` test first. Here's another reason to appreciate the short circuit && operator—if the `instanceof` test fails, we'll never get to the code that does the cast, so we're always safe at runtime with the following:

```
if ((o instanceof Moof) && (((Moof)o).getMoofValue()
      == this.moofValue)) {
    return true;
} else {
    return false;
}
```

So that takes care of `equals()`…

Whoa…not so fast. If you look at the Object class in the Java API spec, you'll find what we call a contract specified in the `equals()` method. A Java contract is a set of rules that should be followed, or rather must be followed if you want to provide a "correct" implementation as others will expect it to be. Or to put it another way, if you don't follow the contract, your code may still compile and run, but your code (or someone else's) may break at runtime in some unexpected way.

exam Watch

Remember that the `equals()`, `hashCode()`, ***and*** `toString()` ***methods are all*** `public`. ***The following would not be a valid override of the*** `equals()` ***method, although it might appear to be if you don't look closely enough during the exam:***

```
class Foo { boolean equals(Object o) { } }
```

And watch out for the argument types as well. The following method is an overload, but not an override of the `equals()` ***method:***

```
class Boo { public boolean equals(Boo b) { } }
```

exam Watch

Be sure you're very comfortable with the rules of overriding so that you can identify whether a method from Object is being overridden, overloaded, or illegally redeclared in a class. The `equals()` method in class Boo changes the argument from Object to Boo, so it becomes an overloaded method and won't be called unless it's from your own code that knows about this new, different method that happens to also be named `equals()`.

The equals() Contract

Pulled straight from the Java docs, the `equals()` contract says

- It is **reflexive**. For any reference value `x`, `x.equals(x)` should return `true`.
- It is **symmetric**. For any reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
- It is **transitive**. For any reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` must return `true`.
- It is **consistent**. For any reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in equals comparisons on the object is modified.
- For any non-`null` reference value `x`, `x.equals(null)` should return `false`.

And you're so not off the hook yet. We haven't looked at the `hashCode()` method, but `equals()` and `hashCode()` are bound together by a joint contract that specifies if two objects are considered equal using the `equals()` method, then they must have identical hashcode values. So to be truly safe, your rule of thumb should be, if you override `equals()`, override `hashCode()` as well. So let's switch over to `hashCode()` and see how that method ties in to `equals()`.

Overriding hashCode()

Hashcodes are typically used to increase the performance of large collections of data. The hashcode value of an object is used by some collection classes (we'll look

at the collections later in this chapter). Although you can think of it as kind of an object ID number, it isn't necessarily unique. Collections such as HashMap and HashSet use the hashcode value of an object to determine how the object should be *stored* in the collection, and the hashcode is used again to help *locate* the object in the collection. For the exam you do not need to understand the deep details of how the collection classes that use hashing are implemented, but you do need to know which collections use them (but, um, they all have "hash" in the name so you should be good there). You must also be able to recognize an appropriate or correct implementation of `hashCode()`. This does not mean legal and does not even mean efficient. It's perfectly legal to have a terribly inefficient hashcode method in your class, as long as it doesn't violate the contract specified in the Object class documentation (we'll look at that contract in a moment). So for the exam, if you're asked to pick out an appropriate or correct use of hashcode, don't mistake appropriate for legal or efficient.

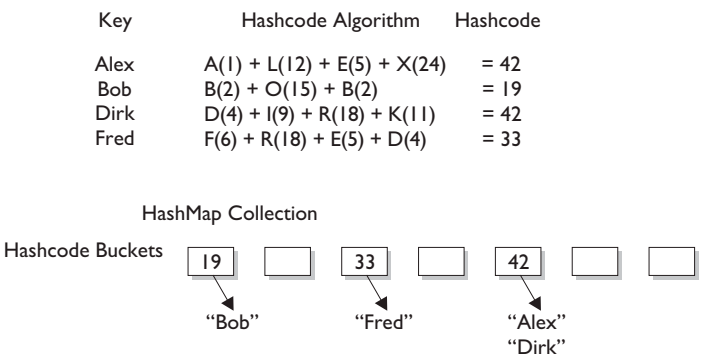
Understanding Hashcodes

In order to understand what's appropriate and correct, we have to look at how some of the collections use hashcodes.

Imagine a set of buckets lined up on the floor. Someone hands you a piece of paper with a name on it. You take the name and calculate an integer code from it by using A is 1, B is 2, and so on, and adding the numeric values of all the letters in the name together. A given name will always result in the same code; see Figure 7-1.

FIGURE 7-1

A simplified hashcode example



We don't introduce anything random, we simply have an algorithm that will always run the same way given a specific input, so the output will always be identical for any two identical inputs. So far so good? Now the way you use that code (and we'll call it a hashcode now) is to determine which bucket to place the piece of

paper into (imagine that each bucket represents a different code number you might get). Now imagine that someone comes up and shows you a name and says, "Please retrieve the piece of paper that matches this name." So you look at the name they show you, and run the same hashcode-generating algorithm. The hashcode tells you in which bucket you should look to find the name.

You might have noticed a little flaw in our system, though. Two different names might result in the same value. For example, the names `Amy` and `May` have the same letters, so the hashcode will be identical for both names. That's acceptable, but it does mean that when someone asks you (the bucket-clerk) for the `Amy` piece of paper, you'll still have to search through the target bucket reading each name until we find `Amy` rather than `May`. The hashcode tells you only which bucket to go into, but not how to locate the name once we're in that bucket.

exam Watch

In real-life hashing, it's not uncommon to have more than one entry in a bucket. Hashing retrieval is a two-step process.

1. ***Find the right bucket (using*** `hashCode()`***)***
2. ***Search the bucket for the right element (using*** `equals()` ***).***

So for efficiency, your goal is to have the papers distributed as evenly as possible across all buckets. Ideally, you might have just one name per bucket so that when someone asked for a paper you could simply calculate the hashcode and just grab the one paper from the correct bucket (without having to go flipping through different papers in that bucket until you locate the exact one you're looking for). The least efficient (but still functional) hashcode generator would return the same hashcode (say, 42) regardless of the name, so that all the papers landed in the same bucket while the others stood empty. The bucket-clerk would have to keep going to that one bucket and flipping painfully through each one of the names in the bucket until the right one was found. And if that's how it works, they might as well not use the hashcodes at all but just go to the one big bucket and start from one end and look through each paper until they find the one they want.

This distributed-across-the-buckets example is similar to the way hashcodes are used in collections. When you put an object in a collection that uses hashcodes, the collection uses the hashcode of the object to decide in which bucket/slot the object

should land. Then when you want to fetch that object (or, for a hashtable, retrieve the associated value for that object), you have to give the collection a reference to an object that the collection compares to the objects it holds in the collection. As long as the object (stored in the collection, like a paper in the bucket) you're trying to search for has the same hashcode as the object you're using for the search (the name you show to the person working the buckets), then the object will be found. But…and this is a Big One, imagine what would happen if, going back to our name example, you showed the bucket-worker a name and they calculated the code based on only half the letters in the name instead of all of them. They'd never find the name in the bucket because they wouldn't be looking in the correct bucket!

Now can you see why if two objects are considered equal, their hashcodes must also be equal? Otherwise, you'd never be able to find the object since the default hashcode method in class Object virtually always comes up with a unique number for each object, even if the `equals()` method is overridden in such a way that two or more objects are considered equal. It doesn't matter how equal the objects are if their hashcodes don't reflect that. So one more time: If two objects are equal, their hashcodes must be equal as well.

Implementing hashCode()

What the heck does a real hashcode algorithm look like? People get their PhDs on hashing algorithms, so from a computer science viewpoint, it's beyond the scope of the exam. The part we care about here is the issue of whether you follow the contract. And to follow the contract, think about what you do in the `equals()` method. You compare attributes. Because that comparison almost always involves instance variable values (remember when we looked at two Moof objects and considered them equal if their `int moofValues` were the same?). Your `hashCode()` implementation should use the same instance variables. Here's an example:

```
class HasHash {
  public int x;
  HasHash(int xVal) { x = xVal; }

  public boolean equals(Object o) {
    HasHash h = (HasHash) o; // Don't try at home without
                             // instanceof test
    if (h.x == this.x) {
      return true;
    } else {
      return false;
    }
```

```
  }
  public int hashCode() { return (x * 17); }
}
```

This `equals()` method says two objects are equal if they have the same `x` value, so objects with the same `x` value will have to return identical hashcodes.

exam Watch

A `hashCode()` that returns the same value for all instances whether they're equal or not is still a legal—even appropriate—`hashCode()` method! For example,

```
public int hashCode() { return 1492; }
```

This does not violate the contract. Two objects with an `x` value of `8` will have the same hashcode. But then again, so will two unequal objects, one with an `x` value of `12` and the other a value of `-920`. This `hashCode()` method is horribly inefficient, remember, because it makes all objects land in the same bucket, but even so, the object can still be found as the collection cranks through the one and only bucket—using `equals()`—trying desperately to finally, painstakingly, locate the correct object. In other words, the hashcode was really no help at all in speeding up the search, even though improving search speed is hashcode's intended purpose! Nonetheless, this one-hash-fits-all method would be considered appropriate and even correct because it doesn't violate the contract. Once more, correct does not necessarily mean good.

Typically, you'll see `hashCode()` methods that do some combination of ^-ing (XOR-ing) a class's instance variables (in other words, twiddling their bits), along with perhaps multiplying them by a prime number. In any case, while the goal is to get a wide and random distribution of objects across buckets, the contract (and whether or not an object can be found) requires only that two equal objects have equal hashcodes. The exam does not expect you to rate the efficiency of a `hashCode()` method, but you must be able to recognize which ones will and will not work (work meaning "will cause the object to be found in the collection").

Now that we know that two equal objects must have identical hashcodes, is the reverse true? Do two objects with identical hashcodes have to be considered equal? Think about it—you might have lots of objects land in the same bucket because their hashcodes are identical, but unless they also pass the `equals()` test, they won't come up as a match in a search through the collection. This is exactly what you'd

get with our very inefficient everybody-gets-the-same-hashcode method. It's legal and correct, just slooooow.

So in order for an object to be located, the search object and the object in the collection must have both identical hashcode values and return `true` for the `equals()` method. So there's just no way out of overriding both methods to be absolutely certain that your objects can be used in Collections that use hashing.

The hashCode() Contract

Now coming to you straight from the fabulous Java API documentation for class Object, may we present (drum roll) the `hashCode()` contract:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode()` method must consistently return the same integer, provided no information used in `equals()` comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode()` method on each of the two objects must produce the same integer result.
- It is NOT required that if two objects are unequal according to the `equals(java.lang.Object)` method, then calling the `hashCode()` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.

And what this means to you is…

Condition	Required	Not Required (But Allowed)
`x.equals(y) == true`	`x.hashCode() == y.hashCode()`	
`x.hashCode() == y.hashCode()`		`x.equals(y) == true`
`x.equals(y) == false`		`No hashCode() requirements`
`x.hashCode() != y.hashCode()`	`x.equals(y) == false`	

So let's look at what else might cause a `hashCode()` method to fail. What happens if you include a transient variable in your `hashCode()` method? While that's legal (compiler won't complain), under some circumstances an object you put in a collection won't be found. As you know, serialization saves an object so that it can be reanimated later by deserializing it back to full objectness. But danger Will Robinson—remember that transient variables are not saved when an object is serialized. A bad scenario might look like this:

```
class SaveMe implements Serializable{
  transient int x;
  int y;
   SaveMe(int xVal, int yVal) {
      x = xVal;
      y = yVal;
   }
  public int hashCode() {
     return (x ^ y);      // Legal, but not correct to
                          // use a transient variable
  }
  public boolean equals(Object o) {
     SaveMe test = (SaveMe)o;
     if (test.y == y && test.x == x) { // Legal, not correct
       return true;
     } else {
       return false;
     }
  }
}
```

Here's what could happen using code like the preceding example:

1. Give an object some state (assign values to its instance variables).
2. Put the object in a HashMap, using the object as a key.
3. Save the object to a file using serialization without altering any of its state.
4. Retrieve the object from the file through deserialization.
5. Use the deserialized (brought back to life on the heap) object to get the object out of the HashMap.

Oops. The object in the collection and the supposedly same object brought back to life are no longer identical. The object's transient variable will come

back with a default value rather than the value the variable had at the time it was saved (or put into the HashMap). So using the preceding SaveMe code, if the value of `x` is `9` when the instance is put in the HashMap, then since `x` is used in the calculation of the hashcode, when the value of `x` changes, the hashcode changes too. And when that same instance of SaveMe is brought back from deserialization, `x == 0`, regardless of the value of `x` at the time the object was serialized. So the new hashcode calculation will give a different hashcode, and the `equals()` method fails as well since `x` is used to determine object equality.

Bottom line: `transient` variables can really mess with your `equals()` and `hashCode()` implementations. Keep variables non-`transient` or, if they must be marked `transient`, don't use them to determine hashcodes or equality.

CERTIFICATION OBJECTIVE

Collections (Exam Objective 6.1)

6.1 Given a design scenario, determine which collection classes and/or interfaces should be used to properly implement that design, including the use of the Comparable interface.

Can you imagine trying to write object-oriented applications without using data structures like hashtables or linked lists? What would you do when you needed to maintain a sorted list of, say, all the members in your Simpsons fan club? Obviously you can do it yourself; Amazon.com must have thousands of algorithm books you can buy. But with the kind of schedules programmers are under today, it's almost too painful to consider.

The Collections Framework in Java, which took shape with the release of JDK 1.2 and was expanded in 1.4 and again in Java 5, and yet again in Java 6, gives you lists, sets, maps, and queues to satisfy most of your coding needs. They've been tried, tested, and tweaked. Pick the best one for your job and you'll get—at the least—reasonable performance. And when you need something a little more custom, the Collections Framework in the java.util package is loaded with interfaces and utilities.

So What Do You Do with a Collection?

There are a few basic operations you'll normally use with collections:

- Add objects to the collection.
- Remove objects from the collection.

- Find out if an object (or group of objects) is in the collection.
- Retrieve an object from the collection (without removing it).
- Iterate through the collection, looking at each element (object) one after another.

Key Interfaces and Classes of the Collections Framework

For the exam you'll need to know which collection to choose based on a stated requirement. The collections API begins with a group of interfaces, but also gives you a truckload of concrete classes. The core interfaces you need to know for the exam (and life in general) are the following nine:

`Collection`	`Set`	`SortedSet`
`List`	`Map`	`SortedMap`
`Queue`	`NavigableSet`	`NavigableMap`

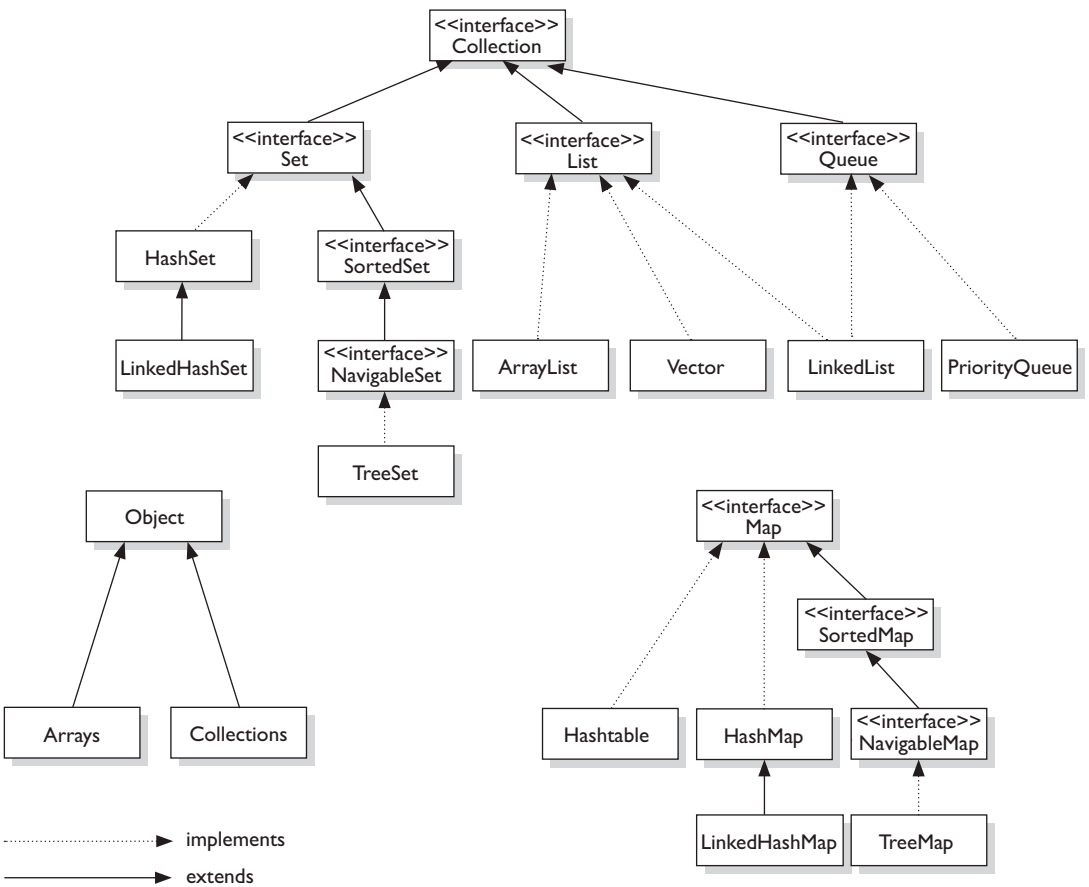
The core concrete implementation classes you need to know for the exam are the following 13 (there are others, but the exam doesn't specifically cover them):

Maps	Sets	Lists	Queues	Utilities
`HashMap`	`HashSet`	`ArrayList`	`PriorityQueue`	`Collections`
`Hashtable`	`LinkedHashSet`	`Vector`		`Arrays`
`TreeMap`	`TreeSet`	`LinkedList`		
`LinkedHashMap`				

Not all collections in the Collections Framework actually implement the Collection interface. In other words, not all collections pass the IS-A test for Collection. Specifically, none of the Map-related classes and interfaces extend from Collection. So while SortedMap, Hashtable, HashMap, TreeMap, and LinkedHashMap are all thought of as collections, none are actually extended from Collection-with-a-capital-C (see Figure 7-2). To make things a little more confusing, there are really three overloaded uses of the word "collection":

- collection (lowercase *c*), which represents any of the data structures in which objects are stored and iterated over.
- Collection (capital C), which is actually the java.util.Collection interface from which Set, List, and Queue extend. (That's right, extend, not implement. There are no direct implementations of Collection.)
- Collections (capital C and ends with *s*) is the java.util.Collections class that holds a pile of `static` utility methods for use with collections.

FIGURE 7-2 The interface and class hierarchy for collections



exam Watch

You can so easily mistake "Collections" for "Collection"—be careful. Keep in mind that Collections is a class, with static utility methods, while Collection is an interface with declarations of the methods common to most collections including `add()`, `remove()`, `contains()`, `size()`, ***and*** `iterator()`.

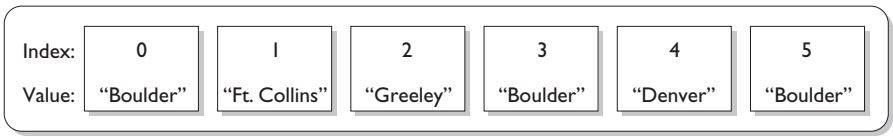
Collections come in four basic flavors:

- **Lists** *Lists* of things (classes that implement List).
- **Sets** *Unique* things (classes that implement Set).
- **Maps** Things with a *unique* ID (classes that implement Map).
- **Queues** Things arranged by the order in which they are to be processed.

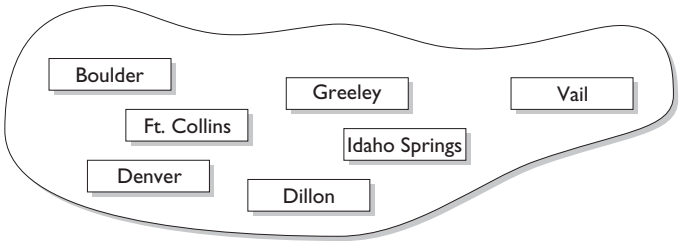
Figure 7-3 illustrates the structure of a List, a Set, and a Map.

FIGURE 7-3

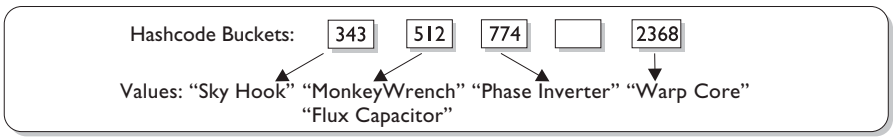
The structure of a List, a Set, and a Map



List: The salesman's itinerary (Duplicates allowed)



Set: The salesman's territory (No duplicates allowed)



HashMap: the salesman's products (Keys generated from product IDs)

But there are sub-flavors within those four flavors of collections:

Sorted	Unsorted	Ordered	Unordered

An implementation class can be unsorted and unordered, ordered but unsorted, or both ordered and sorted. But an implementation can never be sorted but unordered, because sorting is a specific type of ordering, as you'll see in a moment. For example, a HashSet is an unordered, unsorted set, while a LinkedHashSet is an ordered (but not sorted) set that maintains the order in which objects were inserted.

Maybe we should be explicit about the difference between sorted and ordered, but first we have to discuss the idea of iteration. When you think of iteration, you may think of iterating over an array using, say, a `for` loop to access each element in the array in order ([0], [1], [2], and so on). Iterating through a collection usually means walking through the elements one after another starting from the first element. Sometimes, though, even the concept of *first* is a little strange—in a Hashtable there really isn't a notion of first, second, third, and so on. In a Hashtable, the elements are placed in a (seemingly) chaotic order based on the hashcode of the key. But something has to go first when you iterate; thus, when you iterate over a Hashtable, there will indeed be an order. But as far as you can tell, it's completely arbitrary and can change in apparently random ways as the collection changes.

Ordered When a collection is ordered, it means you can iterate through the collection in a specific (not-random) order. A Hashtable collection is not ordered. Although the Hashtable itself has internal logic to determine the order (based on hashcodes and the implementation of the collection itself), you won't find any order when you iterate through the Hashtable. An ArrayList, however, keeps the order established by the elements' index position (just like an array). LinkedHashSet keeps the order established by insertion, so the last element inserted is the last element in the LinkedHashSet (as opposed to an ArrayList, where you can insert an element at a specific index position). Finally, there are some collections that keep an order referred to as the natural order of the elements, and those collections are then not just ordered, but also sorted. Let's look at how natural order works for sorted collections.

Sorted A *sorted* collection means that the order in the collection is determined according to some rule or rules, known as the sort order. A sort order has nothing to do with when an object was added to the collection, or when was the last time it was accessed, or what "position" it was added at. Sorting is done based on properties of the objects themselves. You put objects into the collection, and the collection will figure out what order to put them in, based on the sort order. A collection that keeps an order (such as any List, which uses insertion order) is not really considered *sorted* unless it sorts using some kind of sort order. Most commonly, the sort order used is something called the *natural* order. What does that mean?

You know how to sort alphabetically—A comes before B, F comes before G, and so on. For a collection of String objects, then, the natural order is alphabetical. For Integer objects, the natural order is by numeric value—1 before 2, and so on. And for Foo objects, the natural order is…um…we don't know. There is no natural order for Foo unless or until the Foo developer provides one, through an interface (*Comparable*)that defines how instances of a class can be compared to one another (does instance `a` come before `b`, or does instance `b` come before `a`?). If the developer decides that Foo objects should be compared using the value of some instance variable (let's say there's one called `bar`), then a sorted collection will order the Foo objects according to the rules in the Foo class for how to use the bar instance variable to determine the order. Of course, the Foo class might also inherit a natural order from a superclass rather than define its own order, in some cases.

Aside from natural order as specified by the Comparable interface, it's also possible to define other, different sort orders using another interface: *Comparator*. We will discuss how to use both Comparable and Comparator to define sort orders later in this chapter. But for now, just keep in mind that sort order (including natural order) is not the same as ordering by insertion, access, or index.

Now that we know about ordering and sorting, we'll look at each of the four interfaces, and we'll dive into the concrete implementations of those interfaces.

List Interface

A List cares about the index. The one thing that List has that non-lists don't have is a set of methods related to the index. Those key methods include things like `get(int index)`, `indexOf(Object o)`, `add(int index, Object obj)`, and so on. All three List implementations are ordered by index position—a position that you determine either by setting an object at a specific index or by adding it without specifying position, in which case the object is added to the end. The three List implementations are described in the following sections.

ArrayList Think of this as a growable array. It gives you fast iteration and fast random access. To state the obvious: it is an ordered collection (by index), but not sorted. You might want to know that as of version 1.4, ArrayList now implements the new RandomAccess interface—a marker interface (meaning it has no methods) that says, "this list supports fast (generally constant time) random access." Choose this over a LinkedList when you need fast iteration but aren't as likely to be doing a lot of insertion and deletion.

Vector Vector is a holdover from the earliest days of Java; Vector and Hashtable were the two original collections, the rest were added with Java 2 versions 1.2 and 1.4. A Vector is basically the same as an ArrayList, but Vector methods are synchronized for thread safety. You'll normally want to use ArrayList instead of Vector because the synchronized methods add a performance hit you might not need. And if you do need thread safety, there are utility methods in class Collections that can help. Vector is the only class other than ArrayList to implement RandomAccess.

LinkedList A LinkedList is ordered by index position, like ArrayList, except that the elements are doubly-linked to one another. This linkage gives you new methods (beyond what you get from the List interface) for adding and removing from the beginning or end, which makes it an easy choice for implementing a stack or queue. Keep in mind that a LinkedList may iterate more slowly than an ArrayList, but it's a good choice when you need fast insertion and deletion. As of Java 5, the LinkedList class has been enhanced to implement the java.util.Queue interface. As such, it now supports the common queue methods: `peek()`, `poll()`, and `offer()`.

Set Interface

A Set cares about uniqueness—it doesn't allow duplicates. Your good friend the `equals()` method determines whether two objects are identical (in which case only one can be in the set). The three Set implementations are described in the following sections.

HashSet A HashSet is an unsorted, unordered Set. It uses the hashcode of the object being inserted, so the more efficient your `hashCode()` implementation the better access performance you'll get. Use this class when you want a collection with no duplicates and you don't care about order when you iterate through it.

LinkedHashSet A LinkedHashSet is an ordered version of HashSet that maintains a doubly-linked List across all elements. Use this class instead of HashSet when you care about the iteration order. When you iterate through a HashSet the order is unpredictable, while a LinkedHashSet lets you iterate through the elements in the order in which they were inserted.

exam Watch

When using HashSet or LinkedHashSet, the objects you add to them must override `hashCode()`***. If they don't override*** `hashCode()`***, the default*** `Object.hashCode()` ***method will allow multiple objects that you might consider "meaningfully equal" to be added to your "no duplicates allowed" set.***

TreeSet The TreeSet is one of two sorted collections (the other being TreeMap). It uses a Red-Black tree structure (but you knew that), and guarantees that the elements will be in ascending order, according to natural order. Optionally, you can construct a TreeSet with a constructor that lets you give the collection your own rules for what the order should be (rather than relying on the ordering defined by the elements' class) by using a Comparable or Comparator. As of Java 6, TreeSet implements NavigableSet.

Map Interface

A Map cares about unique identifiers. You map a unique key (the ID) to a specific value, where both the key and the value are, of course, objects. You're probably quite familiar with Maps since many languages support data structures that use a key/value or name/value pair. The Map implementations let you do things like search for a value based on the key, ask for a collection of just the values, or ask for a collection of just the keys. Like Sets, Maps rely on the `equals()` method to determine whether two keys are the same or different.

HashMap The HashMap gives you an unsorted, unordered Map. When you need a Map and you don't care about the order (when you iterate through it), then HashMap is the way to go; the other maps add a little more overhead. Where the keys land in the Map is based on the key's hashcode, so, like HashSet, the more efficient your `hashCode()` implementation, the better access performance you'll get. HashMap allows one `null` key and multiple `null` values in a collection.

Hashtable Like Vector, Hashtable has existed from prehistoric Java times. For fun, don't forget to note the naming inconsistency: HashMap vs. Hashtable. Where's the capitalization of *t*? Oh well, you won't be expected to spell it. Anyway, just as Vector is a synchronized counterpart to the sleeker, more modern ArrayList, Hashtable is the synchronized counterpart to HashMap. Remember that you don't synchronize a class, so when we say that Vector and Hashtable are synchronized, we just mean that the key methods of the class are synchronized. Another difference, though, is that while HashMap lets you have `null` values as well as one `null` key, a Hashtable doesn't let you have anything that's `null`.

LinkedHashMap Like its Set counterpart, LinkedHashSet, the LinkedHashMap collection maintains insertion order (or, optionally, access order). Although it will be somewhat slower than HashMap for adding and removing elements, you can expect faster iteration with a LinkedHashMap.

TreeMap You can probably guess by now that a TreeMap is a sorted Map. And you already know that by default, this means "sorted by the natural order of the elements." Like TreeSet, TreeMap lets you define a custom sort order (via a Comparable or Comparator) when you construct a TreeMap, that specifies how the elements should be compared to one another when they're being ordered. As of Java 6, TreeMap implements NavigableMap.

Queue Interface

A Queue is designed to hold a list of "to-dos," or things to be processed in some way. Although other orders are possible, queues are typically thought of as FIFO (first-in, first-out). Queues support all of the standard Collection methods and they also add methods to add and subtract elements and review queue elements.

PriorityQueue This class is new with Java 5. Since the LinkedList class has been enhanced to implement the Queue interface, basic queues can be handled with a LinkedList. The purpose of a PriorityQueue is to create a "priority-in, priority out" queue as opposed to a typical FIFO queue. A PriorityQueue's elements are ordered either by natural ordering (in which case the elements that are sorted first will be accessed first) or according to a Comparator. In either case, the elements' ordering represents their relative priority.

exam Watch

You can easily eliminate some answers right away if you recognize that, for example, a Map can't be the class to choose when you need a name/value pair collection, since Map is an interface and not a concrete implementation class. The wording on the exam is explicit when it matters, so if you're asked to choose an interface, choose an interface rather than a class that implements that interface. The reverse is also true—if you're asked to choose a class, don't choose an interface type.

Table 7-2 summarizes the 11 of the 13 concrete collection-oriented classes you'll need to understand for the exam. (Arrays and Collections are coming right up!)

TABLE 7-2 Collection Interface Concrete Implementation Classes

Class	Map	Set	List	Ordered	Sorted
`HashMap`	x			No	No
`Hashtable`	x			No	No
`TreeMap`	x			Sorted	By *natural order* or custom comparison rules
`LinkedHashMap`	x			By insertion order or last access order	No
`HashSet`		x		No	No
`TreeSet`		x		Sorted	By *natural order* or custom comparison rules
`LinkedHashSet`		x		By insertion order	No
`ArrayList`			x	By index	No
`Vector`			x	By index	No
`LinkedList`			x	By index	No
`PriorityQueue`				Sorted	By to-do order

exam Watch

Be sure you know how to interpret Table 7-2 in a practical way. For the exam, you might be expected to choose a collection based on a particular requirement, where that need is expressed as a scenario. For example, which collection would you use if you needed to maintain and search on a list of parts, identified by their unique alphanumeric serial number where the part would be of type Part? Would you change your answer at all if we modified the requirement such that you also need to be able to print out the parts in order, by their serial number? For the first question, you can see that since you have a Part class, but need to search for the objects based on a serial number, you need a Map. The key will be the serial number as a String, and the value will be the Part instance. The default choice should be HashMap, the quickest Map for access. But now when we amend the requirement to include getting the parts in order of their serial number, then we need a TreeMap—which maintains the natural order of the keys. Since the key is a String, the natural order for a String will be a standard alphabetical sort. If the requirement had been to keep track of which part was last accessed, then we'd probably need a LinkedHashMap. But since a LinkedHashMap loses the natural order (replacing it with last-accessed order), if we need to list the parts by serial number, we'll have to explicitly sort the collection, using a utility method.

CERTIFICATION OBJECTIVE

Using the Collections Framework (Objectives 6.3 and 6.5)

6.3 Write code that uses the NavigableSet and NavigableMap interfaces.

6.5 Use capabilities in the java.util package to write code to manipulate a list by sorting, performing a binary search, or converting the list to an array. Use capabilities in the java.util package to write code to manipulate an array by sorting, performing a binary search, or converting the array to a list. Use the java.util.Comparator and java.lang.Comparable interfaces to affect the sorting of lists and arrays. Furthermore, recognize the effect of the "natural ordering" of primitive wrapper classes and java.lang.String on sorting.

We've taken a high-level, theoretical look at the key interfaces and classes in the Collections Framework, now let's see how they work in practice.

ArrayList Basics

The `java.util.ArrayList` class is one of the most commonly used of all the classes in the Collections Framework. It's like an array on vitamins. Some of the advantages `ArrayList` has over arrays are

- It can grow dynamically.
- It provides more powerful insertion and search mechanisms than arrays.

Let's take a look at using an `ArrayList` that contains Strings. A key design goal of the Collections Framework was to provide rich functionality at the level of the main interfaces: List, Set, and Map. In practice, you'll typically want to instantiate an `ArrayList` polymorphically like this:

```
List myList = new ArrayList();
```

As of Java 5 you'll want to say

```
List<String> myList = new ArrayList<String>();
```

This kind of declaration follows the object oriented programming principle of "coding to an interface", and it makes use of generics. We'll say lots more about generics later in this chapter, but for now just know that, as of Java 5, the <String> syntax is the way that you declare a collection's type. (Prior to Java 5 there was no way to specify the type of a collection, and when we cover generics, we'll talk about the implications of mixing Java 5 (typed) and pre-Java 5 (untyped) collections.)

In many ways, `ArrayList<String>` is similar to a `String[]` in that it declares a container that can hold only Strings, but it's more powerful than a `String[]`. Let's look at some of the capabilities that an `ArrayList` has

```
List<String> test = new ArrayList<String>();
String s = "hi";
test.add("string");
test.add(s);
test.add(s+s);
System.out.println(test.size());
System.out.println(test.contains(42));
```

```
        System.out.println(test.contains("hihi"));
        test.remove("hi");
        System.out.println(test.size());
```

which produces

```
3
false
true
2
```

There's lots going on in this small program. Notice that when we declared the `ArrayList` we didn't give it a size. Then we were able to ask the `ArrayList` for its size, we were able to ask it whether it contained specific objects, we removed an object right out from the middle of it, and then we rechecked its size.

Autoboxing with Collections

In general, collections can hold Objects but not primitives. Prior to Java 5, a very common use for the wrapper classes was to provide a way to get a primitive into a collection. Prior to Java 5, you had to wrap a primitive by hand before you could put it into a collection. With Java 5, primitives still have to be wrapped, but autoboxing takes care of it for you.

```
List myInts = new ArrayList();   // pre Java 5 declaration
myInts.add(new Integer(42));     // had to wrap an int
```

As of Java 5 we can say

```
myInts.add(42);                  // autoboxing handles it!
```

In this last example, we are still adding an `Integer` object to `myInts` (not an `int` primitive); it's just that autoboxing handles the wrapping for us.

Sorting Collections and Arrays

Sorting and searching topics have been added to the exam for Java 5. Both collections and arrays can be sorted and searched using methods in the API.

Sorting Collections

Let's start with something simple like sorting an ArrayList of Strings alphabetically. What could be easier? Okay, we'll wait while you go find ArrayList's `sort()` method…got it? Of course, ArrayList doesn't give you any way to sort its contents, but the java.util.Collections class does

```
import java.util.*;
class TestSort1 {
  public static void main(String[] args) {
    ArrayList<String> stuff = new ArrayList<String>(); // #1
    stuff.add("Denver");
    stuff.add("Boulder");
    stuff.add("Vail");
    stuff.add("Aspen");
    stuff.add("Telluride");
    System.out.println("unsorted " + stuff);
    Collections.sort(stuff);                          // #2
    System.out.println("sorted   " + stuff);
  }
}
```

This produces something like this:

```
unsorted [Denver, Boulder, Vail, Aspen, Telluride]
sorted   [Aspen, Boulder, Denver, Telluride, Vail]
```

Line 1 is declaring an ArrayList of Strings, and line 2 is sorting the ArrayList alphabetically. We'll talk more about the Collections class, along with the Arrays class in a later section, for now let's keep sorting stuff.

Let's imagine we're building the ultimate home-automation application. Today we're focused on the home entertainment center, and more specifically the DVD control center. We've already got the file I/O software in place to read and write data between the `dvdInfo.txt` file and instances of class `DVDInfo`. Here are the key aspects of the class:

```
class DVDInfo {
  String title;
  String genre;
  String leadActor;
  DVDInfo(String t, String g, String a) {
```

```
    title = t;  genre = g;  leadActor = a;
  }
  public String toString() {
    return title + " " + genre + " " + leadActor + "\n";
  }
  // getters and setter go here
}
```

Here's the DVD data that's in the `dvdinfo.txt` file:

```
Donnie Darko/sci-fi/Gyllenhall, Jake
Raiders of the Lost Ark/action/Ford, Harrison
2001/sci-fi/??
Caddy Shack/comedy/Murray, Bill
Star Wars/sci-fi/Ford, Harrison
Lost in Translation/comedy/Murray, Bill
Patriot Games/action/Ford, Harrison
```

In our home-automation application, we want to create an instance of `DVDInfo` for each line of data we read in from the `dvdInfo.txt` file. For each instance, we will parse the line of data (remember `String.split()`?) and populate `DVDInfo`'s three instance variables. Finally, we want to put all of the `DVDInfo` instances into an ArrayList. Imagine that the `populateList()` method (below) does all of this. Here is a small piece of code from our application:

```
ArrayList<DVDInfo> dvdList = new ArrayList<DVDInfo>();
populateList();   // adds the file data to the ArrayList
System.out.println(dvdList);
```

You might get output like this:

```
[Donnie Darko sci-fi Gyllenhall, Jake
, Raiders of the Lost Ark action Ford, Harrison
, 2001 sci-fi ??
, Caddy Shack comedy Murray, Bill
, Star Wars sci-fi Ford, Harrison
, Lost in Translation comedy Murray, Bill
, Patriot Games action Ford, Harrison
]
```

(Note: We overrode `DVDInfo`'s `toString()` method, so when we invoked `println()` on the ArrayList it invoked `toString()` for each instance.)

Now that we've got a populated ArrayList, let's sort it:

```
Collections.sort(dvdlist);
```

Oops!, you get something like this:

```
TestDVD.java:13: cannot find symbol
symbol  : method sort(java.util.ArrayList<DVDInfo>)
location: class java.util.Collections
    Collections.sort(dvdlist);
```

What's going on here? We know that the Collections class has a `sort()` method, yet this error implies that Collections does NOT have a `sort()` method that can take a `dvdlist`. That means there must be something wrong with the argument we're passing (`dvdlist`).

If you've already figured out the problem, our guess is that you did it without the help of the obscure error message shown above…How the heck do you sort instances of `DVDInfo`? Why were we able to sort instances of String? When you look up `Collections.sort()` in the API your first reaction might be to panic. Hang tight, once again the generics section will help you read that weird looking method signature. If you read the description of the one-arg `sort()` method, you'll see that the `sort()` method takes a List argument, and that the objects in the List must implement an interface called Comparable. It turns out that String implements Comparable, and that's why we were able to sort a list of Strings using the `Collections.sort()` method.

The Comparable Interface

The `Comparable` interface is used by the `Collections.sort()` method and the `java.util.Arrays.sort()` method to sort `Lists` and arrays of objects, respectively. To implement `Comparable`, a class must implement a single method, `compareTo()`. Here's an invocation of `compareTo()`:

```
int x = thisObject.compareTo(anotherObject);
```

The `compareTo()` method returns an int with the following characteristics:

- negative — If `thisObject < anotherObject`
- zero — If `thisObject == anotherObject`
- positive — If `thisObject > anotherObject`

The `sort()` method uses `compareTo()` to determine how the List or object array should be sorted. Since you get to implement `compareTo()` for your own classes, you can use whatever weird criteria you prefer, to sort instances of your classes. Returning to our earlier example for class DVDInfo, we can take the easy way out and use the String class's implementation of `compareTo()`:

```
class DVDInfo implements Comparable<DVDInfo> {   // #1
  // existing code
  public int compareTo(DVDInfo d) {
    return title.compareTo(d.getTitle());         // #2
} }
```

In line 1 we declare that class `DVDInfo` implements `Comparable` in such a way that `DVDInfo` objects can be compared to other DVDInfo objects. In line 2 we implement `compareTo()` by comparing the two DVDInfo object's titles. Since we know that the titles are Strings, and that String implements Comparable, this is an easy way to sort our DVDInfo objects, by title. Before generics came along in Java 5, you would have had to implement Comparable something like this:

```
class DVDInfo implements Comparable {
  // existing code
  public int compareTo(Object o) {   // takes an Object rather
                                     // than a specific type
    DVDInfo d = (DVDInfo)o;
    return title.compareTo(d.getTitle());
} }
```

This is still legal, but you can see that it's both painful and risky, because you have to do a cast, and you need to verify that the cast will not fail before you try it.

exam Watch

It's important to remember that when you override `equals()` **you MUST take an argument of type** `Object`**, but that when you override** `compareTo()` **you should take an argument of the type you're sorting.**

Putting it all together, our DVDInfo class should now look like this:

```
class DVDInfo implements Comparable<DVDInfo> {
  String title;
  String genre;
  String leadActor;
  DVDInfo(String t, String g, String a) {
    title = t;  genre = g;  leadActor = a;
  }
  public String toString() {
    return title + " " + genre + " " + leadActor + "\n";
  }
  public int compareTo(DVDInfo d) {
    return title.compareTo(d.getTitle());
  }
  public String getTitle() {
    return title;
  }
  // other getters and setters
}
```

Now, when we invoke `Collections.sort(dvdList);` we get

```
[2001 sci-fi ??
, Caddy Shack comedy Murray, Bill
, Donnie Darko sci-fi Gyllenhall, Jake
, Lost in Translation comedy Murray, Bill
, Patriot Games action Ford, Harrison
, Raiders of the Lost Ark action Ford, Harrison
, Star Wars sci-fi Ford, Harrison
]
```

Hooray! Our ArrayList has been sorted by title. Of course, if we want our home automation system to really rock, we'll probably want to sort DVD collections in lots of different ways. Since we sorted our ArrayList by implementing the `compareTo()` method, we seem to be stuck. We can only implement `compareTo()` once in a class, so how do we go about sorting our classes in an order different than what we specify in our `compareTo()` method? Good question. As luck would have it, the answer is coming up next.

Sorting with Comparator

While you were looking up the `Collections.sort()` method you might have noticed that there is an overloaded version of `sort()` that takes both a List AND something called a *Comparator*. The Comparator interface gives you the capability to sort a given collection any number of different ways. The other handy thing about the Comparator interface is that you can use it to sort instances of any class—even classes you can't modify—unlike the Comparable interface, which forces you to change the class whose instances you want to sort. The Comparator interface is also very easy to implement, having only one method, `compare()`. Here's a small class that can be used to sort a List of DVDInfo instances, by `genre`.

```
import java.util.*;
class GenreSort implements Comparator<DVDInfo> {
  public int compare(DVDInfo one, DVDInfo two) {
    return one.getGenre().compareTo(two.getGenre());
  }
}
```

The `Comparator.compare()` method returns an `int` whose meaning is the same as the `Comparable.compareTo()` method's return value. In this case we're taking advantage of that by asking `compareTo()` to do the actual comparison work for us. Here's a test program that lets us test both our Comparable code and our new Comparator code:

```
import java.util.*;
import java.io.*;          // populateList() needs this
public class TestDVD {
  ArrayList<DVDInfo> dvdlist = new ArrayList<DVDInfo>();
  public static void main(String[] args) {
    new TestDVD().go();
  }
  public void go() {
    populateList();
    System.out.println(dvdlist);     // output as read from file
    Collections.sort(dvdlist);
    System.out.println(dvdlist);     // output sorted by title

    GenreSort gs = new GenreSort();
    Collections.sort(dvdlist, gs);
    System.out.println(dvdlist);     // output sorted by genre
  }
```

```
  public void populateList() {
     // read the file, create DVDInfo instances, and
     // populate the ArrayList dvdlist with these instances
  }
}
```

You've already seen the first two output lists, here's the third:

```
[Patriot Games action Ford, Harrison
, Raiders of the Lost Ark action Ford, Harrison
, Caddy Shack comedy Murray, Bill
, Lost in Translation comedy Murray, Bill
, 2001 sci-fi ??
, Donnie Darko sci-fi Gyllenhall, Jake
, Star Wars sci-fi Ford, Harrison
]
```

Because the Comparable and Comparator interfaces are so similar, expect the exam to try to confuse you. For instance you might be asked to implement the `compareTo()` method in the Comparator interface. Study Table 7-3 to burn in the differences between these two interfaces.

TABLE 7-3 Comparing Comparable to Comparator

java.lang.Comparable	java.util.Comparator
`int objOne.compareTo(objTwo)`	`int compare(objOne, objTwo)`
Returns negative if `objOne < objTwo` zero if `objOne == objTwo` positive if `objOne > objTwo`	Same as Comparable
You must modify the class whose instances you want to sort.	You build a class separate from the class whose instances you want to sort.
Only **one** sort sequence can be created	**Many** sort sequences can be created
Implemented frequently in the API by: String, Wrapper classes, Date, Calendar...	Meant to be implemented to sort instances of third-party classes.

Sorting with the Arrays Class

We've been using the java.util.Collections class to sort collections; now let's look at using the java.util.Arrays class to sort arrays. The good news is that sorting arrays of objects is just like sorting collections of objects. The `Arrays.sort()` method is overridden in the same way the `Collections.sort()` method is.

- `Arrays.sort(arrayToSort)`
- `Arrays.sort(arrayToSort, Comparator)`

In addition, the `Arrays.sort()` method is overloaded about a million times to provide a couple of sort methods for every type of primitive. The `Arrays.sort()` methods that sort primitives always sort based on natural order. Don't be fooled by an exam question that tries to sort a primitive array using a Comparator.

Finally, remember that the `sort()` methods for both the Collections class and the Arrays class are `static` methods, and that they alter the objects they are sorting, instead of returning a different sorted object.

exam Watch

We've talked a lot about sorting by natural order and using Comparators to sort. The last rule you'll need to burn in is that, whenever you want to sort an array or a collection, the elements inside must all be *mutually comparable.* ***In other words, if you have an*** `Object[]` ***and you put*** `Cat` ***and*** `Dog` ***objects into it, you won't be able to sort it. In general, objects of different types should be considered NOT mutually comparable, unless specifically stated otherwise.***

Searching Arrays and Collections

The Collections class and the Arrays class both provide methods that allow you to search for a specific element. When searching through collections or arrays, the following rules apply:

- Searches are performed using the `binarySearch()` method.
- Successful searches return the `int` index of the element being searched.
- Unsuccessful searches return an `int` index that represents the *insertion point*. The insertion point is the place in the collection/array where the element would be inserted to keep the collection/array properly sorted. Because posi-

tive return values and `0` indicate successful searches, the `binarySearch()` method uses negative numbers to indicate insertion points. Since `0` is a valid result for a successful search, the first available insertion point is `-1`. Therefore, the actual insertion point is represented as `(-(insertion point) -1)`. For instance, if the insertion point of a search is at element `2`, the actual insertion point returned will be `-3`.

- The collection/array being searched must be sorted before you can search it.
- If you attempt to search an array or collection that has not already been sorted, the results of the search will not be predictable.
- If the collection/array you want to search was sorted in natural order, it *must* be searched in natural order. (Usually this is accomplished by NOT sending a Comparator as an argument to the `binarySearch()` method.)
- If the collection/array you want to search was sorted using a Comparator, it *must* be searched using the same Comparator, which is passed as the second argument to the `binarySearch()` method. Remember that Comparators cannot be used when searching arrays of primitives.

Let's take a look at a code sample that exercises the `binarySearch()` method:

```
import java.util.*;
class SearchObjArray {
  public static void main(String [] args) {
    String [] sa = {"one", "two", "three", "four"};

    Arrays.sort(sa);                                          // #1
    for(String s : sa)
      System.out.print(s + " ");
    System.out.println("\none = "
                       + Arrays.binarySearch(sa,"one"));  // #2

    System.out.println("now reverse sort");
    ReSortComparator rs = new ReSortComparator();          // #3
    Arrays.sort(sa,rs);
    for(String s : sa)
      System.out.print(s + " ");
    System.out.println("\none = "
                       + Arrays.binarySearch(sa,"one"));  // #4
    System.out.println("one = "
                    + Arrays.binarySearch(sa,"one",rs));  // #5
  }
```

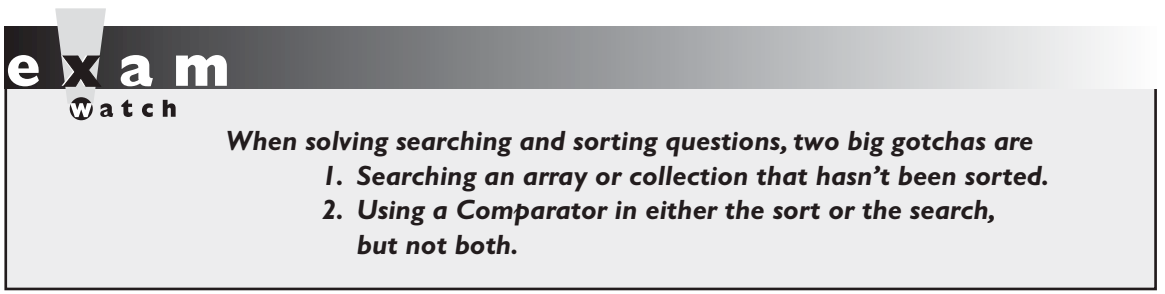
```
  static class ReSortComparator
                  implements Comparator<String> {          // #6
    public int compare(String a, String b) {
      return b.compareTo(a);                               // #7
    }
  }
}
```

which produces something like this:

```
four one three two
one = 1
now reverse sort
two three one four
one = -1
one = 2
```

Here's what happened:

Line 1 Sort the `sa` array, alphabetically (the natural order).
Line 2 Search for the location of element "one", which is 1.
Line 3 Make a Comparator instance. On the next line we re-sort the array using the Comparator.
Line 4 Attempt to search the array. We didn't pass the `binarySearch()` method the Comparator we used to sort the array, so we got an incorrect (undefined) answer.
Line 5 Search again, passing the Comparator to `binarySearch()`. This time we get the correct answer, 2
Line 6 We define the Comparator; it's okay for this to be an inner class.
Line 7 By switching the use of the arguments in the invocation of `compareTo()`, we get an inverted sort.



When solving searching and sorting questions, two big gotchas are

1. ***Searching an array or collection that hasn't been sorted.***
2. ***Using a Comparator in either the sort or the search, but not both.***

Converting Arrays to Lists to Arrays

There are a couple of methods that allow you to convert arrays to Lists, and Lists to arrays. The List and Set classes have `toArray()` methods, and the Arrays class has a method called `asList()`.

The `Arrays.asList()` method copies an array into a List. The API says, "Returns a fixed-size list backed by the specified array. (Changes to the returned list 'write through' to the array.)" When you use the `asList()` method, the array and the List become joined at the hip. When you update one of them, the other gets updated automatically. Let's take a look:

```
String[] sa = {"one", "two", "three", "four"};
List sList = Arrays.asList(sa);                 // make a List
System.out.println("size  " + sList.size());
System.out.println("idx2  " + sList.get(2));

sList.set(3,"six");                             // change List
sa[1] = "five";                                 // change array
for(String s : sa)
  System.out.print(s + " ");
System.out.println("\nsl[1] " + sList.get(1));
```

This produces

```
size 4
idx2 three
one five three six
sl[1] five
```

Notice that when we print the final state of the array and the List, they have both been updated with each other's changes. Wouldn't something like this behavior make a great exam question?

Now let's take a look at the `toArray()` method. There's nothing too fancy going on with the `toArray()` method; it comes in two flavors: one that returns a new Object array, and one that uses the array you send it as the destination array:

```
List<Integer> iL = new ArrayList<Integer>();
for(int x=0; x<3; x++)
  iL.add(x);
Object[] oa = iL.toArray();          // create an Object array
Integer[] ia2 = new Integer[3];
ia2 = iL.toArray(ia2);               // create an Integer array
```

Using Lists

Remember that Lists are usually used to keep things in some kind of order. You can use a LinkedList to create a first-in, first-out queue. You can use an ArrayList to keep track of what locations were visited, and in what order. Notice that in both of these examples it's perfectly reasonable to assume that duplicates might occur. In addition, Lists allow you to manually override the ordering of elements by adding or removing elements via the element's index. Before Java 5, and the enhanced `for` loop, the most common way to examine a List "element by element" was by the use of an Iterator. You'll still find Iterators in use in the Java code you encounter, and you might just find an Iterator or two on the exam. An Iterator is an object that's associated with a specific collection. It lets you loop through the collection step by step. The two Iterator methods you need to understand for the exam are

- **`boolean hasNext()`** Returns `true` if there is at least one more element in the collection being traversed. Invoking `hasNext()` does NOT move you to the next element of the collection.
- **`Object next()`** This method returns the next object in the collection, AND moves you forward to the element after the element just returned.

Let's look at a little code that uses a List and an Iterator:

```
import java.util.*;
class Dog {
  public String name;
  Dog(String n) { name = n; }
}
class ItTest {
  public static void main(String[] args) {
    List<Dog> d = new ArrayList<Dog>();
    Dog dog = new Dog("aiko");
    d.add(dog);
    d.add(new Dog("clover"));
    d.add(new Dog("magnolia"));
    Iterator<Dog> i3 = d.iterator();  // make an iterator
    while (i3.hasNext()) {
      Dog d2 = i3.next();             // cast not required
      System.out.println(d2.name);
    }
    System.out.println("size " + d.size());
    System.out.println("get1 " + d.get(1).name);
    System.out.println("aiko " + d.indexOf(dog));
```

```
    d.remove(2);
    Object[] oa = d.toArray();
    for(Object o : oa) {
      Dog d2 = (Dog)o;
      System.out.println("oa " + d2.name);
    }
  }
}
```

This produces

```
aiko
clover
magnolia
size 3
get1 clover
aiko 0
oa aiko
oa clover
```

First off, we used generics syntax to create the Iterator (an Iterator of type Dog). Because of this, when we used the `next()` method, we didn't have to cast the Object returned by `next()` to a Dog. We could have declared the Iterator like this:

```
Iterator i3 = d.iterator();  // make an iterator
```

But then we would have had to cast the returned value:

```
Dog d2 = (Dog)i3.next();
```

The rest of the code demonstrates using the `size()`, `get()`, `indexOf()`, and `toArray()` methods. There shouldn't be any surprises with these methods. In a few pages Table 7-5 will list all of the List, Set, and Map methods you should be familiar with for the exam. As a last warning, remember that List is an interface!

Using Sets

Remember that Sets are used when you don't want any duplicates in your collection. If you attempt to add an element to a set that already exists in the set, the duplicate element will not be added, and the `add()` method will return `false`. Remember, HashSets tend to be very fast because, as we discussed earlier, they use hashcodes.

You can also create a TreeSet, which is a Set whose elements are sorted. You must use caution when using a TreeSet (we're about to explain why):

```
import java.util.*;
class SetTest {
  public static void main(String[] args) {
    boolean[] ba = new boolean[5];
    // insert code here

    ba[0] = s.add("a");
    ba[1] = s.add(new Integer(42));
    ba[2] = s.add("b");
    ba[3] = s.add("a");
    ba[4] = s.add(new Object());
    for(int x=0; x<ba.length; x++)
      System.out.print(ba[x] + " ");
    System.out.println("\n");
    for(Object o : s)
      System.out.print(o + " ");
  }
}
```

If you insert the following line of code you'll get output something like this:

```
Set s = new HashSet();       // insert this code

true true true false true
a java.lang.Object@e09713 42 b
```

It's important to know that the order of objects printed in the second `for` loop is not predictable: HashSets do not guarantee any ordering. Also, notice that the fourth invocation of `add()` failed, because it attempted to insert a duplicate entry (a String with the value `a`) into the Set.

If you insert this line of code you'll get something like this:

```
Set s = new TreeSet();        // insert this code

Exception in thread "main" java.lang.ClassCastException: java.
lang.String
        at java.lang.Integer.compareTo(Integer.java:35)
        at java.util.TreeMap.compare(TreeMap.java:1093)
```

```
        at java.util.TreeMap.put(TreeMap.java:465)
        at java.util.TreeSet.add(TreeSet.java:210)
```

The issue is that whenever you want a collection to be sorted, its elements must be mutually comparable. Remember that unless otherwise specified, objects of different types are not mutually comparable.

Using Maps

Remember that when you use a class that implements Map, any classes that you use as a part of the keys for that map must override the `hashCode()` and `equals()` methods. (Well, you only have to override them if you're interested in retrieving stuff from your Map. Seriously, it's legal to use a class that doesn't override `equals()` and `hashCode()` as a key in a Map; your code will compile and run, you just won't find your stuff.) Here's some crude code demonstrating the use of a HashMap:

```
import java.util.*;
class Dog {
  public Dog(String n) { name = n; }
  public String name;
  public boolean equals(Object o) {
    if((o instanceof Dog) &&
       (((Dog)o).name == name)) {
      return true;
    } else {
      return false;
    }
  }
  public int hashCode() {return name.length(); }
}
class Cat { }

enum Pets {DOG, CAT, HORSE }

class MapTest {
  public static void main(String[] args) {
    Map<Object, Object> m = new HashMap<Object, Object>();

    m.put("k1", new Dog("aiko"));   // add some key/value pairs
    m.put("k2", Pets.DOG);
    m.put(Pets.CAT, "CAT key");
    Dog d1 = new Dog("clover");     // let's keep this reference
```

```
        m.put(d1, "Dog key");
        m.put(new Cat(), "Cat key");

        System.out.println(m.get("k1"));                    // #1
        String k2 = "k2";
        System.out.println(m.get(k2));                      // #2
        Pets p = Pets.CAT;
        System.out.println(m.get(p));                       // #3
        System.out.println(m.get(d1));                      // #4
        System.out.println(m.get(new Cat()));               // #5
        System.out.println(m.size());                       // #6
    }
}
```

which produces something like this:

```
Dog@1c
DOG
CAT key
Dog key
null
5
```

Let's review the output. The first value retrieved is a Dog object (your value will vary). The second value retrieved is an enum value (DOG). The third value retrieved is a String; note that the key was an enum value. Pop quiz: What's the implication of the fact that we were able to successfully use an enum as a key?

The implication of this is that enums override `equals()` and `hashCode()`. And, if you look at the java.lang.Enum class in the API, you will see that, in fact, these methods have been overridden.

The fourth output is a String. The important point about this output is that the key used to retrieve the String was made of a Dog object. The fifth output is `null`. The important point here is that the `get()` method failed to find the Cat object that was inserted earlier. (The last line of output confirms that indeed, 5 key/value pairs exist in the Map.) Why didn't we find the `Cat key` String? Why did it work to use an instance of Dog as a key, when using an instance of Cat as a key failed?

It's easy to see that Dog overrode `equals()` and `hashCode()` while Cat didn't.

Let's take a quick look at hashcodes. We used an incredibly simplistic hashcode formula in the Dog class—the hashcode of a Dog object is the length of the instance's `name`. So in this example the hashcode = `6`. Let's compare the following two `hashCode()` methods:

```
public int hashCode() {return name.length(); }   // #1
public int hashCode() {return 4; }               // #2
```

Time for another pop quiz: Are the preceding two hashcodes legal? Will they successfully retrieve objects from a Map? Which will be faster?

The answer to the first two questions is Yes and Yes. Neither of these hashcodes will be very efficient (in fact they would both be incredibly inefficient), but they are both legal, and they will both work. The answer to the last question is that the first hashcode will be a little bit faster than the second hashcode. In general, the more *unique* hashcodes a formula creates, the faster the retrieval will be. The first hashcode formula will generate a different code for each `name` length (for instance the name `Robert` will generate one hashcode and the name `Benchley` will generate a different hashcode). The second hashcode formula will always produce the same result, `4`, so it will be slower than the first.

Our last Map topic is what happens when an object used as a key has its values changed? If we add two lines of code to the end of the earlier `MapTest.main()`,

```
d1.name = "magnolia";
System.out.println(m.get(d1));
```

we get something like this:

```
Dog@4
DOG
CAT key
Dog key
null
5
null
```

The Dog that was previously found now cannot be found. Because the `Dog.name` variable is used to create the hashcode, changing the `name` changed the value of the hashcode. As a final quiz for hashcodes, determine the output for the following lines of code if they're added to the end of `MapTest.main()`:

```
d1.name = "magnolia";
System.out.println(m.get(d1));                  // #1
d1.name = "clover";
System.out.println(m.get(new Dog("clover")));   // #2
d1.name = "arthur";
System.out.println(m.get(new Dog("clover")));   // #3
```

Remember that the hashcode is equal to the length of the `name` variable. When you study a problem like this, it can be useful to think of the two stages of retrieval:

1. Use the `hashCode()` method to find the correct bucket
2. Use the `equals()` method to find the object in the bucket

In the first call to `get()`, the hashcode is 8 (`magnolia`) and it should be 6 (`clover`), so the retrieval fails at step 1 and we get `null`. In the second call to `get()`, the hashcodes are both 6, so step 1 succeeds. Once in the correct bucket (the "length of name = 6" bucket), the `equals()` method is invoked, and since Dog's `equals()` method compares names, `equals()` succeeds, and the output is `Dog key`. In the third invocation of `get()`, the hashcode test succeeds, but the `equals()` test fails because `arthur` is NOT equal to `clover`.

Navigating (Searching) TreeSets and TreeMaps

We've talked about searching lists and arrays. Let's turn our attention to searching TreeSets and TreeMaps. Java 6 introduced (among others) two new interfaces: `java.util.NavigableSet` and `java.util.NavigableMap`. For the purposes of the exam, you're interested in how TreeSet and TreeMap implement these interfaces.

Imagine that the Santa Cruz–Monterey ferry has an irregular schedule. Let's say that we have the daily Santa Cruz departure times stored, in military time, in a TreeSet. Let's look at some code that determines two things:

1. The last ferry that leaves before 4 PM (1600 hours)
2. The first ferry that leaves after 8 PM (2000 hours)

```
import java.util.*;
public class Ferry {
  public static void main(String[] args) {
    TreeSet<Integer> times = new TreeSet<Integer>();
    times.add(1205);   // add some departure times
    times.add(1505);
    times.add(1545);
    times.add(1830);
    times.add(2010);
    times.add(2100);
```

```
    // Java 5 version

    TreeSet<Integer> subset = new TreeSet<Integer>();
    subset = (TreeSet)times.headSet(1600);
    System.out.println("J5 - last before 4pm is: " + subset.last());

    TreeSet<Integer> sub2 = new TreeSet<Integer>();
    sub2 = (TreeSet)times.tailSet(2000);
    System.out.println("J5 - first after 8pm is: " + sub2.first());

    // Java 6 version using the new lower() and higher() methods

    System.out.println("J6 - last before 4pm is: " + times.lower(1600));
    System.out.println("J6 - first after 8pm is: " + times.higher(2000));
  }
}
```

This should produce the following:

```
J5 - last before 4pm is: 1545
J5 - first after 8pm is: 2010
J6 - last before 4pm is: 1545
J6 - first after 8pm is: 2010
```

As you can see in the preceding code, before the addition of the NavigableSet interface, zeroing in on an arbitrary spot in a Set—using the methods available in Java 5—was a compute-expensive and clunky proposition. On the other hand, using the new Java 6 methods `lower()` and `higher()`, the code becomes a lot cleaner.

For the purpose of the exam, the NavigableSet methods related to this type of navigation are `lower()`, `floor()`, `higher()`, `ceiling()`, and the mostly parallel NavigableMap methods are `lowerKey()`, `floorKey()`, `ceilingKey()`, and `higherKey()`. The difference between `lower()` and `floor()` is that `lower()` returns the element less than the given element, and `floor()` returns the element less than *or equal to* the given element. Similarly, `higher()` returns the element greater than the given element, and `ceiling()` returns the element greater than *or equal to* the given element. Table 7-4 summarizes the methods you should know for the exam.

Other Navigation Methods

In addition to the methods we just discussed there are a few more new Java 6 methods that could be considered "navigation" methods. (Okay, it's a little stretch to call these "navigation" methods, but just play along.)

Polling

Although the idea of polling isn't new to Java 6 (as you'll see in a minute, `PriorityQueue` had a `poll()` method before Java 6), it *is* new to TreeSet and TreeMap. The idea of polling is that we want *both* to retrieve *and* remove an element from either the beginning or the end of a collection. In the case of TreeSet, `pollFirst()` returns and removes the first entry in the set, and `pollLast()` returns and removes the last. Similarly, TreeMap now provides `pollFirstEntry()` and `pollLastEntry()` to retrieve and remove key-value pairs.

Descending Order

Also new to Java 6 for TreeSet and TreeMap are methods that return a collection in the reverse order of the collection on which the method was invoked. The important methods for the exam are TreeSet.descendingSet() and TreeMap .descendingMap().

Table 7-4 summarizes the "navigation" methods you'll need to know for the exam.

TABLE 7-4 Important "Navigation" Related Methods

Method	Description
`TreeSet.ceiling(e)`	Returns the lowest element >= e
`TreeMap.ceilingKey(key)`	Returns the lowest key >= key
`TreeSet.higher(e)`	Returns the lowest element > e
`TreeMap.higherKey(key)`	Returns the lowest key > key
`TreeSet.floor(e)`	Returns the highest element <= e
`TreeMap.floorKey(key)`	Returns the highest key <= key
`TreeSet.lower(e)`	Returns the highest element < e
`TreeMap.lowerKey(key)`	Returns the highest key < key
`TreeSet.pollFirst()`	Returns and removes the first entry
`TreeMap.pollFirstEntry()`	Returns and removes the first key-value pair
`TreeSet.pollLast()`	Returns and removes the last entry
`TreeMap.pollLastEntry()`	Returns and removes the last key-value pair
`TreeSet.descendingSet()`	Returns a NavigableSet in reverse order
`TreeMap.descendingMap()`	Returns a NavigableMap in reverse order

Backed Collections

Some of the classes in the java.util package support the concept of "backed collections". We'll use a little code to help explain the idea:

```
TreeMap<String, String> map = new TreeMap<String, String>();
map.put("a", "ant"); map.put("d", "dog"); map.put("h", "horse");

SortedMap<String, String> submap;
submap = map.subMap("b", "g");              // #1 create a backed collection

System.out.println(map + " " + submap);     // #2 show contents

map.put("b", "bat");                        // #3 add to original
submap.put("f", "fish");                    // #4 add to copy

map.put("r", "raccoon");                    // #5 add to original - out of range
// submap.put("p", "pig");                  // #6 add to copy - out of range

System.out.println(map + " " + submap);     // #7 show final contents
```

This should produce something like this:

```
{a=ant, d=dog, h=horse} {d=dog}
{a=ant, b=bat, d=dog, f=fish, h=horse, r=raccoon} {b=bat, d=dog, f=fish}
```

The important method in this code is the `TreeMap.subMap()` method. It's easy to guess (and it's correct), that the `subMap()` method is making a copy of a portion of the TreeMap named `map`. The first line of output verifies the conclusions we've just drawn.

What happens next is powerful and a little bit unexpected (now we're getting to why they're called *backed* collections). When we add key-value pairs to either the original TreeMap or the partial-copy SortedMap, the new entries were automatically added to the other collection—sometimes. When `submap` was created, we provided a value range for the new collection. This range defines not only what should be included when the partial copy is created, but also defines the range of values that can be added to the copy. As we can verify by looking at the second line of output, we can add new entries to either collection within the range of the copy, and the new entries will show up in both collections. In addition, we can add a new entry to the original collection, even if it's outside the range of the copy. In this case, the new entry will show up only in the original—it won't be added to the copy because it's outside the copy's range. Notice that we commented out line #6. If you attempt to add an out-of-range entry to the copied collection an exception will be thrown.

For the exam, you'll need to understand the basics just explained, plus a few more details about three methods from TreeSet—(`headSet()`, `subSet()`, and `tailSet()`), and three methods from TreeMap (`headMap()`, `subMap()`, and `tailMap()`). As with the navigation-oriented methods we just discussed, we can see a lot of parallels between the TreeSet and the TreeMap methods. The `headSet()` / `headMap()` methods create a subset that starts at the beginning of the original collection and ends at the point specified by the method's argument. The `tailSet()` / `tailMap()` methods create a subset that starts at the point specified by the method's argument and goes to the end of the original collection. Finally, the `subSet()` / `subMap()` methods allow you to specify both the start and end points for the subset collection you're creating.

As you might expect, the question of whether the subsetted collection's end points are inclusive or exclusive is a little tricky. The good news is that for the exam you have to remember only that when these methods are invoked with endpoint *and* boolean arguments, the boolean always means "is inclusive?". A little more good news is that all you have to know for the exam is that unless specifically indicated by a boolean argument, a subset's starting point will always be inclusive. Finally, you'll notice when you study the API that all of the methods we've been discussing here have an overloaded version that's new to Java 6. The older methods return either a SortedSet or a SortedMap, the new Java 6 methods return either a NavigableSet or a NavigableMap. Table 7-5 summarizes these methods.

TABLE 7-5 Important "Backed Collection" Methods for TreeSet and TreeMap

Method	Description
`headSet(e, b*)`	Returns a subset ending at element `e` and *exclusive* of `e`
`headMap(k, b*)`	Returns a submap ending at key `k` and *exclusive* of key `k`
`tailSet(e, b*)`	Returns a subset starting at and *inclusive* of element `e`
`tailMap(k, b*)`	Returns a submap starting at and *inclusive* of key `k`
`subSet(s, b*, e, b*)`	Returns a subset starting at element `s` and ending just before element `e`
`subMap(s, b*, e, b*)`	Returns a submap starting at key `s` and ending just before key `s`

* NOTE: These boolean arguments are optional. If they exist it's a Java 6 method that lets you specify whether the endpoint is exclusive, and these methods return a NavigableXxx. If the boolean argument(s) don't exist, the method returns either a SortedSet or a SortedMap.

exam Watch

Let's say that you've created a backed collection using either a `tailXxx()` ***or*** `subXxx()` ***method. Typically in these cases the original and copy collections have different "first" elements. For the exam it's important that you remember that the*** `pollFirstXxx()` ***methods will always remove the first entry from the collection on which they're invoked, but they will remove an element from the other collection only if it has same value. So it's most likely that invoking*** `pollFirstXxx()` ***on the copy will remove an entry from both collections, but invoking*** `pollFirstXxx()` ***on the original will remove only the entry from the original.***

Using the PriorityQueue Class

The last collection class you'll need to understand for the exam is the PriorityQueue. Unlike basic queue structures that are first-in, first-out by default, a PriorityQueue orders its elements using a user-defined priority. The priority can be as simple as natural ordering (in which, for instance, an entry of `1` would be a higher priority than an entry of `2`). In addition, a PriorityQueue can be ordered using a Comparator, which lets you define any ordering you want. Queues have a few methods not found in other collection interfaces: `peek()`, `poll()`, and `offer()`.

```
import java.util.*;
class PQ {
  static class PQsort
          implements Comparator<Integer> {  // inverse sort
    public int compare(Integer one, Integer two) {
      return two - one;                     // unboxing
    }
  }
  public static void main(String[] args) {
    int[] ia = {1,5,3,7,6,9,8 };            // unordered data
    PriorityQueue<Integer> pq1 =
      new PriorityQueue<Integer>();         // use natural order

    for(int x : ia)                         // load queue
```

```
      pq1.offer(x);
    for(int x : ia)                          // review queue
      System.out.print(pq1.poll() + " ");
    System.out.println("");

    PQsort pqs = new PQsort();               // get a Comparator
    PriorityQueue<Integer> pq2 =
      new PriorityQueue<Integer>(10,pqs);    // use Comparator

    for(int x : ia)                          // load queue
      pq2.offer(x);
    System.out.println("size " + pq2.size());
    System.out.println("peek " + pq2.peek());
    System.out.println("size " + pq2.size());
    System.out.println("poll " + pq2.poll());
    System.out.println("size " + pq2.size());
    for(int x : ia)                          // review queue
      System.out.print(pq2.poll() + " ");
  }
}
```

This code produces something like this:

```
1 3 5 6 7 8 9
size 7
peek 9
size 7
poll 9
size 6
8 7 6 5 3 1 null
```

Let's look at this in detail. The first `for` loop iterates through the `ia` array, and uses the `offer()` method to add elements to the PriorityQueue named `pq1`. The second `for` loop iterates through `pq1` using the `poll()` method, which returns the highest priority entry in `pq1` AND removes the entry from the queue. Notice that the elements are returned in priority order (in this case, natural order). Next, we create a Comparator—in this case, a Comparator that orders elements in the opposite of natural order. We use this Comparator to build a second PriorityQueue, `pq2`, and we load it with the same array we used earlier. Finally, we check the size of `pq2` before and after calls to `peek()` and `poll()`. This confirms that `peek()` returns the highest priority element in the queue without removing it, and `poll()` returns the highest priority element, AND removes it from the queue. Finally, we review the remaining elements in the queue.

Method Overview for Arrays and Collections

For these two classes, we've already covered the trickier methods you might encounter on the exam. Table 7-6 lists a summary of the methods you should be aware of. (Note: The `T[]` syntax will be explained later in this chapter; for now, think of it as meaning "any array that's NOT an array of primitives.")

TABLE 7-6 Key Methods in Arrays and Collections

Key Methods in java.util.Arrays	Descriptions
`static List asList(T[])`	Convert an array to a List (and bind them).
`static int binarySearch(Object[], key)` `static int binarySearch(primitive[], key)`	Search a sorted array for a given value, return an index or insertion point.
`static int binarySearch(T[], key, Comparator)`	Search a Comparator-sorted array for a value.
`static boolean equals(Object[], Object[])` `static boolean equals(primitive[], primitive[])`	Compare two arrays to determine if their contents are equal.
`public static void sort(Object[ ] )` `public static void sort(primitive[ ] )`	Sort the elements of an array by natural order.
`public static void sort(T[], Comparator)`	Sort the elements of an array using a Comparator.
`public static String toString(Object[])` `public static String toString(primitive[])`	Create a String containing the contents of an array.
Key Methods in java.util.Collections	**Descriptions**
`static int binarySearch(List, key)` `static int binarySearch(List, key, Comparator)`	Search a "sorted" List for a given value, return an index or insertion point.
`static void reverse(List)`	Reverse the order of elements in a List.
`static Comparator reverseOrder()` `static Comparator reverseOrder(Comparator)`	Return a Comparator that sorts the reverse of the collection's current sort sequence.
`static void sort(List)` `static void sort(List, Comparator)`	Sort a List either by natural order or by a Comparator.

Method Overview for List, Set, Map, and Queue

For these four interfaces, we've already covered the trickier methods you might encounter on the exam. Table 7-7 lists a summary of the List, Set, and Map methods you should be aware of, **but don't forget the new "Navigable" methods floor, lower, ceiling, and higher that we discussed a few pages back**.

TABLE 7-7 Key Methods in List, Set, and Map

Key Interface Methods	List	Set	Map	Descriptions
`boolean add(element)` `boolean add(index, element)`	X X	X		Add an element. For Lists, optionally add the element at an index point.

`boolean contains(object)`
`boolean containsKey(object key)`
`boolean containsValue(object value)`
	X	X	
			X
			X
				Search a collection for an object (or, optionally for Maps a key), return the result as a `boolean`.

`object get(index)` `object get(key)`	X		 X	Get an object from a collection, via an index or a key.
`int indexOf(object)`	X			Get the location of an object in a List.
`Iterator iterator()`	X	X		Get an Iterator for a List or a Set.
`Set keySet()`			X	Return a Set containing a Map's keys.
`put(key, value)`			X	Add a key/value pair to a Map.

`remove(index)`
`remove(object)`
`remove(key)`
	X
	X
		
		X
			
			X
				Remove an element via an index, or via the element's value, or via a key.

`int size()`	X	X	X	Return the number of elements in a collection.
`Object[] toArray()` `T[] toArray(T[])`	X	X		Return an array containing the elements of the collection.

For the exam, the PriorityQueue methods that are important to understand are `offer()` (which is similar to `add()`), `peek()` (which retrieves the element at the head of the queue, but doesn't delete it), and `poll()` (which retrieves the head element and removes it from the queue).

exam Watch

It's important to know some of the details of natural ordering. The following code will help you understand the relative positions of uppercase characters, lowercase characters, and spaces in a natural ordering:

```
String[] sa = {">ff<", "> f<", ">f <", ">FF<" }; // ordered?
PriorityQueue<String> pq3 = new PriorityQueue<String>();
for(String s : sa)
  pq3.offer(s);
for(String s : sa)
  System.out.print(pq3.poll() + " ");
```

This produces

```
> f< >FF< >f < >ff<
```

If you remember that spaces sort before characters and that uppercase letters sort before lowercase characters, you should be good to go for the exam.

CERTIFICATION OBJECTIVE

Generic Types (Objectives 6.3 and 6.4)

6.3 Write code that uses the generic versions of the Collections API, in particular the Set, List, and Map interfaces and implementation classes. Recognize the limitations of the non-generic Collections API and how to refactor code to use the generic versions.

6.4 Develop code that makes proper use of type parameters in class/interface declarations, instance variables, method arguments, and return types; and write generic methods or methods that make use of wildcard types and understand the similarities and differences between these two approaches. Write code that uses the NavigableSet and NavigableMap interfaces.

Arrays in Java have always been type safe—an array declared as type String (`String []`) can't accept Integers (or `int`s), Dogs, or anything other than Strings. But remember that before Java 5 there was no syntax for declaring a type safe collection. To make an ArrayList of Strings, you said,

```
ArrayList myList = new ArrayList();
```

or, the polymorphic equivalent

```
List myList = new ArrayList();
```

There was no syntax that let you specify that `myList` will take Strings and only Strings. And with no way to specify a type for the ArrayList, the compiler couldn't enforce that you put only things of the specified type into the list. As of Java 5, we can use generics, and while they aren't only for making type safe collections, that's just about all most developers use generics for. So, while generics aren't just for collections, think of collections as the overwhelming reason and motivation for adding generics to the language.

And it was not an easy decision, nor has it been an entirely welcome addition. Because along with all the nice happy type safety, generics come with a lot of baggage—most of which you'll never see or care about, but there are some gotchas that come up surprisingly quickly. We'll cover the ones most likely to show up in your own code, and those are also the issues that you'll need to know for the exam.

The biggest challenge for Sun in adding generics to the language (and the main reason it took them so long) was how to deal with legacy code built without generics. Sun's Java engineers obviously didn't want to break everyone's existing Java code, so they had to find a way for Java classes with both type safe (generic) and non-type safe (non-generic/pre-Java 5) collections to still work together. Their solution isn't the friendliest, but it does let you use older non-generic code, as well as use generic code that plays with non-generic code. But notice we said "plays," and not "plays WELL."

While you can integrate Java 5 and Java 6 generic code with legacy non-generic code, the consequences can be disastrous, and unfortunately, most of the disasters happen at runtime, not compile time. Fortunately, though, most compilers will generate warnings to tell you when you're using unsafe (meaning non-generic) collections.

The Java 5 exam covers both pre-Java 5 (non-generic) and Java 5 style collections, and you'll see questions that expect you to understand the tricky

problems that can come from mixing non-generic and generic code together. And like some of the other topics in this book, you could fill an entire book if you really wanted to cover every detail about generics, but the exam (and this book) covers more than most developers will ever need to use.

The Legacy Way to Do Collections

Here's a review of a pre-Java 5 ArrayList intended to hold Strings. (We say "intended" because that's about all you had—good intentions—to make sure that the ArrayList would hold only Strings).

```
List myList = new ArrayList();  // can't declare a type

myList.add("Fred");             // OK, it will hold Strings

myList.add(new Dog());          // and it will hold Dogs too

myList.add(new Integer(42));    // and Integers...
```

A non-generic collection can hold any kind of object! A non-generic collection is quite happy to hold anything that is NOT a primitive.

This meant it was entirely up to the programmer to be…careful. Having no way to guarantee collection type wasn't very programmer-friendly for such a strongly typed language. We're so used to the compiler stopping us from, say, assigning an `int` to a `boolean` reference or a String to a Dog reference, but with collections, it was, "Come on in! The door is always open! All objects are welcome here any time!"

And since a collection could hold anything, the methods that get objects out of the collection could have only one kind of return type—java.lang.Object. That meant that getting a String back out of our only-Strings-intended list required a cast:

```
String s = (String) myList.get(0);
```

And since you couldn't guarantee that what was coming out really was a String (since you were allowed to put anything in the list), the cast could fail at runtime.

So, generics takes care of both ends (the putting in and getting out) by enforcing the type of your collections. Let's update the String list:

```
List<String> myList = new ArrayList<String>();
myList.add("Fred");        // OK, it will hold Strings
myList.add(new Dog());     // compiler error!!
```

Perfect. That's exactly what we want. By using generics syntax—which means putting the type in angle brackets <String>, we're telling the compiler that this collection can hold only String objects. The type in angle brackets is referred to as either the "parameterized type," "type parameter," or of course just old-fashioned "type." In this chapter, we'll refer to it both new ways.

So, now that what you put IN is guaranteed, you can also guarantee what comes OUT, and that means you can get rid of the cast when you get something from the collection. Instead of

```
String s = (String)myList.get(0);  // pre-generics, when a
                                   // String wasn't guaranteed
```

we can now just say

```
String s = myList.get(0);
```

The compiler already knows that `myList` contains only things that can be assigned to a String reference, so now there's no need for a cast. So far, it seems pretty simple. And with the new `for` loop, you can of course iterate over the guaranteed-to-be-String list:

```
for (String s : myList) {
   int x = s.length();
   // no need for a cast before calling a String method! The
   // compiler already knew "s" was a String coming from MyList
}
```

And of course you can declare a type parameter for a method argument, which then makes the argument a type safe reference:

```
void takeListOfStrings(List<String> strings) {
    strings.add("foo");  // no problem adding a String
}
```

The method above would NOT compile if we changed it to

```
void takeListOfStrings(List<String> strings) {
   strings.add(new Integer(42)); // NO!! strings is type safe
}
```

Return types can obviously be declared type safe as well:

```
public List<Dog> getDogList() {
    List<Dog> dogs = new ArrayList<Dog>();
    // more code to insert dogs
    return dogs;
}
```

The compiler will stop you from returning anything not compatible with a List<Dog> (although what is and is not compatible is going to get very interesting in a minute). And since the compiler guarantees that only a type safe Dog List is returned, those calling the method won't need a cast to take Dogs from the List:

```
Dog d = getDogList().get(0);  // we KNOW a Dog is coming out
```

With pre-Java 5, non-generic code, the `getDogList()` method would be

```
public List getDogList() {
    List dogs = new ArrayList();
     // code to add only Dogs... fingers crossed...
    return dogs; // a List of ANYTHING will work here
}
```

and the caller would need a cast:

```
Dog d = (Dog) getDogList().get(0);
```

(The cast in this example applies to what comes from the List's `get()` method; we aren't casting what is returned from the `getDogList()` method, which is a List.)

But what about the benefit of a completely heterogeneous collection? In other words, what if you liked the fact that before generics you could make an ArrayList that could hold any kind of object?

```
List myList = new ArrayList();  // old-style, non-generic
```

is almost identical to

```
List<Object> myList = new
                  ArrayList<Object>(); // holds ANY object type
```

Declaring a List with a type parameter of <Object> makes a collection that works in almost the same way as the original pre-Java 5, non-generic collection—you can put ANY Object type into the collection. You'll see a little later that non-generic collections and collections of type <Object> aren't entirely the same, but most of the time the differences do not matter.

Oh, if only this were the end of the story...but there are still a few tricky issues with methods arguments, polymorphism, and integrating generic and non-generic code, so we're just getting warmed up here.

Generics and Legacy Code

The easiest generics thing you'll need to know for the exam is how to update non-generic code to make it generic. You just add a type in angle brackets (<>) immediately following the collection type in BOTH the variable declaration and the constructor call, including any place you declare a variable (so that means arguments and return types too). A pre-Java 5 List meant to hold only Integers:

```
List myList = new ArrayList();
```

becomes

```
List<Integer> myList = new ArrayList<Integer>();
```

and a list meant to hold only Strings goes from

```
public List changeStrings(ArrayList s) { }
```

to this:

```
public List<String> changeStrings(ArrayList<String> s) { }
```

Easy. And if there's code that used the earlier non-generic version and performed a cast to get things out, that won't break anyone's code:

```
Integer i = (Integer) list.get(0); // cast no longer needed,
                                   // but it won't hurt
```

Mixing Generic and Non-generic Collections

Now here's where it starts to get interesting…imagine we have an ArrayList, of type Integer, and we're passing it into a method from a class whose source code we don't have access to. Will this work?

```
// a Java 5 class using a generic collection
import java.util.*;
public class TestLegacy {
    public static void main(String[] args) {
        List<Integer> myList = new ArrayList<Integer>();
                                        // type safe collection
        myList.add(4);
        myList.add(6);
        Adder adder = new Adder();
        int total = adder.addAll(myList);
                              // pass it to an untyped argument
        System.out.println(total);
    }
}
```

The older, non-generics class we want to use:

```
import java.util.*;
class Adder {
   int addAll(List list) {
      // method with a non-generic List argument,
      // but assumes (with no guarantee) that it will be Integers
      Iterator it = list.iterator();
      int total = 0;
      while (it.hasNext()) {
        int i = ((Integer)it.next()).intValue();
        total += i;
      }
      return total;
   }
}
```

Yes, this works just fine. You can mix correct generic code with older non-generic code, and everyone is happy.

In the previous example, the `addAll()` legacy method assumed (trusted? hoped?) that the list passed in was indeed restricted to Integers, even though when the code was written, there was no guarantee. It was up to the programmers to be careful.

Since the `addAll()` method wasn't doing anything except getting the Integer (using a cast) from the list and accessing its value, there were no problems. In that example, there was no risk to the caller's code, but the legacy method might have blown up if the list passed in contained anything but Integers (which would cause a ClassCastException).

But now imagine that you call a legacy method that doesn't just *read* a value but *adds* something to the ArrayList? Will this work?

```
import java.util.*;
public class TestBadLegacy {
    public static void main(String[] args) {
        List<Integer> myList = new ArrayList<Integer>();
        myList.add(4);
        myList.add(6);
        Inserter in  = new Inserter();
        in.insert(myList);  // pass List<Integer> to legacy code
      }
}
class Inserter {
     // method with a non-generic List argument
     void insert(List list) {
       list.add(new Integer(42)); // adds to the incoming list
    }
}
```

Sure, this code works. It compiles, and it runs. The `insert()` method puts an Integer into the list that was originally typed as <Integer>, so no problem.

But…what if we modify the `insert()` method like this:

```
void insert(List list) {
   list.add(new String("42"));  // put a String in the list
                                // passed in
}
```

Will that work? Yes, sadly, it does! It both compiles and runs. No runtime exception. Yet, someone just stuffed a String into a *supposedly* type safe ArrayList of type <Integer>. How can that be?

Remember, the older legacy code was allowed to put anything at all (except primitives) into a collection. And in order to support legacy code, Java 5 and Java 6 allows your newer type safe code to make use of older code (the last thing Sun wanted to do was ask several million Java developers to modify all their existing code).

So, the Java 5 or Java 6 compiler is *forced* into letting you compile your new type safe code even though your code invokes a method of an older class that takes a non-type safe argument and does who knows what with it.

However, just because the Java 5 compiler allows this code to compile doesn't mean it has to be HAPPY about it. In fact the compiler will warn you that you're taking a big, big risk sending your nice protected ArrayList<Integer> into a dangerous method that can have its way with your list and put in Floats, Strings, or even Dogs.

When you called the `addAll()` method in the earlier example, it didn't insert anything to the list (it simply added up the values within the collection), so there was no risk to the caller that his list would be modified in some horrible way. It compiled and ran just fine. But in the second version, with the legacy `insert()` method that adds a String, the compiler generated a warning:

```
javac TestBadLegacy.java
Note: TestBadLegacy.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Remember that *compiler warnings are NOT considered a compiler failure*. The compiler generated a perfectly valid class file from the compilation, but it was kind enough to tell you by saying, in so many words, "I seriously hope you know what you are doing because this old code has NO respect (or even knowledge) of your <Integer> typing, and can do whatever the heck it wants to your precious ArrayList<Integer>."

exam Watch

Be sure you know the difference between "compilation fails" and "compiles without error" and "compiles without warnings" and "compiles with warnings." In most questions on the exam, you care only about compiles vs. compilation fails—compiler warnings don't matter for most of the exam. But when you are using generics, and mixing both typed and untyped code, warnings matter.

Back to our example with the legacy code that does an insert, keep in mind that for BOTH versions of the `insert()` method (one that adds an Integer and one that adds a String) the compiler issues warnings. The compiler does NOT know whether

the `insert()` method is adding the right thing (Integer) or wrong thing (String). The reason the compiler produces a warning is because the method is ADDING something to the collection! In other words, the compiler knows there's a chance the method might add the wrong thing to a collection the caller thinks is type safe.

exam Watch

For the purposes of the exam, unless the question includes an answer that mentions warnings, then even if you know compilation will produce warnings, that is still a successful compile! Compiling with warnings is NEVER considered a compilation failure.

One more time—if you see code that you know will compile with warnings, you must NOT choose "Compilation fails." as an answer. The bottom line is this: code that compiles with warnings is still a successful compile. If the exam question wants to test your knowledge of whether code will produce a warning (or what you can do to the code to ELIMINATE warnings), the question (or answer) will explicitly include the word "warnings."

So far, we've looked at how the compiler will generate warnings if it sees that there's a chance your type safe collection could be harmed by older, non-type-safe code. But one of the questions developers often ask is, "Okay, sure, it compiles, but why does it RUN? Why does the code that inserts the wrong thing into my list work at runtime?" In other words, why does the JVM let old code stuff a String into your ArrayList<Integer>, without any problems at all? No exceptions, nothing. Just a quiet, behind-the-scenes, total violation of your type safety that you might not discover until the worst possible moment.

There's one Big Truth you need to know to understand why it runs without problems—the JVM has no idea that your ArrayList was supposed to hold only Integers. The typing information does not exist at runtime! All your generic code is strictly for the compiler. Through a process called "type erasure," the compiler does all of its verifications on your generic code and then strips the type information out of the class bytecode. At runtime, ALL collection code—both legacy and new Java 5 code you write using generics—looks exactly like the pre-generic version of collections. None of your typing information exists at runtime. In other words, even though you WROTE

```
List<Integer> myList = new ArrayList<Integer>();
```

By the time the compiler is done with it, the JVM sees what it always saw before Java 5 and generics:

```
List myList = new ArrayList();
```

The compiler even inserts the casts for you—the casts you had to do to get things out of a pre-Java 5 collection.

Think of generics as strictly a compile-time protection. The compiler uses generic type information (the <type> in the angle brackets) to make sure that your code doesn't put the wrong things into a collection, and that you do not assign what you get from a collection to the wrong reference type. But NONE of this protection exists at runtime.

This is a little different from arrays, which give you BOTH compile-time protection and runtime protection. Why did they do generics this way? Why is there no type information at runtime? To support legacy code. At runtime, collections are collections just like the old days. What you gain from using generics is compile-time protection that guarantees that you won't put the wrong thing into a typed collection, and it also eliminates the need for a cast when you get something out, since the compiler already knows that only an Integer is coming out of an Integer list.

The fact is, you don't NEED runtime protection…until you start mixing up generic and non-generic code, as we did in the previous example. Then you can have disasters at runtime. The only advice we have is to pay very close attention to those compiler warnings:

```
javac TestBadLegacy.java
Note: TestBadLegacy.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

This compiler warning isn't very descriptive, but the second note suggests that you recompile with `-Xlint:unchecked`. If you do, you'll get something like this:

```
javac -Xlint:unchecked TestBadLegacy.java
TestBadLegacy.java:17: warning: [unchecked] unchecked call to
add(E) as a member of the raw type java.util.List
       list.add(new String("42"));
                ^
1 warning
```

When you compile with the `-Xlint:unchecked` flag, the compiler shows you exactly which method(s) might be doing something dangerous. In this example,

since the list argument was not declared with a type, the compiler treats it as legacy code and assumes no risk for what the method puts into the "raw" list.

On the exam, you must be able to recognize when you are compiling code that will produce warnings but still compile. And any code that compiles (even with warnings) will run! No type violations will be caught at runtime by the JVM, *until* those type violations mess with your code in some other way. In other words, the act of adding a String to an <Integer> list won't fail at runtime *until* you try to treat that String-you-think-is-an-Integer as an Integer.

For example, imagine you want your code to pull something out of your *supposedly* type safe ArrayList<Integer> that older code put a String into. It compiles (with warnings). It runs...or at least the code that actually adds the String to the list runs. But when you take the String-that-wasn't-supposed-to-be-there out of the list, and try to assign it to an Integer reference or invoke an Integer method, you're dead.

Keep in mind, then, that the problem of putting the wrong thing into a typed (generic) collection does not show up at the time you actually do the `add()` to the collection. It only shows up later, when you try to use something in the list and it doesn't match what you were expecting. In the old (pre-Java 5) days, you always assumed that you might get the wrong thing out of a collection (since they were all non-type safe), so you took appropriate defensive steps in your code. The problem with mixing generic with non-generic code is that you won't be expecting those problems if you have been lulled into a false sense of security by having written type safe code. Just remember that the moment you turn that type safe collection over to older, non-type safe code, your protection vanishes.

Again, pay very close attention to compiler warnings, and be prepared to see issues like this come up on the exam.

When using legacy (non-type safe) collections—watch out for unboxing problems! If you declare a non-generic collection, the `get()` ***method ALWAYS returns a reference of type java.lang.Object. Remember that unboxing can't convert a plain old Object to a primitive, even if that Object reference refers to an Integer (or some other wrapped primitive) on the heap. Unboxing converts only from a wrapper class reference (like an Integer or a Long) to a primitive.***

exam Watch

Unboxing gotcha, continued:

```
List test = new ArrayList();
test.add(43);
int x = (Integer)test.get(0);    // you must cast !!

List<Integer> test2 = new ArrayList<Integer>();
test2.add(343);
int x2 = test2.get(0);           // cast not necessary
```

Watch out for missing casts associated with pre-Java 5, non-generic collections.

Polymorphism and Generics

Generic collections give you the same benefits of type safety that you've always had with arrays, but there are some crucial differences that can bite you if you aren't prepared. Most of these have to do with polymorphism.

You've already seen that polymorphism applies to the "base" type of the collection:

```
List<Integer> myList = new ArrayList<Integer>();
```

In other words, we were able to assign an ArrayList to a List reference, because List is a supertype of ArrayList. Nothing special there—this polymorphic assignment works the way it always works in Java, regardless of the generic typing.

But what about this?

```
class Parent { }
class Child extends Parent { }
List<Parent> myList = new ArrayList<Child>();
```

Think about it for a minute.

Keep thinking...

No, it doesn't work. There's a very simple rule here—the type of the variable declaration must match the type you pass to the actual object type. If you declare List<Foo> foo then whatever you assign to the foo reference MUST be of the generic type <Foo>. Not a subtype of <Foo>. Not a supertype of <Foo>. Just <Foo>.

These are wrong:

```
List<Object> myList = new ArrayList<JButton>();  // NO!
List<Number> numbers = new ArrayList<Integer>(); // NO!
// remember that Integer is a subtype of Number
```

But these are fine:

```
List<JButton> myList = new ArrayList<JButton>(); // yes
List<Object> myList = new ArrayList<Object>();   // yes
List<Integer> myList = new ArrayList<Integer>(); // yes
```

So far so good. Just keep the generic type of the reference and the generic type of the object to which it refers identical. In other words, polymorphism applies here to only the "base" type. And by "base," we mean the type of the collection class itself—the class that can be customized with a type. In this code,

```
List<JButton> myList = new ArrayList<JButton>();
```

List and ArrayList are the *base* type and JButton is the *generic* type. So an ArrayList can be assigned to a List, but a collection of <JButton> cannot be assigned to a reference of <Object>, even though JButton is a subtype of Object.

The part that feels wrong for most developers is that this is NOT how it works with arrays, where you *are* allowed to do this,

```
import java.util.*;
class Parent { }
class Child extends Parent { }
public class TestPoly {
    public static void main(String[] args) {
        Parent[] myArray = new Child[3];  // yes
    }
}
```

which means you're also allowed to do this

```
Object[] myArray = new JButton[3];  // yes
```

but not this:

```
List<Object> list = new ArrayList<JButton>();  // NO!
```

Why are the rules for typing of arrays different from the rules for generic typing? We'll get to that in a minute. For now, just burn it into your brain that polymorphism does not work the same way for generics as it does with arrays.

Generic Methods

If you weren't already familiar with generics, you might be feeling very uncomfortable with the implications of the previous no-polymorphic-assignment-for-generic-types thing. And why shouldn't you be uncomfortable? One of the biggest benefits of polymorphism is that you can declare, say, a method argument of a particular type and at runtime be able to have that argument refer to any subtype—including those you'd never known about at the time you wrote the method with the supertype argument.

For example, imagine a classic (simplified) polymorphism example of a veterinarian (AnimalDoctor) class with a method `checkup()`. And right now, you have three Animal subtypes—Dog, Cat, and Bird—each implementing the `abstract checkup()` method from Animal:

```
abstract class Animal {
    public abstract void checkup();
}
class Dog extends Animal {
    public void checkup() {   // implement Dog-specific code
      System.out.println("Dog checkup");
    }
}
class Cat extends Animal {
    public void checkup() {   // implement Cat-specific code
      System.out.println("Cat checkup");
    }
}
class Bird extends Animal {
    public void checkup() {   // implement Bird-specific code
      System.out.println("Bird checkup");
}   }
```

Forgetting collections/arrays for a moment, just imagine what the AnimalDoctor class needs to look like in order to have code that takes any kind of Animal and invokes the Animal `checkup()` method. Trying to overload the AnimalDoctor class with checkup() methods for every possible kind of animal is ridiculous, and obviously not extensible. You'd have to change the AnimalDoctor class every time someone added a new subtype of Animal.

So in the AnimalDoctor class, you'd probably have a polymorphic method:

```
public void checkAnimal(Animal a) {
     a.checkup(); // does not matter which animal subtype each
                  // Animal's overridden checkup() method runs
}
```

And of course we do want the AnimalDoctor to also have code that can take arrays of Dogs, Cats, or Birds, for when the vet comes to the dog, cat, or bird kennel. Again, we don't want overloaded methods with arrays for each potential Animal subtype, so we use polymorphism in the AnimalDoctor class:

```
  public void checkAnimals(Animal[] animals) {
      for(Animal a : animals) {
        a.checkup();
      }
    }
```

Here is the entire example, complete with a test of the array polymorphism that takes any type of animal array (`Dog[]`, `Cat[]`, `Bird[]`).

```
import java.util.*;
abstract class Animal {
     public abstract void checkup();
}
class Dog extends Animal {
    public void checkup() {   // implement Dog-specific code
      System.out.println("Dog checkup");
    }
}
class Cat extends Animal {
   public void checkup() {    // implement Cat-specific code
      System.out.println("Cat checkup");
   }
}
```

```
class Bird extends Animal {
   public void checkup() {     // implement Bird-specific code
      System.out.println("Bird checkup");
   }
}
public class AnimalDoctor  {
    // method takes an array of any animal subtype
    public void checkAnimals(Animal[] animals) {
      for(Animal a : animals) {
        a.checkup();
      }
    }
    public static void main(String[] args) {
        // test it
       Dog[] dogs = {new Dog(), new Dog()};
       Cat[] cats = {new Cat(), new Cat(), new Cat()};
       Bird[] birds = {new Bird()};

       AnimalDoctor doc = new AnimalDoctor();
       doc.checkAnimals(dogs);  // pass the Dog[]
       doc.checkAnimals(cats);  // pass the Cat[]
       doc.checkAnimals(birds); // pass the Bird[]
   }
}
```

This works fine, of course (we know, we know, this is old news). But here's why we brought this up as refresher—this approach does NOT work the same way with type safe collections!

In other words, a method that takes, say, an ArrayList<Animal> will NOT be able to accept a collection of any Animal subtype! That means ArrayList<Dog> cannot be passed into a method with an argument of ArrayList<Animal>, even though we already know that this works just fine with plain old arrays.

Obviously this difference between arrays and ArrayList is consistent with the polymorphism assignment rules we already looked at—the fact that you cannot assign an object of type ArrayList<JButton> to a List<Object>. But this is where you really start to feel the pain of the distinction between typed arrays and typed collections.

We know it won't work correctly, but let's try changing the AnimalDoctor code to use generics instead of arrays:

```
public class AnimalDoctorGeneric  {
```

```
    // change the argument from Animal[] to ArrayList<Animal>
    public void checkAnimals(ArrayList<Animal> animals) {
      for(Animal a : animals) {
        a.checkup();
      }
    }
    public static void main(String[] args) {
       // make ArrayLists instead of arrays for Dog, Cat, Bird
       List<Dog> dogs = new ArrayList<Dog>();
       dogs.add(new Dog());
       dogs.add(new Dog());
       List<Cat> cats = new ArrayList<Cat>();
       cats.add(new Cat());
       cats.add(new Cat());
       List<Bird> birds = new ArrayList<Bird>();
       birds.add(new Bird());
       // this code is the same as the Array version
       AnimalDoctorGeneric doc = new AnimalDoctorGeneric();
       // this worked when we used arrays instead of ArrayLists
       doc.checkAnimals(dogs);  // send a List<Dog>
       doc.checkAnimals(cats);  // send a List<Cat>
       doc.checkAnimals(birds); // send a List<Bird>
    }
}
```

So what does happen?

```
javac AnimalDoctorGeneric.java
AnimalDoctorGeneric.java:51: checkAnimals(java.util.
ArrayList<Animal>) in AnimalDoctorGeneric cannot be applied to
(java.util.List<Dog>)
       doc.checkAnimals(dogs);
          ^
AnimalDoctorGeneric.java:52: checkAnimals(java.util.
ArrayList<Animal>) in AnimalDoctorGeneric cannot be applied to
(java.util.List<Cat>)
       doc.checkAnimals(cats);
          ^
AnimalDoctorGeneric.java:53: checkAnimals(java.util.
ArrayList<Animal>) in AnimalDoctorGeneric cannot be applied to
(java.util.List<Bird>)
       doc.checkAnimals(birds);
          ^
3 errors
```

The compiler stops us with errors, not warnings. You simply CANNOT assign the individual ArrayLists of Animal subtypes (<Dog>, <Cat>, or <Bird>) to an ArrayList of the supertype <Animal>, which is the declared type of the argument.

This is one of the biggest gotchas for Java programmers who are so familiar with using polymorphism with arrays, where the same scenario (`Animal[]` can refer to `Dog[]`, `Cat[]`, or `Bird[]`) works as you would expect. So we have two real issues:

1. Why doesn't this work?
2. How do you get around it?

You'd hate us and all of the Sun engineers if we told you that there wasn't a way around it—that you had to accept it and write horribly inflexible code that tried to anticipate and code overloaded methods for each specific <type>. Fortunately, there is a way around it.

But first, why can't you do it if it works for arrays? Why can't you pass an ArrayList<Dog> into a method with an argument of ArrayList<Animal>?

We'll get there, but first let's step way back for a minute and consider this perfectly legal scenario:

```
Animal[] animals = new Animal[3];
animals[0] = new Cat();
animals[1] = new Dog();
```

Part of the benefit of declaring an array using a more abstract supertype is that the array itself can hold objects of multiple subtypes of the supertype, and then you can manipulate the array assuming everything in it can respond to the Animal interface (in other words, everything in the array can respond to method calls defined in the Animal class). So here, we're using polymorphism not for the object that the array reference points to, but rather what the array can actually HOLD—in this case, any subtype of Animal. You can do the same thing with generics:

```
List<Animal> animals = new ArrayList<Animal>();
animals.add(new Cat());  // OK
animals.add(new Dog());  // OK
```

So this part works with both arrays and generic collections—we can add an instance of a subtype into an array or collection declared with a supertype. You can add Dogs and Cats to an Animal array (`Animal[]`) or an Animal collection (ArrayList<Animal>).

And with arrays, this applies to what happens within a method:

```
public void addAnimal(Animal[] animals) {
      animals[0] = new Dog();  // no problem, any Animal works
                               // in Animal[]
}
```

So if this is true, and if you can put Dogs into an ArrayList<Animal>, then why can't you use that same kind of method scenario? Why can't you do this?

```
public void addAnimal(ArrayList<Animal> animals) {
     animals.add(new Dog()); // sometimes allowed...
}
```

Actually, you CAN do this under certain conditions. The code above WILL compile just fine IF what you pass into the method is also an ArrayList<Animal>. This is the part where it differs from arrays, because in the array version, you COULD pass a `Dog[]` into the method that takes an `Animal[]`.

The ONLY thing you can pass to a method argument of ArrayList<Animal> is an ArrayList<Animal>! (Assuming you aren't trying to pass a subtype of ArrayList, since remember—the "base" type can be polymorphic.)

The question is still out there—why is this bad? And why is it bad for ArrayList but not arrays? Why can't you pass an ArrayList<Dog> to an argument of ArrayList<Animal>? Actually, the problem IS just as dangerous whether you're using arrays or a generic collection. It's just that the compiler and JVM behave differently for arrays vs. generic collections.

The reason it is dangerous to pass a collection (array or ArrayList) of a subtype into a method that takes a collection of a supertype, is because you might add something. And that means you might add the WRONG thing! This is probably really obvious, but just in case (and to reinforce), let's walk through some scenarios. The first one is simple:

```
public void foo() {
  Dog[] dogs = {new Dog(), new Dog()};
  addAnimal(dogs);  // no problem, send the Dog[] to the method
}
public void addAnimal(Animal[] animals) {
  animals[0] = new Dog();  // ok, any Animal subtype works
}
```

This is no problem. We passed a `Dog[]` into the method, and added a Dog to the array (which was allowed since the method parameter was type `Animal[]`, which can hold any Animal subtype). But what if we changed the calling code to

```
public void foo() {
  Cat[] cats = {new Cat(), new Cat()};
  addAnimal(cats);  // no problem, send the Cat[] to the method
}
```

and the original method stays the same:

```
public void addAnimal(Animal[] animals) {
   animals[0] = new Dog();  // Eeek! We just put a Dog
                            // in a Cat array!
}
```

The compiler thinks it is perfectly fine to add a Dog to an `Animal[]` array, since a Dog can be assigned to an Animal reference. The problem is, if you passed in an array of an Animal subtype (Cat, Dog, or Bird), the compiler does not know. The compiler does not realize that out on the heap somewhere is an array of type `Cat[]`, not `Animal[]`, and you're about to try to add a Dog to it. To the compiler, you have passed in an array of type Animal, so it has no way to recognize the problem.

THIS is the scenario we're trying to prevent, regardless of whether it's an array or an ArrayList. The difference is, the compiler lets you get away with it for arrays, but not for generic collections.

The reason the compiler won't let you pass an ArrayList<Dog> into a method that takes an ArrayList<Animal>, is because within the method, that parameter is of type ArrayList<Animal>, and that means you could put *any* kind of Animal into it. There would be no way for the compiler to stop you from putting a Dog into a List that was originally declared as <Cat>, but is now referenced from the <Animal> parameter.

We still have two questions…how do you get around it and why the heck does the compiler allow you to take that risk for arrays but not for ArrayList (or any other generic collection)?

The reason you can get away with compiling this for arrays is because there is a runtime exception (ArrayStoreException) that will prevent you from putting the wrong type of object into an array. If you send a `Dog` array into the method that takes an `Animal` array, and you add only Dogs (including Dog subtypes, of course) into the array now referenced by `Animal`, no problem. But if you DO try to add a Cat to the object that is actually a `Dog` array, you'll get the exception.

But there IS no equivalent exception for generics, because of type erasure! In other words, at runtime the JVM KNOWS the type of arrays, but does NOT know the type of a collection. All the generic type information is removed during compilation, so by the time it gets to the JVM, there is simply no way to recognize the disaster of putting a Cat into an ArrayList<Dog> and vice versa (and it becomes exactly like the problems you have when you use legacy, non-type safe code).

So this actually IS legal code:

```
public void addAnimal(List<Animal> animals) {
    animals.add(new Dog());  // this is always legal,
                             // since Dog can
                             // be assigned to an Animal
                             // reference
 }
  public static void main(String[] args) {
     List<Animal> animals = new ArrayList<Animal>();
     animals.add(new Dog());
     animals.add(new Dog());
     AnimalDoctorGeneric doc = new AnimalDoctorGeneric();
     doc.addAnimal(animals); // OK, since animals matches
                             // the method arg
  }
```

As long as the only thing you pass to the `addAnimals(List<Animal>)` is an ArrayList<Animal>, the compiler is pleased—knowing that any Animal subtype you add will be valid (you can always add a Dog to an Animal collection, yada, yada, yada). But if you try to invoke `addAnimal()` with an argument of any OTHER ArrayList type, the compiler will stop you, since at runtime the JVM would have no way to stop you from adding a Dog to what was created as a Cat collection.

For example, this code that changes the generic type to <Dog>, but without changing the `addAnimal()` method, will NOT compile:

```
public void addAnimal(List<Animal> animals) {
  animals.add(new Dog());  // still OK as always
}
public static void main(String[] args) {
  List<Dog> animals = new ArrayList<Dog>();
  animals.add(new Dog());
  animals.add(new Dog());
  AnimalDoctorGeneric doc = new AnimalDoctorGeneric();
  doc.addAnimal(animals); // THIS is where it breaks!
}
```

The compiler says something like:

```
javac AnimalDoctorGeneric.java
AnimalDoctorGeneric.java:49: addAnimal(java.util.List<Animal>)
in AnimalDoctorGeneric cannot be applied to (java.util.
List<Dog>)
       doc.addAnimal(animals);
           ^
1 error
```

Notice that this message is virtually the same one you'd get trying to invoke any method with the wrong argument. It's saying that you simply cannot invoke `addAnimal(List<Animal>)` using something whose reference was declared as List<Dog>. (It's the reference type, not the actual object type that matters—but remember—the generic type of an object is ALWAYS the same as the generic type declared on the reference. List<Dog> can refer ONLY to collections that are subtypes of List, but which were instantiated as generic type <Dog>.)

Once again, remember that once inside the `addAnimals()` method, all that matters is the type of the parameter—in this case, List<Animal>. (We changed it from ArrayList to List to keep our "base" type polymorphism cleaner.)

Back to the key question—how do we get around this? If the problem is related only to the danger of adding the wrong thing to the collection, what about the `checkup()` method that used the collection passed in as read-only? In other words, what about methods that invoke Animal methods on each thing in the collection, which will work regardless of which kind of ArrayList subtype is passed in?

And that's a clue! It's the `add()` method that is the problem, so what we need is a way to tell the compiler, "Hey, I'm using the collection passed in just to invoke methods on the elements—and I promise not to ADD anything into the collection." And there IS a mechanism to tell the compiler that you can take any generic subtype of the declared argument type because you won't be putting anything in the collection. And that mechanism is the wildcard `<?>`.

The method signature would change from

```
public void addAnimal(List<Animal> animals)
```

to

```
public void addAnimal(List<? extends Animal> animals)
```

By saying `<? extends Animal>`, we're saying, "I can be assigned a collection that is a subtype of List and typed for <Animal> or anything that *extends* Animal. And oh yes, I SWEAR that I will not ADD anything into the collection." (There's a little more to the story, but we'll get there.)

So of course the `addAnimal()` method above won't actually compile even with the wildcard notation, because that method DOES add something.

```
public void addAnimal(List<? extends Animal> animals) {
      animals.add(new Dog());  // NO! Can't add if we
                               // use <? extends Animal>
}
```

You'll get a very strange error that might look something like this:

```
javac AnimalDoctorGeneric.java
AnimalDoctorGeneric.java:38: cannot find symbol
symbol  : method add(Dog)
location: interface java.util.List<capture of ? extends Animal>
      animals.add(new Dog());
              ^
1 error
```

which basically says, "you can't add a Dog here." If we change the method so that it doesn't add anything, it works.

But wait—there's more. (And by the way, everything we've covered in this generics section is likely to be tested for on the exam, with the exception of "type erasure," for which you aren't required to know any details.)

First, the `<? extends Animal>` means that you can take any subtype of Animal; however, that subtype can be EITHER a subclass of a class (abstract or concrete) OR a type that implements the interface after the word `extends`. In other words, the keyword `extends` in the context of a wildcard represents BOTH subclasses and interface implementations. There is no `<? implements Serializable>` syntax. If you want to declare a method that takes anything that is of a type that implements Serializable, you'd still use `extends` like this:

```
void foo(List<? extends Serializable> list) // odd, but correct
                                            // to use "extends"
```

This looks strange since you would never say this in a class declaration because Serializable is an interface, not a class. But that's the syntax, so burn it in!

One more time—there is only ONE wildcard keyword that represents *both* interface implementations and subclasses. And that keyword is `extends`. But when you see it, think "Is-a", as in something that passes the `instanceof` test.

However, there is another scenario where you can use a wildcard AND still add to the collection, but in a safe way—the keyword `super`.

Imagine, for example, that you declared the method this way:

```
public void addAnimal(List<? super Dog> animals) {
  animals.add(new Dog());  // adding is sometimes OK with super
}
public static void main(String[] args) {
  List<Animal> animals = new ArrayList<Animal>();
  animals.add(new Dog());
  animals.add(new Dog());
  AnimalDoctorGeneric doc = new AnimalDoctorGeneric();
  doc.addAnimal(animals); // passing an Animal List
}
```

Now what you've said in this line

```
public void addAnimal(List<? super Dog> animals)
```

is essentially, "Hey compiler, please accept any List with a generic type that is of type Dog, or a supertype of Dog. Nothing lower in the inheritance tree can come in, but anything higher than Dog is OK."

You probably already recognize why this works. If you pass in a list of type Animal, then it's perfectly fine to add a Dog to it. If you pass in a list of type Dog, it's perfectly fine to add a Dog to it. And if you pass in a list of type Object, it's STILL fine to add a Dog to it. When you use the `<? super ...>` syntax, you are telling the compiler that you can accept the type on the right-hand side of `super` or any of its supertypes, since—and this is the key part that makes it work—a collection declared as any supertype of Dog will be able to accept a Dog as an element. List<Object> can take a Dog. List<Animal> can take a Dog. And List<Dog> can take a Dog. So passing any of those in will work. So the `super` keyword in wildcard notation lets you have a restricted, but still possible way to add to a collection.

So, the wildcard gives you polymorphic assignments, but with certain restrictions that you don't have for arrays. Quick question: are these two identical?

```
public void foo(List<?> list) { }
public void foo(List<Object> list) { }
```

If there IS a difference (and we're not yet saying there is), what is it?

There IS a huge difference. `List<?>`, which is the wildcard `<?>` without the keywords `extends` or `super`, simply means "any type." So that means any type of List can be assigned to the argument. That could be a List of <Dog>, <Integer>, <JButton>, <Socket>, whatever. And using the wildcard alone, without the keyword `super` (followed by a type), means that you cannot ADD anything to the list referred to as `List<?>`.

`List<Object>` is completely different from `List<?>`. `List<Object>` means that the method can take ONLY a `List<Object>`. Not a `List<Dog>`, or a `List<Cat>`. It does, however, mean that you can add to the list, since the compiler has already made certain that you're passing only a valid `List<Object>` into the method.

Based on the previous explanations, figure out if the following will work:

```
import java.util.*;
public class TestWildcards {
  public static void main(String[] args) {
     List<Integer> myList = new ArrayList<Integer>();
     Bar bar = new Bar();
     bar.doInsert(myList);
  }
}
class Bar {
  void doInsert(List<?> list) {
     list.add(new Dog());
  }
}
```

If not, where is the problem?

The problem is in the `list.add()` method within `doInsert()`. The `<?>` wildcard allows a list of ANY type to be passed to the method, but the `add()` method is not valid, for the reasons we explored earlier (that you could put the wrong kind of thing into the collection). So this time, the TestWildcards class is fine, but the Bar class won't compile because it does an `add()` in a method that uses a wildcard (without `super`). What if we change the `doInsert()` method to this:

```
public class TestWildcards {
  public static void main(String[] args) {
     List<Integer> myList = new ArrayList<Integer>();
     Bar bar = new Bar();
     bar.doInsert(myList);
  }
}
```

```
class Bar {
   void doInsert(List<Object> list) {
     list.add(new Dog());
   }
}
```

Now will it work? If not, why not?

This time, class Bar, with the `doInsert()` method, compiles just fine. The problem is that the TestWildcards code is trying to pass a List<Integer> into a method that can take ONLY a List<Object>. And *nothing* else can be substituted for <Object>.

By the way, `List<? extends Object>` and `List<?>` are absolutely identical! They both say, "I can refer to any type of object." But as you can see, neither of them are the same as List<Object>. One way to remember this is that if you see the wildcard notation (a question mark `?`), this means "many possibilities". If you do NOT see the question mark, then it means the <type> in the brackets, and absolutely NOTHING ELSE. List<Dog> means List<Dog> and not List<Beagle>, List<Poodle>, or any other subtype of Dog. But List<? extends Dog> could mean List<Beagle>, List<Poodle>, and so on. Of course List<?> could be... anything at all.

Keep in mind that the wildcards can be used only for reference declarations (including arguments, variables, return types, and so on). They can't be used as the type parameter when you create a new typed collection. Think about that—while a reference can be abstract and polymorphic, the actual object created must be of a specific type. You have to lock down the type when you make the object using `new`.

As a little review before we move on with generics, look at the following statements and figure out which will compile:

```
1) List<?> list = new ArrayList<Dog>();
2) List<? extends Animal> aList = new ArrayList<Dog>();
3) List<?> foo = new ArrayList<? extends Animal>();
4) List<? extends Dog> cList = new ArrayList<Integer>();
5) List<? super Dog> bList = new ArrayList<Animal>();
6) List<? super Animal> dList = new ArrayList<Dog>();
```

The correct answers (the statements that compile) are 1, 2, and 5.

The three that won't compile are

- Statement: `List<?> foo = new ArrayList<? extends Animal>();`
  Problem: you cannot use wildcard notation in the object creation. So the `new ArrayList<? extends Animal>()` will not compile.

- Statement: `List<? extends Dog> cList = new ArrayList<Integer>();`

  Problem: You cannot assign an Integer list to a reference that takes only a Dog (including any subtypes of Dog, of course).

- Statement: `List<? super Animal> dList = new ArrayList<Dog>();`

  Problem: You cannot assign a Dog to <? super Animal>. The Dog is too "low" in the class hierarchy. Only <Animal> or <Object> would have been legal.

Generic Declarations

Until now, we've talked about how to create type safe collections, and how to declare reference variables including arguments and return types using generic syntax. But here are a few questions: How do we even know that we're allowed/supposed to specify a type for these collection classes? And does generic typing work with any other classes in the API? And finally, can we declare our own classes as generic types? In other words, can we make a class that requires that someone pass a type in when they declare it and instantiate it?

First, the one you obviously know the answer to—the API tells you when a parameterized type is expected. For example, this is the API declaration for the java.util.List interface:

```
public interface List<E>
```

The <E> is a placeholder for the type you pass in. The List interface is behaving as a generic "template" (sort of like C++ templates), and when you write your code, you change it from a generic List to a List<Dog> or List<Integer>, and so on.

The E, by the way, is only a convention. Any valid Java identifier would work here, but E stands for "Element," and it's used when the template is a collection. The other main convention is T (stands for "type"), used for, well, things that are NOT collections.

Now that you've seen the interface declaration for List, what do you think the `add()` method looks like?

```
boolean add(E o)
```

In other words, whatever E is when you declare the List, *that's what you can add to it*. So imagine this code:

```
List<Animal> list = new ArrayList<Animal>();
```

The E in the List API suddenly has its waveform collapsed, and goes from the abstract <your type goes here>, to a List of Animals. And if it's a List of Animals, then the `add()` method of List must obviously behave like this:

```
boolean add(Animal a)
```

When you look at an API for a generics class or interface, pick a type parameter (Dog, JButton, even Object) and do a mental find and replace on each instance of E (or whatever identifier is used as the placeholder for the type parameter).

Making Your Own Generic Class

Let's try making our own generic class, to get a feel for how it works, and then we'll look at a few remaining generics syntax details. Imagine someone created a class Rental, that manages a pool of rentable items.

```
public class Rental {
   private List rentalPool;
   private int maxNum;
   public Rental(int maxNum, List rentalPool) {
      this.maxNum = maxNum;
      this.rentalPool = rentalPool;
   }
   public Object getRental() {
      // blocks until there's something available
      return rentalPool.get(0);
   }
   public void returnRental(Object o) {
      rentalPool.add(o);
   }
}
```

Now imagine you wanted to make a subclass of Rental that was just for renting cars. You might start with something like this:

```
import java.util.*;
public class CarRental extends Rental {
   public CarRental(int maxNum, List<Car> rentalPool) {
      super(maxNum, rentalPool);
   }
   public Car getRental() {
      return (Car) super.getRental();
   }
```

```
   public void returnRental(Car c) {
      super.returnRental(c);
   }
   public void returnRental(Object o) {
     if (o instanceof Car) {
       super.returnRental(o);
     } else {
        System.out.println("Cannot add a non-Car");
        // probably throw an exception
}  } }
```

But then the more you look at it, the more you realize:

1. You are doing your own type checking in the `returnRental()` method. You can't change the argument type of `returnRental()` to take a Car, since it's an override (not an overload) of the method from class Rental. (Overloading would take away your polymorphic flexibility with Rental).
2. You really don't want to make separate subclasses for every possible kind of rentable thing (cars, computers, bowling shoes, children, and so on).

But given your natural brilliance (heightened by this contrived scenario), you quickly realize that you can make the Rental class a generic type—a template for any kind of Rentable thing—and you're good to go.

(We did say contrived…since in reality, you might very well want to have different behaviors for different kinds of rentable things, but even that could be solved cleanly through some kind of behavior composition as opposed to inheritance (using the Strategy design pattern, for example). And no, design patterns aren't on the exam, but we still think you should read our design patterns book. Think of the kittens.) So here's your new and improved generic Rental class:

```
import java.util.*;
public class RentalGeneric<T> {  // "T" is for the type
                                 // parameter
   private List<T> rentalPool;   // Use the class type for the
                                 // List type
   private int maxNum;
   public RentalGeneric(
     int maxNum, List<T> rentalPool) { // constructor takes a
                                       // List of the class type
     this.maxNum = maxNum;
     this.rentalPool = rentalPool;
   }
```

```
   public T getRental() {                    // we rent out a T
     // blocks until there's something available
     return rentalPool.get(0);
   }
   public void returnRental(T returnedThing) { // and the renter
                                               // returns a T
     rentalPool.add(returnedThing);
   }
}
```

Let's put it to the test:

```
class TestRental {
  public static void main (String[] args) {
     //make some Cars for the pool
     Car c1 = new Car();
     Car c2 = new Car();
     List<Car> carList = new ArrayList<Car>();
     carList.add(c1);
     carList.add(c2);
     RentalGeneric<Car> carRental = new
                         RentalGeneric<Car>(2, carList);
     // now get a car out, and it won't need a cast
     Car carToRent = carRental.getRental();
     carRental.returnRental(carToRent);
     // can we stick something else in the original carList?
     carList.add(new Cat("Fluffy"));
  }
}
```

We get one error:

```
kathy% javac1.5 RentalGeneric.java
RentalGeneric.java:38: cannot find symbol
symbol  : method add(Cat)
location: interface java.util.List<Car>
     carList.add(new Cat("Fluffy"));
             ^
1 error
```

Now we have a Rental class that can be *typed* to whatever the programmer chooses, and the compiler will enforce it. In other words, it works just as the

Collections classes do. Let's look at more examples of generic syntax you might find in the API or source code. Here's another simple class that uses the parameterized type of the class in several ways:

```
public class TestGenerics<T> {  // as the class type
   T anInstance;                // as an instance variable type
   T [] anArrayOfTs;            // as an array type

   TestGenerics(T anInstance) {     // as an argument type
      this.anInstance = anInstance;
   }
   T getT() {                       // as a return type
      return anInstance;
   }
}
```

Obviously this is a ridiculous use of generics, and in fact you'll see generics only rarely outside of collections. But, you do need to understand the different kinds of generic syntax you might encounter, so we'll continue with these examples until we've covered them all.

You can use more than one parameterized type in a single class definition:

```
public class UseTwo<T, X> {
  T one;
  X two;
  UseTwo(T one, X two) {
    this.one = one;
    this.two = two;
  }
  T getT() { return one; }
  X getX() { return two; }

// test it by creating it with <String, Integer>

  public static void main (String[] args) {
    UseTwo<String, Integer> twos =
                    new UseTwo<String, Integer>("foo", 42);

    String theT = twos.getT(); // returns a String
    int theX = twos.getX();   // returns Integer, unboxes to int
  }
}
```

And you can use a form of wildcard notation in a class definition, to specify a range (called "bounds") for the type that can be used for the type parameter:

```
public class AnimalHolder<T extends Animal> { // use "T" instead
                                               // of "?"
 T animal;
 public static void main(String[] args) {
   AnimalHolder<Dog> dogHolder = new AnimalHolder<Dog>(); // OK
   AnimalHolder<Integer> x = new AnimalHolder<Integer>(); // NO!
 }
}
```

Creating Generic Methods

Until now, every example we've seen uses the class parameter type—the type declared with the class name. For example, in the UseTwo<T,X> declaration, we used the T and X placeholders throughout the code. But it's possible to define a parameterized type at a more granular level—a method.

Imagine you want to create a method that takes an instance of any type, instantiates an ArrayList of that type, and adds the instance to the ArrayList. The class itself doesn't need to be generic; basically we just want a utility method that we can pass a type to and that can use that type to construct a type safe collection. Using a generic method, we can declare the method without a specific type and then get the type information based on the type of the object passed to the method. For example:

```
import java.util.*;
public class CreateAnArrayList {
  public <T> void makeArrayList(T t) { // take an object of an
                                       // unknown type and use a
                                       // "T" to represent the type
    List<T> list = new ArrayList<T>(); // now we can create the
                                       // list using "T"
    list.add(t);
   }
}
```

In the preceding code, if you invoke the `makeArrayList()` method with a Dog instance, the method will behave as though it looked like this all along:

```
  public void makeArrayList(Dog t) {
     List<Dog> list = new ArrayList<Dog>();
     list.add(t);
  }
```

And of course if you invoke the method with an Integer, then the T is replaced by Integer (not in the bytecode, remember—we're describing how it appears to behave, not how it actually gets it done).

The strangest thing about generic methods is that you must declare the type variable BEFORE the return type of the method:

```
public <T> void makeArrayList(T t)
```

The <T> before `void` simply defines what T is before you use it as a type in the argument. You MUST declare the type like that unless the type is specified for the class. In CreateAnArrayList, the class is not generic, so there's no type parameter placeholder we can use.

You're also free to put boundaries on the type you declare, for example, if you want to restrict the `makeArrayList()` method to only Number or its subtypes (Integer, Float, and so on) you would say

```
public <T extends Number> void makeArrayList(T t)
```

exam Watch

It's tempting to forget that the method argument is NOT where you declare the type parameter variable T. In order to use a type variable like T, you must have declared it either as the class parameter type or in the method, before the return type. The following might look right,

```
public void makeList(T t) { }
```

But the only way for this to be legal is if there is actually a class named T, in which case the argument is like any other type declaration for a variable. And what about constructor arguments? They, too, can be declared with a generic type, but then it looks even stranger since constructors have no return type at all:

```
public class Radio {
   public <T> Radio(T t) { }  // legal constructor
}
```

exam Watch

If you REALLY want to get ridiculous (or fired), you can declare a class with a name that is the same as the type parameter placeholder:

```
class X { public <X> X(X x) { } }
```

Yes, this works. The X that is the constructor name has no relationship to the <X> type declaration, which has no relationship to the constructor argument identifier, which is also, of course, X. The compiler is able to parse this and treat each of the different uses of X independently. So there is no naming conflict between class names, type parameter placeholders, and variable identifiers.

exam Watch

One of the most common mistakes programmers make when creating generic classes or methods is to use a <?> in the wildcard syntax rather than a type variable <T>, <E>, and so on. This code might look right, but isn't:

```
public class NumberHolder<? extends Number> { }
```

While the question mark works when declaring a reference for a variable, it does NOT work for generic class and method declarations. This code is not legal:

```
public class NumberHolder<?> { ? aNum; }    // NO!
```

But if you replace the <?> with a legal identifier, you're good:

```
public class NumberHolder<T> { T aNum; }    // Yes
```

98% of what you're likely to do with generics is simply declare and use type safe collections, including using (and passing) them as arguments. But now you know much more (but by no means everything) about the way generics works.

If this was clear and easy for you, that's excellent. If it was…painful…just know that adding generics to the Java language very nearly caused a revolt among some of the most experienced Java developers. Most of the outspoken critics are simply unhappy with the complexity, or aren't convinced that gaining type safe collections is worth the ten million little rules you have to learn now. It's true that with Java 5, learning Java just got harder. But trust us…we've never seen it take more than two days to "get" generics. That's 48 consecutive hours.

CERTIFICATION SUMMARY

We began with a quick review of the `toString()` method. The `toString()` method is automatically called when you ask `System.out.println()` to print an object—you override it to return a String of meaningful data about your objects.

Next we reviewed the purpose of == (to see if two reference variables refer to the same object) and the `equals()` method (to see if two objects are meaningfully equivalent). You learned the downside of not overriding `equals()`—you may not be able to find the object in a collection. We discussed a little bit about how to write a good `equals()` method—don't forget to use `instanceof` and refer to the object's significant attributes. We reviewed the contracts for overriding `equals()` and `hashCode()`. We learned about the theory behind hashcodes, the difference between legal, appropriate, and efficient hashcoding. We also saw that even though wildly inefficient, it's legal for a `hashCode()` method to always return the same value.

Next we turned to collections, where we learned about Lists, Sets, and Maps, and the difference between ordered and sorted collections. We learned the key attributes of the common collection classes, and when to use which.

We covered the ins and outs of the Collections and Arrays classes: how to sort, and how to search. We learned about converting arrays to Lists and back again.

Finally we tackled generics. Generics let you enforce compile-time type-safety on collections or other classes. Generics help assure you that when you get an item from a collection it will be of the type you expect, with no casting required. You can mix legacy code with generics code, but this can cause exceptions. The rules for polymorphism change when you use generics, although by using wildcards you can still create polymorphic collections. Some generics declarations allow reading of a collection, but allow very limited updating of the collection.

All in all, one fascinating chapter.

TWO-MINUTE DRILL

Here are some of the key points from this chapter.

Overriding hashCode() and equals() (Objective 6.2)

- ❑ `equals()`, `hashCode()`, and `toString()` are `public`.
- ❑ Override `toString()` so that `System.out.println()` or other methods can see something useful, like your object's state.
- ❑ Use == to determine if two reference variables refer to the same object.
- ❑ Use `equals()` to determine if two objects are meaningfully equivalent.
- ❑ If you don't override `equals()`, your objects won't be useful hashing keys.
- ❑ If you don't override `equals()`, different objects can't be considered equal.
- ❑ Strings and wrappers override `equals()` and make good hashing keys.
- ❑ When overriding `equals()`, use the `instanceof` operator to be sure you're evaluating an appropriate class.
- ❑ When overriding `equals()`, compare the objects' significant attributes.
- ❑ Highlights of the `equals()` contract:
  - ❑ Reflexive: `x.equals(x)` is `true`.
  - ❑ Symmetric: If `x.equals(y)` is `true`, then `y.equals(x)` must be `true`.
  - ❑ Transitive: If `x.equals(y)` is `true`, and `y.equals(z)` is `true`, then `z.equals(x)` is `true`.
  - ❑ Consistent: Multiple calls to `x.equals(y)` will return the same result.
  - ❑ Null: If x is not `null`, then `x.equals(null)` is false.
- ❑ If `x.equals(y)` is true, then `x.hashCode() == y.hashCode()` is true.
- ❑ If you override `equals()`, override `hashCode()`.
- ❑ HashMap, HashSet, Hashtable, LinkedHashMap, & LinkedHashSet use hashing.
- ❑ An appropriate `hashCode()` override sticks to the `hashCode()` contract.
- ❑ An efficient `hashCode()` override distributes keys evenly across its buckets.
- ❑ An overridden `equals()` must be at least as precise as its `hashCode()` mate.
- ❑ To reiterate: if two objects are equal, their hashcodes must be equal.
- ❑ It's legal for a `hashCode()` method to return the same value for all instances (although in practice it's very inefficient).

- Highlights of the `hashCode()` contract:
  - Consistent: multiple calls to `x.hashCode()` return the same integer.
  - If `x.equals(y)` is `true`, `x.hashCode() == y.hashCode()` is `true`.
  - If `x.equals(y)` is `false`, then `x.hashCode() == y.hashCode()` can be either `true` or `false`, but `false` will tend to create better efficiency.
- `transient` variables aren't appropriate for `equals()` and `hashCode()`.

Collections (Objective 6.1)

- Common collection activities include adding objects, removing objects, verifying object inclusion, retrieving objects, and iterating.
- Three meanings for "collection":
  - **collection** Represents the data structure in which objects are stored
  - **Collection** java.util interface from which Set and List extend
  - **Collections** A class that holds static collection utility methods
- Four basic flavors of collections include Lists, Sets, Maps, Queues:
  - **Lists of things** Ordered, duplicates allowed, with an index.
  - **Sets of things** May or may not be ordered and/or sorted; duplicates not allowed.
  - **Maps of things with keys** May or may not be ordered and/or sorted; duplicate keys are not allowed.
  - **Queues of things to process** Ordered by FIFO or by priority.
- Four basic sub-flavors of collections Sorted, Unsorted, Ordered, Unordered.
  - **Ordered** Iterating through a collection in a specific, non-random order.
  - **Sorted** Iterating through a collection in a sorted order.
- Sorting can be alphabetic, numeric, or programmer-defined.

Key Attributes of Common Collection Classes (Objective 6.1)

- ArrayList: Fast iteration and fast random access.
- Vector: It's like a slower ArrayList, but it has synchronized methods.
- LinkedList: Good for adding elements to the ends, i.e., stacks and queues.
- HashSet: Fast access, assures no duplicates, provides no ordering.
- LinkedHashSet: No duplicates; iterates by insertion order.
- TreeSet: No duplicates; iterates in sorted order.

- ❑ HashMap: Fastest updates (key/values); allows one `null` key, many `null` values.
- ❑ Hashtable: Like a slower HashMap (as with Vector, due to its synchronized methods). No `null` values or `null` keys allowed.
- ❑ LinkedHashMap: Faster iterations; iterates by insertion order or last accessed; allows one `null` key, many `null` values.
- ❑ TreeMap: A sorted map.
- ❑ PriorityQueue: A to-do list ordered by the elements' priority.

Using Collection Classes (Objective 6.3)

- ❑ Collections hold only Objects, but primitives can be autoboxed.
- ❑ Iterate with the enhanced `for`, or with an Iterator via `hasNext()` & `next()`.
- ❑ `hasNext()` determines if more elements exist; the Iterator does NOT move.
- ❑ `next()` returns the next element AND moves the Iterator forward.
- ❑ To work correctly, a Map's keys must override `equals()` and `hashCode()`.
- ❑ Queues use `offer()` to add an element, `poll()` to remove the head of the queue, and `peek()` to look at the head of a queue.
- ❑ As of Java 6 TreeSets and TreeMaps have new navigation methods like floor() and higher().
- ❑ You can create/extend "backed" sub-copies of TreeSets and TreeMaps.

Sorting and Searching Arrays and Lists (Objective 6.5)

- ❑ Sorting can be in natural order, or via a Comparable or many Comparators.
- ❑ Implement Comparable using `compareTo()`; provides only one sort order.
- ❑ Create many Comparators to sort a class many ways; implement `compare()`.
- ❑ To be sorted and searched, a List's elements must be *comparable*.
- ❑ To be searched, an array or List must first be sorted.

Utility Classes: Collections and Arrays (Objective 6.5)

- ❑ Both of these java.util classes provide
  - ❑ A `sort()` method. Sort using a Comparator or sort using natural order.
  - ❑ A `binarySearch()` method. Search a pre-sorted array or List.

- `Arrays.asList()` creates a List from an array and links them together.
- `Collections.reverse()` reverses the order of elements in a List.
- `Collections.reverseOrder()` returns a Comparator that sorts in reverse.
- Lists and Sets have a `toArray()` method to create arrays.

Generics (Objective 6.4)

- Generics let you enforce compile-time type safety on Collections (or other classes and methods declared using generic type parameters).
- An ArrayList<Animal> can accept references of type Dog, Cat, or any other subtype of Animal (subclass, or if Animal is an interface, implementation).
- When using generic collections, a cast is not needed to get (declared type) elements out of the collection. With non-generic collections, a cast is required:

```
List<String> gList = new ArrayList<String>();
List list = new ArrayList();
// more code
String s = gList.get(0);           // no cast needed
String s = (String)list.get(0);    // cast required
```

- You can pass a generic collection into a method that takes a non-generic collection, but the results may be disastrous. The compiler can't stop the method from inserting the wrong type into the previously type safe collection.
- If the compiler can recognize that non-type-safe code is potentially endangering something you originally declared as type-safe, you will get a compiler warning. For instance, if you pass a List<String> into a method declared as

```
void foo(List aList) { aList.add(anInteger); }
```

  You'll get a warning because add() is potentially "unsafe".
- "Compiles without error" is not the same as "compiles without warnings." A compilation *warning* is not considered a compilation *error* or *failure*.
- Generic type information does not exist at runtime—it is for compile-time safety only. Mixing generics with legacy code can create compiled code that may throw an exception at runtime.
- Polymorphic assignments applies only to the base type, not the generic type parameter. You can say

```
List<Animal> aList = new ArrayList<Animal>();   // yes
```

  You can't say

```
List<Animal> aList = new ArrayList<Dog>();      // no
```

- The polymorphic assignment rule applies everywhere an assignment can be made. The following are NOT allowed:

```
void foo(List<Animal> aList) { }  // cannot take a List<Dog>
List<Animal> bar() { }            // cannot return a List<Dog>
```

- Wildcard syntax allows a generic method, accept subtypes (or supertypes) of the declared type of the method argument:

```
void addD(List<Dog> d) {}  // can take only <Dog>
void addD(List<? extends Dog>) {} // take a <Dog> or <Beagle>
```

- The wildcard keyword `extends` is used to mean either "extends" or "implements." So in `<? extends Dog>`, `Dog` can be a class or an interface.
- When using a wildcard, `List<? extends Dog>`, the collection can be accessed but not modified.
- When using a wildcard, `List<?>`, any generic type can be assigned to the reference, but for access only, no modifications.
- `List<Object>` refers only to a `List<Object>`, while `List<?>` or `List<? extends Object>` can hold any type of object, but for access only.
- Declaration conventions for generics use T for type and E for element:

```
public interface List<E>  // API declaration for List
boolean add(E o)          // List.add() declaration
```

- The generics type identifier can be used in class, method, and variable declarations:

```
class Foo<t> { }      // a class
T anInstance;         // an instance variable
Foo(T aRef) {}        // a constructor argument
void bar(T aRef) {}   // a method argument
T baz() {}            // a return type
```

  The compiler will substitute the actual type.

- You can use more than one parameterized type in a declaration:

```
public class UseTwo<T, X> { }
```

- You can declare a generic method using a type not defined in the class:

```
public <T> void makeList(T t) { }
```

  is NOT using T as the return type. This method has a `void` return type, but to use T within the method's argument you must declare the <T>, which happens before the return type.

SELF TEST

1. Given:

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

  // INSERT DECLARATION HERE
  for (int i = 0; i <= 10; i++) {
    List<Integer> row = new ArrayList<Integer>();
    for (int j = 0; j <= 10; j++)
      row.add(i * j);
    table.add(row);
  }
  for (List<Integer> row : table)
    System.out.println(row);
  }
```

Which statements could be inserted at `// INSERT DECLARATION HERE` to allow this code to compile and run? (Choose all that apply.)

A. `List<List<Integer>> table = new List<List<Integer>>();`

B. `List<List<Integer>> table = new ArrayList<List<Integer>>();`

C. `List<List<Integer>> table = new ArrayList<ArrayList<Integer>>();`

D. `List<List, Integer> table = new List<List, Integer>();`

E. `List<List, Integer> table = new ArrayList<List, Integer>();`

F. `List<List, Integer> table = new ArrayList<ArrayList, Integer>();`

G. None of the above

2. Which statements are true about comparing two instances of the same class, given that the `equals()` and `hashCode()` methods have been properly overridden? (Choose all that apply.)

A. If the `equals()` method returns true, the `hashCode()` comparison `==` might return false

B. If the `equals()` method returns false, the `hashCode()` comparison `==` might return true

C. If the `hashCode()` comparison `==` returns true, the `equals()` method must return true

D. If the `hashCode()` comparison `==` returns true, the `equals()` method might return true

E. If the `hashCode()` comparison `!=` returns true, the `equals()` method might return true

3. Given:

```
public static void before() {
  Set set = new TreeSet();
  set.add("2");
  set.add(3);
  set.add("1");
  Iterator it = set.iterator();
    while (it.hasNext())
  System.out.print(it.next() + " ");
}
```

Which statements are true?

A. The `before()` method will print `1 2`

B. The `before()` method will print `1 2 3`

C. The `before()` method will print three numbers, but the order cannot be determined

D. The `before()` method will not compile

E. The `before()` method will throw an exception at runtime

4. Given:

```
import java.util.*;
class MapEQ {
  public static void main(String[] args) {
    Map<ToDos, String> m = new HashMap<ToDos, String>();
    ToDos t1 = new ToDos("Monday");
    ToDos t2 = new ToDos("Monday");
    ToDos t3 = new ToDos("Tuesday");
    m.put(t1, "doLaundry");
    m.put(t2, "payBills");
    m.put(t3, "cleanAttic");
    System.out.println(m.size());
  }
}
class ToDos{
  String day;
  ToDos(String d) { day = d; }
  public boolean equals(Object o) {
    return ((ToDos)o).day == this.day;
  }
  // public int hashCode() { return 9; }
}
```

Which is correct? (Choose all that apply.)

A. As the code stands it will not compile

B. As the code stands the output will be `2`

C. As the code stands the output will be `3`

D. If the `hashCode()` method is uncommented the output will be `2`

E. If the `hashCode()` method is uncommented the output will be `3`

F. If the `hashCode()` method is uncommented the code will not compile

5. Given:

```
12. public class AccountManager {
13.     private Map accountTotals = new HashMap();
14.     private int retirementFund;
15.
16.     public int getBalance(String accountName) {
17.         Integer total = (Integer) accountTotals.get(accountName);
18.         if (total == null)
19.             total = Integer.valueOf(0);
20.         return total.intValue();
21.     }
23.     public void setBalance(String accountName, int amount) {
24.         accountTotals.put(accountName, Integer.valueOf(amount));
25.     }
26. }
```

This class is to be updated to make use of appropriate generic types, with no changes in behavior (for better or worse). Which of these steps could be performed? (Choose three.)

A. Replace line 13 with
```
private Map<String, int> accountTotals = new HashMap<String, int>();
```

B. Replace line 13 with
```
private Map<String, Integer> accountTotals = new HashMap<String, Integer>();
```

C. Replace line 13 with
```
private Map<String<Integer>> accountTotals = new HashMap<String<Integer>>();
```

D. Replace lines 17–20 with
```
int total = accountTotals.get(accountName);
if (total == null)
    total = 0;
return total;
```

E. Replace lines 17–20 with
```
Integer total = accountTotals.get(accountName);
if (total == null)
    total = 0;
return total;
```

F. Replace lines 17–20 with
```
return accountTotals.get(accountName);
```

G. Replace line 24 with
```
accountTotals.put(accountName, amount);
```

H. Replace line 24 with
```
accountTotals.put(accountName, amount.intValue());
```

6. Given:

```
interface Hungry<E> { void munch(E x); }
interface Carnivore<E extends Animal> extends Hungry<E> {}
interface Herbivore<E extends Plant> extends Hungry<E> {}
abstract class Plant {}
class Grass extends Plant {}
abstract class Animal {}
class Sheep extends Animal implements Herbivore<Sheep> {
  public void munch(Sheep x) {}
}
class Wolf extends Animal implements Carnivore<Sheep> {
  public void munch(Sheep x) {}
}
```

Which of the following changes (taken separately) would allow this code to compile? (Choose all that apply.)

A. Change the Carnivore interface to
```
interface Carnivore<E extends Plant> extends Hungry<E> {}
```

B. Change the Herbivore interface to
```
interface Herbivore<E extends Animal> extends Hungry<E> {}
```

C. Change the Sheep class to
```
class Sheep extends Animal implements Herbivore<Plant> {
    public void munch(Grass x) {}
}
```

D. Change the Sheep class to
```
class Sheep extends Plant implements Carnivore<Wolf> {
    public void munch(Wolf x) {}
}
```

E. Change the Wolf class to

```
class Wolf extends Animal implements Herbivore<Grass> {
    public void munch(Grass x) {}
}
```

F. No changes are necessary

7. Which collection class(es) allows you to grow or shrink its size and provides indexed access to its elements, but whose methods are not synchronized? (Choose all that apply.)

A. `java.util.HashSet`

B. `java.util.LinkedHashSet`

C. `java.util.List`

D. `java.util.ArrayList`

E. `java.util.Vector`

F. `java.util.PriorityQueue`

8. Given a method declared as

```
public static <E extends Number> List<E> process(List<E> nums)
```

A programmer wants to use this method like this

```
// INSERT DECLARATIONS HERE

output = process(input);
```

Which pairs of declarations could be placed at `// INSERT DECLARATIONS HERE` to allow the code to compile? (Choose all that apply.)

A.
```
ArrayList<Integer> input = null;
ArrayList<Integer> output = null;
```

B.
```
ArrayList<Integer> input = null;
List<Integer> output = null;
```

C.
```
ArrayList<Integer> input = null;
List<Number> output = null;
```

D.
```
List<Number> input = null;
ArrayList<Integer> output = null;
```

E.
```
List<Number> input = null;
List<Number> output = null;
```

F.
```
List<Integer> input = null;
List<Integer> output = null;
```

G. None of the above

9. Given the proper import statement(s), and

```
13.    PriorityQueue<String> pq = new PriorityQueue<String>();
14.    pq.add("2");
15.    pq.add("4");
16.    System.out.print(pq.peek() + " ");
17.    pq.offer("1");
18.    pq.add("3");
19.    pq.remove("1");
20.    System.out.print(pq.poll() + " ");
21.    if(pq.remove("2")) System.out.print(pq.poll() + " ");
22.    System.out.println(pq.poll() + " " + pq.peek());
```

What is the result?

A. 2 2 3 3

B. 2 2 3 4

C. 4 3 3 4

D. 2 2 3 3 3

E. 4 3 3 3 3

F. 2 2 3 3 4

G. Compilation fails

H. An exception is thrown at runtime

10. Given:

```
3. import java.util.*;
4. public class Mixup {
5.   public static void main(String[] args) {
```

```
 6.     Object o = new Object();
 7.     // insert code here
 8.     s.add("o");
 9.     s.add(o);
10.   }
11. }
```

And these three fragments:

```
I.   Set s = new HashSet();
II.  TreeSet s = new TreeSet();
III. LinkedHashSet s = new LinkedHashSet();
```

When fragments I, II, or III are inserted, independently, at line 7, which are true? (Choose all that apply.)

A. Fragment I compiles

B. Fragment II compiles

C. Fragment III compiles

D. Fragment I executes without exception

E. Fragment II executes without exception

F. Fragment III executes without exception

11. Given:

```
 3. import java.util.*;
 4. class Turtle {
 5.   int size;
 6.   public Turtle(int s) { size = s; }
 7.   public boolean equals(Object o) { return (this.size == ((Turtle)o).size); }
 8.   // insert code here
 9. }
10. public class TurtleTest {
11.   public static void main(String[] args) {
12.     LinkedHashSet<Turtle> t = new LinkedHashSet<Turtle>();
13.     t.add(new Turtle(1));   t.add(new Turtle(2));   t.add(new Turtle(1));
14.     System.out.println(t.size());
15.   }
16. }
```

And these two fragments:

```
I.   public int hashCode() { return size/5; }
II.  // no hashCode method declared
```

If fragment I or II is inserted, independently, at line 8, which are true? (Choose all that apply.)

A. If fragment I is inserted, the output is 2
B. If fragment I is inserted, the output is 3
C. If fragment II is inserted, the output is 2
D. If fragment II is inserted, the output is 3
E. If fragment I is inserted, compilation fails
F. If fragment II is inserted, compilation fails

12. Given the proper import statement(s), and:

```
13.    TreeSet<String> s = new TreeSet<String>();
14.    TreeSet<String> subs = new TreeSet<String>();
15.    s.add("a"); s.add("b"); s.add("c"); s.add("d"); s.add("e");
16.
17.    subs = (TreeSet)s.subSet("b", true, "d", true);
18.    s.add("g");
19.    s.pollFirst();
20.    s.pollFirst();
21.    s.add("c2");
22.    System.out.println(s.size() +" "+ subs.size());
```

Which are true? (Choose all that apply.)

A. The size of `s` is 4
B. The size of `s` is 5
C. The size of `s` is 7
D. The size of `subs` is 1
E. The size of `subs` is 2
F. The size of `subs` is 3
G. The size of `subs` is 4
H. An exception is thrown at runtime

13. Given:

```
 3. import java.util.*;
 4. public class Magellan {
 5.   public static void main(String[] args) {
 6.     TreeMap<String, String> myMap = new TreeMap<String, String>();
 7.     myMap.put("a", "apple"); myMap.put("d", "date");
 8.     myMap.put("f", "fig"); myMap.put("p", "pear");
 9.     System.out.println("1st after mango: " +   // sop 1
10.       myMap.higherKey("f"));
11.     System.out.println("1st after mango: " +   // sop 2
12.       myMap.ceilingKey("f"));
13.     System.out.println("1st after mango: " +   // sop 3
14.       myMap.floorKey("f"));
15.     SortedMap<String, String> sub = new TreeMap<String, String>();
16.     sub = myMap.tailMap("f");
17.     System.out.println("1st after mango: " +   // sop 4
18.       sub.firstKey());
19.   }
20. }
```

Which of the System.out.println statements will produce the output `1st after mango: p`? (Choose all that apply.)

A. `sop 1`

B. `sop 2`

C. `sop 3`

D. `sop 4`

E. None; compilation fails

F. None; an exception is thrown at runtime

14. Given:

```
 3. import java.util.*;
 4. class Business { }
 5. class Hotel extends Business { }
 6. class Inn extends Hotel { }
 7. public class Travel {
 8.   ArrayList<Hotel> go() {
 9.     // insert code here
10.   }
11. }
```

Which, inserted independently at line 9, will compile? (Choose all that apply.)

A. `return new ArrayList<Inn>();`

B. `return new ArrayList<Hotel>();`

C. `return new ArrayList<Object>();`

D. `return new ArrayList<Business>();`

15. Given:

```
3. import java.util.*;
4. class Dog { int size; Dog(int s) { size = s; } }
5. public class FirstGrade {
6.   public static void main(String[] args) {
7.     TreeSet<Integer> i = new TreeSet<Integer>();
8.     TreeSet<Dog> d = new TreeSet<Dog>();
9.
10.    d.add(new Dog(1));   d.add(new Dog(2));   d.add(new Dog(1));
11.    i.add(1);            i.add(2);            i.add(1);
12.    System.out.println(d.size() + " " + i.size());
13.  }
14. }
```

What is the result?

A. 1 2

B. 2 2

C. 2 3

D. 3 2

E. 3 3

F. Compilation fails

G. An exception is thrown at runtime

16. Given:

```
3. import java.util.*;
4. public class GeoCache {
5.   public static void main(String[] args) {
6.     String[] s = {"map", "pen", "marble", "key"};
7.     Othello o = new Othello();
```

```
 8.     Arrays.sort(s,o);
 9.     for(String s2: s) System.out.print(s2 + " ");
10.     System.out.println(Arrays.binarySearch(s, "map"));
11.   }
12.   static class Othello implements Comparator<String> {
13.     public int compare(String a, String b) { return b.compareTo(a); }
14.   }
15. }
```

Which are true? (Choose all that apply.)

A. Compilation fails

B. The output will contain a 1

C. The output will contain a 2

D. The output will contain a –1

E. An exception is thrown at runtime

F. The output will contain `"key map marble pen"`

G. The output will contain `"pen marble map key"`

SELF TEST ANSWERS

1. Given:

```
public static void main(String[] args) {
  // INSERT DECLARATION HERE
  for (int i = 0; i <= 10; i++) {
    List<Integer> row = new ArrayList<Integer>();
    for (int j = 0; j <= 10; j++)
      row.add(i * j);
    table.add(row);
  }
  for (List<Integer> row : table)
    System.out.println(row);
  }
```

Which statements could be inserted at `// INSERT DECLARATION HERE` to allow this code to compile and run? (Choose all that apply.)

A. `List<List<Integer>> table = new List<List<Integer>>();`

B. `List<List<Integer>> table = new ArrayList<List<Integer>>();`

C. `List<List<Integer>> table = new ArrayList<ArrayList<Integer>>();`

D. `List<List, Integer> table = new List<List, Integer>();`

E. `List<List, Integer> table = new ArrayList<List, Integer>();`

F. `List<List, Integer> table = new ArrayList<ArrayList, Integer>();`

G. None of the above

Answer:

☑ **B** is correct.

☒ **A** is incorrect because List is an interface, so you can't say `new List()` regardless of any generic types. **D, E,** and **F** are incorrect because List only takes one type parameter (a Map would take two, not a List). **C** is tempting, but incorrect. The type argument `<List<Integer>>` must be the same for both sides of the assignment, even though the constructor `new ArrayList()` on the right side is a subtype of the declared type List on the left. (Objective 6.4)

2. Which statements are true about comparing two instances of the same class, given that the `equals()` and `hashCode()` methods have been properly overridden? (Choose all that apply.)

A. If the `equals()` method returns true, the `hashCode()` comparison `==` might return false

B. If the `equals()` method returns false, the `hashCode()` comparison `==` might return true

C. If the `hashCode()` comparison `==` returns true, the `equals()` method must return true

D. If the `hashCode()` comparison `==` returns true, the `equals()` method might return true

E. If the `hashCode()` comparison `!=` returns true, the `equals()` method might return true

Answer:

☑ **B** and **D**. **B** is true because often two dissimilar objects can return the same hashcode value. **D** is true because if the `hashCode()` comparison returns ==, the two objects might or might not be equal.

☒ **A, C,** and **E** are incorrect. **C** is incorrect because the `hashCode()` method is very flexible in its return values, and often two dissimilar objects can return the same hash code value. **A** and **E** are a negation of the `hashCode()` and `equals()` contract. (Objective 6.2)

3. Given:

```
public static void before() {
  Set set = new TreeSet();
  set.add("2");
  set.add(3);
  set.add("1");
  Iterator it = set.iterator();
    while (it.hasNext())
  System.out.print(it.next() + " ");
}
```

Which statements are true?

A. The `before()` method will print `1 2`

B. The `before()` method will print `1 2 3`

C. The `before()` method will print three numbers, but the order cannot be determined

D. The `before()` method will not compile

E. The `before()` method will throw an exception at runtime

Answer:

☑ **E** is correct. You can't put both Strings and `int`s into the same TreeSet. Without generics, the compiler has no way of knowing what type is appropriate for this TreeSet, so it allows everything to compile. At runtime, the TreeSet will try to sort the elements as they're added, and when it tries to compare an Integer with a String it will throw a ClassCastException. Note that although the `before()` method does not use generics, it does use autoboxing. Watch out for code that uses some new features and some old features mixed together.

☒ **A, B, C,** and **D** are incorrect based on the above. (Objective 6.5)

4. Given:

```
import java.util.*;
class MapEQ {
  public static void main(String[] args) {
    Map<ToDos, String> m = new HashMap<ToDos, String>();
    ToDos t1 = new ToDos("Monday");
    ToDos t2 = new ToDos("Monday");
    ToDos t3 = new ToDos("Tuesday");
    m.put(t1, "doLaundry");
    m.put(t2, "payBills");
    m.put(t3, "cleanAttic");
    System.out.println(m.size());
} }
class ToDos{
  String day;
  ToDos(String d) { day = d; }
  public boolean equals(Object o) {
    return ((ToDos)o).day == this.day;
  }
  // public int hashCode() { return 9; }
}
```

Which is correct? (Choose all that apply.)

A. As the code stands it will not compile

B. As the code stands the output will be `2`

C. As the code stands the output will be `3`

D. If the `hashCode()` method is uncommented the output will be `2`

E. If the `hashCode()` method is uncommented the output will be `3`

F. If the `hashCode()` method is uncommented the code will not compile

Answer:

☑ **C** and **D** are correct. If `hashCode()` is not overridden then every entry will go into its own bucket, and the overridden `equals()` method will have no effect on determining equivalency. If `hashCode()` is overridden, then the overridden `equals()` method will view `t1` and `t2` as duplicates.

☒ **A, B, E,** and **F** are incorrect based on the above. (Objective 6.2)

5. Given:

```
12. public class AccountManager {
13.     private Map accountTotals = new HashMap();
14.     private int retirementFund;
15.
16.     public int getBalance(String accountName) {
17.         Integer total = (Integer) accountTotals.get(accountName);
18.         if (total == null)
19.             total = Integer.valueOf(0);
20.         return total.intValue();
21.     }
23.     public void setBalance(String accountName, int amount) {
24.         accountTotals.put(accountName, Integer.valueOf(amount));
25.  }  }
```

This class is to be updated to make use of appropriate generic types, with no changes in behavior (for better or worse). Which of these steps could be performed? (Choose three.)

A. Replace line 13 with
```
private Map<String, int> accountTotals = new HashMap<String, int>();
```

B. Replace line 13 with
```
private Map<String, Integer> accountTotals = new HashMap<String, Integer>();
```

C. Replace line 13 with
```
private Map<String<Integer>> accountTotals = new HashMap<String<Integer>>();
```

D. Replace lines 17–20 with
```
int total = accountTotals.get(accountName);
if (total == null)  total = 0;
return total;
```

E. Replace lines 17–20 with
```
Integer total = accountTotals.get(accountName);
if (total == null)  total = 0;
return total;
```

F. Replace lines 17–20 with
```
return accountTotals.get(accountName);
```
G. Replace line 24 with
```
accountTotals.put(accountName, amount);
```
H. Replace line 24 with
```
accountTotals.put(accountName, amount.intValue());
```

Answer:

☑ **B, E,** and **G** are correct.

☒ **A** is wrong because you can't use a primitive type as a type parameter. **C** is wrong because a Map takes two type parameters separated by a comma. **D** is wrong because an int can't autobox to a null, and **F** is wrong because a `null` can't unbox to 0. **H** is wrong because you can't autobox a primitive just by trying to invoke a method with it. (Objective 6.4)

6. Given:

```
interface Hungry<E> { void munch(E x); }
interface Carnivore<E extends Animal> extends Hungry<E> {}
interface Herbivore<E extends Plant> extends Hungry<E> {}
abstract class Plant {}
class Grass extends Plant {}
abstract class Animal {}
class Sheep extends Animal implements Herbivore<Sheep> {
  public void munch(Sheep x) {}
}
class Wolf extends Animal implements Carnivore<Sheep> {
  public void munch(Sheep x) {}
}
```

Which of the following changes (taken separately) would allow this code to compile? (Choose all that apply.)

A. Change the Carnivore interface to
```
interface Carnivore<E extends Plant> extends Hungry<E> {}
```
B. Change the Herbivore interface to
```
interface Herbivore<E extends Animal> extends Hungry<E> {}
```
C. Change the Sheep class to
```
class Sheep extends Animal implements Herbivore<Plant> {
    public void munch(Grass x) {}
}
```

D. Change the Sheep class to

```
class Sheep extends Plant implements Carnivore<Wolf> {
    public void munch(Wolf x) {}
}
```

E. Change the Wolf class to

```
class Wolf extends Animal implements Herbivore<Grass> {
    public void munch(Grass x) {}
}
```

F. No changes are necessary

Answer:

☑ **B** is correct. The problem with the original code is that Sheep tries to implement `Herbivore<Sheep>` and Herbivore declares that its type parameter `E` can be any type that extends Plant. Since a Sheep is not a Plant, `Herbivore<Sheep>` makes no sense—the type Sheep is outside the allowed range of Herbivore's parameter `E`. Only solutions that either alter the definition of a Sheep or alter the definition of Herbivore will be able to fix this. So **A, E,** and **F** are eliminated. **B** works, changing the definition of an Herbivore to allow it to eat Sheep solves the problem. **C** doesn't work because an `Herbivore<Plant>` must have a `munch(Plant)` method, not `munch(Grass)`. And **D** doesn't work, because in **D** we made Sheep extend Plant, now the Wolf class breaks because its `munch(Sheep)` method no longer fulfills the contract of Carnivore. (Objective 6.4)

7. Which collection class(es) allows you to grow or shrink its size and provides indexed access to its elements, but whose methods are not synchronized? (Choose all that apply.)

A. `java.util.HashSet`
B. `java.util.LinkedHashSet`
C. `java.util.List`
D. `java.util.ArrayList`
E. `java.util.Vector`
F. `java.util.PriorityQueue`

Answer:

☑ **D** is correct. All of the collection classes allow you to grow or shrink the size of your collection. `ArrayList` provides an index to its elements. The newer collection classes tend not to have synchronized methods. Vector is an older implementation of `ArrayList` functionality and has synchronized methods; it is slower than `ArrayList`.

☒ **A, B, C, E,** and **F** are incorrect based on the logic described above; Notes: **C**, List is an interface, and **F**, `PriorityQueue` does not offer access by index. (Objective 6.1)

8. Given a method declared as

```
public static <E extends Number> List<E> process(List<E> nums)
```

A programmer wants to use this method like this

```
// INSERT DECLARATIONS HERE
output = process(input);
```

Which pairs of declarations could be placed at `// INSERT DECLARATIONS HERE` to allow the code to compile? (Choose all that apply.)

A.
```
ArrayList<Integer> input = null;
ArrayList<Integer> output = null;
```

B.
```
ArrayList<Integer> input = null;
List<Integer> output = null;
```

C.
```
ArrayList<Integer> input = null;
List<Number> output = null;
```

D.
```
List<Number> input = null;
ArrayList<Integer> output = null;
```

E.
```
List<Number> input = null;
List<Number> output = null;
```

F.
```
List<Integer> input = null;
List<Integer> output = null;
```

G. None of the above

Answer:

☑ **B, E,** and **F** are correct.

☒ The return type of process is definitely declared as a `List`, not an `ArrayList`, so **A** and **D** are wrong. **C** is wrong because the return type evaluates to `List<Integer>`, and that can't be assigned to a variable of type `List<Number>`. Of course all these would probably cause a `NullPointerException` since the variables are still null—but the question only asked us to get the code to compile. (Objective 6.4)

9. Given the proper import statement(s), and

```
13.    PriorityQueue<String> pq = new PriorityQueue<String>();
14.    pq.add("2");
15.    pq.add("4");
```

```
16.    System.out.print(pq.peek() + " ");
17.    pq.offer("1");
18.    pq.add("3");
19.    pq.remove("1");
20.    System.out.print(pq.poll() + " ");
21.    if(pq.remove("2")) System.out.print(pq.poll() + " ");
22.    System.out.println(pq.poll() + " " + pq.peek());
```

What is the result?

A. 2 2 3 3

B. 2 2 3 4

C. 4 3 3 4

D. 2 2 3 3 3

E. 4 3 3 3 3

F. 2 2 3 3 4

G. Compilation fails

H. An exception is thrown at runtime

Answer:

☑ **B** is correct. For the sake of the exam, `add()` and `offer()` both add to (in this case), naturally sorted queues. The calls to `poll()` both return and then remove the first item from the queue, so the if test fails.

☒ **A, C, D, E, F, G,** and **H** are incorrect based on the above. (Objective 6.1)

10. Given:

```
3. import java.util.*;
4. public class Mixup {
5.   public static void main(String[] args) {
6.     Object o = new Object();
7.     // insert code here
8.     s.add("o");
9.     s.add(o);
10.  }
11. }
```

And these three fragments:

```
I.   Set s = new HashSet();
II.  TreeSet s = new TreeSet();
III. LinkedHashSet s = new LinkedHashSet();
```

When fragments I, II, or III are inserted, independently, at line 7, which are true? (Choose all that apply.)

A. Fragment I compiles

B. Fragment II compiles

C. Fragment III compiles

D. Fragment I executes without exception

E. Fragment II executes without exception

F. Fragment III executes without exception

Answer:

☑ **A, B, C, D,** and **F** are all correct.

☒ Only **E** is incorrect. Elements of a TreeSet must in some way implement `Comparable`. (Objective 6.1)

11. Given:

```
3. import java.util.*;
4. class Turtle {
5.   int size;
6.   public Turtle(int s) { size = s; }
7.   public boolean equals(Object o) { return (this.size == ((Turtle)o).size); }
8.   // insert code here
9. }
10. public class TurtleTest {
11.   public static void main(String[] args) {
12.     LinkedHashSet<Turtle> t = new LinkedHashSet<Turtle>();
13.     t.add(new Turtle(1));   t.add(new Turtle(2));   t.add(new Turtle(1));
14.     System.out.println(t.size());
15.   }
16. }
```

And these two fragments:

```
I.   public int hashCode() { return size/5; }
II.  // no hashCode method declared
```

If fragment I or II is inserted, independently, at line 8, which are true? (Choose all that apply.)

A. If fragment I is inserted, the output is 2

B. If fragment I is inserted, the output is 3

C. If fragment II is inserted, the output is 2

D. If fragment II is inserted, the output is 3

E. If fragment I is inserted, compilation fails

F. If fragment II is inserted, compilation fails

Answer:

☑ **A** and **D** are correct. While fragment II wouldn't fulfill the `hashCode()` contract (as you can see by the results), it is legal Java. For the purpose of the exam, if you don't override `hashCode()`, every object will have a unique hashcode.

☒ **B, C, E,** and **F** are incorrect based on the above. (Objective 6.2)

12. Given the proper import statement(s), and:

```
13.    TreeSet<String> s = new TreeSet<String>();
14.    TreeSet<String> subs = new TreeSet<String>();
15.    s.add("a"); s.add("b"); s.add("c"); s.add("d"); s.add("e");
16.
17.    subs = (TreeSet)s.subSet("b", true, "d", true);
18.    s.add("g");
19.    s.pollFirst();
20.    s.pollFirst();
21.    s.add("c2");
22.    System.out.println(s.size() +" "+ subs.size());
```

Which are true? (Choose all that apply.)

A. The size of `s` is 4

B. The size of `s` is 5

C. The size of `s` is 7

D. The size of `subs` is 1

E. The size of `subs` is 2

F. The size of `subs` is 3

G. The size of `subs` is 4

H. An exception is thrown at runtime

Answer:

☑ **B** and **F** are correct. After `"g"` is added, TreeSet `s` contains six elements and TreeSet `subs` contains three (`b`, `c`, `d`), because `"g"` is out of the range of `subs`. The first `pollFirst()` finds and removes only the `"a"`. The second `pollFirst()` finds and removes the `"b"` from *both* TreeSets (remember they are backed). The final `add()` is in range of both TreeSets. The final contents are [c,c2,d,e,g] and [c,c2,d].

☒ **A, C, D, E, G,** and **H** are incorrect based on the above. (Objective 6.3)

13. Given:

```
 3. import java.util.*;
 4. public class Magellan {
 5.   public static void main(String[] args) {
 6.     TreeMap<String, String> myMap = new TreeMap<String, String>();
 7.     myMap.put("a", "apple"); myMap.put("d", "date");
 8.     myMap.put("f", "fig"); myMap.put("p", "pear");
 9.     System.out.println("1st after mango: " +   // sop 1
10.       myMap.higherKey("f"));
11.     System.out.println("1st after mango: " +   // sop 2
12.       myMap.ceilingKey("f"));
13.     System.out.println("1st after mango: " +   // sop 3
14.       myMap.floorKey("f"));
15.     SortedMap<String, String> sub = new TreeMap<String, String>();
16.     sub = myMap.tailMap("f");
17.     System.out.println("1st after mango: " +   // sop 4
18.       sub.firstKey());
19.   }
20. }
```

Which of the System.out.println statements will produce the output `1st after mango: p`? (Choose all that apply.)

A. `sop 1`

B. `sop 2`

C. `sop 3`

D. `sop 4`

E. None; compilation fails

F. None; an exception is thrown at runtime

Answer:

☑ **A** is correct. The `ceilingKey()` method's argument is inclusive. The `floorKey()` method would be used to find keys before the specified key. The `firstKey()` method's argument is also inclusive.

☒ **B, C, D, E,** and **F** are incorrect based on the above. (Objective 6.3)

14. Given:

```
 3. import java.util.*;
 4. class Business { }
 5. class Hotel extends Business { }
 6. class Inn extends Hotel { }
 7. public class Travel {
 8.   ArrayList<Hotel> go() {
 9.     // insert code here
10.   }
11. }
```

Which, inserted independently at line 9, will compile? (Choose all that apply.)

A. `return new ArrayList<Inn>();`

B. `return new ArrayList<Hotel>();`

C. `return new ArrayList<Object>();`

D. `return new ArrayList<Business>();`

Answer:

☑ **B** is correct.

☒ **A** is incorrect because polymorphic assignments don't apply to generic type parameters. **C** and **D** are incorrect because they don't follow basic polymorphism rules. (Objective 6.4)

15. Given:

```
 3. import java.util.*;
 4. class Dog { int size; Dog(int s) { size = s; } }
 5. public class FirstGrade {
 6.   public static void main(String[] args) {
 7.     TreeSet<Integer> i = new TreeSet<Integer>();
 8.     TreeSet<Dog> d = new TreeSet<Dog>();
 9.
10.     d.add(new Dog(1));   d.add(new Dog(2));   d.add(new Dog(1));
11.     i.add(1);            i.add(2);            i.add(1);
12.     System.out.println(d.size() + " " + i.size());
13.   }
14. }
```

What is the result?

A. 1 2

B. 2 2

C. 2 3

D. 3 2

E. 3 3

F. Compilation fails

G. An exception is thrown at runtime

Answer:

☑ **G** is correct. Class `Dog` needs to implement `Comparable` in order for a TreeSet (which keeps its elements sorted) to be able to contain `Dog` objects.

☒ **A, B, C, D, E,** and **F** are incorrect based on the above. (Objective 6.5)

16. Given:

```
3. import java.util.*;
4. public class GeoCache {
5.   public static void main(String[] args) {
6.     String[] s = {"map", "pen", "marble", "key"};
7.     Othello o = new Othello();
8.     Arrays.sort(s,o);
```

```
 9.     for(String s2: s) System.out.print(s2 + " ");
10.     System.out.println(Arrays.binarySearch(s, "map"));
11.   }
12.   static class Othello implements Comparator<String> {
13.     public int compare(String a, String b) { return b.compareTo(a); }
14.   }
15. }
```

Which are true? (Choose all that apply.)

A. Compilation fails

B. The output will contain a 1

C. The output will contain a 2

D. The output will contain a –1

E. An exception is thrown at runtime

F. The output will contain `"key map marble pen"`

G. The output will contain `"pen marble map key"`

Answer:

☑ **D** and **G** are correct. First, the `compareTo()` method will reverse the normal sort. Second, the `sort()` is valid. Third, the `binarySearch()` gives –1 because it needs to be invoked using the same `Comparator` (`o`), as was used to sort the array. Note that when the `binarySearch()` returns an "undefined result" it doesn't officially have to be a -1, but it usually is, so if you selected only G, you get full credit!

☒ **A, B, C, E,** and **F** are incorrect based on the above. (Objective 6.5)

8

Inner Classes

CERTIFICATION OBJECTIVES

- Inner Classes
- Method-Local Inner Classes
- Anonymous Inner Classes
- Static Nested Classes

✓ Two-Minute Drill

Q&A Self Test

Inner classes (including static nested classes) appear throughout the exam. Although there are no official exam objectives specifically about inner classes, Objective 1.1 includes inner (a.k.a. nested) classes. More important, the code used to represent questions on virtually *any* topic on the exam can involve inner classes. Unless you deeply understand the rules and syntax for inner classes, you're likely to miss questions you'd otherwise be able to answer. *As if the exam weren't already tough enough.*

This chapter looks at the ins and outs (inners and outers?) of inner classes, and exposes you to the kinds of (often strange-looking) syntax examples you'll see scattered throughout the entire exam. So you've really got two goals for this chapter—to learn what you'll need to answer questions testing your inner class knowledge, and to learn how to read and understand inner class code so that you can correctly process questions testing your knowledge of *other* topics.

So what's all the hoopla about inner classes? Before we get into it, we have to warn you (if you don't already know) that inner classes have inspired passionate love 'em or hate 'em debates since first introduced in version 1.1 of the language. For once, we're going to try to keep our opinions to ourselves here and just present the facts as you'll need to know them for the exam. It's up to you to decide how—and to what extent—you should use inner classes in your own development. We mean it. We believe they have some powerful, efficient uses in very specific situations, including code that's easier to read and maintain, but they can also be abused and lead to code that's as clear as a cornfield maze, and to the syndrome known as "reuseless": *code that's useless over and over again*.

Inner classes let you define one class within another. They provide a type of scoping for your classes since you can make one class *a member of another class*. Just as classes have member *variables* and *methods*, a class can also have member *classes*. They come in several flavors, depending on how and where you define the inner class, including a special kind of inner class known as a "top-level nested class" (an inner class marked `static`), which technically isn't really an inner class. Because a static nested class is still a class defined within the scope of another class, we're still going to cover them in this chapter on inner classes.

Unlike the other chapters in this book, the certification objectives for inner classes don't have official exam objective numbers since they're part of other objectives covered elsewhere. So for this chapter, the Certification Objective headings in the following list represent the four inner class *topics* discussed in this chapter, rather than four official exam *objectives*:

- Inner classes
- Method-local inner classes
- Anonymous inner classes
- Static nested classes

CERTIFICATION OBJECTIVE

Inner Classes

You're an OO programmer, so you know that for reuse and flexibility/extensibility you need to keep your classes specialized. In other words, a class should have code *only* for the things an object of that particular type needs to do; any *other* behavior should be part of another class better suited for *that* job. Sometimes, though, you find yourself designing a class where you discover you need behavior that belongs in a separate, specialized class, but also needs to be intimately tied to the class you're designing.

Event handlers are perhaps the best example of this (and are, in fact, one of the main reasons inner classes were added to the language in the first place). If you have a GUI class that performs some job like, say, a chat client, you might want the chat-client–specific methods (accept input, read new messages from server, send user input back to server, and so on) to be in the class. But how do those methods get invoked in the first place? A user clicks a button. Or types some text in the input field. Or a separate thread doing the I/O work of getting messages from the server has messages that need to be displayed in the GUI. So you have chat-client–specific methods, but you also need methods for handling the "events" (button presses, keyboard typing, I/O available, and so on) that drive the calls on those chat-client methods. The ideal scenario—from an OO perspective—is to keep the chat-client–specific methods in the ChatClient class, and put the event-handling *code* in a separate event-handling *class*.

Nothing unusual about that so far; after all, that's how you're *supposed* to design OO classes. As *specialists*. But here's the problem with the chat-client scenario: the event-handling code is intimately tied to the chat-client–specific code! Think about it: when the user presses a Send button (indicating that they want their typed-in message to be sent to the chat server), the chat-client code that sends the message needs to read from a *particular* text field. In other words, if the user clicks Button A, the program is supposed to extract the text from the TextField B, *of a particular*

ChatClient instance. Not from some *other* text field from some *other* object, but specifically the text field that a specific instance of the ChatClient class has a reference to. So the event-handling code needs access to the members of the ChatClient object, to be useful as a "helper" to a particular ChatClient instance.

And what if the ChatClient class needs to inherit from one class, but the event handling code is better off inheriting from some *other* class? You can't make a class extend more than one class, so putting all the code (the chat-client– specific code and the event-handling code) in one class won't work in that case. So what you'd really like to have is the benefit of putting your event code in a separate class (better OO, encapsulation, and the ability to extend a class other than the class the ChatClient extends) but still allow the event-handling code to have easy access to the members of the ChatClient (so the event-handling code can, for example, update the ChatClient's private instance variables). You *could* manage it by making the members of the ChatClient accessible to the event-handling class by, for example, marking them `public`. But that's not a good solution either.

You already know where this is going—one of the key benefits of an inner class is the "special relationship" an *inner class instance* shares with *an instance of the outer class*. That "special relationship" gives code in the inner class access to members of the enclosing (outer) class, *as if the inner class were part of the outer class*. In fact, that's exactly what it means: the inner class *is* a part of the outer class. Not just a "part" but a full-fledged, card-carrying *member* of the outer class. Yes, an inner class instance has access to all members of the outer class, *even those marked private*. (Relax, that's the whole point, remember? We want this separate inner class instance to have an intimate relationship with the outer class instance, but we still want to keep everyone *else* out. And besides, if you wrote the outer class, then you also wrote the inner class! So you're not violating encapsulation; you *designed* it this way.)

Coding a "Regular" Inner Class

We use the term *regular* here to represent inner classes that are not

- Static
- Method-local
- Anonymous

For the rest of this section, though, we'll just use the term "inner class" and drop the "regular". (When we switch to one of the other three types in the preceding list, you'll know it.) You define an inner class within the curly braces of the outer class:

```
class MyOuter {
   class MyInner { }
}
```

Piece of cake. And if you compile it,

```
%javac MyOuter.java
```

you'll end up with *two* class files:

MyOuter.class
MyOuter$MyInner.class

The inner class is still, in the end, a separate class, so a separate class file is generated for it. But the inner class file isn't accessible to you in the usual way. You can't say

```
%java MyOuter$MyInner
```

in hopes of running the `main()` method of the inner class, because a *regular* inner class can't have static declarations of any kind. *The only way you can access the inner class is through a live instance of the outer class!* In other words, only at runtime when there's already an instance of the outer class to tie the inner class instance to. You'll see all this in a moment. First, let's beef up the classes a little:

```
class MyOuter {
   private int x = 7;

   // inner class definition
   class MyInner {
      public void seeOuter() {
         System.out.println("Outer x is " + x);
      }
   } // close inner class definition

} // close outer class
```

The preceding code is perfectly legal. Notice that the inner class is indeed accessing a private member of the outer class. That's fine, because the inner class is also a member of the outer class. So just as any member of the outer class (say, an instance method) can access any other member of the outer class, `private` or not, the inner class—also a member—can do the same.

OK, so now that we know how to write the code giving an inner class access to members of the outer class, how do you actually use it?

Instantiating an Inner Class

To create an instance of an inner class, *you must have an instance of the outer class* to tie to the inner class. There are no exceptions to this rule: an inner class instance can never stand alone without a direct relationship to an instance of the outer class.

Instantiating an Inner Class from Within the Outer Class Most often, it is the outer class that creates instances of the inner class, since it is usually the outer class wanting to use the inner instance as a helper for its own personal use. We'll modify the MyOuter class to create an instance of MyInner:

```
class MyOuter {
   private int x = 7;
   public void makeInner() {
      MyInner in = new MyInner();  // make an inner instance
      in.seeOuter();
   }

   class MyInner {
      public void seeOuter() {
         System.out.println("Outer x is " + x);
      }
   }
}
```

You can see in the preceding code that the MyOuter code treats MyInner just as though MyInner were any other accessible class—it instantiates it using the class name (`new MyInner()`), and then invokes a method on the reference variable (`in.seeOuter()`). But the only reason this syntax works is because the outer class instance method code is doing the instantiating. In other words, *there's already an instance of the outer class—the instance running the* `makeInner()` *method*. So how do you instantiate a MyInner object from somewhere outside the MyOuter class? Is it even possible? (Well, since we're going to all the trouble of making a whole new subhead for it, as you'll see next, there's no big mystery here.)

Creating an Inner Class Object from Outside the Outer Class Instance Code Whew. Long subhead there, but it does explain what we're trying to do. If we want to create an instance of the inner class, we must have an instance of the outer class. You already know that, but think about the

implications…it means that, without a reference to an instance of the outer class, you can't instantiate the inner class from a `static` method of the outer class (because, don't forget, in `static` code *there is no* `this` *reference*), or from any other code in any other class. Inner class instances are always handed an implicit reference to the outer class. The compiler takes care of it, so you'll never see anything but the end result—the ability of the inner class to access members of the outer class. The code to make an instance from anywhere outside nonstatic code of the outer class is simple, but you must memorize this for the exam!

```
public static void main(String[] args) {
  MyOuter mo = new MyOuter();     // gotta get an instance!
  MyOuter.MyInner inner = mo.new MyInner();
  inner.seeOuter();
}
```

The preceding code is the same regardless of whether the `main()` method is within the MyOuter class or some *other* class (assuming the other class has access to MyOuter, and since MyOuter has default access, that means the code must be in a class within the same package as MyOuter).

If you're into one-liners, you can do it like this:

```
public static void main(String[] args) {
  MyOuter.MyInner inner = new MyOuter().new MyInner();
  inner.seeOuter();
}
```

You can think of this as though you're invoking a method on the outer instance, but the method happens to be a special inner class instantiation method, and it's invoked using the keyword `new`. Instantiating an inner class is the *only* scenario in which you'll invoke `new` *on* an instance as opposed to invoking `new` to *construct* an instance.

Here's a quick summary of the differences between inner class instantiation code that's *within* the outer class (but not `static`), and inner class instantiation code that's *outside* the outer class:

- From *inside* the outer class instance code, use the inner class name in the normal way:

```
MyInner mi = new MyInner();
```

- From *outside* the outer class instance code (including static method code within the outer class), the inner class name must now include the outer class's name:

```
MyOuter.MyInner
```

  To instantiate it, you must use a reference to the outer class:

```
new MyOuter().new MyInner(); or outerObjRef.new MyInner();
```

  if you already have an instance of the outer class.

Referencing the Inner or Outer Instance from Within the Inner Class

How does an object refer to itself normally? By using the `this` reference. Here is a quick review of `this`:

- The keyword `this` can be used only from within instance code. In other words, not within `static` code.
- The `this` reference is a reference to the currently executing object. In other words, the object whose reference was used to invoke the currently running method.
- The `this` reference is the way an object can pass a reference to itself to some other code, as a method argument:

```
public void myMethod() {
 MyClass mc = new MyClass();
 mc.doStuff(this);  // pass a ref to object running myMethod
}
```

Within an inner class code, the `this` reference refers to the instance of the inner class, as you'd probably expect, since `this` always refers to the currently executing object. But what if the inner class code wants an explicit reference to the outer class instance that the inner instance is tied to? In other words, *how do you reference the "outer* `this`*"*? Although normally the inner class code doesn't need a reference to the outer class, since it already has an implicit one it's using to access the members of the outer class, it would need a reference to the outer class if it needed to pass that reference to some other code as follows:

```
class MyInner {
   public void seeOuter() {
      System.out.println("Outer x is " + x);
      System.out.println("Inner class ref is " + this);
      System.out.println("Outer class ref is " + MyOuter.this);
   }
}
```

If we run the complete code as follows:

```
class MyOuter {
   private int x = 7;
   public void makeInner() {
      MyInner in = new MyInner();
      in.seeOuter();
   }
   class MyInner {
      public void seeOuter() {
         System.out.println("Outer x is " + x);
         System.out.println("Inner class ref is " + this);
         System.out.println("Outer class ref is " + MyOuter.this);
      }
   }
   public static void main (String[] args) {
      MyOuter.MyInner inner = new MyOuter().new MyInner();
      inner.seeOuter();
   }
}
```

the output is something like this:

```
Outer x is 7
Inner class ref is MyOuter$MyInner@113708
Outer class ref is MyOuter@33f1d7
```

So the rules for an inner class referencing itself or the outer instance are as follows:

- To reference the inner class instance itself, from *within* the inner class code, use `this`.
- To reference the "*outer* `this`" (the outer class instance) from within the inner class code, use `NameOfOuterClass.this` (example, `MyOuter.this`).

Member Modifiers Applied to Inner Classes A regular inner class is a member of the outer class just as instance variables and methods are, so the following modifiers can be applied to an inner class:

- `final`
- `abstract`
- `public`
- `private`
- `protected`
- `static`—*but* `static` *turns it into a* `static` *nested class not an inner class*
- `strictfp`

CERTIFICATION OBJECTIVE

Method-Local Inner Classes

A regular inner class is scoped inside another class's curly braces, but outside any method code (in other words, at the same level that an instance variable is declared). But you can also define an inner class within a method:

```
class MyOuter2 {
     private String x = "Outer2";

     void doStuff() {
        class MyInner {
           public void seeOuter() {
             System.out.println("Outer x is " + x);
           } // close inner class method
        } // close inner class definition
      } // close outer class method doStuff()

} // close outer class
```

The preceding code declares a class, `MyOuter2`, with one method, `doStuff()`. But *inside* `doStuff()`, another class, `MyInner`, is declared, and it has a method of its own, `seeOuter()`. The code above is completely useless, however, because *it*

never instantiates the inner class! Just because you *declared* the class doesn't mean you created an *instance* of it. So to *use* the inner class you must make an instance of it somewhere *within the method but below the inner class definition* (or the compiler won't be able to find the inner class). The following legal code shows how to instantiate and use a method-local inner class:

```
class MyOuter2 {
     private String x = "Outer2";
     void doStuff() {
        class MyInner {
           public void seeOuter() {
             System.out.println("Outer x is " + x);
           } // close inner class method
        } // close inner class definition

        MyInner mi = new MyInner();  // This line must come
                                     // after the class
        mi.seeOuter();
     } // close outer class method doStuff()
} // close outer class
```

What a Method-Local Inner Object Can and Can't Do

A method-local inner class can be instantiated only within the method where the inner class is defined. In other words, no other code running in any other method—inside or outside the outer class—can ever instantiate the method-local inner class. Like regular inner class objects, the method-local inner class object shares a special relationship with the enclosing (outer) class object, and can access its `private` (or any other) members. However, *the inner class object cannot use the local variables of the method the inner class is in*. Why not?

Think about it. The local variables of the method live on the stack, and exist only for the lifetime of the method. You already know that the scope of a local variable is limited to the method the variable is declared in. When the method ends, the stack frame is blown away and the variable is history. But even after the method completes, the inner class object created within it might still be alive on the heap if, for example, a reference to it was passed into some other code and then stored in an instance variable. Because the local variables aren't guaranteed to be alive as long as the method-local inner class object, the inner class object can't use them. *Unless the local variables are marked* `final`*!* The following code attempts to access a local variable from within a method-local inner class.

```
class MyOuter2 {
   private String x = "Outer2";
   void doStuff() {
     String z = "local variable";
     class MyInner {
       public void seeOuter() {
         System.out.println("Outer x is " + x);
         System.out.println("Local variable z is " + z);  // Won't Compile!
       } // close inner class method
     }   // close inner class definition
   }     // close outer class method doStuff()
}        // close outer class
```

Compiling the preceding code *really* upsets the compiler:

```
MyOuter2.java:8: local variable z is accessed from within inner class;
needs to be declared final
           System.out.println("Local variable z is " + z);
                                                        ^
```

Marking the local variable `z` as `final` fixes the problem:

```
final String z = "local variable";  // Now inner object can use it
```

And just a reminder about modifiers within a method: the same rules apply to method-local inner classes as to local variable declarations. You can't, for example, mark a method-local inner class `public`, `private`, `protected`, `static`, `transient`, and the like. For the purpose of the exam, the only modifiers you *can* apply to a method-local inner class are `abstract` and `final`, but as always, never both at the same time.

exam Watch

Remember that a local class declared in a `static` ***method has access to only*** `static` ***members of the enclosing class, since there is no associated instance of the enclosing class. If you're in a*** `static` ***method there is no*** `this`***, so an inner class in a*** `static` ***method is subject to the same restrictions as the*** `static` ***method. In other words, no access to instance variables.***

CERTIFICATION OBJECTIVE

Anonymous Inner Classes

So far we've looked at defining a class within an enclosing class (a regular inner class) and within a method (a method-local inner class). Finally, we're going to look at the most unusual syntax you might ever see in Java; inner classes declared without any class name at all (hence the word *anonymous*). And if that's not weird enough, you can define these classes not just within a method, but even within an *argument* to a method. We'll look first at the *plain-old* (as if there is such a thing as a plain-old anonymous inner class) version (actually, even the plain-old version comes in two flavors), and then at the argument-declared anonymous inner class.

Perhaps your most important job here is to *learn to not be thrown when you see the syntax*. The exam is littered with anonymous inner class code: you might see it on questions about threads, wrappers, overriding, garbage collection, and... well, you get the idea.

Plain-Old Anonymous Inner Classes, Flavor One

Check out the following legal-but-strange-the-first-time-you-see-it code:

```
class Popcorn {
   public void pop() {
      System.out.println("popcorn");
    }
}
class Food {
   Popcorn p = new Popcorn() {
     public void pop() {
        System.out.println("anonymous popcorn");
     }
   };
}
```

Let's look at what's in the preceding code:

- We define two classes, Popcorn and Food.
- Popcorn has one method, `pop()`.
- Food has one instance variable, declared as type Popcorn. That's it for Food. Food has *no* methods.

And here's the big thing to get

The Popcorn reference variable refers *not* to an instance of Popcorn, but to *an instance of an anonymous (unnamed) subclass of Popcorn*.

Let's look at just the anonymous class code:

```
2. Popcorn p = new Popcorn() {
3.   public void pop() {
4.      System.out.println("anonymous popcorn");
5.   }
6. };
```

Line 2 Line 2 starts out as an instance variable declaration of type `Popcorn`. But instead of looking like this:

```
Popcorn p = new Popcorn(); // notice the semicolon at the end
```

there's a curly brace at the end of line 2, where a semicolon would normally be.

```
Popcorn p = new Popcorn() { // a curly brace, not a semicolon
```

You can read line 2 as saying,

Declare a reference variable, `p`, of type `Popcorn`. Then declare a new class that has no name, but that is a *subclass* of `Popcorn`. And here's the curly brace that opens the class definition…

Line 3 Line 3, then, is actually the first statement within the new class definition. And what is it doing? Overriding the `pop()` method of the superclass `Popcorn`. This is the whole point of making an anonymous inner class—to *override one or more methods of the superclass!* (Or to implement methods of an interface, but we'll save that for a little later.)

Line 4 Line 4 is the first (and in this case *only*) statement within the overriding `pop()` method. Nothing special there.

Line 5 Line 5 is the closing curly brace of the `pop()` method. Nothing special.

Line 6 Here's where you have to pay attention: line 6 includes a *curly brace closing off the anonymous class definition* (it's the companion brace to the one

on line 2), but there's more! Line 6 also has *the semicolon that ends the statement started on line 2*—the statement where it all began—the statement declaring and initializing the `Popcorn` reference variable. And what you're left with is a `Popcorn` reference to a brand-new *instance* of a brand-new, just-in-time, anonymous (no name) *subclass* of Popcorn.

exam Watch

The closing semicolon is hard to spot. Watch for code like this:

```
2. Popcorn p = new Popcorn() {
3.   public void pop() {
4.      System.out.println("anonymous popcorn");
5.   }
6. }                 //  Missing the semicolon needed to end
                     //  the statement started on 2!
7. Foo f = new Foo();
```

You'll need to be especially careful about the syntax when inner classes are involved, because the code on line 6 looks perfectly natural. It's rare to see semicolons following curly braces.

Polymorphism is in play when anonymous inner classes are involved. Remember that, as in the preceding Popcorn example, we're using a superclass reference variable type to refer to a subclass object. What are the implications? You can only call methods on an anonymous inner class reference that are defined in the reference variable type! This is no different from any other polymorphic references, for example,

```
class Horse extends Animal{
   void buck() { }
}
class Animal {
   void eat() { }
}
```

```
class Test {
   public static void main (String[] args) {
      Animal h = new Horse();
      h.eat();  // Legal, class Animal has an eat() method
      h.buck();  // Not legal! Class Animal doesn't have buck()
   }
}
```

So on the exam, you must be able to spot an anonymous inner class that—rather than overriding a method of the superclass—defines its own new method. The method definition isn't the problem, though; the real issue is how do you invoke that new method? The reference variable type (the superclass) won't know anything about that new method (defined in the anonymous subclass), so the compiler will complain if you try to invoke any method on an anonymous inner class reference that is not in the superclass class definition.

Check out the following, illegal code:

```
class Popcorn {
   public void pop() {
      System.out.println("popcorn");
   }
}

class Food {
   Popcorn p = new Popcorn() {
      public void sizzle() {
        System.out.println("anonymous sizzling popcorn");
      }
      public void pop() {
         System.out.println("anonymous popcorn");
      }
   };

   public void popIt() {
      p.pop();     // OK, Popcorn has a pop() method
      p.sizzle();  // Not Legal! Popcorn does not have sizzle()
   }
}
```

Compiling the preceding code gives us something like,

```
Anon.java:19: cannot resolve symbol
symbol  : method sizzle  ()
```

```
location: class Popcorn
      p.sizzle();
        ^
```

which is the compiler's way of saying, "I can't find method `sizzle()` in class `Popcorn`," followed by, "Get a clue."

Plain-Old Anonymous Inner Classes, Flavor Two

The only difference between flavor one and flavor two is that flavor one creates an anonymous *subclass* of the specified *class* type, whereas flavor two creates an anonymous *implementer* of the specified *interface* type. In the previous examples, we defined a new anonymous subclass of type `Popcorn` as follows:

```
Popcorn p = new Popcorn() {
```

But if `Popcorn` were an *interface* type instead of a *class* type, then the new anonymous class would be an *implementer* of the *interface* rather than a *subclass* of the *class*. Look at the following example:

```
interface Cookable {
   public void cook();
}
class Food {
   Cookable c = new Cookable() {
     public void cook() {
        System.out.println("anonymous cookable implementer");
     }
   };
}
```

The preceding code, like the Popcorn example, still creates an instance of an anonymous inner class, but this time the new just-in-time class is an implementer of the `Cookable` interface. And note that this is the only time you will ever see the syntax

```
new Cookable()
```

where `Cookable` is an *interface* rather than a nonabstract class type. Because think about it, *you can't instantiate an interface*, yet that's what the code *looks* like it's doing. But of course it's not instantiating a Cookable object, it's creating an instance of a new, anonymous, implementer of Cookable. You can read this line:

```
Cookable c = new Cookable() {
```

as, "Declare a reference variable of type `Cookable` that, obviously, will refer to an object from a class that implements the `Cookable` interface. But, oh yes, we don't yet *have* a class that implements `Cookable`, so we're going to make one right here, right now. We don't need a name for the class, but it will be a class that implements `Cookable`, and this curly brace starts the definition of the new implementing class."

One more thing to keep in mind about anonymous interface implementers—*they can implement only one interface*. There simply isn't any mechanism to say that your anonymous inner class is going to implement multiple interfaces. In fact, an anonymous inner class can't even extend a class and implement an interface at the same time. The inner class has to choose either to be a subclass of a named class—and not directly implement any interfaces at all—*or* to implement a single interface. By directly, we mean actually using the keyword `implements` as part of the class declaration. If the anonymous inner class is a subclass of a class type, it automatically becomes an implementer of any interfaces implemented by the superclass.

exam Watch

Don't be fooled by any attempts to instantiate an interface except in the case of an anonymous inner class. The following is not legal,

```
Runnable r = new Runnable(); // can't instantiate interface
```

whereas the following is legal, because it's instantiating an* implementer *of the* `Runnable` *interface (an anonymous implementation class):

```
Runnable r = new Runnable() {  // curly brace, not semicolon
  public void run() { }
};
```

Argument-Defined Anonymous Inner Classes

If you understood what we've covered so far in this chapter, then this last part will be simple. If you *are* still a little fuzzy on anonymous classes, however, then you should reread the previous sections. If they're not completely clear, we'd like to take full responsibility for the confusion. But we'll be happy to share.

Okay, if you've made it to this sentence, then we're all going to assume you understood the preceding section, and now we're just going to add one new twist. Imagine the following scenario. You're typing along, creating the Perfect Class, when you write code calling a method on a `Bar` object, and that method takes an object of type `Foo` (an interface).

```
class MyWonderfulClass {
   void go() {
      Bar b = new Bar();
      b.doStuff(ackWeDoNotHaveAFoo!); // Don't try to compile this at home
   }
}
interface Foo {
   void foof();
}
class Bar {
   void doStuff(Foo f) { }
}
```

No *problemo*, except that you don't *have* an object from a class that implements Foo, and you can't instantiate one, either, because *you don't even have a class that implements Foo*, let alone an instance of one. So you first need a class that implements Foo, and then you need an instance of that class to pass to the Bar class's `doStuff()` method. Savvy Java programmer that you are, you simply define an anonymous inner class, *right inside the argument*. That's right, just where you least expect to find a class. And here's what it looks like:

```
 1. class MyWonderfulClass {
 2.   void go() {
 3.     Bar b = new Bar();
 4.     b.doStuff(new Foo() {
 5.       public void foof() {
 6.         System.out.println("foofy");
 7.       } // end foof method
 8.     }); // end inner class def, arg, and b.doStuff stmt.
 9.   } // end go()
10. } // end class
11.
12. interface Foo {
13.   void foof();
14. }
15. class Bar {
16.   void doStuff(Foo f) { }
17. }
```

All the action starts on line 4. We're calling `doStuff()` on a `Bar` object, but the method takes an instance that IS-A `Foo`, where `Foo` is an interface. So we must make both an *implementation* class and an *instance* of that class, all right here in the argument to `doStuff()`. So that's what we do. We write

```
new Foo() {
```

to start the new class definition for the anonymous class that implements the `Foo` interface. `Foo` has a single method to implement, `foof()`, so on lines 5, 6, and 7 we implement the `foof()` method. Then on line 8—whoa!—more strange syntax appears. The first curly brace closes off the new anonymous class definition. But don't forget that this all happened as part of a method argument, so the close parenthesis,), finishes off the method invocation, and then we must still end the statement that began on line 4, so we end with a semicolon. Study this syntax! You will see anonymous inner classes on the exam, and you'll have to be very, very picky about the way they're closed. If they're *argument local*, they end like this:

```
});
```

but if they're just plain-old anonymous classes, then they end like this:

```
};
```

Regardless, the syntax is not what you use in virtually any other part of Java, so be careful. Any question from any part of the exam might involve anonymous inner classes as part of the code.

CERTIFICATION OBJECTIVE

Static Nested Classes

We saved the easiest for last, as a kind of treat :)

You'll sometimes hear static nested classes referred to as *static inner classes*, but they really aren't inner classes at all, by the standard definition of an inner class. While an inner class (regardless of the flavor) enjoys that *special relationship* with the outer class (or rather the *instances* of the two classes share a relationship), a static nested class does not. It is simply a non-inner (also called "top-level") class scoped within another. So with static classes it's really more about name-space resolution than about an implicit relationship between the two classes.

A static nested class is simply *a class that's a static member of the enclosing class*:

```
class BigOuter {
   static class Nested { }
}
```

The class itself isn't really "static"; there's no such thing as a static class. The `static` modifier in this case says that the nested class is *a static member of the outer class*. That means it can be accessed, as with other static members, *without having an instance of the outer class*.

Instantiating and Using Static Nested Classes

You use standard syntax to access a static nested class from its enclosing class. The syntax for instantiating a static nested class from a non-enclosing class is a little different from a normal inner class, and looks like this:

```
class BigOuter {
  static class Nest {void go() { System.out.println("hi"); } }
}
class Broom {
  static class B2 {void goB2() { System.out.println("hi 2"); } }
  public static void main(String[] args) {
    BigOuter.Nest n = new BigOuter.Nest();   // both class names
    n.go();
    B2 b2 = new B2();     // access the enclosed class
    b2.goB2();
  }
}
```

Which produces

```
hi
hi 2
```

exam Watch

Just as a static method does not have access to the instance variables and nonstatic methods of the class, a static nested class does not have access to the instance variables and nonstatic methods of the outer class. Look for static nested classes with code that behaves like a nonstatic (regular inner) class.

CERTIFICATION SUMMARY

Inner classes will show up throughout the exam, in any topic, and these are some of the exam's hardest questions. You should be comfortable with the sometimes bizarre syntax, and know how to spot legal and illegal inner class definitions.

We looked first at "regular" inner classes, where one class is a member of another. You learned that coding an inner class means putting the class definition of the inner class inside the curly braces of the enclosing (outer) class, but outside of any method or other code block. You learned that an inner class *instance* shares a special relationship with a specific *instance* of the outer class, and that this special relationship lets the inner class access all members of the outer class, including those marked `private`. You learned that to instantiate an inner class, you *must* have a reference to an instance of the outer class.

Next we looked at method-local inner classes—classes defined *inside* a method. The code for a method-local inner class looks virtually the same as the code for any other class definition, except that you can't apply an access modifier the way you can with a regular inner class. You learned why method-local inner classes cannot use non-`final` local variables declared within the method—the inner class instance may outlive the stack frame, so the local variable might vanish while the inner class object is still alive. You saw that to *use* the inner class you need to instantiate it, and that the instantiation must come *after* the class declaration in the method.

We also explored the strangest inner class type of all—the *anonymous* inner class. You learned that they come in two forms, normal and argument-defined. Normal, ho-hum, anonymous inner classes are created as part of a variable assignment, while argument-defined inner classes are actually declared, defined, and automatically instantiated *all within the argument to a method!* We covered the way anonymous inner classes can be either a subclass of the named class type, or an *implementer* of the named interface. Finally, we looked at how polymorphism applies to anonymous inner classes: you can invoke on the new instance only those methods defined in the named class or interface type. In other words, even if the anonymous inner class defines its own new method, no code from anywhere outside the inner class will be able to invoke that method.

As if we weren't already having enough fun for one day, we pushed on to static nested classes, which really aren't inner classes at all. Known as static nested classes, a nested class marked with the `static` modifier is quite similar to any other non-inner class, except that to access it, code must have access to both the nested and enclosing class. We saw that because the class is `static`, no instance of the enclosing class is needed, and thus the static nested class *does not share a special relationship with any instance of the enclosing class*. Remember, static inner classes can't access instance methods or variables.

TWO-MINUTE DRILL

Here are some of the key points from this chapter.

Inner Classes

- A "regular" inner class is declared *inside* the curly braces of another class, but *outside* any method or other code block.
- An inner class is a full-fledged member of the enclosing (outer) class, so it can be marked with an access modifier as well as the `abstract` or `final` modifiers. (Never both `abstract` and `final` together— remember that `abstract` *must* be subclassed, whereas `final` *cannot* be subclassed).
- An inner class instance shares a special relationship with an instance of the enclosing class. This relationship gives the inner class access to *all* of the outer class's members, including those marked `private`.
- To instantiate an inner class, you must have a reference to an instance of the outer class.
- From code within the enclosing class, you can instantiate the inner class using only the name of the inner class, as follows:

```
MyInner mi = new MyInner();
```

- From code outside the enclosing class's instance methods, you can instantiate the inner class only by using both the inner and outer class names, and a reference to the outer class as follows:

```
MyOuter mo = new MyOuter();
MyOuter.MyInner inner = mo.new MyInner();
```

- From code within the inner class, the keyword `this` holds a reference to the inner class instance. To reference the *outer* `this` (in other words, the instance of the outer class that this inner instance is tied to) precede the keyword `this` with the outer class name as follows: `MyOuter.this;`

Method-Local Inner Classes

- A method-local inner class is defined within a method of the enclosing class.
- For the inner class to be used, you must instantiate it, and that instantiation must happen within the same method, but *after* the class definition code.
- A method-local inner class cannot use variables declared within the method (including parameters) unless those variables are marked `final`.

- The only modifiers you can apply to a method-local inner class are `abstract` and `final`. (Never both at the same time, though.)

Anonymous Inner Classes

- Anonymous inner classes have no name, and their type must be either a subclass of the named type or an implementer of the named interface.
- An anonymous inner class is always created as part of a statement; don't forget to close the statement after the class definition with a curly brace. This is a rare case in Java, a curly brace followed by a semicolon.
- Because of polymorphism, the only methods you can call on an anonymous inner class reference are those defined in the reference variable class (or interface), even though the anonymous class is really a subclass or implementer of the reference variable type.
- An anonymous inner class can extend one subclass *or* implement one interface. Unlike non-anonymous classes (inner or otherwise), an anonymous inner class cannot do both. In other words, it cannot both extend a class *and* implement an interface, nor can it implement more than one interface.
- An argument-defined inner class is declared, defined, and automatically instantiated as part of a method invocation. The key to remember is that the class is being defined within a method argument, so the syntax will end the class definition with a curly brace, followed by a closing parenthesis to end the method call, followed by a semicolon to end the statement: `});`

Static Nested Classes

- Static nested classes are inner classes marked with the `static` modifier.
- A static nested class is *not* an inner class, it's a top-level nested class.
- Because the nested class is static, it does not share any special relationship with an instance of the outer class. In fact, you don't need an instance of the outer class to instantiate a static nested class.
- Instantiating a static nested class requires using both the outer and nested class names as follows:

```
BigOuter.Nested n = new BigOuter.Nested();
```

- A static nested class cannot access non-static members of the outer class, since it does not have an implicit reference to any outer instance (in other words, the nested class instance does not get an *outer* `this` reference).

SELF TEST

The following questions will help you measure your understanding of the dynamic and life-altering material presented in this chapter. Read all of the choices carefully. Take your time. Breathe.

1. Which are true about a static nested class? (Choose all that apply.)
   A. You must have a reference to an instance of the enclosing class in order to instantiate it
   B. It does not have access to non-`static` members of the enclosing class
   C. Its variables and methods must be `static`
   D. If the outer class is named `MyOuter`, and the nested class is named `MyInner`, it can be instantiated using `new MyOuter.MyInner();`
   E. It must extend the enclosing class

2. Given:

```
class Boo {
   Boo(String s) { }
   Boo() { }
}
class Bar extends Boo {
   Bar() { }
   Bar(String s) {super(s);}
   void zoo() {
   // insert code here
   }
}
```

   Which create an anonymous inner class from within class Bar? (Choose all that apply.)
   A. `Boo f = new Boo(24) { };`
   B. `Boo f = new Bar() { };`
   C. `Boo f = new Boo() {String s; };`
   D. `Bar f = new Boo(String s) { };`
   E. `Boo f = new Boo.Bar(String s) { };`

3. Which are true about a method-local inner class? (Choose all that apply.)
   A. It must be marked `final`
   B. It can be marked `abstract`

C. It can be marked `public`
D. It can be marked `static`
E. It can access private members of the enclosing class

4. Given:

```
1. public class TestObj {
2.   public static void main(String[] args) {
3.     Object o = new Object() {
4.       public boolean equals(Object obj) {
5.         return true;
6.       }
7.     }
8.     System.out.println(o.equals("Fred"));
9.   }
10. }
```

What is the result?
A. An exception occurs at runtime
B. `true`
C. `Fred`
D. Compilation fails because of an error on line 3
E. Compilation fails because of an error on line 4
F. Compilation fails because of an error on line 8
G. Compilation fails because of an error on a line other than 3, 4, or 8

5. Given:

```
1. public class HorseTest {
2.   public static void main(String[] args) {
3.     class Horse {
4.       public String name;
5.       public Horse(String s) {
6.         name = s;
7.       }
8.     }
9.     Object obj = new Horse("Zippo");
10.    System.out.println(obj.name);
11.  }
12. }
```

What is the result?

A. An exception occurs at runtime at line 10
B. `Zippo`
C. Compilation fails because of an error on line 3
D. Compilation fails because of an error on line 9
E. Compilation fails because of an error on line 10

6. Given:

```
public abstract class AbstractTest {
   public int getNum() {
      return 45;
   }
   public abstract class Bar {
     public int getNum() {
       return 38;
     }
   }
   public static void main(String[] args) {
      AbstractTest t = new AbstractTest() {
         public int getNum() {
           return 22;
         }
      };
      AbstractTest.Bar f = t.new Bar() {
         public int getNum() {
           return 57;
         }
      };
      System.out.println(f.getNum() + " " + t.getNum());
   }
}
```

What is the result?

A. `57 22`
B. `45 38`
C. `45 57`
D. An exception occurs at runtime
E. Compilation fails

7. Given:

```
3. public class Tour {
4.   public static void main(String[] args) {
5.     Cathedral c = new Cathedral();
6.     // insert code here
7.     s.go();
8.   }
9. }
10. class Cathedral {
11.   class Sanctum {
12.     void go() { System.out.println("spooky"); }
13.   }
14. }
```

Which, inserted independently at line 6, compile and produce the output "spooky"? (Choose all that apply.)

A. `Sanctum s = c.new Sanctum();`
B. `c.Sanctum s = c.new Sanctum();`
C. `c.Sanctum s = Cathedral.new Sanctum();`
D. `Cathedral.Sanctum s = c.new Sanctum();`
E. `Cathedral.Sanctum s = Cathedral.new Sanctum();`

8. Given:

```
5. class A { void m() { System.out.println("outer"); } }
6.
7. public class TestInners {
8.   public static void main(String[] args) {
9.     new TestInners().go();
10.   }
11.   void go() {
12.     new A().m();
13.     class A { void m() { System.out.println("inner"); } }
14.   }
15.   class A { void m() { System.out.println("middle"); } }
16. }
```

What is the result?

A. `inner`
B. `outer`

C. `middle`
D. Compilation fails
E. An exception is thrown at runtime

9. Given:

```
3. public class Car {
4.   class Engine {
5.     // insert code here
6.   }
7.   public static void main(String[] args) {
8.     new Car().go();
9.   }
10.  void go() {
11.    new Engine();
12.  }
13.  void drive() { System.out.println("hi"); }
14. }
```

Which, inserted independently at line 5, produce the output "hi"? (Choose all that apply.)

A. `{ Car.drive(); }`
B. `{ this.drive(); }`
C. `{ Car.this.drive(); }`
D. `{ this.Car.this.drive(); }`
E. `Engine() { Car.drive(); }`
F. `Engine() { this.drive(); }`
G. `Engine() { Car.this.drive(); }`

10. Given:

```
3. public class City {
4.   class Manhattan {
5.     void doStuff() throws Exception { System.out.print("x "); }
6.   }
7.   class TimesSquare extends Manhattan {
8.     void doStuff() throws Exception { }
9.   }
10.  public static void main(String[] args) throws Exception {
11.    new City().go();
12.  }
13.  void go() throws Exception { new TimesSquare().doStuff(); }
14. }
```

What is the result?

A. `x`
B. `x x`
C. No output is produced
D. Compilation fails due to multiple errors
E. Compilation fails due only to an error on line 4
F. Compilation fails due only to an error on line 7
G. Compilation fails due only to an error on line 10
H. Compilation fails due only to an error on line 13

11. Given:

```
 3. public class Navel {
 4.   private int size = 7;
 5.   private static int length = 3;
 6.   public static void main(String[] args) {
 7.     new Navel().go();
 8.   }
 9.   void go() {
10.     int size = 5;
11.     System.out.println(new Gazer().adder());
12.   }
13.   class Gazer {
14.     int adder() { return size * length; }
15.   }
16. }
```

What is the result?

A. `15`
B. `21`
C. An exception is thrown at runtime
D. Compilation fails due to multiple errors
E. Compilation fails due only to an error on line 4
F. Compilation fails due only to an error on line 5

12. Given:

```
 3. import java.util.*;
 4. public class Pockets {
 5.   public static void main(String[] args) {
 6.     String[] sa = {"nickel", "button", "key", "lint"};
 7.     Sorter s = new Sorter();
 8.     for(String s2: sa) System.out.print(s2 + " ");
 9.     Arrays.sort(sa,s);
10.     System.out.println();
11.     for(String s2: sa) System.out.print(s2 + " ");
12.   }
13.   class Sorter implements Comparator<String> {
14.     public int compare(String a, String b) {
15.       return b.compareTo(a);
16.     }
17.   }
18. }
```

What is the result?

A. Compilation fails

B.
```
button key lint nickel
nickel lint key button
```

C.
```
nickel button key lint
button key lint nickel
```

D.
```
nickel button key lint
nickel button key lint
```

E.
```
nickel button key lint
nickel lint key button
```

F. An exception is thrown at runtime

SELF TEST ANSWERS

1. Which are true about a static nested class? (Choose all that apply.)
   A. You must have a reference to an instance of the enclosing class in order to instantiate it
   B. It does not have access to non-`static` members of the enclosing class
   C. Its variables and methods must be `static`
   D. If the outer class is named `MyOuter`, and the nested class is named `MyInner`, it can be instantiated using `new MyOuter.MyInner();`
   E. It must extend the enclosing class

Answer:

☑ **B** and **D. B** is correct because a static nested class is not tied to an instance of the enclosing class, and thus can't access the non-`static` members of the class (just as a `static` method can't access non-static members of a class). **D** uses the correct syntax for instantiating a static nested class.

☒ **A** is incorrect because static nested classes do not need (and can't use) a reference to an instance of the enclosing class. **C** is incorrect because static nested classes can declare and define non-static members. **E** is wrong because…it just is. There's no rule that says an inner or nested class has to extend anything.

2. Given:

```
class Boo {
   Boo(String s) { }
   Boo() { }
}
class Bar extends Boo {
   Bar() { }
   Bar(String s) {super(s);}
   void zoo() {
   // insert code here
   }
}
```

Which create an anonymous inner class from within class Bar? (Choose all that apply.)

A. `Boo f = new Boo(24) { };`
B. `Boo f = new Bar() { };`

C. `Boo f = new Boo() {String s; };`
D. `Bar f = new Boo(String s) { };`
E. `Boo f = new Boo.Bar(String s) { };`

Answer:

☑ **B** and **C. B** is correct because anonymous inner classes are no different from any other class when it comes to polymorphism. That means you are always allowed to declare a reference variable of the superclass type and have that reference variable refer to an instance of a subclass type, which in this case is an anonymous subclass of Bar. Since Bar is a subclass of Boo, it all works. **C** uses correct syntax for creating an instance of Boo.

☒ **A** is incorrect because it passes an `int` to the Boo constructor, and there is no matching constructor in the Boo class. **D** is incorrect because it violates the rules of polymorphism; you cannot refer to a superclass type using a reference variable declared as the subclass type. The superclass doesn't have everything the subclass has. **E** uses incorrect syntax.

3. Which are true about a method-local inner class? (Choose all that apply.)

A. It must be marked `final`
B. It can be marked `abstract`
C. It can be marked `public`
D. It can be marked `static`
E. It can access private members of the enclosing class

Answer:

☑ **B** and **E. B** is correct because a method-local inner class can be `abstract`, although it means a subclass of the inner class must be created if the `abstract` class is to be used (so an `abstract` method-local inner class is probably not useful). **E** is correct because a method-local inner class works like any other inner class—it has a special relationship to an instance of the enclosing class, thus it can access all members of the enclosing class.

☒ **A** is incorrect because a method-local inner class does not have to be declared `final` (although it is legal to do so). **C** and **D** are incorrect because a method-local inner class cannot be made `public` (remember—local variables can't be `public`) or `static`.

4. Given:

```
1. public class TestObj {
2.   public static void main(String[] args) {
3.     Object o = new Object() {
```

```
4.        public boolean equals(Object obj) {
5.          return true;
6.        }
7.      }
8.      System.out.println(o.equals("Fred"));
9.    }
10. }
```

What is the result?

A. An exception occurs at runtime
B. `true`
C. `fred`
D. Compilation fails because of an error on line 3
E. Compilation fails because of an error on line 4
F. Compilation fails because of an error on line 8
G. Compilation fails because of an error on a line other than 3, 4, or 8

Answer:

☑ **G**. This code would be legal if line 7 ended with a semicolon. Remember that line 3 is a statement that doesn't end until line 7, and a statement needs a closing semicolon!

☒ **A, B, C, D, E,** and **F** are incorrect based on the program logic described above. If the semicolon were added at line 7, then answer **B** would be correct—the program would print `true`, the return from the `equals()` method overridden by the anonymous subclass of `Object`.

5. Given:

```
1. public class HorseTest {
2.   public static void main(String[] args) {
3.     class Horse {
4.       public String name;
5.       public Horse(String s) {
6.         name = s;
7.       }
8.     }
9.     Object obj = new Horse("Zippo");
10.    System.out.println(obj.name);
11.  }
12. }
```

What is the result?

A. An exception occurs at runtime at line 10
B. `Zippo`
C. Compilation fails because of an error on line 3
D. Compilation fails because of an error on line 9
E. Compilation fails because of an error on line 10

Answer:

☑ **E.** If you use a reference variable of type `Object`, you can access only those members defined in class `Object`.

☒ **A, B, C,** and **D** are incorrect based on the program logic described above.

6. Given:

```
public abstract class AbstractTest {
   public int getNum() {
      return 45;
   }
   public abstract class Bar {
     public int getNum() {
       return 38;
     }
   }
   public static void main(String[] args) {
      AbstractTest t = new AbstractTest() {
         public int getNum() {
           return 22;
         }
      };
      AbstractTest.Bar f = t.new Bar() {
         public int getNum() {
           return 57;
         }
      };
      System.out.println(f.getNum() + " " + t.getNum());
} }
```

What is the result?

A. `57 22`
B. `45 38`
C. `45 57`
D. An exception occurs at runtime
E. Compilation fails

Answer:

☑ **A**. You can define an inner class as `abstract`, which means you can instantiate only concrete subclasses of the abstract inner class. The object referenced by the variable `t` is an instance of an anonymous subclass of AbstractTest, and the anonymous class overrides the `getNum()` method to return `22`. The variable referenced by `f` is an instance of an anonymous subclass of `Bar`, and the anonymous Bar subclass also overrides the `getNum()` method (to return `57`). Remember that to create a Bar instance, we need an instance of the enclosing AbstractTest class to tie to the new `Bar` inner class instance. AbstractTest can't be instantiated because it's `abstract`, so we created an anonymous subclass (non-`abstract`) and then used the instance of that anonymous subclass to tie to the new Bar subclass instance.

☒ **B, C, D,** and **E** are incorrect based on the program logic described above.

7. Given:

```
 3. public class Tour {
 4.   public static void main(String[] args) {
 5.     Cathedral c = new Cathedral();
 6.     // insert code here
 7.     s.go();
 8.   }
 9. }
10. class Cathedral {
11.   class Sanctum {
12.     void go() { System.out.println("spooky"); }
13.   }
14. }
```

Which, inserted independently at line 6, compile and produce the output "spooky"? (Choose all that apply.)

A. `Sanctum s = c.new Sanctum();`
B. `c.Sanctum s = c.new Sanctum();`
C. `c.Sanctum s = Cathedral.new Sanctum();`
D. `Cathedral.Sanctum s = c.new Sanctum();`
E. `Cathedral.Sanctum s = Cathedral.new Sanctum();`

Answer:

☑ **D** is correct. It is the only code that uses the correct inner class instantiation syntax.

☒ **A, B, C,** and **E** are incorrect based on the above. (Objective 1.1)

8. Given:

```
 5. class A { void m() { System.out.println("outer"); } }
 6.
 7. public class TestInners {
 8.   public static void main(String[] args) {
 9.     new TestInners().go();
10.   }
11.   void go() {
12.     new A().m();
13.     class A { void m() { System.out.println("inner"); } }
14.   }
15.   class A { void m() { System.out.println("middle"); } }
16. }
```

What is the result?

A. `inner`
B. `outer`
C. `middle`
D. Compilation fails
E. An exception is thrown at runtime

Answer:

☑ **C** is correct. The `"inner"` version of `class A` isn't used because its declaration comes after the instance of `class A` is created in the `go()` method.

☒ **A, B, D,** and **E** are incorrect based on the above. (Objective 1.1)

9. Given:

```
 3. public class Car {
 4.   class Engine {
 5.     // insert code here
 6.   }
 7.   public static void main(String[] args) {
 8.     new Car().go();
 9.   }
10.   void go() {
11.     new Engine();
12.   }
13.   void drive() { System.out.println("hi"); }
14. }
```

Which, inserted independently at line 5, produce the output "hi"? (Choose all that apply.)

A. `{ Car.drive(); }`
B. `{ this.drive(); }`
C. `{ Car.this.drive(); }`
D. `{ this.Car.this.drive(); }`
E. `Engine() { Car.drive(); }`
F. `Engine() { this.drive(); }`
G. `Engine() { Car.this.drive(); }`

Answer:

☑ **C** and **G** are correct. **C** is the correct syntax to access an inner class's outer instance method from an initialization block, and **G** is the correct syntax to access it from a constructor.

☒ **A, B, D, E,** and **F** are incorrect based on the above. (Objectives 1.1, 1.4)

10. Given:

```
 3. public class City {
 4.   class Manhattan {
 5.     void doStuff() throws Exception { System.out.print("x "); }
 6.   }
 7.   class TimesSquare extends Manhattan {
 8.     void doStuff() throws Exception { }
 9.   }
10.   public static void main(String[] args) throws Exception {
11.     new City().go();
12.   }
13.   void go() throws Exception { new TimesSquare().doStuff(); }
14. }
```

What is the result?

A. `x`
B. `x x`
C. No output is produced
D. Compilation fails due to multiple errors
E. Compilation fails due only to an error on line 4
F. Compilation fails due only to an error on line 7
G. Compilation fails due only to an error on line 10
H. Compilation fails due only to an error on line 13

Answer:

☑ **C** is correct. The inner classes are valid, and all the methods (including `main()`), correctly throw an Exception, given that `doStuff()` throws an Exception. The `doStuff()` in class `TimesSquare` overrides class `Manhattan`'s `doStuff()` and produces no output.

☒ **A, B, D, E, F, G,** and **H** are incorrect based on the above. (Objectives 1.1, 2.4)

11. Given:

```
 3. public class Navel {
 4.   private int size = 7;
 5.   private static int length = 3;
 6.   public static void main(String[] args) {
 7.     new Navel().go();
 8.   }
 9.   void go() {
10.     int size = 5;
11.     System.out.println(new Gazer().adder());
12.   }
13.   class Gazer {
14.     int adder() { return size * length; }
15.   }
16. }
```

What is the result?

A. `15`
B. `21`
C. An exception is thrown at runtime
D. Compilation fails due to multiple errors
E. Compilation fails due only to an error on line 4
F. Compilation fails due only to an error on line 5

Answer:

☑ **B** is correct. The inner class `Gazer` has access to `Navel`'s `private static` and private instance variables.

☒ **A, C, D, E,** and **F** are incorrect based on the above. (Objectives 1.1, 1.4)

12. Given:

```
 3. import java.util.*;
 4. public class Pockets {
 5.   public static void main(String[] args) {
 6.     String[] sa = {"nickel", "button", "key", "lint"};
 7.     Sorter s = new Sorter();
 8.     for(String s2: sa) System.out.print(s2 + " ");
 9.     Arrays.sort(sa,s);
10.     System.out.println();
11.     for(String s2: sa) System.out.print(s2 + " ");
12.   }
13.   class Sorter implements Comparator<String> {
14.     public int compare(String a, String b) {
15.       return b.compareTo(a);
16.     }
17.   }
18. }
```

What is the result?

A. Compilation fails

B. `button key lint nickel`
`nickel lint key button`

C. `nickel button key lint`
`button key lint nickel`

D. `nickel button key lint`
`nickel button key lint`

E. `nickel button key lint`
`nickel lint key button`

F. An exception is thrown at runtime

Answer:

☑ **A** is correct, the inner class `Sorter` must be declared `static` to be called from the `static` method `main()`. If `Sorter` had been `static`, answer E would be correct.

☒ **B, C, D, E,** and **F** are incorrect based on the above. (Objectives 1.1, 1.4, 6.5)

9

Threads

CERTIFICATION OBJECTIVES

- Start New Threads
- Recognize Thread States and Transitions
- Use Object Locking to Avoid Concurrent Access
- Write Code That Uses wait(), notify(), or notifyAll()

✓ Two-Minute Drill

Q&A Self Test

CERTIFICATION OBJECTIVE

Defining, Instantiating, and Starting Threads (Objective 4.1)

4.1 Write code to define, instantiate, and start new threads using both java.lang.Thread and java.lang.Runnable.

Imagine a stockbroker application with a lot of complex capabilities. One of its functions is "download last stock option prices," another is "check prices for warnings," and a third time-consuming operation is "analyze historical data for company XYZ."

In a single-threaded runtime environment, these actions execute one after another. The next action can happen *only* when the previous one is finished. If a historical analysis takes half an hour, and the user selects to perform a download and check afterward, the warning may come too late to, say, buy or sell stock as a result.

We just imagined the sort of application that cries out for multithreading. Ideally, the download should happen in the background (that is, in another thread). That way, other processes could happen at the same time so that, for example, a warning could be communicated instantly. All the while, the user is interacting with other parts of the application. The analysis, too, could happen in a separate thread, so the user can work in the rest of the application while the results are being calculated.

So what exactly is a thread? In Java, "thread" means two different things:

- An instance of class java.lang.Thread
- A thread of execution

An instance of Thread is just…an object. Like any other object in Java, it has variables and methods, and lives and dies on the heap. But a *thread of execution* is an individual process (a "lightweight" process) that has its own call stack. In Java, there is *one thread per call stack*—or, to think of it in reverse, *one call stack per thread*. Even if you don't create any new threads in your program, threads are back there running.

The `main()` method, that starts the whole ball rolling, runs in one thread, called (surprisingly) the *main* thread. If you looked at the main call stack (and you can, any time you get a stack trace from something that happens after main begins, but not within another thread), you'd see that `main()` is the first method on the stack—

the method at the bottom. But as soon as you create a *new* thread, a new stack materializes and methods called from *that* thread run in a call stack that's separate from the `main()` call stack. That second new call stack is said to run concurrently with the main thread, but we'll refine that notion as we go through this chapter.

You might find it confusing that we're talking about code running *concurrently*—as if in *parallel*—given that there's only one CPU on most of the machines running Java. What gives? The JVM, which gets its turn at the CPU by whatever scheduling mechanism the underlying OS uses, operates like a mini-OS and schedules *its* own threads regardless of the underlying operating system. In some JVMs, the Java threads are actually mapped to native OS threads, but we won't discuss that here; native threads are not on the exam. Nor is it required to understand how threads behave in different JVM environments. In fact, the most important concept to understand from this entire chapter is this:

When it comes to threads, very little is guaranteed.

So be very cautious about interpreting the behavior you see on *one* machine as "the way threads work." The exam expects you to know what is and is not guaranteed behavior, so that you can design your program in such a way that it will work regardless of the underlying JVM. *That's part of the whole point of Java.*

Don't make the mistake of designing your program to be dependent on a particular implementation of the JVM. As you'll learn a little later, different JVMs can run threads in profoundly different ways. For example, one JVM might be sure that all threads get their turn, with a fairly even amount of time allocated for each thread in a nice, happy, round-robin fashion. But in other JVMs, a thread might start running and then just hog the whole show, never stepping out so others can have a turn. If you test your application on the "nice turn-taking" JVM, and you don't know what is and is not guaranteed in Java, then you might be in for a big shock when you run it under a JVM with a different thread scheduling mechanism.

The thread questions are among the most difficult questions on the exam. In fact, for most people they *are* the toughest questions on the exam, and with four objectives for threads you'll be answering a *lot* of thread questions. If you're not already familiar with threads, you'll probably need to spend some time experimenting. Also, one final disclaimer: *This chapter makes almost no attempt to teach you how to design a good, safe, multithreaded application. We only scratch*

the surface of that huge topic in this chapter! You're here to learn the basics of threading, and what you need to get through the thread questions on the exam. Before you can write decent multithreaded code, however, you really need to study more on the complexities and subtleties of multithreaded code.

(Note: The topic of daemon threads is NOT on the exam. All of the threads discussed in this chapter are "user" threads. You and the operating system can create a second kind of thread called a daemon thread. The difference between these two types of threads (user and daemon) is that the JVM exits an application only when all user threads are complete—the JVM doesn't care about letting daemon threads complete, so once all user threads are complete, the JVM will shut down, regardless of the state of any daemon threads. Once again, this topic is NOT on the exam.)

Making a Thread

A thread in Java begins as an instance of java.lang.Thread. You'll find methods in the Thread class for managing threads including creating, starting, and pausing them. For the exam, you'll need to know, at a minimum, the following methods:

```
start()
yield()
sleep()
run()
```

The action happens in the `run()` method. Think of the code you want to execute in a separate thread as *the job to do*. In other words, you have some work that needs to be done, say, downloading stock prices in the background while other things are happening in the program, so what you really want is that *job* to be executed in its own thread. So if the *work* you want done is the *job*, the one *doing* the work (actually executing the job code) is the *thread*. And the *job always starts from a* `run()` *method* as follows:

```
public void run() {
  // your job code goes here
}
```

You always write the code that needs to be run in a separate thread in a `run()` method. The `run()` method will call other methods, of course, but the thread of execution—the new call stack—always begins by invoking `run()`. So where does the `run()` method go? In one of the two classes you can use to define your thread job.

You can define and instantiate a thread in one of two ways:

- Extend the java.lang.Thread class.
- Implement the Runnable interface.

You need to know about both for the exam, although in the real world you're much more likely to implement Runnable than extend Thread. Extending the Thread class is the easiest, but it's usually not a good OO practice. Why? Because subclassing should be reserved for specialized versions of more general superclasses. So the only time it really makes sense (from an OO perspective) to extend Thread is when you have a more specialized version of a Thread class. In other words, because *you have more specialized thread-specific behavior*. Chances are, though, that the thread work you want is really just a job to be done *by* a thread. In that case, you should design a class that implements the Runnable interface, which also leaves your class free to extend some *other* class.

Defining a Thread

To define a thread, you need a place to put your `run()` method, and as we just discussed, you can do that by extending the Thread class or by implementing the Runnable interface. We'll look at both in this section.

Extending java.lang.Thread

The simplest way to define code to run in a separate thread is to

- Extend the java.lang.Thread class.
- Override the `run()` method.

It looks like this:

```
class MyThread extends Thread {
   public void run() {
     System.out.println("Important job running in MyThread");
   }
 }
```

The limitation with this approach (besides being a poor design choice in most cases) is that if you extend Thread, *you can't extend anything else*. And it's not as if you really need that inherited Thread class behavior, because in order to use a thread you'll need to instantiate one anyway.

Keep in mind that you're free to overload the `run()` method in your Thread subclass:

```
class MyThread extends Thread {
  public void run() {
    System.out.println("Important job running in MyThread");
  }
  public void run(String s) {
    System.out.println("String in run is " + s);
  }
}
```

But know this: *The overloaded* `run(String s)` *method will be ignored by the Thread class unless you call it yourself. The Thread class expects a* `run()` *method with no arguments, and it will execute this method for you in a separate call stack after the thread has been started.* With a `run(String s)` method, the Thread class won't call the method for you, and even if you call the method directly yourself, execution won't happen in a new thread of execution with a separate call stack. It will just happen in the same call stack as the code that you made the call from, just like any other normal method call.

Implementing java.lang.Runnable

Implementing the Runnable interface gives you a way to extend any class you like, but still define behavior that will be run by a separate thread. It looks like this:

```
class MyRunnable implements Runnable {
   public void run() {
     System.out.println("Important job running in MyRunnable");
   }
}
```

Regardless of which mechanism you choose, you've now got yourself some code that can be run by a thread of execution. So now let's take a look at *instantiating* your thread-capable class, and then we'll figure out how to actually get the thing *running*.

Instantiating a Thread

Remember, every thread of execution begins as an instance of class Thread. Regardless of whether your `run()` method is in a Thread subclass or a Runnable implementation class, you still need a Thread object to do the work.

If you extended the Thread class, instantiation is dead simple (we'll look at some additional overloaded constructors in a moment):

```
MyThread t = new MyThread()
```

If you implement Runnable, instantiation is only slightly less simple. To have code run by a separate thread, *you still need a Thread instance*. But rather than combining both the *thread* and the *job* (the code in the `run()` method) into one class, you've split it into two classes—the Thread class for the *thread-specific* code and your Runnable implementation class for your *job-that-should-be-run-by-a-thread* code. (Another common way to think about this is that the Thread is the "worker," and the Runnable is the "job" to be done.)

First, you instantiate your Runnable class:

```
MyRunnable r = new MyRunnable();
```

Next, you get yourself an instance of java.lang.Thread (*somebody* has to run your job…), and you *give it your job*!

```
Thread t = new Thread(r);  // Pass your Runnable to the Thread
```

If you create a thread using the no-arg constructor, the thread will call its own `run()` method when it's time to start working. That's exactly what you want when you extend Thread, but when you use Runnable, you need to tell the new thread to use *your* `run()` method rather than its own. The Runnable you pass to the Thread constructor is called the *target* or the *target Runnable*.

You can pass a single Runnable instance to multiple Thread objects, so that the same Runnable becomes the target of multiple threads, as follows:

```
public class TestThreads {
   public static void main (String [] args) {
     MyRunnable r = new MyRunnable();
     Thread foo = new Thread(r);
     Thread bar = new Thread(r);
     Thread bat = new Thread(r);
   }
}
```

Giving the same target to multiple threads means that several threads of execution will be running the very same job (and that the same job will be done multiple times).

exam Watch

The Thread class itself implements Runnable. (After all, it has a `run()` ***method that we were overriding.) This means that you could pass a Thread to another Thread's constructor:***

```
Thread t = new Thread(new MyThread());
```

This is a bit silly, but it's legal. In this case, you really just need a Runnnable, and creating a whole other Thread is overkill.

Besides the no-arg constructor and the constructor that takes a Runnable (the target, i.e., the instance with the job to do), there are other overloaded constructors in class Thread. The constructors we care about are

- `Thread()`
- `Thread(Runnable target)`
- `Thread(Runnable target, String name)`
- `Thread(String name)`

You need to recognize all of them for the exam! A little later, we'll discuss some of the other constructors in the preceding list.

So now you've made yourself a Thread instance, and it knows which `run()` method to call. *But nothing is happening yet*. At this point, all we've got is a plain old Java object of type Thread. *It is not yet a thread of execution*. To get an actual thread—a new call stack—we still have to *start* the thread.

When a thread has been instantiated but not started (in other words, the `start()` method has not been invoked on the Thread instance), the thread is said to be in the *new* state. At this stage, the thread is not yet considered to be *alive*. Once the `start()` method is called, the thread is considered to be *alive* (even though the `run()` method may not have actually started executing yet). A thread is considered *dead* (no longer *alive*) after the `run()` method completes. The `isAlive()` method is the best way to determine if a thread has been started but has not yet completed its `run()` method. (Note: The `getState()` method is very useful for debugging, but you won't have to know it for the exam.)

Starting a Thread

You've created a Thread object and it knows its target (either the passed-in Runnable or itself if you extended class Thread). Now it's time to get the whole thread thing happening—to launch a new call stack. It's so simple it hardly deserves its own subheading:

```
t.start();
```

Prior to calling `start()` on a Thread instance, the thread (when we use lowercase `t`, we're referring to the *thread of execution* rather than the Thread class) is said to be in the *new* state as we said. The new state means you have a Thread *object* but you don't yet have a *true thread*. So what happens after you call `start()`? The good stuff:

- A new thread of execution starts (with a new call stack).
- The thread moves from the *new* state to the *runnable* state.
- When the thread gets a chance to execute, its target `run()` method will run.

Be *sure* you remember the following: You start a *Thread*, not a *Runnable*. You call `start()` on a Thread instance, not on a Runnable instance. The following example demonstrates what we've covered so far—defining, instantiating, and starting a thread:

```
class FooRunnable implements Runnable {
   public void run() {
      for(int x = 1; x < 6; x++) {
        System.out.println("Runnable running");
      }
   }
}

public class TestThreads {
   public static void main (String [] args) {
     FooRunnable r = new FooRunnable();
     Thread t = new Thread(r);
     t.start();
   }
}
```

Running the preceding code prints out exactly what you'd expect:

```
% java TestThreads
Runnable running
Runnable running
Runnable running
Runnable running
Runnable running
```

(If this isn't what you expected, go back and re-read everything in this objective.)

exam Watch

There's nothing special about the `run()` ***method as far as Java is concerned. Like*** `main()`***, it just happens to be the name (and signature) of the method that the new thread knows to invoke. So if you see code that calls the*** `run()` ***method on a Runnable (or even on a Thread instance), that's perfectly legal. But it doesn't mean the*** `run()` ***method will run in a separate thread! Calling a*** `run()` ***method directly just means you're invoking a method from whatever thread is currently executing, and the*** `run()` ***method goes onto the current call stack rather than at the beginning of a new call stack. The following code does not start a new thread of execution:***

```
Thread t = new Thread();
t.run();  // Legal, but does not start a new thread
```

So what happens if we start multiple threads? We'll run a simple example in a moment, but first we need to know how to print out which thread is executing. We can use the `getName()` method of class Thread, and have each Runnable print out the name of the thread executing that Runnable object's `run()` method. The following example instantiates a thread and gives it a name, and then the name is printed out from the `run()` method:

```
class NameRunnable implements Runnable {
   public void run() {
       System.out.println("NameRunnable running");
       System.out.println("Run by "
         + Thread.currentThread().getName());
```

```
    }
}
public class NameThread {
    public static void main (String [] args) {
      NameRunnable nr = new NameRunnable();
      Thread t = new Thread(nr);
      t.setName("Fred");
      t.start();
    }
}
```

Running this code produces the following, extra special, output:

```
% java NameThread
NameRunnable running
Run by Fred
```

To get the name of a thread you call—who would have guessed—`getName()` on the Thread instance. But the target Runnable instance doesn't even *have* a reference to the Thread instance, so we first invoked the `static Thread.currentThread()` method, which returns a reference to the currently executing thread, and then we invoked `getName()` on that returned reference.

Even if you don't explicitly name a thread, it still has a name. Let's look at the previous code, commenting out the statement that sets the thread's name:

```
public class NameThread {
    public static void main (String [] args) {
      NameRunnable nr = new NameRunnable();
      Thread t = new Thread(nr);
      // t.setName("Fred");
      t.start();
    }
}
```

Running the preceding code now gives us

```
% java NameThread
NameRunnable running
Run by Thread-0
```

And since we're getting the name of the current thread by using the static `Thread.currentThread()` method, we can even get the name of the thread running our main code,

```
public class NameThreadTwo {
   public static void main (String [] args) {
     System.out.println("thread is "
      + Thread.currentThread().getName());
   }
}
```

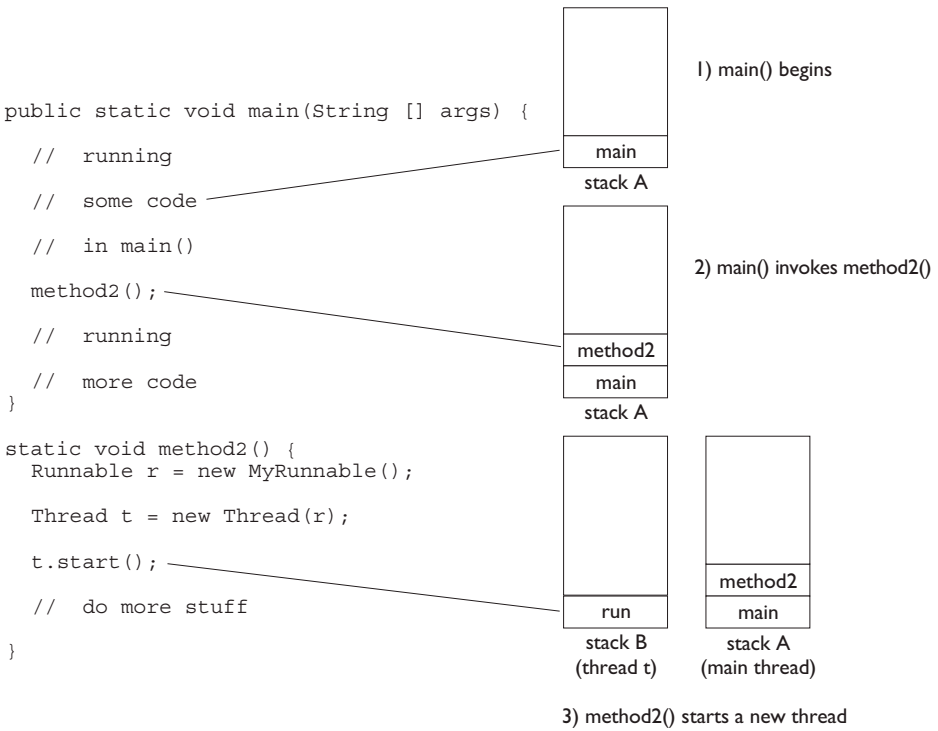
which prints out

```
% java NameThreadTwo
thread is main
```

That's right, the main thread already has a name—*main*. (Once again, what are the odds?) Figure 9-1 shows the process of starting a thread.

FIGURE 9-1

Starting a thread



Starting and Running Multiple Threads

Enough playing around here; let's actually get multiple threads going (more than two, that is). We already had two threads, because the `main()` method starts in a thread of its own, and then `t.start()` started a *second* thread. Now we'll do more. The following code creates a single Runnable instance and three Thread instances. All three Thread instances get the same Runnable instance, and each thread is given a unique name. Finally, all three threads are started by invoking `start()` on the Thread instances.

```
class NameRunnable implements Runnable {
    public void run() {
        for (int x = 1; x <= 3; x++) {
            System.out.println("Run by "
                    + Thread.currentThread().getName()
                    + ", x is " + x);
        }
    }
}
public class ManyNames {
    public static void main(String [] args) {
        // Make one Runnable
        NameRunnable nr = new NameRunnable();
        Thread one = new Thread(nr);
        Thread two = new Thread(nr);
        Thread three = new Thread(nr);

        one.setName("Fred");
        two.setName("Lucy");
        three.setName("Ricky");
        one.start();
        two.start();
        three.start();
    }
}
```

Running this code might produce the following:

```
% java ManyNames
Run by Fred, x is 1
Run by Fred, x is 2
Run by Fred, x is 3
```

```
Run by Lucy, x is 1
Run by Lucy, x is 2
Run by Lucy, x is 3
Run by Ricky, x is 1
Run by Ricky, x is 2
Run by Ricky, x is 3
```

Well, at least that's what it printed when we ran it—this time, on our machine. But the behavior you see above is not guaranteed. This is so crucial that you need to stop right now, take a deep breath, and repeat after me, "The behavior is not guaranteed." You need to know, for your future as a Java programmer as well as for the exam, that there is nothing in the Java specification that says threads will start running in the order in which they were started (in other words, the order in which `start()` was invoked on each thread). And there is no guarantee that once a thread starts executing, it will keep executing until it's done. Or that a loop will complete before another thread begins. No siree Bob. Nothing is guaranteed in the preceding code except this:

Each thread will start, and each thread will run to completion.

Within each thread, things will happen in a predictable order. But the actions of different threads can mix together in unpredictable ways. If you run the program multiple times, or on multiple machines, you may see different output. Even if you don't see different output, you need to realize that the behavior you see is not guaranteed. Sometimes a little change in the way the program is run will cause a difference to emerge. Just for fun we bumped up the loop code so that each `run()` method ran the `for` loop 400 times rather than 3, and eventually we did start to see some wobbling:

```
public void run() {
    for (int x = 1; x <= 400; x++) {
        System.out.println("Run by "
                + Thread.currentThread().getName()
                + ", x is " + x);
    }
}
```

Running the preceding code, with each thread executing its run loop 400 times, started out fine but then became nonlinear. Here's just a snip from the command-

line output of running that code. To make it easier to distinguish each thread, we put Fred's output in italics and Lucy's in bold, and left Ricky's alone:

Run by Fred, x is 345
Run by Ricky, x is 313
Run by Lucy, x is 341
Run by Ricky, x is 314
Run by Lucy, x is 342
Run by Ricky, x is 315
Run by Fred, x is 346
Run by Lucy, x is 343
Run by Fred, x is 347
Run by Lucy, x is 344
... it continues on ...

Notice that there's not really any clear pattern here. If we look at only the output from Fred, we see the numbers increasing one at a time, as expected:

Run by Fred, x is 345
Run by Fred, x is 346
Run by Fred, x is 347

And similarly if we look only at the output from Lucy, or Ricky. Each one individually is behaving in a nice orderly manner. But together—chaos! In the fragment above we see Fred, then Lucy, then Ricky (in the same order we originally started the threads), but then Lucy butts in when it was Fred's turn. What nerve! And then Ricky and Lucy trade back and forth for a while until finally Fred gets another chance. They jump around like this for a while after this. Eventually (after the part shown above) Fred finishes, then Ricky, and finally Lucy finishes with a long sequence of output. So even though Ricky was started third, he actually

completed second. And if we run it again, we'll get a different result. Why? Because it's up to the scheduler, and we don't control the scheduler! Which brings up another key point to remember: Just because a series of threads are started in a particular order doesn't mean they'll run in that order. For any group of started threads, order is not guaranteed by the scheduler. And duration is not guaranteed. You don't know, for example, if one thread will run to completion before the others have a chance to get in or whether they'll all take turns nicely, or whether they'll do a combination of both. There is a way, however, to start a thread but tell it not to run until some other thread has finished. You can do this with the `join()` method, which we'll look at a little later.

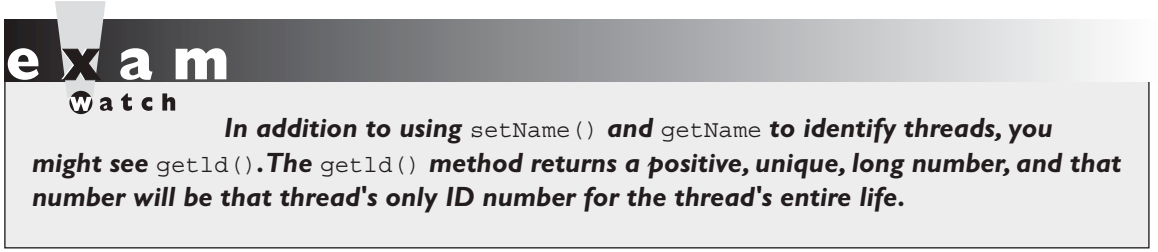
A thread is done being a thread when its target `run()` *method completes.*

When a thread completes its `run()` method, the thread ceases to be a thread of execution. The stack for that thread dissolves, and the thread is considered dead. (Technically the API calls a dead thread "terminated", but we'll use "dead" in this chapter.) Not dead and gone, however, just dead. It's still a Thread *object*, just not a *thread of execution*. So if you've got a reference to a Thread instance, then even when that Thread instance is no longer a thread of execution, you can still call methods on the Thread instance, just like any other Java object. What you can't do, though, is call `start()` again.

Once a thread has been started, it can never be started again.

If you have a reference to a Thread, and you call `start()`, it's started. If you call `start()` a second time, it will cause an exception (an IllegalThreadStateException, which is a kind of RuntimeException, but you don't need to worry about the exact type). This happens whether or not the `run()` method has completed from the first `start()` call. Only a new thread can be started, and then only once. A runnable thread or a dead thread cannot be restarted.

So far, we've seen three thread states: *new*, *runnable*, and *dead*. We'll look at more thread states before we're done with this chapter.



In addition to using `setName()` ***and*** `getName` ***to identify threads, you might see*** `getId()`***. The*** `getId()` ***method returns a positive, unique, long number, and that number will be that thread's only ID number for the thread's entire life.***

The Thread Scheduler

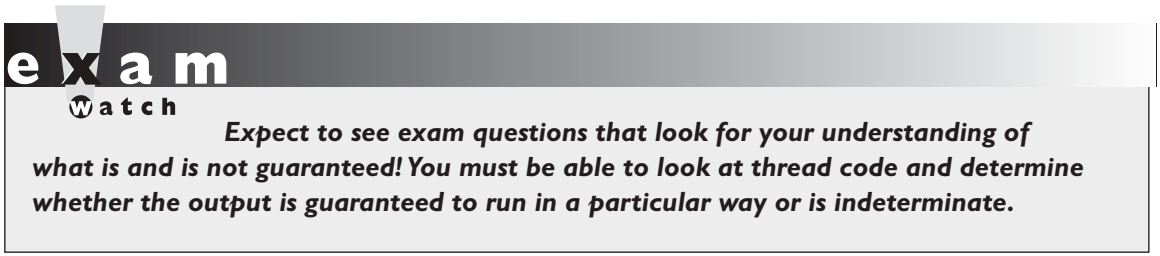
The thread scheduler is the part of the JVM (although most JVMs map Java threads directly to native threads on the underlying OS) that decides which thread should run at any given moment, and also takes threads *out* of the run state. Assuming a single processor machine, only one thread can actually *run* at a time. Only one stack can ever be executing at one time. And it's the thread scheduler that decides *which* thread—of all that are eligible—will actually *run*. When we say *eligible*, we really mean *in the runnable state*.

Any thread in the *runnable* state can be chosen by the scheduler to be the one and only running thread. If a thread is not in a runnable state, then it cannot be chosen to be the *currently running* thread. And just so we're clear about how little is guaranteed here:

The order in which runnable threads are chosen to run is not guaranteed.

Although *queue* behavior is typical, it isn't guaranteed. Queue behavior means that when a thread has finished with its "turn," it moves to the end of the line of the runnable pool and waits until it eventually gets to the front of the line, where it can be chosen again. In fact, we call it a runnable *pool*, rather than a runnable *queue*, to help reinforce the fact that threads aren't all lined up in some guaranteed order.

Although we don't *control* the thread scheduler (we can't, for example, tell a specific thread to run), we can sometimes influence it. The following methods give us some tools for *influencing* the scheduler. Just don't ever mistake influence for control.



Expect to see exam questions that look for your understanding of what is and is not guaranteed! You must be able to look at thread code and determine whether the output is guaranteed to run in a particular way or is indeterminate.

Methods from the java.lang.Thread Class Some of the methods that can help us influence thread scheduling are as follows:

```
public static void sleep(long millis) throws InterruptedException
public static void yield()
public final void join() throws InterruptedException
public final void setPriority(int newPriority)
```

Note that both `sleep()` and `join()` have overloaded versions not shown here.

Methods from the java.lang.Object Class Every class in Java inherits the following three thread-related methods:

```
public final void wait() throws InterruptedException
public final void notify()
public final void notifyAll()
```

The `wait()` method has three overloaded versions (including the one listed here).

We'll look at the behavior of each of these methods in this chapter. First, though, we're going to look at the different states a thread can be in.

CERTIFICATION OBJECTIVE

Thread States and Transitions (Objective 4.2)

4.2 Recognize the states in which a thread can exist, and identify ways in which a thread can transition from one state to another.

We've already seen three thread states— *new*, *runnable*, and *dead*—but wait! There's more! The thread scheduler's job is to move threads in and out of the *running* state. While the thread scheduler can move a thread from the running state back to runnable, other factors can cause a thread to move out of running, but *not* back to runnable. One of these is when the thread's `run()` method completes, in which case the thread moves from the running state directly to the dead state. Next we'll look at some of the other ways in which a thread can leave the running state, and where the thread goes.

Thread States

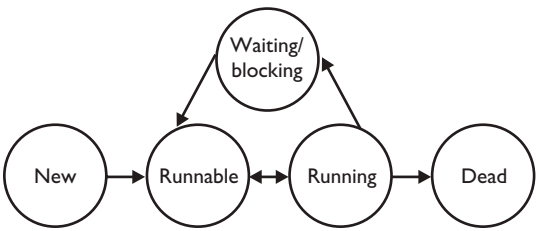
A thread can be only in one of five states (see Figure 9-2):

- **New** This is the state the thread is in after the Thread instance has been created, but the `start()` method has not been invoked on the thread. It is a live Thread object, but not yet a thread of execution. At this point, the thread is considered *not alive*.

- **Runnable** This is the state a thread is in when it's eligible to run, but the scheduler has not selected it to be the running thread. A thread first enters the runnable state when the `start()` method is invoked, but a thread can also return to the runnable state after either running or coming back from a blocked, waiting, or sleeping state. When the thread is in the runnable state, it is considered *alive*.
- **Running** This is it. The "big time." Where the action is. This is the state a thread is in when the thread scheduler selects it (from the runnable pool) to be the currently executing process. A thread can transition out of a running state for several reasons, including because "the thread scheduler felt like it." We'll look at those other reasons shortly. Note that in Figure 9-2, there are several ways to get to the runnable state, but only *one* way to get to the running state: the scheduler chooses a thread from the runnable pool.

FIGURE 9-2

Transitioning between thread states



- **Waiting/blocked/sleeping** This is the state a thread is in when it's not eligible to run. Okay, so this is really three states combined into one, but they all have one thing in common: the thread is still alive, but is currently not eligible to run. In other words, it is not *runnable*, but it might *return* to a runnable state later if a particular event occurs. A thread may be *blocked* waiting for a resource (like I/O or an object's lock), in which case the event that sends it back to runnable is the availability of the resource—for example, if data comes in through the input stream the thread code is reading from, or if the object's lock suddenly becomes available. A thread may be *sleeping* because the thread's run code *tells* it to sleep for some period of time, in which case the event that sends it back to runnable is that it wakes up because its sleep time has expired. Or the thread may be *waiting*, because the thread's run code *causes* it to wait, in which case the event that sends it back to runnable is that another thread sends a notification that it may no longer be necessary for the thread to wait. The important point is that one thread

does not *tell* another thread to block. Some methods may *look* like they tell another thread to block, but they don't. If you have a reference `t` to another thread, you can write something like this:

```
t.sleep();   or     t.yield()
```

But those are actually static methods of the Thread class—*they don't affect the instance* `t`; instead they are defined to always affect the thread that's currently executing. (This is a good example of why it's a bad idea to use an instance variable to access a `static` method—it's misleading. There *is* a method, `suspend()`, in the Thread class, that lets one thread tell another to suspend, but the `suspend()` method has been deprecated and won't be on the exam (nor will its counterpart `resume()`). There is also a `stop()` method, but it too has been deprecated and we won't even go there. Both `suspend()` and `stop()` turned out to be very dangerous, so you shouldn't use them and again, because they're deprecated, they won't appear on the exam. Don't study 'em, don't use 'em. Note also that a thread in a blocked state is still considered to be *alive*.

- **Dead** A thread is considered dead when its `run()` method completes. It may still be a viable Thread object, but it is no longer a separate thread of execution. Once a thread is dead, it can never be brought back to life! (The whole "I see dead threads" thing.) If you invoke `start()` on a dead Thread instance, you'll get a runtime (not compiler) exception. And it probably doesn't take a rocket scientist to tell you that if a thread is dead, it is no longer considered to be *alive*.

Preventing Thread Execution

A thread that's been stopped usually means a thread that's moved to the dead state. But Objective 4.2 is also looking for your ability to recognize when a thread will get kicked out of running but *not* be sent back to either runnable or dead.

For the purpose of the exam, we aren't concerned with a thread blocking on I/O (say, waiting for something to arrive from an input stream from the server). We *are* concerned with the following:

- Sleeping
- Waiting
- Blocked because it needs an object's lock

Sleeping

The `sleep()` method is a `static` method of class Thread. You use it in your code to "slow a thread down" by forcing it to go into a sleep mode before coming back to runnable (where it still has to beg to be the currently running thread). When a thread sleeps, it drifts off somewhere and doesn't return to runnable until it wakes up.

So why would you want a thread to sleep? Well, you might think the thread is moving too quickly through its code. Or you might need to force your threads to take turns, since reasonable turn-taking isn't guaranteed in the Java specification. Or imagine a thread that runs in a loop, downloading the latest stock prices and analyzing them. Downloading prices one after another would be a waste of time, as most would be quite similar—and even more important, it would be an incredible waste of precious bandwidth. The simplest way to solve this is to cause a thread to pause (sleep) for five minutes after each download.

You do this by invoking the static `Thread.sleep()` method, giving it a time in milliseconds as follows:

```
try {
  Thread.sleep(5*60*1000);  // Sleep for 5 minutes
} catch (InterruptedException ex) { }
```

Notice that the `sleep()` method can throw a checked InterruptedException (you'll usually know if that is a possibility, since another thread has to explicitly do the interrupting), so you must acknowledge the exception with a handle or declare. Typically, you wrap calls to `sleep()` in a `try/catch`, as in the preceding code.

Let's modify our Fred, Lucy, Ricky code by using `sleep()` to *try* to force the threads to alternate rather than letting one thread dominate for any period of time. Where do you think the `sleep()` method should go?

```
class NameRunnable implements Runnable {
   public void run() {
      for (int x = 1; x < 4; x++) {
         System.out.println("Run by "
          + Thread.currentThread().getName());
```

```
            try {
              Thread.sleep(1000);
            } catch (InterruptedException ex) { }
          }
      }
    }

    public class ManyNames {
       public static void main (String [] args) {

         // Make one Runnable
         NameRunnable nr = new NameRunnable();

         Thread one = new Thread(nr);
         one.setName("Fred");
         Thread two = new Thread(nr);
         two.setName("Lucy");
         Thread three = new Thread(nr);
         three.setName("Ricky");

         one.start();
         two.start();
         three.start();
      }
    }
```

Running this code shows Fred, Lucy, and Ricky alternating nicely:

```
% java ManyNames
Run by Fred
Run by Lucy
Run by Ricky
Run by Fred
Run by Lucy
Run by Ricky
Run by Fred
Run by Lucy
Run by Ricky
```

Just keep in mind that the behavior in the preceding output is still not guaranteed. You can't be certain how long a thread will actually run *before* it gets put to sleep, so you can't know with certainty that only one of the three threads will be in the runnable state when the running thread goes to sleep. In other words, if there are

two threads awake and in the runnable pool, you can't know with certainty that the least recently used thread will be the one selected to run. *Still, using* `sleep()` *is the best way to help all threads get a chance to run!* Or at least to guarantee that one thread doesn't get in and stay until it's done. When a thread encounters a sleep call, it *must* go to sleep for *at least* the specified number of milliseconds (unless it is interrupted before its wake-up time, in which case it immediately throws the InterruptedException).

exam Watch

Just because a thread's `sleep()` ***expires, and it wakes up, does not mean it will return to running! Remember,* when a thread wakes up, it simply goes back to the runnable state. *So the time specified in* `sleep()` *is the* minimum *duration in which the thread won't run, but it is not the exact duration in which the thread won't run. So you can't, for example, rely on the* `sleep()` *method to give you a perfectly accurate timer. Although in many applications using* `sleep()` *as a timer is certainly good enough, you must know that a* `sleep()` *time is not a guarantee that the thread will start running again as soon as the time expires and the thread wakes.***

Remember that `sleep()` is a static method, so don't be fooled into thinking that one thread can put another thread to sleep. You can put `sleep()` code anywhere, since *all* code is being run by *some* thread. When the executing code (meaning the currently running thread's code) hits a `sleep()` call, it puts the currently running thread to sleep.

EXERCISE 9-1

Creating a Thread and Putting It to Sleep

In this exercise we will create a simple counting thread. It will count to 100, pausing one second between each number. Also, in keeping with the counting theme, it will output a string every ten numbers.

1. Create a class and extend the Thread class. As an option, you can implement the Runnable interface.
2. Override the `run()` method of Thread. This is where the code will go that will output the numbers.
3. Create a `for` loop that will loop 100 times. Use the modulo operation to check whether there are any remainder numbers when divided by 10.
4. Use the `static` method `Thread.sleep()` to pause. (Remember, the one-arg version of `sleep()` specifies the amount of time of sleep in milliseconds.)

Thread Priorities and yield( )

To understand `yield()`, you must understand the concept of thread *priorities*. Threads always run with some priority, usually represented as a number between 1 and 10 (although in some cases the range is less than 10). The scheduler in most JVMs uses preemptive, priority-based scheduling (which implies some sort of time slicing). *This does not mean that all JVMs use time slicing.* The JVM specification does not require a VM to implement a time-slicing scheduler, where each thread is allocated a fair amount of time and then sent back to runnable to give another thread a chance. Although many JVMs do use time slicing, some may use a scheduler that lets one thread stay running until the thread completes its `run()` method.

In most JVMs, however, the scheduler does use thread priorities in one important way: If a thread enters the runnable state, and it has a higher priority than any of the threads in the pool and a higher priority than the currently running thread, *the lower-priority running thread usually will be bumped back to runnable and the highest-priority thread will be chosen to run.* In other words, at any given time the currently running thread usually will not have a priority that is lower than any of the threads in the pool. *In most cases, the running thread will be of equal or greater priority than the highest priority threads in the pool.* This is as close to a guarantee about scheduling as you'll get from the JVM specification, so you must never rely on thread priorities to guarantee the correct behavior of your program.

Don't rely on thread priorities when designing your multithreaded application. Because thread-scheduling priority behavior is not guaranteed, use thread priorities as a way to improve the efficiency of your program, but just be sure your program doesn't depend on that behavior for correctness.

What is also *not* guaranteed is the behavior when threads in the pool are of equal priority, or when the currently running thread has the same priority as threads in the pool. All priorities being equal, a JVM implementation of the scheduler is free to do just about anything it likes. That means a scheduler might do one of the following (among other things):

- Pick a thread to run, and run it there until it blocks or completes.
- Time slice the threads in the pool to give everyone an equal opportunity to run.

Setting a Thread's Priority A thread gets a default priority that is *the priority of the thread of execution that creates it*. For example, in the code

```
public class TestThreads {
  public static void main (String [] args) {
    MyThread t = new MyThread();
  }
}
```

the thread referenced by `t` will have the same priority as the *main* thread, since the main thread is executing the code that creates the MyThread instance.

You can also set a thread's priority directly by calling the `setPriority()` method on a Thread instance as follows:

```
FooRunnable r = new FooRunnable();
Thread t = new Thread(r);
t.setPriority(8);
t.start();
```

Priorities are set using a positive integer, usually between 1 and 10, and the JVM will never change a thread's priority. However, the values 1 through 10 are not guaranteed. Some JVM's might not recognize ten distinct values. Such a JVM might merge values from 1 to 10 down to maybe values from 1 to 5, so if you have, say, ten threads each with a different priority, and the current application is running in a JVM that allocates a range of only five priorities, then two or more threads might be mapped to one priority.

Although *the default priority is 5*, the Thread class has the three following constants (`static final` variables) that define the range of thread priorities:

```
Thread.MIN_PRIORITY  (1)
Thread.NORM_PRIORITY  (5)
Thread.MAX_PRIORITY  (10)
```

The yield( ) Method So what does the `static Thread.yield()` have to do with all this? Not that much, in practice. What `yield()` is *supposed* to do is make the currently running thread head back to runnable to allow other threads of the same priority to get their turn. So the intention is to use `yield()` to promote graceful turn-taking among equal-priority threads. In reality, though, the `yield()` method isn't guaranteed to do what it claims, and even if `yield()` does cause a thread to step out of running and back to runnable, *there's no guarantee the yielding thread won't just be chosen again over all the others!* So while `yield()` might—and often does—make a running thread give up its slot to another runnable thread of the same priority, there's no guarantee.

A `yield()` won't ever cause a thread to go to the waiting/sleeping/ blocking state. At most, a `yield()` will cause a thread to go from running to runnable, but again, it might have no effect at all.

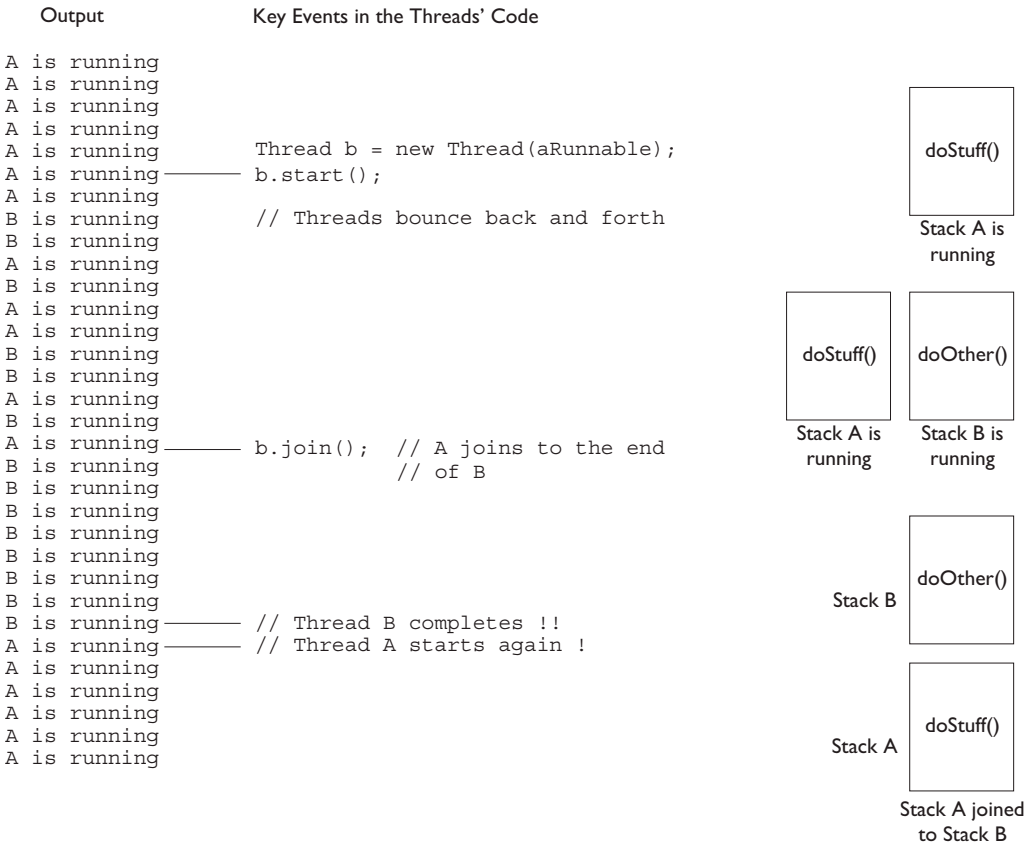
The join( ) Method

The non-`static` `join()` method of class Thread lets one thread "join onto the end" of another thread. If you have a thread B that can't do its work until another thread A has completed *its* work, then you want thread B to "join" thread A. This means that thread B will not become runnable until A has finished (and entered the dead state).

```
Thread t = new Thread();
t.start();
t.join();
```

The preceding code takes the currently running thread (if this were in the `main()` method, then that would be the main thread) and *joins* it to the end of the thread referenced by `t`. This blocks the current thread from becoming runnable until after the thread referenced by `t` is no longer alive. In other words, the code `t.join()` means "Join me (the current thread) to the end of `t`, so that `t` must finish before I (the current thread) can run again." You can also call one of the overloaded versions of `join()` that takes a timeout duration, so that you're saying, "wait until thread `t` is done, but if it takes longer than 5,000 milliseconds, then stop waiting and become runnable anyway." Figure 9-3 shows the effect of the `join()` method.

FIGURE 9-3 The `join()` method



So far we've looked at three ways a running thread could leave the running state:

- **A call to `sleep()`** Guaranteed to cause the current thread to stop executing for at least the specified sleep duration (although it might be *interrupted* before its specified time).
- **A call to `yield()`** Not guaranteed to do much of anything, although typically it will cause the currently running thread to move back to runnable so that a thread of the same priority can have a chance.
- **A call to `join()`** Guaranteed to cause the current thread to stop executing until the thread it joins with (in other words, the thread it calls `join()`

on) completes, or if the thread it's trying to join with is not alive, however, the current thread won't need to back out.

Besides those three, we also have the following scenarios in which a thread might leave the running state:

- The thread's `run()` method completes. Duh.
- A call to `wait()` on an object (we don't call `wait()` on a *thread*, as we'll see in a moment).
- A thread can't acquire the *lock* on the object whose method code it's attempting to run.
- The thread scheduler can decide to move the current thread from running to runnable in order to give another thread a chance to run. No reason is needed—the thread scheduler can trade threads in and out whenever it likes.

CERTIFICATION OBJECTIVE

Synchronizing Code (Objective 4.3)

4.3 Given a scenario, write code that makes appropriate use of object locking to protect static or instance variables from concurrent access problems.

Can you imagine the havoc that can occur when two different threads have access to a single instance of a class, and both threads invoke methods on that object…and those methods modify the state of the object? In other words, what might happen if *two* different threads call, say, a setter method on a *single* object? A scenario like that might corrupt an object's state (by changing its instance variable values in an inconsistent way), and if that object's state is data shared by other parts of the program, well, it's too scary to even visualize.

But just because we enjoy horror, let's look at an example of what might happen. The following code demonstrates what happens when two different threads are accessing the same account data. Imagine that two people each have a checkbook for a single checking account (or two people each have ATM cards, but both cards are linked to only one account).

In this example, we have a class called Account that represents a bank account. To keep the code short, this account starts with a balance of 50, and can be used only for withdrawals. The withdrawal will be accepted even if there isn't enough money in the account to cover it. The account simply reduces the balance by the amount you want to withdraw:

```
class Account {
   private int balance = 50;
   public int getBalance() {
      return balance;
   }
   public void withdraw(int amount) {
      balance = balance - amount;
   }
}
```

Now here's where it starts to get fun. Imagine a couple, Fred and Lucy, who both have access to the account and want to make withdrawals. But they don't want the account to ever be overdrawn, so just before one of them makes a withdrawal, he or she will first check the balance to be certain there's enough to cover the withdrawal. Also, withdrawals are always limited to an amount of 10, so there must be at least 10 in the account balance in order to make a withdrawal. Sounds reasonable. But that's a two-step process:

1. Check the balance.
2. If there's enough in the account (in this example, at least 10), make the withdrawal.

What happens if something separates step 1 from step 2? For example, imagine what would happen if Lucy checks the balance and sees that there's just exactly enough in the account, 10. *But before she makes the withdrawal, Fred checks the balance and also sees that there's enough for his withdrawal.* Since Lucy has verified the balance, but not yet made her withdrawal, Fred is seeing "bad data." He is seeing the account balance *before* Lucy actually debits the account, but at this point that debit is certain to occur. Now both Lucy and Fred believe there's enough to make their withdrawals. So now imagine that Lucy makes *her* withdrawal, and now there isn't enough in the account for Fred's withdrawal, but he thinks there is since when he checked, there was enough! Yikes. In a minute we'll see the actual banking code, with Fred and Lucy, represented by two threads, each acting on the same Runnable, and that Runnable holds a reference to the one and only account instance—so, two threads, one account.

The logic in our code example is as follows:

1. The Runnable object holds a reference to a single account.
2. Two threads are started, representing Lucy and Fred, and each thread is given a reference to the same Runnable (which holds a reference to the actual account)
3. The initial balance on the account is 50, and each withdrawal is exactly 10.
4. In the `run()` method, we loop 5 times, and in each loop we
   - Make a withdrawal (if there's enough in the account).
   - Print a statement *if the account is overdrawn* (which it should never be, since we check the balance *before* making a withdrawal).
5. The `makeWithdrawal()` method in the test class (representing the behavior of Fred or Lucy) will do the following:
   - Check the balance to see if there's enough for the withdrawal.
   - If there is enough, print out the name of the one making the withdrawal.
   - Go to sleep for 500 milliseconds—just long enough to give the other partner a chance to get in before you actually *make* the withdrawal.
   - Upon waking up, complete the withdrawal and print that fact.
   - If there wasn't enough in the first place, print a statement showing who you are and the fact that there wasn't enough.

So what we're really trying to discover is if the following is possible: for one partner to check the account and see that there's enough, but before making the actual withdrawal, the other partner checks the account and *also* sees that there's enough. When the account balance gets to 10, if both partners check it before making the withdrawal, both will think it's OK to withdraw, and the account will overdraw by 10!

Here's the code:

```
public class AccountDanger implements Runnable {
   private Account acct = new Account();
   public static void main (String [] args) {
      AccountDanger r = new AccountDanger();
      Thread one = new Thread(r);
      Thread two = new Thread(r);
      one.setName("Fred");
      two.setName("Lucy");
```

```
      one.start();
      two.start();
    }
  public void run() {
    for (int x = 0; x < 5; x++) {
      makeWithdrawal(10);
      if (acct.getBalance() < 0) {
        System.out.println("account is overdrawn!");
      }
    }
  }
  private void makeWithdrawal(int amt) {
     if (acct.getBalance() >= amt) {
        System.out.println(Thread.currentThread().getName()
                   + " is going to withdraw");
        try {
          Thread.sleep(500);
        } catch(InterruptedException ex) { }
        acct.withdraw(amt);
        System.out.println(Thread.currentThread().getName()
                   + " completes the withdrawal");
     } else {
        System.out.println("Not enough in account for "
                   + Thread.currentThread().getName()
                   + " to withdraw " + acct.getBalance());
     }
   }
}
```

So what happened? Is it possible that, say, Lucy checked the balance, fell asleep, Fred checked the balance, Lucy woke up and completed *her* withdrawal, then Fred completes *his* withdrawal, and in the end they overdraw the account? Look at the (numbered) output:

```
% java AccountDanger
 1. Fred is going to withdraw
 2. Lucy is going to withdraw
 3. Fred completes the withdrawal
 4. Fred is going to withdraw
 5. Lucy completes the withdrawal
 6. Lucy is going to withdraw
 7. Fred completes the withdrawal
 8. Fred is going to withdraw
 9. Lucy completes the withdrawal
```

```
10. Lucy is going to withdraw
11. Fred completes the withdrawal
12. Not enough in account for Fred to withdraw 0
13. Not enough in account for Fred to withdraw 0
14. Lucy completes the withdrawal
15. account is overdrawn!
16. Not enough in account for Lucy to withdraw -10
17. account is overdrawn!
18. Not enough in account for Lucy to withdraw -10
19. account is overdrawn!
```

Although each time you run this code the output might be a little different, let's walk through this particular example using the numbered lines of output. For the first four attempts, everything is fine. Fred checks the balance on line 1, and finds it's OK. At line 2, Lucy checks the balance and finds it OK. At line 3, Fred makes his withdrawal. At this point, the balance Lucy checked for (and believes is still accurate) has actually changed since she last checked. And now Fred checks the balance *again*, before Lucy even completes her first withdrawal. By this point, even Fred is seeing a potentially inaccurate balance, because we know Lucy is going to complete her withdrawal. It is possible, of course, that Fred will complete his before Lucy does, but that's not what happens here.

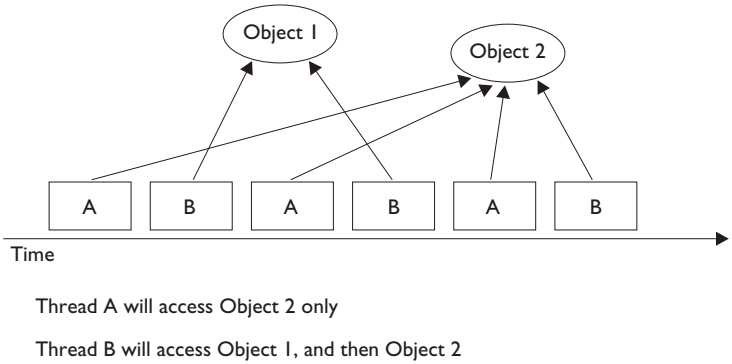
On line 5, Lucy completes her withdrawal and then before Fred completes his, Lucy does another check on the account on line 6. And so it continues until we get to line 8, where Fred checks the balance and sees that it's 20. On line 9, Lucy completes a withdrawal (that she had checked for earlier), and this takes the balance to 10. On line 10, Lucy checks again, sees that the balance is 10, so she knows she can do a withdrawal. *But she didn't know that Fred, too, has already checked the balance on line 8 so he thinks it's safe to do the withdrawal!* On line 11, Fred completes the withdrawal he approved on line 8. This takes the balance to zero. But Lucy still has a pending withdrawal that she got approval for on line 10! You know what's coming.

On lines 12 and 13, Fred checks the balance and finds that there's not enough in the account. But on line 14, Lucy completes her withdrawal and BOOM! The account is now overdrawn by 10—*something we thought we were preventing by doing a balance check prior to a withdrawal*.

Figure 9-4 shows the timeline of what can happen when two threads concurrently access the same object.

FIGURE 9-4

Problems with concurrent access



This problem is known as a "race condition," where multiple threads can access the same resource (typically an object's instance variables), and can produce corrupted data if one thread "races in" too quickly before an operation that should be "atomic" has completed.

Preventing the Account Overdraw So what can be done? The solution is actually quite simple. We must guarantee that the two steps of the withdrawal—*checking* the balance and *making* the withdrawal—are never split apart. We need them to always be performed as one operation, even when the thread falls asleep in between step 1 and step 2! We call this an "atomic operation" (although the physics is a little outdated, in this case "atomic" means "indivisible") because the operation, regardless of the number of actual statements (or underlying byte code instructions), is completed *before* any other thread code that acts on the same data.

You can't guarantee that a single thread will stay running throughout the entire atomic operation. But you *can* guarantee that even if the thread running the atomic operation moves in and out of the running state, *no other running thread will be able to act on the same data.* In other words, If Lucy falls asleep after checking the balance, we can stop Fred from checking the balance until *after* Lucy wakes up and completes her withdrawal.

So how do you protect the data? You must do two things:

- Mark the variables `private`.
- Synchronize the code that modifies the variables.

Remember, you protect the variables in the normal way—using an access control modifier. It's the method code that you must protect, so that only one thread at a time can be executing that code. You do this with the `synchronized` keyword.

We can solve all of Fred and Lucy's problems by adding one word to the code. We mark the `makeWithdrawal()` method `synchronized` as follows:

```
private synchronized void makeWithdrawal(int amt) {
  if (acct.getBalance() >= amt) {
    System.out.println(Thread.currentThread().getName() +
                       " is going to withdraw");
    try {
      Thread.sleep(500);
    } catch(InterruptedException ex) { }
    acct.withdraw(amt);
    System.out.println(Thread.currentThread().getName() +
                        " completes the withdrawal");
  } else {
    System.out.println("Not enough in account for "
                      + Thread.currentThread().getName()
                      + " to withdraw " + acct.getBalance());
  }
}
```

Now we've guaranteed that once a thread (Lucy or Fred) starts the withdrawal process (by invoking `makeWithdrawal()`), the other thread cannot enter that method until the first one completes the process by exiting the method. The new output shows the benefit of synchronizing the `makeWithdrawal()` method:

```
% java AccountDanger
Fred is going to withdraw
Fred completes the withdrawal
Lucy is going to withdraw
Lucy completes the withdrawal
Fred is going to withdraw
Fred completes the withdrawal
Lucy is going to withdraw
Lucy completes the withdrawal
Fred is going to withdraw
Fred completes the withdrawal
Not enough in account for Lucy to withdraw 0
Not enough in account for Fred to withdraw 0
Not enough in account for Lucy to withdraw 0
Not enough in account for Fred to withdraw 0
Not enough in account for Lucy to withdraw 0
```

Notice that now both threads, Lucy and Fred, always check the account balance *and* complete the withdrawal before the other thread can check the balance.

Synchronization and Locks

How does synchronization work? With locks. Every object in Java has a built-in lock that only comes into play when the object has synchronized method code. When we enter a synchronized non-`static` method, we automatically acquire the lock associated with the current instance of the class whose code we're executing (the `this` instance). Acquiring a lock for an object is also known as getting the lock, or locking the object, locking *on* the object, or synchronizing on the object. We may also use the term *monitor* to refer to the object whose lock we're acquiring. Technically the lock and the monitor are two different things, but most people talk about the two interchangeably, and we will too.

Since there is only one lock per object, if one thread has picked up the lock, no other thread can pick up the lock until the first thread releases (or returns) the lock. This means no other thread can enter the synchronized code (which means it can't enter any `synchronized` method of that object) until the lock has been released. Typically, releasing a lock means the thread holding the lock (in other words, the thread currently in the synchronized method) exits the `synchronized` method. At that point, the lock is free until some other thread enters a `synchronized` method on that object. Remember the following key points about locking and synchronization:

- Only methods (or blocks) can be `synchronized`, not variables or classes.
- Each object has just one lock.
- Not all methods in a class need to be synchronized. A class can have both `synchronized` and non-`synchronized` methods.
- If two threads are about to execute a `synchronized` method in a class, and both threads are using the same instance of the class to invoke the method, only one thread at a time will be able to execute the method. The other thread will need to wait until the first one finishes its method call. In other words, once a thread acquires the lock on an object, no other thread can enter any of the synchronized methods in that class (for that object).

- If a class has both `synchronized` and non-`synchronized` methods, multiple threads can still access the class's non-`synchronized` methods! If you have methods that don't access the data you're trying to protect, then you don't need to synchronize them. Synchronization can cause a hit in some cases (or even deadlock if used incorrectly), so you should be careful not to overuse it.
- If a thread goes to sleep, it holds any locks it has—it doesn't release them.
- A thread can acquire more than one lock. For example, a thread can enter a `synchronized` method, thus acquiring a lock, and then immediately invoke a `synchronized` method on a different object, thus acquiring that lock as well. As the stack unwinds, locks are released again. Also, if a thread acquires a lock and then attempts to call a `synchronized` method on that same object, no problem. The JVM knows that this thread already has the lock for this object, so the thread is free to call other `synchronized` methods on the same object, using the lock the thread already has.
- You can synchronize a block of code rather than a method.

Because synchronization does hurt concurrency, you don't want to synchronize any more code than is necessary to protect your data. So if the scope of a method is more than needed, you can reduce the scope of the synchronized part to something less than a full method—to just a block. We call this, strangely, a *synchronized block*, and it looks like this:

```
class SyncTest {
  public void doStuff() {
    System.out.println("not synchronized");
    synchronized(this) {
      System.out.println("synchronized");
    }
  }
}
```

When a thread is executing code from within a `synchronized` block, including any method code invoked from that `synchronized` block, the code is said to be executing in a synchronized context. The real question is, synchronized on what? Or, synchronized on which object's lock?

When you synchronize a method, the object used to invoke the method is the object whose lock must be acquired. But when you synchronize a block of code, you

specify which object's lock you want to use as the lock, so you could, for example, use some third-party object as the lock for this piece of code. That gives you the ability to have more than one lock for code synchronization within a single object.

Or you can synchronize on the current instance (`this`) as in the code above. Since that's the same instance that `synchronized` methods lock on, it means that you could always replace a `synchronized` method with a non-`synchronized` method containing a `synchronized` block. In other words, this:

```
public synchronized void doStuff() {
    System.out.println("synchronized");
}
```

is equivalent to this:

```
public void doStuff() {
    synchronized(this) {
        System.out.println("synchronized");
    }
}
```

These methods both have the exact same effect, in practical terms. The compiled bytecodes may not be exactly the same for the two methods, but they *could* be—and any differences are not really important. The first form is shorter and more familiar to most people, but the second can be more flexible.

So What About Static Methods? Can They Be Synchronized?

`static` methods can be `synchronized`. There is only one copy of the static data you're trying to protect, so you only need one lock per class to synchronize static methods—a lock for the whole class. There is such a lock; every class loaded in Java has a corresponding instance of java.lang.Class representing that class. It's that java.lang.Class instance whose lock is used to protect the `static` methods of the class (if they're `synchronized`). There's nothing special you have to do to synchronize a `static` method:

```
public static synchronized int getCount() {
    return count;
}
```

Again, this could be replaced with code that uses a `synchronized` block. If the method is defined in a class called MyClass, the equivalent code is as follows:

```
public static int getCount() {
    synchronized(MyClass.class) {
        return count;
    }
}
```

Wait—what's that `MyClass.class` thing? That's called a *class literal*. It's a special feature in the Java language that tells the compiler (who tells the JVM): go and find me the instance of Class that represents the class called MyClass. You can also do this with the following code:

```
public static void classMethod() {
    Class cl = Class.forName("MyClass");
    synchronized (cl) {
        // do stuff
    }
}
```

However that's longer, ickier, and most important, *not on the SCJP exam*. But it's quick and easy to use a class literal—just write the name of the class, and add `.class` at the end. No quotation marks needed. Now you've got an expression for the Class object you need to synchronize on.

EXERCISE 9-2

Synchronizing a Block of Code

In this exercise we will attempt to synchronize a block of code. Within that block of code we will get the lock on an object, so that other threads cannot modify it while the block of code is executing. We will be creating three threads that will all attempt to manipulate the same object. Each thread will output a single letter 100 times, and then increment that letter by one. The object we will be using is StringBuffer.

We could synchronize on a String object, but strings cannot be modified once they are created, so we would not be able to increment the letter without generating a new String object. The final output should have 100 As, 100 Bs, and 100 Cs all in unbroken lines.

1. Create a class and extend the Thread class.
2. Override the `run()` method of Thread. This is where the `synchronized` block of code will go.
3. For our three thread objects to share the same object, we will need to create a constructor that accepts a StringBuffer object in the argument.
4. The `synchronized` block of code will obtain a lock on the StringBuffer object from step 3.
5. Within the block, output the StringBuffer 100 times and then increment the letter in the StringBuffer. You can check Chapter 6 for StringBuffer methods that will help with this.
6. Finally, in the `main()` method, create a single StringBuffer object using the letter A, then create three instances of our class and start all three of them.

What Happens If a Thread Can't Get the Lock?

If a thread tries to enter a `synchronized` method and the lock is already taken, the thread is said to be blocked on the object's lock. Essentially, the thread goes into a kind of pool for that particular object and has to sit there until the lock is released and the thread can again become runnable/running. Just because a lock is released doesn't mean any particular thread will get it. There might be three threads waiting for a single lock, for example, and there's no guarantee that the thread that has waited the longest will get the lock first.

When thinking about blocking, it's important to pay attention to which objects are being used for locking.

- Threads calling non-`static` `synchronized` methods in the same class will only block each other if they're invoked using the same instance. That's because they each lock on `this` instance, and if they're called using two different instances, they get two locks, which do not interfere with each other.
- Threads calling `static` `synchronized` methods in the same class will always block each other—they all lock on the same Class instance.
- A `static` `synchronized` method and a non-`static` `synchronized` method will not block each other, ever. The `static` method locks on a Class instance while the non-`static` method locks on the `this` instance—these actions do not interfere with each other at all.

- For `synchronized` blocks, you have to look at exactly what object has been used for locking. (What's inside the parentheses after the word `synchronized`?) Threads that synchronize on the same object will block each other. Threads that synchronize on different objects will not.

Table 9-1 lists the thread-related methods and whether the thread gives up its lock as a result of the call.

TABLE 9-1 Methods and Lock Status

Give Up Locks	Keep Locks	Class Defining the Method
`wait ()`	`notify()` (Although the thread will probably exit the synchronized code shortly after this call, and thus give up its locks.)	`java.lang.Object`
	`join()`	`java.lang.Thread`
	`sleep()`	`java.lang.Thread`
	`yield()`	`java.lang.Thread`

So When Do I Need to Synchronize?

Synchronization can get pretty complicated, and you may be wondering why you would want to do this at all if you can help it. But remember the earlier "race conditions" example with Lucy and Fred making withdrawals from their account. When we use threads, we usually need to use some synchronization somewhere to make sure our methods don't interrupt each other at the wrong time and mess up our data. Generally, any time more than one thread is accessing mutable (changeable) data, you synchronize to protect that data, to make sure two threads aren't changing it at the same time (or that one isn't changing it at the same time the other is reading it, which is also confusing). You don't need to worry about local variables—each thread gets its own copy of a local variable. Two threads executing the same method at the same time will use different copies of the local variables, and they won't bother each other. However, you do need to worry about `static` and non-`static` fields, if they contain data that can be changed.

For changeable data in a non-`static` field, you usually use a non-`static` method to access it. By synchronizing that method, you will ensure that any threads trying

to run that method *using the same instance* will be prevented from simultaneous access. But a thread working with a *different* instance will not be affected, because it's acquiring a lock on the other instance. That's what we want—threads working with the same data need to go one at a time, but threads working with different data can just ignore each other and run whenever they want to; it doesn't matter.

For changeable data in a `static` field, you usually use a static method to access it. And again, by synchronizing the method you ensure that any two threads trying to access the data will be prevented from simultaneous access, because both threads will have to acquire locks on the Class object for the class the `static` method's defined in. Again, that's what we want.

However—what if you have a non-`static` method that accesses a `static` field? Or a `static` method that accesses a non-`static` field (using an instance)? In these cases things start to get messy quickly, and there's a very good chance that things will not work the way you want. If you've got a `static` method accessing a non-`static` field, and you synchronize the method, you acquire a lock on the Class object. But what if there's another method that also accesses the non-`static` field, this time using a non-`static` method? It probably synchronizes on the current instance (`this`) instead. Remember that a `static synchronized` method and a non-`static synchronized` method will not block each other—they can run at the same time. Similarly, if you access a `static` field using a non-`static` method, two threads might invoke that method using two different `this` instances. Which means they won't block each other, because they use different locks. Which means two threads are simultaneously accessing the same `static` field—exactly the sort of thing we're trying to prevent.

It gets very confusing trying to imagine all the weird things that can happen here. To keep things simple: in order to make a class thread-safe, methods that access changeable fields need to be `synchronized`.

Access to static fields should be done from `static synchronized` *methods. Access to non-*`static` *fields should be done from non-*`static synchronized` *methods*. For example:

```
public class Thing {
    private static int staticField;
    private int nonstaticField;
    public static synchronized int getStaticField() {
        return staticField;
    }
    public static synchronized void setStaticField(
                                            int staticField) {
```

```
        Thing.staticField = staticField;
    }
    public synchronized int getNonstaticField() {
        return nonstaticField;
    }
    public synchronized void setNonstaticField(
                                          int nonstaticField) {
        this.nonstaticField = nonstaticField;
    }
}
```

What if you need to access both `static` and `non-static` fields in a method? Well, there are ways to do that, but it's beyond what you need for the exam. You will live a longer, happier life if you JUST DON'T DO IT. Really. Would we lie?

Thread-Safe Classes

When a class has been carefully synchronized to protect its data (using the rules just given, or using more complicated alternatives), we say the class is "thread-safe." Many classes in the Java APIs already use synchronization internally in order to make the class "thread-safe." For example, StringBuffer and StringBuilder are nearly identical classes, except that all the methods in StringBuffer are `synchronized` when necessary, while those in StringBuilder are not. Generally, this makes StringBuffer safe to use in a multithreaded environment, while StringBuilder is not. (In return, StringBuilder is a little bit faster because it doesn't bother synchronizing.) However, even when a class is "thread-safe," it is often dangerous to rely on these classes to provide the thread protection you need. (C'mon, the repeated quotes used around "thread-safe" had to be a clue, right?) You still need to think carefully about how you use these classes, As an example, consider the following class.

```
import java.util.*;
public class NameList {
    private List names = Collections.synchronizedList(
                                            new LinkedList());
    public void add(String name) {
        names.add(name);
    }
    public String removeFirst() {
        if (names.size() > 0)
            return (String) names.remove(0);
        else
            return null;
```

```
    }
}
```

The method `Collections.synchronizedList()` returns a List whose methods are all `synchronized` and "thread-safe" according to the documentation (like a Vector—but since this is the 21st century, we're not going to use a Vector here). The question is, can the NameList class be used safely from multiple threads? It's tempting to think that yes, since the data in names is in a synchronized collection, the NameList class is "safe" too. However that's not the case—the `removeFirst()` may sometimes throw a NoSuchElementException. What's the problem? Doesn't it correctly check the `size()` of `names` before removing anything, to make sure there's something there? How could this code fail? Let's try to use NameList like this:

```
public static void main(String[] args) {
    final NameList nl = new NameList();
    nl.add("Ozymandias");
    class NameDropper extends Thread {
        public void run() {
            String name = nl.removeFirst();
            System.out.println(name);
        }
    }
    Thread t1 = new NameDropper();
    Thread t2 = new NameDropper();
    t1.start();
    t2.start();
}
```

What might happen here is that one of the threads will remove the one name and print it, then the other will try to remove a name and get `null`. If we think just about the calls to `names.size()` and `names.get(0)`, they occur in this order:

Thread `t1` executes `names.size()`, which returns 1.
Thread `t1` executes `names.remove(0)`, which returns `Ozymandias`.
Thread `t2` executes `names.size()`, which returns 0.
Thread `t2` does not call `remove(0)`.

The output here is

```
Ozymandias
null
```

However, if we run the program again something different might happen:

Thread `t1` executes `names.size()`, which returns 1.
Thread `t2` executes `names.size()`, which returns 1.
Thread `t1` executes `names.remove(0)`, which returns `Ozymandias`.
Thread `t2` executes `names.remove(0)`, which throws an exception because the list is now empty.

The thing to realize here is that in a "thread-safe" class like the one returned by `synchronizedList()`, each *individual* method is synchronized. So `names.size()` is `synchronized`, and `names.remove(0)` is `synchronized`. But nothing prevents another thread from doing something else to the list *in between* those two calls. And that's where problems can happen.

There's a solution here: don't rely on `Collections.synchronizedList()`. Instead, synchronize the code yourself:

```
import java.util.*;
public class NameList {
    private List names = new LinkedList();
    public synchronized void add(String name) {
        names.add(name);
    }
    public synchronized String removeFirst() {
        if (names.size() > 0)
            return (String) names.remove(0);
        else
            return null;
    }
}
```

Now the entire `removeFirst()` method is `synchronized`, and once one thread starts it and calls `names.size()`, there's no way the other thread can cut in and steal the last name. The other thread will just have to wait until the first thread completes the `removeFirst()` method.

The moral here is that just because a class is described as "thread-safe" doesn't mean it is *always* thread-safe. If individual methods are synchronized, that may not be enough—you may be better off putting in synchronization at a higher level (i.e., put it in the block or method that *calls* the other methods). Once you do that, the original synchronization (in this case, the synchronization inside the object returned by `Collections.synchronizedList()`) may well become redundant.

Thread Deadlock

Perhaps the scariest thing that can happen to a Java program is deadlock. Deadlock occurs when two threads are blocked, with each waiting for the other's lock. Neither can run until the other gives up its lock, so they'll sit there forever.

This can happen, for example, when thread A hits `synchronized` code, acquires a lock B, and then enters another method (still within the `synchronized` code it has the lock on) that's also `synchronized`. But thread A can't get the lock to enter this `synchronized` code—block C—because another thread D has the lock already. So thread A goes off to the waiting-for-the-C-lock pool, hoping that thread D will hurry up and release the lock (by completing the `synchronized` method). But thread A will wait a very long time indeed, because while thread D picked up lock C, it then entered a method `synchronized` on lock B. Obviously, thread D can't get the lock B because thread A has it. And thread A won't release it until thread D releases lock C. But thread D won't release lock C until after it can get lock B and continue. And there they sit. The following example demonstrates deadlock:

```
 1. public class DeadlockRisk {
 2.   private static class Resource {
 3.     public int value;
 4.   }
 5.   private Resource resourceA = new Resource();
 6.   private Resource resourceB = new Resource();
 7.   public int read() {
 8.     synchronized(resourceA) { // May deadlock here
 9.       synchronized(resourceB) {
10.         return resourceB.value + resourceA.value;
11.       }
12.     }
13.   }
14.
15.   public void write(int a, int b) {
16.     synchronized(resourceB) { // May deadlock here
17.       synchronized(resourceA) {
18.         resourceA.value = a;
19.         resourceB.value = b;
20.       }
21.     }
22.   }
23. }
```

Assume that `read()` is started by one thread and `write()` is started by another. If there are two different threads that may read and write independently, there is a risk of deadlock at line 8 or 16. The reader thread will have `resourceA`, the writer thread will have `resourceB`, and both will get stuck waiting for the other.

Code like this almost never results in deadlock because the CPU has to switch from the reader thread to the writer thread at a particular point in the code, and the chances of deadlock occurring are very small. The application may work fine 99.9 percent of the time.

The preceding simple example is easy to fix; just swap the order of locking for either the reader or the writer at lines 16 and 17 (or lines 8 and 9). More complex deadlock situations can take a long time to figure out.

Regardless of how little chance there is for your code to deadlock, the bottom line is, if you deadlock, you're dead. There are design approaches that can help avoid deadlock, including strategies for always acquiring locks in a predetermined order.

But that's for you to study and is beyond the scope of this book. We're just trying to get you through the exam. If you learn everything in this chapter, though, you'll still know more about threads than most experienced Java programmers.

CERTIFICATION OBJECTIVE

Thread Interaction (Objective 4.4)

4.4 Given a scenario, write code that makes appropriate use of wait, notify, or notifyAll.

The last thing we need to look at is how threads can interact with one another to communicate about—among other things—their locking status. The Object class has three methods, `wait()`, `notify()`, and `notifyAll()` that help threads communicate about the status of an event that the threads care about. For example, if one thread is a mail-delivery thread and one thread is a mail-processor thread, the mail-processor thread has to keep checking to see if there's any mail to process. Using the wait and notify mechanism, the mail-processor thread could check for mail, and if it doesn't find any it can say, "Hey, I'm not going to waste my time checking for mail every two seconds. I'm going to go hang out, and when the mail deliverer puts something in the mailbox, have him notify me so I can go back to runnable and do some work." In other words, using `wait()` and `notify()` lets one

thread put itself into a "waiting room" until some *other* thread notifies it that there's a reason to come back out.

One key point to remember (and keep in mind for the exam) about wait/notify is this:

> *`wait()`, `notify()`, and `notifyAll()` must be called from within a synchronized context! A thread can't invoke a wait or notify method on an object unless it owns that object's lock.*

Here we'll present an example of two threads that depend on each other to proceed with their execution, and we'll show how to use `wait()` and `notify()` to make them interact safely and at the proper moment.

Think of a computer-controlled machine that cuts pieces of fabric into different shapes and an application that allows users to specify the shape to cut. The current version of the application has one thread, which loops, first asking the user for instructions, and then directs the hardware to cut the requested shape:

```
public void run(){
   while(true){
      // Get shape from user
      // Calculate machine steps from shape
      // Send steps to hardware
   }
}
```

This design is not optimal because the user can't do anything while the machine is busy and while there are other shapes to define. We need to improve the situation.

A simple solution is to separate the processes into two different threads, one of them interacting with the user and another managing the hardware. The user thread sends the instructions to the hardware thread and then goes back to interacting with the user immediately. The hardware thread receives the instructions from the user thread and starts directing the machine immediately. Both threads use a common object to communicate, which holds the current design being processed.

The following pseudocode shows this design:

```
public void userLoop(){
   while(true){
      // Get shape from user
      // Calculate machine steps from shape
      // Modify common object with new machine steps
```

```
    }
}

public void hardwareLoop(){
    while(true){
        // Get steps from common object
        // Send steps to hardware
    }
}
```

The problem now is to get the hardware thread to process the machine steps as soon as they are available. Also, the user thread should not modify them until they have all been sent to the hardware. The solution is to use `wait()` and `notify()`, and also to synchronize some of the code.

The methods `wait()` and `notify()`, remember, are instance methods of Object. In the same way that every object has a lock, every object can have a list of threads that are waiting for a signal (a notification) from the object. A thread gets on this waiting list by executing the `wait()` method of the target object. From that moment, it doesn't execute any further instructions until the `notify()` method of the target object is called. If many threads are waiting on the same object, only one will be chosen (in no guaranteed order) to proceed with its execution. If there are no threads waiting, then no particular action is taken. Let's take a look at some real code that shows one object waiting for another object to notify it (take note, it is somewhat complex):

```
 1.  class ThreadA {
 2.     public static void main(String [] args) {
 3.        ThreadB b = new ThreadB();
 4.        b.start();
 5.
 6.        synchronized(b) {
 7.           try {
 8.             System.out.println("Waiting for b to complete...");
 9.             b.wait();
10.           } catch (InterruptedException e) {}
11.           System.out.println("Total is: " + b.total);
12.        }
13.     }
14.  }
15.
16.  class ThreadB extends Thread {
17.     int total;
```

```
18.
19.     public void run() {
20.        synchronized(this) {
21.           for(int i=0;i<100;i++) {
22.              total += i;
23.           }
24.           notify();
25.        }
26.     }
27.  }
```

This program contains two objects with threads: ThreadA contains the main thread and ThreadB has a thread that calculates the sum of all numbers from 0 through 99. As soon as line 4 calls the `start()` method, ThreadA will continue with the next line of code in its own class, which means it could get to line 11 before ThreadB has finished the calculation. To prevent this, we use the `wait()` method in line 9.

Notice in line 6 the code synchronizes itself with the object `b`—this is because in order to call `wait()` on the object, ThreadA must own a lock on `b`. For a thread to call `wait()` or `notify()`, the thread has to be the owner of the lock for that object. When the thread waits, it temporarily releases the lock for other threads to use, but it will need it again to continue execution. It's common to find code like this:

```
synchronized(anotherObject) { // this has the lock on anotherObject
   try {
      anotherObject.wait();
      // the thread releases the lock and waits
      // To continue, the thread needs the lock,
      // so it may be blocked until it gets it.
   } catch(InterruptedException e){}
}
```

The preceding code waits until `notify()` is called on `anotherObject`.

```
synchronized(this) { notify(); }
```

This code notifies a single thread currently waiting on the `this` object. The lock can be acquired much earlier in the code, such as in the calling method. Note that if the thread calling `wait()` does not own the lock, it will throw an `IllegalMonitorStateException`. This exception is not a checked exception,

so you don't have to *catch* it explicitly. You should always be clear whether a thread has the lock of an object in any given block of code.

Notice in lines 7–10 there is a `try/catch` block around the `wait()` method. A waiting thread can be interrupted in the same way as a sleeping thread, so you have to take care of the exception:

```
try {
   wait();
} catch(InterruptedException e) {
   // Do something about it
}
```

In the fabric example, the way to use these methods is to have the hardware thread wait on the shape to be available and the user thread to notify after it has written the steps. The machine steps may comprise global steps, such as moving the required fabric to the cutting area, and a number of substeps, such as the direction and length of a cut. As an example they could be

```
int fabricRoll;
int cuttingSpeed;
Point startingPoint;
float[] directions;
float[] lengths;
etc..
```

It is important that the user thread does not modify the machine steps while the hardware thread is using them, so this reading and writing should be synchronized.

The resulting code would look like this:

```
class Operator extends Thread {
   public void run(){
      while(true){
         // Get shape from user
         synchronized(this){
            // Calculate new machine steps from shape
            notify();
         }
      }
   }
}
class Machine extends Thread {
   Operator operator; // assume this gets initialized
```

```
    public void run(){
       while(true){
          synchronized(operator){
             try {
                operator.wait();
             } catch(InterruptedException ie) {}
             // Send machine steps to hardware
          }
       }
    }
}
```

The machine thread, once started, will immediately go into the waiting state and will wait patiently until the operator sends the first notification. At that point it is the operator thread that owns the lock for the object, so the hardware thread gets stuck for a while. It's only after the operator thread abandons the `synchronized` block that the hardware thread can really start processing the machine steps.

While one shape is being processed by the hardware, the user may interact with the system and specify another shape to be cut. When the user is finished with the shape and it is time to cut it, the operator thread attempts to enter the `synchronized` block, maybe blocking until the machine thread has finished with the previous machine steps. When the machine thread has finished, it repeats the loop, going again to the waiting state (and therefore releasing the lock). Only then can the operator thread enter the `synchronized` block and overwrite the machine steps with the new ones.

Having two threads is definitely an improvement over having one, although in this implementation there is still a possibility of making the user wait. A further improvement would be to have many shapes in a queue, thereby reducing the possibility of requiring the user to wait for the hardware.

There is also a second form of `wait()` that accepts a number of milliseconds as a maximum time to wait. If the thread is not interrupted, it will continue normally whenever it is notified or the specified timeout has elapsed. This normal continuation consists of getting out of the waiting state, but to continue execution it will have to get the lock for the object:

```
synchronized(a){ // The thread gets the lock on 'a'
  a.wait(2000); // Thread releases the lock and waits for notify
  // only for a maximum of two seconds, then goes back to Runnable
  // The thread reacquires the lock
  // More instructions here
}
```

exam Watch

When the `wait()` method is invoked on an object, the thread executing that code gives up its lock on the object immediately. However, when `notify()` is called, that doesn't mean the thread gives up its lock at that moment. If the thread is still completing synchronized code, the lock is not released until the thread moves out of synchronized code. So just because `notify()` is called doesn't mean the lock becomes available at that moment.

Using notifyAll( ) When Many Threads May Be Waiting

In most scenarios, it's preferable to notify *all* of the threads that are waiting on a particular object. If so, you can use `notifyAll()` on the object to let all the threads rush out of the waiting area and back to runnable. This is especially important if you have several threads waiting on one object, but for different reasons, and you want to be sure that the *right* thread (along with all of the others) gets notified.

```
notifyAll(); // Will notify all waiting threads
```

All of the threads will be notified and start competing to get the lock. As the lock is used and released by each thread, all of them will get into action without a need for further notification.

As we said earlier, an object can have many threads waiting on it, and using `notify()` will affect only one of them. Which one, exactly, is not specified and depends on the JVM implementation, so you should never rely on a particular thread being notified in preference to another.

In cases in which there might be a lot more waiting, the best way to do this is by using `notifyAll()`. Let's take a look at this in some code. In this example, there is one class that performs a calculation and many readers that are waiting to receive the completed calculation. At any given moment many readers may be waiting.

```
1.  class Reader extends Thread {
2.     Calculator c;
3.
4.     public Reader(Calculator calc) {
5.        c = calc;
```

```
 6.      }
 7.
 8.      public void run() {
 9.         synchronized(c) {
10.            try {
11.               System.out.println("Waiting for calculation...");
12.               c.wait();
13.            } catch (InterruptedException e) {}
14.            System.out.println("Total is: " + c.total);
15.         }
16.      }
17.
18.      public static void main(String [] args) {
19.         Calculator calculator = new Calculator();
20.         new Reader(calculator).start();
21.         new Reader(calculator).start();
22.         new Reader(calculator).start();
23.         calculator.start();
24.      }
25.   }
26.
27.   class Calculator extends Thread {
28.      int total;
29.
30.      public void run() {
31.         synchronized(this) {
32.            for(int i=0;i<100;i++) {
33.               total += i;
34.            }
35.            notifyAll();
36.         }
37.      }
38.   }
```

The program starts three threads that are all waiting to receive the finished calculation (lines 18–24), and then starts the calculator with its calculation. Note that if the `run()` method at line 30 used `notify()` instead of `notifyAll()`, only one reader would be notified instead of all the readers.

Using wait( ) in a Loop

Actually both of the previous examples (Machine/Operator and Reader/Calculator) had a common problem. In each one, there was at least one thread calling `wait()`, and another thread calling `notify()` or `notifyAll()`. This works well enough

as long as the waiting threads have actually started waiting before the other thread executes the `notify()` or `notifyAll()`. But what happens if, for example, the Calculator runs first and calls `notify()` before the Readers have started waiting? This could happen, since we can't guarantee what order the different parts of the thread will execute in. Unfortunately, when the Readers run, they just start waiting right away. They don't do anything to see if the event they're waiting for has already happened. So if the Calculator has already called `notifyAll()`, it's not going to call `notifyAll()` again—and the waiting Readers will keep waiting forever. This is probably *not* what the programmer wanted to happen. Almost always, when you want to wait for something, you also need to be able to check if it has already happened. Generally the best way to solve this is to put in some sort of loop that checks on some sort of conditional expressions, and only waits if the thing you're waiting for has not yet happened. Here's a modified, safer version of the earlier fabric-cutting machine example:

```
class Operator extends Thread {
  Machine machine; // assume this gets initialized
  public void run() {
    while (true) {
      Shape shape = getShapeFromUser();
      MachineInstructions job =
                          calculateNewInstructionsFor(shape);
      machine.addJob(job);
    }
  }
}
```

The operator will still keep on looping forever, getting more shapes from users, calculating new instructions for those shapes, and sending them to the machine. But now the logic for `notify()` has been moved into the `addJob()` method in the Machine class:

```
class Machine extends Thread {
  List<MachineInstructions> jobs =
                         new ArrayList<MachineInstructions>();

  public void addJob(MachineInstructions job) {
    synchronized (jobs) {
      jobs.add(job);
      jobs.notify();
```

```
    }
  }
  public void run() {
    while (true) {
      synchronized (jobs) {
        // wait until at least one job is available
        while (jobs.isEmpty()) {
          try {
            jobs.wait();
          } catch (InterruptedException ie) { }
        }
        // If we get here, we know that jobs is not empty
        MachineInstructions instructions = jobs.remove(0);
        // Send machine steps to hardware
      }
    }
  }
}
```

A machine keeps a list of the jobs it's scheduled to do. Whenever an operator adds a new job to the list, it calls the `addJob()` method and adds the new job to the list. Meanwhile the `run()` method just keeps looping, looking for any jobs on the list. If there are no jobs, it will start waiting. If it's notified, it will stop waiting and then recheck the loop condition: is the list still empty? In practice this double-check is probably not necessary, as the only time a `notify()` is ever sent is when a new job has been added to the list. However, it's a good idea to require the thread to recheck the `isEmpty()` condition whenever it's been woken up, because it's possible that a thread has accidentally sent an extra `notify()` that was not intended. There's also a possible situation called *spontaneous wakeup* that may exist in some situations—a thread may wake up even though no code has called `notify()` or `notifyAll()`. (At least, no code you know about has called these methods. Sometimes the JVM may call `notify()` for reasons of its own, or code in some other class calls it for reasons you just don't know.) What this means is, when your thread wakes up from a `wait()`, you don't know for sure why it was awakened. By putting the `wait()` method in a `while` loop and re-checking the condition that represents what we were waiting for, we ensure that *whatever* the reason we woke up, we will re-enter the `wait()` if (and only if) the thing we were waiting for has not happened yet. In the Machine class, the thing we were waiting for is for the jobs list to not be empty. If it's empty, we wait, and if it's not, we don't.

Note also that both the `run()` method and the `addJob()` method synchronize on the same object—the jobs list. This is for two reasons. One is because we're calling `wait()` and `notify()` on this instance, so we need to synchronize in order to avoid an IllegalThreadState exception. The other reason is, the data in the jobs list is changeable data stored in a field that is accessed by two different threads. We need to synchronize in order to access that changeable data safely. Fortunately, the same `synchronized` blocks that allow us to `wait()` and `notify()` also provide the required thread safety for our other access to changeable data. In fact this is a main reason why synchronization is required to use `wait()` and `notify()` in the first place—you almost always need to share some mutable data between threads at the same time, and that means you need synchronization. Notice that the `synchronized` block in `addJob()` is big enough to also include the call to `jobs.add(job)`—which modifies shared data. And the `synchronized` block in `run()` is large enough to include the whole `while` loop—which includes the call to `jobs.isEmpty()`, which accesses shared data.

The moral here is that when you use `wait()` and `notify()` or `notifyAll()`, you should almost always also have a `while` loop around the `wait()` that checks a condition and forces continued waiting until the condition is met. And you should also make use of the required synchronization for the `wait()` and `notify()` calls, to also protect whatever other data you're sharing between threads. If you see code which fails to do this, there's usually something wrong with the code—even if you have a hard time seeing what exactly the problem is.

exam Watch

The methods `wait()`, `notify()`, and `notifyAll()` are methods of only java.lang.Object, not of java.lang.Thread or java.lang.Runnable. Be sure you know which methods are defined in Thread, which in Object, and which in Runnable (just `run()`, so that's an easy one). Of the key methods in Thread, be sure you know which are `static`—`sleep()` and `yield()`, and which are not `static`—`join()` and `start()`. Table 9-2 lists the key methods you'll need to know for the exam, with the `static` methods shown in italics.

TABLE 9-2 Key Thread Methods

Class Object	Class Thread	Interface Runnable
wait ()	start()	run()
notify()	*yield()*	
notifyAll()	*sleep()*	
	join()	

CERTIFICATION SUMMARY

This chapter covered the required thread knowledge you'll need to apply on the certification exam. Threads can be created by either extending the Thread class or implementing the Runnable interface. The only method that must be overridden in the Runnable interface is the `run()` method, but the thread doesn't become a *thread of execution* until somebody calls the Thread object's `start()` method. We also looked at how the `sleep()` method can be used to pause a thread, and we saw that when an object goes to sleep, it holds onto any locks it acquired prior to sleeping.

We looked at five thread states: new, runnable, running, blocked/waiting/sleeping, and dead. You learned that when a thread is dead, it can never be restarted even if it's still a valid object on the heap. We saw that there is only one way a thread can transition to running, and that's from runnable. However, once running, a thread can become dead, go to sleep, wait for another thread to finish, block on an object's lock, wait for a notification, or return to runnable.

You saw how two threads acting on the same data can cause serious problems (remember Lucy and Fred's bank account?). We saw that, to let one thread execute a method, but prevent other threads from running the same object's method, we use the `synchronized` keyword. To coordinate activity between different threads, use the `wait()`, `notify()`, and `notifyAll()` methods.

TWO-MINUTE DRILL

Here are some of the key points from each certification objective in this chapter. Photocopy it and sleep with it under your pillow for complete absorption.

Defining, Instantiating, and Starting Threads (Objective 4.1)

- Threads can be created by extending Thread and overriding the `public void run()` method.
- Thread objects can also be created by calling the Thread constructor that takes a Runnable argument. The Runnable object is said to be the *target* of the thread.
- You can call `start()` on a Thread object only once. If `start()` is called more than once on a Thread object, it will throw a RuntimeException.
- It is legal to create many Thread objects using the same Runnable object as the target.
- When a Thread object is created, it does not become a *thread of execution* until its `start()` method is invoked. When a Thread object exists but hasn't been started, it is in the *new* state and is not considered *alive*.

Transitioning Between Thread States (Objective 4.2)

- Once a new thread is started, it will always enter the runnable state.
- The thread scheduler can move a thread back and forth between the runnable state and the running state.
- For a typical single-processor machine, only one thread can be running at a time, although many threads may be in the runnable state.
- There is no guarantee that the order in which threads were started determines the order in which they'll run.
- There's no guarantee that threads will take turns in any fair way. It's up to the thread scheduler, as determined by the particular virtual machine implementation. If you want a guarantee that your threads will take turns regardless of the underlying JVM, you can use the `sleep()` method. This prevents one thread from hogging the running process while another thread starves. (In most cases, though, `yield()` works well enough to encourage your threads to play together nicely.)
- A running thread may enter a blocked/waiting state by a `wait()`, `sleep()`, or `join()` call.

- ❑ A running thread may enter a blocked/waiting state because it can't acquire the lock for a synchronized block of code.
- ❑ When the sleep or wait is over, or an object's lock becomes available, the thread can only reenter the runnable state. It will *go* directly from waiting to running (well, for all practical purposes anyway).
- ❑ A dead thread cannot be started again.

Sleep, Yield, and Join (Objective 4.2)

- ❑ Sleeping is used to delay execution for a period of time, and no locks are released when a thread goes to sleep.
- ❑ A sleeping thread is guaranteed to sleep for at least the time specified in the argument to the `sleep()` method (unless it's interrupted), but there is no guarantee as to when the newly awakened thread will actually return to running.
- ❑ The `sleep()` method is a `static` method that sleeps the currently executing thread's state. One thread *cannot* tell another thread to sleep.
- ❑ The `setPriority()` method is used on Thread objects to give threads a priority of between 1 (low) and 10 (high), although priorities are not guaranteed, and not all JVMs recognize 10 distinct priority levels—some levels may be treated as effectively equal.
- ❑ If not explicitly set, a thread's priority will have the same priority as the priority of the thread that created it.
- ❑ The `yield()` method *may* cause a running thread to back out if there are runnable threads of the same priority. There is no guarantee that this will happen, and there is no guarantee that when the thread backs out there will be a *different* thread selected to run. A thread might yield and then immediately reenter the running state.
- ❑ The closest thing to a guarantee is that at any given time, when a thread is running it will usually not have a lower priority than any thread in the runnable state. If a low-priority thread is running when a high-priority thread enters runnable, the JVM will usually preempt the running low-priority thread and put the high-priority thread in.
- ❑ When one thread calls the `join()` method of another thread, the currently running thread will wait until the thread it joins with has completed. Think of the `join()` method as saying, "Hey thread, I want to join on to the end of you. Let me know when you're done, so I can enter the runnable state."

Concurrent Access Problems and Synchronized Threads (Objective 4.3)

- `synchronized` methods prevent more than one thread from accessing an object's critical method code simultaneously.
- You can use the `synchronized` keyword as a method modifier, or to start a synchronized block of code.
- To synchronize a block of code (in other words, a scope smaller than the whole method), you must specify an argument that is the object whose lock you want to synchronize on.
- While only one thread can be accessing synchronized code of a particular instance, multiple threads can still access the same object's *un*synchronized code.
- When a thread goes to sleep, its locks will be unavailable to other threads.
- `static` methods can be `synchronized`, using the lock from the java.lang.Class instance representing that class.

Communicating with Objects by Waiting and Notifying (Objective 4.4)

- The `wait()` method lets a thread say, "there's nothing for me to do now, so put me in your waiting pool and notify me when something happens that I care about." Basically, a `wait()` call means "wait me in your pool," or "add me to your waiting list."
- The `notify()` method is used to send a signal to one and only one of the threads that are waiting in that same object's waiting pool.
- The `notify()` method can NOT specify which waiting thread to notify.
- The method `notifyAll()` works in the same way as `notify()`, only it sends the signal to *all* of the threads waiting on the object.
- All three methods—`wait()`, `notify()`, and `notifyAll()`—must be called from within a `synchronized` context! A thread invokes `wait()` or `notify()` on a particular object, and the thread must currently hold the lock on that object.

Deadlocked Threads (Objective 4.3)

- Deadlocking is when thread execution grinds to a halt because the code is waiting for locks to be removed from objects.
- Deadlocking can occur when a locked object attempts to access another locked object that is trying to access the first locked object. In other words, both threads are waiting for each other's locks to be released; therefore, the locks will *never* be released!
- Deadlocking is bad. Don't do it.

SELF TEST

The following questions will help you measure your understanding of the material presented in this chapter. If you have a rough time with some of these at first, don't beat yourself up. Some of these questions are long and intricate, expect long and intricate questions on the real exam too!

1. The following block of code creates a Thread using a Runnable target:

```
Runnable target = new MyRunnable();
Thread myThread = new Thread(target);
```

Which of the following classes can be used to create the target, so that the preceding code compiles correctly?

A. `public class MyRunnable extends Runnable{public void run(){}}`
B. `public class MyRunnable extends Object{public void run(){}}`
C. `public class MyRunnable implements Runnable{public void run(){}}`
D. `public class MyRunnable implements Runnable{void run(){}}`
E. `public class MyRunnable implements Runnable{public void start(){}}`

2. Given:

```
 3.  class MyThread extends Thread {
 4.     public static void main(String [] args) {
 5.        MyThread t = new MyThread();
 6.        Thread x = new Thread(t);
 7.        x.start();
 8.     }
 9.     public void run() {
10.        for(int i=0;i<3;++i) {
11.           System.out.print(i + "..");
12.        }
13.     }
14.  }
```

What is the result of this code?

A. Compilation fails
B. 1..2..3..
C. 0..1..2..3..
D. 0..1..2..
E. An exception occurs at runtime

3. Given:

```
3.  class Test {
4.     public static void main(String [] args) {
5.        printAll(args);
6.     }
7.     public static void printAll(String[] lines) {
8.        for(int i=0;i<lines.length;i++){
9.           System.out.println(lines[i]);
10.          Thread.currentThread().sleep(1000);
11.       }
12.    }
13. }
```

The `static` method `Thread.currentThread()` returns a reference to the currently executing Thread object. What is the result of this code?

A. Each String in the array `lines` will output, with a 1-second pause between lines

B. Each String in the array `lines` will output, with no pause in between because this method is not executed in a Thread

C. Each String in the array `lines` will output, and there is no guarantee there will be a pause because `currentThread()` may not retrieve this thread

D. This code will not compile

E. Each String in the `lines` array will print, with at least a one-second pause between lines

4. Assume you have a class that holds two `private` variables: `a` and `b`. Which of the following pairs can prevent concurrent access problems in that class? (Choose all that apply.)

A.
```
public int read(){return a+b;}
public void set(int a, int b){this.a=a;this.b=b;}
```

B.
```
public synchronized int read(){return a+b;}
public synchronized void set(int a, int b){this.a=a;this.b=b;}
```

C.
```
public int read(){synchronized(a){return a+b;}}
public void set(int a, int b){synchronized(a){this.a=a;this.b=b;}}
```

D.
```
public int read(){synchronized(a){return a+b;}}
public void set(int a, int b){synchronized(b){this.a=a;this.b=b;}}
```

E.
```
public synchronized(this) int read(){return a+b;}
public synchronized(this) void set(int a, int b){this.a=a;this.b=b;}
```

F.
```
public int read(){synchronized(this){return a+b;}}
public void set(int a, int b){synchronized(this){this.a=a;this.b=b;}}
```

5. Given:

```
 1.  public class WaitTest {
 2.     public static void main(String [] args) {
 3.        System.out.print("1 ");
 4.        synchronized(args){
 5.           System.out.print("2 ");
 6.           try {
 7.              args.wait();
 8.           }
 9.           catch(InterruptedException e){}
10.        }
11.        System.out.print("3 ");
12.     }
13.  }
```

What is the result of trying to compile and run this program?

A. It fails to compile because the `IllegalMonitorStateException` of `wait()` is not dealt with in line 7

B. `1 2 3`

C. `1 3`

D. `1 2`

E. At runtime, it throws an `IllegalMonitorStateException` when trying to wait

F. It will fail to compile because it has to be synchronized on the `this` object

6. Assume the following method is properly synchronized and called from a thread A on an object B:

```
wait(2000);
```

After calling this method, when will the thread A become a candidate to get another turn at the CPU?

A. After object B is notified, or after two seconds

B. After the lock on B is released, or after two seconds

C. Two seconds after object B is notified

D. Two seconds after lock B is released

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

A. The `notifyAll()` method must be called from a synchronized context

B. To call `wait()`, an object must own the lock on the thread

C. The `notify()` method is defined in class `java.lang.Thread`

D. When a thread is waiting as a result of `wait()`, it releases its lock

E. The `notify()` method causes a thread to immediately release its lock

F. The difference between `notify()` and `notifyAll()` is that `notifyAll()` notifies all waiting threads, regardless of the object they're waiting on

8. Given the scenario: This class is intended to allow users to write a series of messages, so that each message is identified with a timestamp and the name of the thread that wrote the message:

```
public class Logger {
    private StringBuilder contents = new StringBuilder();
    public void log(String message) {
        contents.append(System.currentTimeMillis());
        contents.append(": ");
        contents.append(Thread.currentThread().getName());
        contents.append(message);
        contents.append("\n");
    }
    public String getContents() { return contents.toString(); }
}
```

How can we ensure that instances of this class can be safely used by multiple threads?

A. This class is already thread-safe

B. Replacing `StringBuilder` with `StringBuffer` will make this class thread-safe

C. Synchronize the `log()` method only

D. Synchronize the `getContents()` method only

E. Synchronize both `log()` and `getContents()`

F. This class cannot be made thread-safe

9. Given:

```
public static synchronized void main(String[] args) throws
 InterruptedException {
    Thread t = new Thread();
    t.start();
    System.out.print("X");
    t.wait(10000);
    System.out.print("Y");
}
```

What is the result of this code?

A. It prints `X` and exits

B. It prints `X` and never exits

C. It prints `XY` and exits almost immeditately

D. It prints `XY` with a 10-second delay between `X` and `Y`

E. It prints `XY` with a 10000-second delay between `X` and `Y`

F. The code does not compile

G. An exception is thrown at runtime

10. Given:

```
class MyThread extends Thread {
   MyThread() {
     System.out.print(" MyThread");
   }
   public void run() {
     System.out.print(" bar");
   }
   public void run(String s) {
      System.out.print(" baz");
  }
}
public class TestThreads {
  public static void main (String [] args) {
    Thread t = new MyThread() {
      public void run() {
        System.out.print(" foo");
      }
    };
    t.start();
} }
```

What is the result?

A. `foo`

B. `MyThread foo`

C. `MyThread bar`

D. `foo bar`

E. `foo bar baz`

F. `bar foo`

G. Compilation fails

H. An exception is thrown at runtime

11. Given:

```
public class ThreadDemo {
    synchronized void a() { actBusy(); }
    static synchronized void b() { actBusy(); }
    static void actBusy() {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {}
    }
    public static void main(String[] args) {
        final ThreadDemo x = new ThreadDemo();
        final ThreadDemo y = new ThreadDemo();
        Runnable runnable = new Runnable() {
            public void run() {
                int option = (int) (Math.random() * 4);
                switch (option) {
                    case 0: x.a(); break;
                    case 1: x.b(); break;
                    case 2: y.a(); break;
                    case 3: y.b(); break;
                }
            }
        };
        Thread thread1 = new Thread(runnable);
        Thread thread2 = new Thread(runnable);
        thread1.start();
        thread2.start();
    }
}
```

Which of the following pairs of method invocations could NEVER be executing at the same time? (Choose all that apply.)

A. `x.a()` in `thread1`, and `x.a()` in `thread2`

B. `x.a()` in `thread1`, and `x.b()` in `thread2`

C. `x.a()` in `thread1`, and `y.a()` in `thread2`

D. `x.a()` in `thread1`, and `y.b()` in `thread2`

E. `x.b()` in `thread1`, and `x.a()` in `thread2`

F. `x.b()` in `thread1`, and `x.b()` in `thread2`

G. `x.b()` in `thread1`, and `y.a()` in `thread2`

H. `x.b()` in `thread1`, and `y.b()` in `thread2`

12. Given:

```
public class TwoThreads {
    static Thread laurel, hardy;
    public static void main(String[] args) {
        laurel = new Thread() {
            public void run() {
                System.out.println("A");
                try {
                    hardy.sleep(1000);
                } catch (Exception e) {
                    System.out.println("B");
                }
                System.out.println("C");
            }
        };
        hardy = new Thread() {
            public void run() {
                System.out.println("D");
                try {
                    laurel.wait();
                } catch (Exception e) {
                    System.out.println("E");
                }
                System.out.println("F");
            }
        };
        laurel.start();
        hardy.start();
    }
}
```

Which letters will eventually appear somewhere in the output? (Choose all that apply.)

A. `A`
B. `B`
C. `C`
D. `D`
E. `E`
F. `F`
G. The answer cannot be reliably determined
H. The code does not compile

13. Given:

```
 3. public class Starter implements Runnable {
 4.   void go(long id) {
 5.     System.out.println(id);
 6.   }
 7.   public static void main(String[] args) {
 8.     System.out.print(Thread.currentThread().getId() + " ");
 9.     // insert code here
10.   }
11.   public void run() { go(Thread.currentThread().getId()); }
12. }
```

And given the following five fragments:

```
I.   new Starter().run();
II.  new Starter().start();
III. new Thread(new Starter());
IV.  new Thread(new Starter()).run();
V.   new Thread(new Starter()).start();
```

When the five fragments are inserted, one at a time at line 9, which are true? (Choose all that apply.)

A. All five will compile
B. Only one might produce the output `4 4`
C. Only one might produce the output `4 2`
D. Exactly two might produce the output `4 4`
E. Exactly two might produce the output `4 2`
F. Exactly three might produce the output `4 4`
G. Exactly three might produce the output `4 2`

14. Given:

```
 3. public class Leader implements Runnable {
 4.   public static void main(String[] args) {
 5.     Thread t = new Thread(new Leader());
 6.     t.start();
 7.     System.out.print("m1 ");
 8.     t.join();
 9.     System.out.print("m2 ");
10.   }
11.   public void run() {
12.     System.out.print("r1 ");
13.     System.out.print("r2 ");
14.   }
15. }
```

Which are true? (Choose all that apply.)

A. Compilation fails

B. The output could be `r1 r2 m1 m2`

C. The output could be `m1 m2 r1 r2`

D. The output could be `m1 r1 r2 m2`

E. The output could be `m1 r1 m2 r2`

F. An exception is thrown at runtime

15. Given:

```
 3. class Dudes {
 4.   static long flag = 0;
 5.   // insert code here
 6.     if(flag == 0) flag = id;
 7.     for(int x = 1; x < 3; x++) {
 8.       if(flag == id) System.out.print("yo ");
 9.       else System.out.print("dude ");
10.     }
11.   }
12. }
13. public class DudesChat implements Runnable {
14.   static Dudes d;
15.   public static void main(String[] args) {
16.     new DudesChat().go();
17.   }
18.   void go() {
19.     d = new Dudes();
```

```
20.     new Thread(new DudesChat()).start();
21.     new Thread(new DudesChat()).start();
22.   }
23.   public void run() {
24.     d.chat(Thread.currentThread().getId());
25.   }
26. }
```

And given these two fragments:

```
I.  synchronized void chat(long id) {
II. void chat(long id) {
```

When fragment I or fragment II is inserted at line 5, which are true? (Choose all that apply.)

A. An exception is thrown at runtime
B. With fragment I, compilation fails
C. With fragment II, compilation fails
D. With fragment I, the output could be `yo dude dude yo`
E. With fragment I, the output could be `dude dude yo yo`
F. With fragment II, the output could be `yo dude dude yo`

16. Given:

```
 3. class Chicks {
 4.   synchronized void yack(long id) {
 5.     for(int x = 1; x < 3; x++) {
 6.       System.out.print(id + " ");
 7.       Thread.yield();
 8.     }
 9.   }
10. }
11. public class ChicksYack implements Runnable {
12.   Chicks c;
13.   public static void main(String[] args) {
14.     new ChicksYack().go();
15.   }
16.   void go() {
17.     c = new Chicks();
18.     new Thread(new ChicksYack()).start();
19.     new Thread(new ChicksYack()).start();
20.   }
21.   public void run() {
```

```
22.      c.yack(Thread.currentThread().getId());
23.    }
24. }
```

Which are true? (Choose all that apply.)

A. Compilation fails
B. The output could be `4 4 2 3`
C. The output could be `4 4 2 2`
D. The output could be `4 4 4 2`
E. The output could be `2 2 4 4`
F. An exception is thrown at runtime

17. Given:

```
 3. public class Chess implements Runnable {
 4.   public void run() {
 5.     move(Thread.currentThread().getId());
 6.   }
 7.   // insert code here
 8.     System.out.print(id + " ");
 9.     System.out.print(id + " ");
10.   }
11.   public static void main(String[] args) {
12.     Chess ch = new Chess();
13.     new Thread(ch).start();
14.     new Thread(new Chess()).start();
15.   }
16. }
```

And given these two fragments:

```
I.  synchronized void move(long id) {
II.  void move(long id) {
```

When either fragment I or fragment II is inserted at line 7, which are true? (Choose all that apply.)

A. Compilation fails
B. With fragment I, an exception is thrown
C. With fragment I, the output could be `4 2 4 2`
D. With fragment I, the output could be `4 4 2 3`
E. With fragment II, the output could be `2 4 2 4`

SELF TEST ANSWERS

1. The following block of code creates a Thread using a Runnable target:

```
Runnable target = new MyRunnable();
Thread myThread = new Thread(target);
```

Which of the following classes can be used to create the target, so that the preceding code compiles correctly?

A. `public class MyRunnable extends Runnable{public void run(){}}`
B. `public class MyRunnable extends Object{public void run(){}}`
C. `public class MyRunnable implements Runnable{public void run(){}}`
D. `public class MyRunnable implements Runnable{void run(){}}`
E. `public class MyRunnable implements Runnable{public void start(){}}`

Answer:

☑ **C** is correct. The class implements the Runnable interface with a legal `run()` method.

☒ **A** is incorrect because interfaces are implemented, not extended. **B** is incorrect because even though the class has a valid `public void run()` method, it does not implement the Runnable interface. **D** is incorrect because the `run()` method must be `public`. **E** is incorrect because the method to implement is `run()`, not `start()`. (Objective 4.1)

2. Given:

```
 3.  class MyThread extends Thread {
 4.     public static void main(String [] args) {
 5.        MyThread t = new MyThread();
 6.        Thread x = new Thread(t);
 7.        x.start();
 8.     }
 9.     public void run() {
10.        for(int i=0;i<3;++i) {
11.           System.out.print(i + "..");
12.  }  }  }
```

What is the result of this code?

A. Compilation fails

B. 1..2..3..

C. 0..1..2..3..

D. 0..1..2..

E. An exception occurs at runtime

Answer:

☑ **D** is correct. The thread `MyThread` will start and loop three times (from 0 to 2).

☒ **A** is incorrect because the Thread class implements the Runnable interface; therefore, in line 5, Thread can take an object of type Thread as an argument in the constructor (this is NOT recommended). **B** and **C** are incorrect because the variable `i` in the `for` loop starts with a value of 0 and ends with a value of 2. **E** is incorrect based on the above. (Objective 4.1)

3. Given:

```
3.  class Test {
4.     public static void main(String [] args) {
5.        printAll(args);
6.     }
7.     public static void printAll(String[] lines) {
8.        for(int i=0;i<lines.length;i++){
9.           System.out.println(lines[i]);
10.          Thread.currentThread().sleep(1000);
11. }  }  }
```

The `static` method `Thread.currentThread()` returns a reference to the currently executing Thread object. What is the result of this code?

A. Each String in the array `lines` will print, with exactly a 1-second pause between lines

B. Each String in the array `lines` will print, with no pause in between because this method is not executed in a Thread

C. Each String in the array `lines` will print, and there is no guarantee there will be a pause because `currentThread()` may not retrieve this thread

D. This code will not compile

E. Each String in the `lines` array will print, with at least a one-second pause between lines

Answer:

☑ **D** is correct. The `sleep()` method must be enclosed in a `try/catch` block, or the method `printAll()` must declare it throws the `InterruptedException`.

☒ **E** is incorrect, but it would be correct if the `InterruptedException` was dealt with (**A** is too precise). **B** is incorrect (even if the `InterruptedException` was dealt with) because all Java code, including the `main()` method, runs in threads. **C** is incorrect. The `sleep()` method is `static`, it always affects the currently executing thread. (Objective 4.2)

4. Assume you have a class that holds two `private` variables: `a` and `b`. Which of the following pairs can prevent concurrent access problems in that class? (Choose all that apply.)

A.
```
public int read(){return a+b;}
public void set(int a, int b){this.a=a;this.b=b;}
```

B.
```
public synchronized int read(){return a+b;}
public synchronized void set(int a, int b){this.a=a;this.b=b;}
```

C.
```
public int read(){synchronized(a){return a+b;}}
public void set(int a, int b){synchronized(a){this.a=a;this.b=b;}}
```

D.
```
public int read(){synchronized(a){return a+b;}}
public void set(int a, int b){synchronized(b){this.a=a;this.b=b;}}
```

E.
```
public synchronized(this) int read(){return a+b;}
public synchronized(this) void set(int a, int b){this.a=a;this.b=b;}
```

F.
```
public int read(){synchronized(this){return a+b;}}
public void set(int a, int b){synchronized(this){this.a=a;this.b=b;}}
```

Answer:

☑ **B** and **F** are correct. By marking the methods as `synchronized`, the threads will get the lock of the `this` object before proceeding. Only one thread will be setting or reading at any given moment, thereby assuring that `read()` always returns the addition of a valid pair.

☒ **A** is incorrect because it is not synchronized; therefore, there is no guarantee that the values added by the `read()` method belong to the same pair. **C** and **D** are incorrect; only objects can be used to synchronize on. **E** fails—it is not possible to select other objects (even `this`) to synchronize on when declaring a method as `synchronized`. (Objective 4.3)

5. Given:

```
1.  public class WaitTest {
2.     public static void main(String [] args) {
3.        System.out.print("1 ");
4.        synchronized(args){
```

```
 5.            System.out.print("2 ");
 6.            try {
 7.               args.wait();
 8.            }
 9.            catch(InterruptedException e){}
10.         }
11.         System.out.print("3 ");
12.   }  }
```

What is the result of trying to compile and run this program?

A. It fails to compile because the `IllegalMonitorStateException` of `wait()` is not dealt with in line 7

B. `1 2 3`

C. `1 3`

D. `1 2`

E. At runtime, it throws an `IllegalMonitorStateException` when trying to wait

F. It will fail to compile because it has to be synchronized on the `this` object

Answer:

☑ **D** is correct. 1 and 2 will be printed, but there will be no return from the `wait` call because no other thread will notify the main thread, so 3 will never be printed. It's frozen at line 7.

☒ **A** is incorrect; `IllegalMonitorStateException` is an unchecked exception. **B** and **C** are incorrect; 3 will never be printed, since this program will wait forever. **E** is incorrect because `IllegalMonitorStateException` will never be thrown because the `wait()` is done on `args` within a block of code synchronized on `args`. **F** is incorrect because any object can be used to synchronize on and `this` and `static` don't mix. (Objective 4.4)

6. Assume the following method is properly synchronized and called from a thread A on an object B:

```
wait(2000);
```

After calling this method, when will the thread A become a candidate to get another turn at the CPU?

A. After object B is notified, or after two seconds

B. After the lock on B is released, or after two seconds

C. Two seconds after object B is notified

D. Two seconds after lock B is released

Answer:

☑ **A** is correct. Either of the two events will make the thread a candidate for running again.

☒ **B** is incorrect because a waiting thread will not return to runnable when the lock is released, unless a notification occurs. **C** is incorrect because the thread will become a candidate immediately after notification. **D** is also incorrect because a thread will not come out of a waiting pool just because a lock has been released. (Objective 4.4)

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

A. The `notifyAll()` method must be called from a synchronized context

B. To call `wait()`, an object must own the lock on the thread

C. The `notify()` method is defined in class `java.lang.Thread`

D. When a thread is waiting as a result of `wait()`, it releases its lock

E. The `notify()` method causes a thread to immediately release its lock

F. The difference between `notify()` and `notifyAll()` is that `notifyAll()` notifies all waiting threads, regardless of the object they're waiting on

Answer:

☑ **A** is correct because `notifyAll()` (and `wait()` and `notify()`) must be called from within a synchronized context. **D** is a correct statement.

☒ **B** is incorrect because to call `wait()`, the thread must own the lock on the object that `wait()` is being invoked on, not the other way around. **C** is wrong because `notify()` is defined in `java.lang.Object`. **E** is wrong because `notify()` will not cause a thread to release its locks. The thread can only release its locks by exiting the synchronized code. **F** is wrong because `notifyAll()` notifies all the threads waiting on a particular locked object, not all threads waiting on *any* object. (Objective 4.4)

8. Given the scenario: This class is intended to allow users to write a series of messages, so that each message is identified with a timestamp and the name of the thread that wrote the message:

```
public class Logger {
    private StringBuilder contents = new StringBuilder();
    public void log(String message) {
        contents.append(System.currentTimeMillis());
        contents.append(": ");
        contents.append(Thread.currentThread().getName());
```

```
            contents.append(message);
            contents.append("\n");
        }
        public String getContents() { return contents.toString(); }
    }
```

How can we ensure that instances of this class can be safely used by multiple threads?

A. This class is already thread-safe
B. Replacing `StringBuilder` with `StringBuffer` will make this class thread-safe
C. Synchronize the `log()` method only
D. Synchronize the `getContents()` method only
E. Synchronize both `log()` and `getContents()`
F. This class cannot be made thread-safe

Answer:

☑ **E** is correct. Synchronizing the `public` methods is sufficient to make this safe, so **F** is false. This class is not thread-safe unless some sort of synchronization protects the changing data.

☒ **B** is not correct because although a `StringBuffer` is synchonized internally, we call `append()` multiple times, and nothing would prevent two simultaneous `log()` calls from mixing up their messages. **C** and **D** are not correct because if one method remains unsynchronized, it can run while the other is executing, which could result in reading the contents while one of the messages is incomplete, or worse. (You don't want to call `getString()` on the `StringBuffer` as it's resizing its internal character array.) (Objective 4.3)

9. Given:

```
public static synchronized void main(String[] args) throws
 InterruptedException {
    Thread t = new Thread();
    t.start();
    System.out.print("X");
    t.wait(10000);
    System.out.print("Y");
}
```

What is the result of this code?

A. It prints `X` and exits
B. It prints `X` and never exits
C. It prints `XY` and exits almost immeditately

D. It prints `XY` with a 10-second delay between `X` and `Y`
E. It prints `XY` with a 10000-second delay between `X` and `Y`
F. The code does not compile
G. An exception is thrown at runtime

Answer:

☑ **G** is correct. The code does not acquire a lock on `t` before calling `t.wait()`, so it throws an `IllegalMonitorStateException`. The method is `synchronized`, but it's not synchronized on `t` so the exception will be thrown. If the wait were placed inside a `synchronized(t)` block, then the answer would have been **D**.

☒ **A, B, C, D, E,** and **F** are incorrect based the logic described above. (Objective 4.2)

10. Given:

```
class MyThread extends Thread {
   MyThread() {
     System.out.print(" MyThread");
   }
   public void run() { System.out.print(" bar"); }
   public void run(String s) { System.out.print(" baz"); }
}
public class TestThreads {
  public static void main (String [] args) {
    Thread t = new MyThread() {
      public void run() { System.out.print(" foo"); }
    };
    t.start();
} }
```

What is the result?

A. `foo`
B. `MyThread foo`
C. `MyThread bar`
D. `foo bar`
E. `foo bar baz`
F. `bar foo`
G. Compilation fails
H. An exception is thrown at runtime

Answer:

☑ **B** is correct. The first line of main we're constructing an instance of an anonymous inner class extending from `MyThread`. So the `MyThread` constructor runs and prints `MyThread`. Next, `main()` invokes `start()` on the new thread instance, which causes the overridden `run()` method (the `run()` method in the anonymous inner class) to be invoked.

☒ **A, C, D, E, F, G,** and **H** are incorrect based on the logic described above. (Objective 4.1)

11. Given:

```
public class ThreadDemo {
    synchronized void a() { actBusy(); }
    static synchronized void b() { actBusy(); }
    static void actBusy() {
        try { Thread.sleep(1000); }
        catch (InterruptedException e) {}
    }
    public static void main(String[] args) {
        final ThreadDemo x = new ThreadDemo();
        final ThreadDemo y = new ThreadDemo();
        Runnable runnable = new Runnable() {
            public void run() {
                int option = (int) (Math.random() * 4);
                switch (option) {
                    case 0: x.a(); break;
                    case 1: x.b(); break;
                    case 2: y.a(); break;
                    case 3: y.b(); break;
            }  }
        };
        Thread thread1 = new Thread(runnable);
        Thread thread2 = new Thread(runnable);
        thread1.start();
        thread2.start();
}   }
```

Which of the following pairs of method invocations could NEVER be executing at the same time? (Choose all that apply.)

A. `x.a()` in `thread1`, and `x.a()` in `thread2`

B. `x.a()` in `thread1`, and `x.b()` in `thread2`

C. `x.a()` in `thread1`, and `y.a()` in `thread2`

D. `x.a()` in `thread1`, and `y.b()` in `thread2`

E. `x.b()` in `thread1`, and `x.a()` in `thread2`

F. `x.b()` in `thread1`, and `x.b()` in `thread2`

G. `x.b()` in `thread1`, and `y.a()` in `thread2`

H. `x.b()` in `thread1`, and `y.b()` in `thread2`

Answer:

☑ **A, F,** and **H**. **A** is a right answer because when `synchronized` instance methods are called on the same *instance*, they block each other. **F** and **H** can't happen because `synchronized static` methods in the same class block each other, regardless of which instance was used to call the methods. (An instance is not required to call `static` methods; only the class.)

☒ **C** could happen because `synchronized` instance methods called on different instances do not block each other. **B, D, E,** and **G** could all happen because instance methods and `static` methods lock on different objects, and do not block each other. (Objective 4.3)

12. Given:

```
public class TwoThreads {
    static Thread laurel, hardy;
    public static void main(String[] args) {
        laurel = new Thread() {
            public void run() {
                System.out.println("A");
                try {
                    hardy.sleep(1000);
                } catch (Exception e) {
                    System.out.println("B");
                }
                System.out.println("C");
            }
        };
        hardy = new Thread() {
            public void run() {
                System.out.println("D");
                try {
                    laurel.wait();
                } catch (Exception e) {
                    System.out.println("E");
                }
                System.out.println("F");
```

```
                }
            };
            laurel.start();
            hardy.start();
        }
    }
```

Which letters will eventually appear somewhere in the output? (Choose all that apply.)

A. `A`
B. `B`
C. `C`
D. `D`
E. `E`
F. `F`
G. The answer cannot be reliably determined
H. The code does not compile

Answer:

☑ **A, C, D, E,** and **F** are correct. This may look like `laurel` and `hardy` are battling to cause the other to `sleep()` or `wait()`—but that's not the case. Since `sleep()` is a `static` method, it affects the current thread, which is `laurel` (even though the method is invoked using a reference to `hardy`). That's misleading but perfectly legal, and the Thread `laurel` is able to sleep with no exception, printing `A` and `C` (after at least a 1-second delay). Meanwhile `hardy` tries to call `laurel.wait()`—but hardy has not `synchronized` on `laurel`, so calling `laurel.wait()` immediately causes an `IllegalMonitorStateException`, and so `hardy` prints `D`, `E`, and `F`. Although the *order* of the output is somewhat indeterminate (we have no way of knowing whether `A` is printed before `D`, for example) it is guaranteed that `A`, `C`, `D`, `E`, and `F` will all be printed in some order, eventually—so **G** is incorrect.

☒ **B, G,** and **H** are incorrect based on the above. (Objective 4.4)

13. Given:

```
3. public class Starter implements Runnable {
4.   void go(long id) {
5.     System.out.println(id);
6.   }
7.   public static void main(String[] args) {
8.     System.out.print(Thread.currentThread().getId() + " ");
9.     // insert code here
```

```
10.   }
11.   public void run() { go(Thread.currentThread().getId()); }
12. }
```

And given the following five fragments:

```
I.   new Starter().run();
II.  new Starter().start();
III. new Thread(new Starter());
IV.  new Thread(new Starter()).run();
V.   new Thread(new Starter()).start();
```

When the five fragments are inserted, one at a time at line 9, which are true? (Choose all that apply.)

A. All five will compile
B. Only one might produce the output `4 4`
C. Only one might produce the output `4 2`
D. Exactly two might produce the output `4 4`
E. Exactly two might produce the output `4 2`
F. Exactly three might produce the output `4 4`
G. Exactly three might produce the output `4 2`

Answer:

☑ **C** and **D** are correct. Fragment I doesn't start a new thread. Fragment II doesn't compile. Fragment III creates a new thread but doesn't start it. Fragment IV creates a new thread and invokes `run()` directly, but it doesn't start the new thread. Fragment V creates *and* starts a new thread.

☒ **A, B, E, F,** and **G** are incorrect based on the above. (Objective 4.1)

14. Given:

```
 3. public class Leader implements Runnable {
 4.   public static void main(String[] args) {
 5.     Thread t = new Thread(new Leader());
 6.     t.start();
 7.     System.out.print("m1 ");
 8.     t.join();
 9.     System.out.print("m2 ");
10.   }
```

```
11.   public void run() {
12.     System.out.print("r1 ");
13.     System.out.print("r2 ");
14.   }
15. }
```

Which are true? (Choose all that apply.)

A. Compilation fails

B. The output could be `r1 r2 m1 m2`

C. The output could be `m1 m2 r1 r2`

D. The output could be `m1 r1 r2 m2`

E. The output could be `m1 r1 m2 r2`

F. An exception is thrown at runtime

Answer:

☑ **A** is correct. The `join()` must be placed in a try/catch block. If it were, answers B and D would be correct. The `join()` causes the main thread to pause and join the end of the other thread, meaning `"m2"` must come last.

☒ **B, C, D, E,** and **F** are incorrect based on the above. (Objective 4.2)

15. Given:

```
 3. class Dudes {
 4.   static long flag = 0;
 5.   // insert code here
 6.     if(flag == 0) flag = id;
 7.     for(int x = 1; x < 3; x++) {
 8.       if(flag == id) System.out.print("yo ");
 9.       else System.out.print("dude ");
10.     }
11.   }
12. }
13. public class DudesChat implements Runnable {
14.   static Dudes d;
15.   public static void main(String[] args) {
16.     new DudesChat().go();
17.   }
18.   void go() {
19.     d = new Dudes();
```

```
20.     new Thread(new DudesChat()).start();
21.     new Thread(new DudesChat()).start();
22.   }
23.   public void run() {
24.     d.chat(Thread.currentThread().getId());
25.   }
26. }
```

And given these two fragments:

```
I.  synchronized void chat(long id) {
II. void chat(long id) {
```

When fragment I or fragment II is inserted at line 5, which are true? (Choose all that apply.)

A. An exception is thrown at runtime
B. With fragment I, compilation fails
C. With fragment II, compilation fails
D. With fragment I, the output could be `yo dude dude yo`
E. With fragment I, the output could be `dude dude yo yo`
F. With fragment II, the output could be `yo dude dude yo`

Answer:

☑ **F** is correct. With fragment I, the `chat` method is synchronized, so the two threads can't swap back and forth. With either fragment, the first output must be `yo`.

☒ **A, B, C, D,** and **E** are incorrect based on the above. (Objective 4.3)

16. Given:

```
 3. class Chicks {
 4.   synchronized void yack(long id) {
 5.     for(int x = 1; x < 3; x++) {
 6.       System.out.print(id + " ");
 7.       Thread.yield();
 8.     }
 9.   }
10. }
11. public class ChicksYack implements Runnable {
12.   Chicks c;
13.   public static void main(String[] args) {
14.     new ChicksYack().go();
15.   }
```

```
16.   void go() {
17.     c = new Chicks();
18.     new Thread(new ChicksYack()).start();
19.     new Thread(new ChicksYack()).start();
20.   }
21.   public void run() {
22.     c.yack(Thread.currentThread().getId());
23.   }
24. }
```

Which are true? (Choose all that apply.)

A. Compilation fails

B. The output could be `4 4 2 3`

C. The output could be `4 4 2 2`

D. The output could be `4 4 4 2`

E. The output could be `2 2 4 4`

F. An exception is thrown at runtime

Answer:

☑ **F** is correct. When `run()` is invoked, it is with a new instance of `ChicksYack` and `c` has not been assigned to an object. If `c` were static, then because `yack` is synchronized, answers C and E would have been correct.

☒ **A, B, C, D,** and **E** are incorrect based on the above. (Objective 4.3)

17. Given:

```
3. public class Chess implements Runnable {
4.   public void run() {
5.     move(Thread.currentThread().getId());
6.   }
7.   // insert code here
8.     System.out.print(id + " ");
9.     System.out.print(id + " ");
10.  }
11.  public static void main(String[] args) {
12.    Chess ch = new Chess();
13.    new Thread(ch).start();
14.    new Thread(new Chess()).start();
15.  }
16. }
```

And given these two fragments:

```
I.  synchronized void move(long id) {
II.  void move(long id) {
```

When either fragment I or fragment II is inserted at line 7, which are true? (Choose all that apply.)

A. Compilation fails
B. With fragment I, an exception is thrown
C. With fragment I, the output could be `4 2 4 2`
D. With fragment I, the output could be `4 4 2 3`
E. With fragment II, the output could be `2 4 2 4`

Answer:

☑ **C** and **E** are correct. **E** should be obvious. **C** is correct because even though `move()` is synchronized, it's being invoked on two different objects.

☒ **A, B,** and **D** are incorrect based on the above. (Objective 4.3)

EXERCISE ANSWERS

Exercise 9-1: Creating a Thread and Putting It to Sleep

The final code should look something like this:

```
class TheCount extends Thread {
   public void run() {
      for(int i = 1;i<=100;++i) {
         System.out.print(i + "  ");
         if(i % 10 == 0)  System.out.println("Hahaha");
         try { Thread.sleep(1000); }
         catch(InterruptedException e) {}
      }
   }
   public static void main(String [] args) {
      new TheCount().start();
   }
}
```

Exercise 9-2: Synchronizing a Block of Code

Your code might look something like this when completed:

```
class InSync extends Thread {
  StringBuffer letter;
  public InSync(StringBuffer letter) { this.letter = letter; }
  public void run() {
    synchronized(letter) {      // #1
      for(int i = 1;i<=100;++i) System.out.print(letter);
      System.out.println();
      char temp = letter.charAt(0);
      ++temp;         // Increment the letter in StringBuffer:
      letter.setCharAt(0, temp);
    }     // #2
  }
  public static void main(String [] args) {
    StringBuffer sb = new StringBuffer("A");
    new InSync(sb).start();  new InSync(sb).start();
    new InSync(sb).start();
  }
}
```

Just for fun, try removing lines 1 and 2 then run the program again. It will be unsynchronized—watch what happens.

This page intentionally left blank

10

Development

CERTIFICATION OBJECTIVES

- Use Packages and Imports
- Determine Runtime Behavior for Classes and Command-Lines
- Use Classes in JAR Files
- Use Classpaths to Compile Code

✓ Two-Minute Drill

Q&A Self Test

You want to keep your classes organized. You need to have powerful ways for your classes to find each other. You want to make sure that when you're looking for a particular class you get the one you want, and not another class that happens to have the same name. In this chapter we'll explore some of the advanced capabilities of the java and javac commands. We'll revisit the use of packages in Java, and how to search for classes that live in packages.

CERTIFICATION OBJECTIVES

Using the javac and java Commands (Exam Objectives 7.1, 7.2, and 7.5)

7.1 Given a code example and a scenario, write code that uses the appropriate access modifiers, package declarations, and import statements to interact with (through access or inheritance) the code in the example.

7.2 Given an example of a class and a command-line, determine the expected runtime behavior.

7.5 Given the fully-qualified name of a class that is deployed inside and/or outside a JAR file, construct the appropriate directory structure for that class. Given a code example and a classpath, determine whether the classpath will allow the code to compile successfully.

So far in this book, we've probably talked about invoking the `javac` and `java` commands about 1000 times; now we're going to take a closer look.

Compiling with javac

The `javac` command is used to invoke Java's compiler. In Chapter 5 we talked about the assertion mechanism and when you might use the `-source` option when compiling a file. There are many other options you can specify when running `javac`, options to generate debugging information or compiler warnings for example. For the exam, you'll need to understand the `-classpath` and `-d` options, which we'll cover in the next few pages. Here's the structural overview for `javac`:

```
javac [options] [source files]
```

There are additional command-line options called `@argfiles`, but you won't need to study them for the exam. Both the `[options]` and the `[source files]` are optional parts of the command, and both allow multiple entries. The following are both legal `javac` commands:

```
javac -help
javac -classpath com:. -g Foo.java Bar.java
```

The first invocation doesn't compile any files, but prints a summary of valid options. The second invocation passes the compiler two options (`-classpath`, which itself has an argument of `com:.` and `-g`), and passes the compiler two `.java` files to compile (`Foo.java` and `Bar.java`). Whenever you specify multiple options and/or files they should be separated by spaces.

Compiling with -d

By default, the compiler puts a `.class` file in the same directory as the `.java` source file. This is fine for very small projects, but once you're working on a project of any size at all, you'll want to keep your `.java` files separated from your `.class` files. (This helps with version control, testing, deployment...) The `-d` option lets you tell the compiler in which directory to put the `.class` file(s) it generates (`d` is for destination). Let's say you have the following directory structure:

```
myProject
       |
       |--source
       |      |
       |      |-- MyClass.java
       |
       |-- classes
              |
              |--
```

The following command, issued from the `myProject` directory, will compile `MyClass.java` and put the resulting `MyClass.class` file into the `classes` directory. (Note: This assumes that MyClass does not have a package statement; we'll talk about packages in a minute.)

```
cd myProject
javac -d classes source/MyClass.java
```

This command also demonstrates selecting a `.java` file from a subdirectory of the directory from which the command was invoked. Now let's take a quick look at how packages work in relationship to the `-d` option.

Suppose we have the following `.java` file in the following directory structure:

```
package com.wickedlysmart;
public class MyClass { }
```

```
myProject
        |
        |--source
        |       |
        |       |--com
        |            |
        |            |--wickedlysmart
        |                        |
        |                        |--MyClass.java
        |
        |--classes
        |       |
        |       |--com
        |            |
        |            |--wickedlysmart
        |                        |
        |                        |-- (MyClass.class goes here)
```

If you were in the `source` directory, you would compile `MyClass.java` and put the resulting `MyClass.class` file into the `classes/com/wickedlysmart` directory by invoking the following command:

```
javac -d ../classes com/wickedlysmart/MyClass.java
```

This command could be read: "To set the destination directory, `cd` back to the `myProject` directory then `cd` into the `classes` directory, which will be your destination. Then compile the file named `MyClass.java`. Finally, put the resulting `MyClass.class` file into the directory structure that matches its package, in this case, `classes/com/wickedlysmart`." Because `MyClass.java` is in a package, the compiler knew to put the resulting `.class` file into the `classes/com/wickedlysmart` directory.

Somewhat amazingly, the `javac` command can sometimes help you out by building directories it needs! Suppose we have the following:

```
package com.wickedlysmart;
public class MyClass { }
```

```
myProject
       |
       |--source
       |      |
       |      |--com
       |           |
       |           |--wickedlysmart
       |                       |
       |                       |--MyClass.java
       |
       |--classes
       |      |
```

And the following command (the same as last time):

```
javac -d ../classes com/wickedlysmart/MyClass.java
```

In this case, the compiler will build two directories called `com` and `com/wickedlysmart` in order to put the resulting `MyClass.class` file into the correct package directory (`com/wickedlysmart/`) which it builds within the existing `.../classes` directory.

The last thing about `-d` that you'll need to know for the exam is that if the destination directory you specify doesn't exist, you'll get a compiler error. If, in the previous example, the `classes` directory did NOT exist, the compiler would say something like:

```
java:5: error while writing MyClass: classes/MyClass.class (No
such file or directory)
```

Launching Applications with java

The `java` command is used to invoke the Java virtual machine. In Chapter 5 we talked about the assertion mechanism and when you might use flags such as `-ea` or `-da` when launching an application. There are many other options you can specify

when running the `java` command, but for the exam, you'll need to understand the `-classpath` (and its twin `-cp`) and `-D` options, which we'll cover in the next few pages. In addition, it's important to understand the structure of this command. Here's the overview:

```
java [options] class [args]
```

The `[options]` and `[args]` parts of the `java` command are optional, and they can both have multiple values. You must specify exactly one class file to execute, and the `java` command assumes you're talking about a `.class` file, so you don't specify the `.class` extension on the command line. Here's an example:

```
java -DmyProp=myValue MyClass x 1
```

Sparing the details for later, this command can be read as "Create a *system property* called `myProp` and set its value to `myValue`. Then launch the file named `MyClass.class` and send it two String *arguments* whose values are `x` and `1`."

Let's look at system properties and command-line arguments more closely.

Using System Properties

Java has a class called `java.util.Properties` that can be used to access a system's persistent information such as the current versions of the operating system, the Java compiler, and the Java virtual machine. In addition to providing such default information, you can also add and retrieve your own properties. Take a look at the following:

```
import java.util.*;
public class TestProps {
  public static void main(String[] args) {
    Properties p = System.getProperties();
    p.setProperty("myProp", "myValue");
    p.list(System.out);
  }
}
```

If this file is compiled and invoked as follows:

```
java -DcmdProp=cmdVal TestProps
```

You'll get something like this:

```
...
os.name=Mac OS X
myProp=myValue
...
java.specification.vendor=Sun Microsystems Inc.
user.language=en
java.version=1.6.0_05
...
cmdProp=cmdVal
...
```

where the . . . represent lots of other name=value pairs. (The *name* and *value* are sometimes called the *key* and the *property*.) Two name=value properties were added to the system's properties: `myProp=myValue` was added via the `setProperty` method, and `cmdProp=cmdVal` was added via the `-D` option at the command line. When using the `-D` option, if your value contains white space the entire value should be placed in quotes like this:

```
java -DcmdProp="cmdVal take 2" TestProps
```

Just in case you missed it, when you use `-D`, the name=value pair must follow *immediately*, no spaces allowed.

The `getProperty()` method is used to retrieve a single property. It can be invoked with a single argument (a String that represents the name (or key)), or it can be invoked with two arguments, (a String that represents the name (or key), and a default String value to be used as the property if the property does not already exist). In both cases, `getProperty()` returns the property as a String.

Handling Command-Line Arguments

Let's return to an example of launching an application and passing in arguments from the command line. If we have the following code:

```
public class CmdArgs {
  public static void main(String[] args) {
    int x = 0;
    for(String s : args)
      System.out.println(x++ + " element = " + s);
  }
}
```

compiled and then invoked as follows

```
java CmdArgs x 1
```

the output will be

```
0 element = x
1 element = 1
```

Like all arrays, `args` index is zero based. Arguments on the command line directly follow the class name. The first argument is assigned to `args[0]`, the second argument is assigned to `args[1]`, and so on.

Finally, there is some flexibility in the declaration of the `main()` method that is used to start a Java application. The order of `main()`'s modifiers can be altered a little, the String array doesn't have to be named `args`, and as of Java 5 it can be declared using var-args syntax. The following are all legal declarations for `main()`:

```
static public void main(String[] args)
public static void main(String... x)
static public void main(String bang_a_gong[])
```

Searching for Other Classes

In most cases, when we use the `java` and `javac` commands, we want these commands to search for other classes that will be necessary to complete the operation. The most obvious case is when classes we create use classes that Sun provides with J2SE (now sometimes called Java SE), for instance when we use classes in `java.lang` or `java.util`. The next common case is when we want to compile a file or run a class that uses other classes that have been created outside of what Sun provides, for instance our own previously created classes. Remember that for any given class, the `java` virtual machine will need to find exactly the same supporting classes that the `javac` compiler needed to find at compilation time. In other words, if `javac` needed access to `java.util.HashMap` then the `java` command will need to find `java.util.HashMap` as well.

Both `java` and `javac` use the same basic search algorithm:

1. They both have the same list of places (directories) they search, to look for classes.

2. They both search through this list of directories in the same order.
3. As soon as they find the class they're looking for, they stop searching for that class. In the case that their search lists contain two or more files with the same name, the first file found will be the file that is used.
4. The first place they look is in the directories that contain the classes that come standard with J2SE.
5. The second place they look is in the directories defined by classpaths.
6. Classpaths should be thought of as "class search paths." They are lists of directories in which classes might be found.
7. There are two places where classpaths can be declared:

   A classpath can be declared as an operating system environment variable. The classpath declared here is used by default, whenever `java` or `javac` are invoked.

   A classpath can be declared as a command-line option for either `java` or `javac`. *Classpaths declared as command-line options override the classpath declared as an environment variable, but they persist only for the length of the invocation.*

Declaring and Using Classpaths

Classpaths consist of a variable number of directory locations, separated by delimiters. For Unix-based operating systems, forward slashes are used to construct directory locations, and the separator is the colon (:). For example:

```
-classpath /com/foo/acct:/com/foo
```

specifies two directories in which classes can be found: `/com/foo/acct` and `/com/foo`. In both cases, these directories are absolutely tied to the root of the file system, which is specified by the leading forward slash. It's important to remember that when you specify a subdirectory, you're NOT specifying the directories above it. For instance, in the preceding example the directory `/com` will NOT be searched.

exam Watch

Most of the path-related questions on the exam will use Unix conventions. If you are a Windows user, your directories will be declared using backslashes (\) and the separator character you use will be a semicolon (;). But again, you will NOT need any shell-specific knowledge for the exam.

A very common situation occurs in which `java` or `javac` complains that it can't find a class file, and yet you can see that the file is IN the current directory! When searching for class files, the `java` and `javac` commands don't search the current directory by default. You must *tell* them to search there. The way to tell `java` or `javac` to search in the current directory is to add a dot (.) to the classpath:

```
-classpath /com/foo/acct:/com/foo:.
```

This classpath is identical to the previous one EXCEPT that the dot (.) at the end of the declaration instructs `java` or `javac` to *also* search for class files in the current directory. (Remember, we're talking about class files—when you're telling `javac` which `.java` file to compile, `javac` looks in the current directory by default.)

It's also important to remember that classpaths are searched from left to right. Therefore in a situation where classes with duplicate names are located in several different directories in the following classpaths, different results will occur:

```
-classpath /com:/foo:.
```

is not the same as

```
-classpath .:/foo:/com
```

Finally, the `java` command allows you to abbreviate `-classpath` with `-cp`. The Java documentation is inconsistent about whether the `javac` command allows the `-cp` abbreviation. On most machines it does, but there are no guarantees.

Packages and Searching

When you start to put classes into packages, and then start to use classpaths to find these classes, things can get tricky. The exam creators knew this, and they tried to create an especially devilish set of package/classpath questions with which to confound you. Let's start off by reviewing packages. In the following code:

```
package com.foo;
public class MyClass { public void hi() { } }
```

We're saying that `MyClass` is a member of the `com.foo` package. This means that the fully qualified name of the class is now `com.foo.MyClass`. Once a class is in a package, the package part of its fully qualified name is *atomic*—it can never be divided. You can't split it up on the command line, and you can't split it up in an `import` statement.

Now let's see how we can use `com.foo.MyClass` in another class:

```
package com.foo;
public class MyClass { public void hi() { } }
```

And in another file:

```
import com.foo.MyClass;   // either import will work
import com.foo.*;

public class Another {
  void go() {
    MyClass m1 = new MyClass();                    // alias name
    com.foo.MyClass m2 = new com.foo.MyClass();  // full name
    m1.hi();
    m2.hi();
  }
}
```

It's easy to get confused when you use `import` statements. The preceding code is perfectly legal. The `import` statement is like an alias for the class's fully qualified name. You define the fully qualified name for the class with an `import` statement (or with a wildcard in an `import` statement of the package). Once you've defined the fully qualified name, you can use the "alias" in your code—but the alias is referring back to the fully qualified name.

Now that we've reviewed packages, let's take a look at how they work in conjunction with classpaths and command lines. First we'll start off with the idea that when you're searching for a class using its fully qualified name, that fully qualified name relates closely to a specific directory structure. For instance, relative to your current directory, the class whose source code is

```
package com.foo;
public class MyClass { public void hi() { } }
```

would *have* to be located here:

```
com/foo/MyClass.class
```

In order to find a class in a package, you have to have a directory in your classpath that has the package's leftmost entry (the package's "root") as a subdirectory.

This is an important concept, so let's look at another example:

```
import com.wickedlysmart.Utils;
class TestClass {
  void doStuff() {
    Utils u = new Utils();                      // simple name
    u.doX("arg1", "arg2");
    com.wickedlysmart.Date d =
                new com.wickedlysmart.Date();   // full name
    d.getMonth("Oct");
  }
}
```

In this case we're using two classes from the package `com.wickedlysmart`. For the sake of discussion we imported the fully qualified name for the `Utils` class, and we didn't for the `Date` class. The *only* difference is that because we listed `Utils` in an `import` statement, we didn't have to type its fully qualified name inside the class. In both cases the package is `com.wickedlysmart`. When it's time to compile or run `TestClass`, the classpath will have to include a directory with the following attributes:

- A subdirectory named `com` (we'll call this the "package root" directory)
- A subdirectory in `com` named `wickedlysmart`
- Two files in `wickedlysmart` named `Utils.class` and `Date.class`

Finally, the directory that has all of these attributes has to be accessible (via a classpath) in one of two ways:

1. The path to the directory must be absolute, in other words, from the root (the file system root, not the package root).

   or

2. The path to the directory has to be correct relative to the current directory.

Relative and Absolute Paths

A classpath is a collection of one or more paths. Each path in a classpath is either an absolute path or a relative path. An absolute path in Unix begins with a forward slash (/) (on Windows it would be something like c:\). The leading slash indicates that this path is starting from the root directory of the system. Because it's starting from the root, it doesn't *matter* what the current directory is—*a directory's absolute path is always the same*. A *relative* path is one that does NOT start with a slash. Here's an example of a full directory structure, and a classpath:

```
/ (root)
     |
     |--dirA
          |
          |-- dirB
                |
                |--dirC

-cp dirB:dirB/dirC
```

In this example, `dirB` and `dirB/dirC` are relative paths (they don't start with a slash /). Both of these relative paths are meaningful *only* when the current directory is `dirA`. Pop Quiz! If the current directory is `dirA`, and you're searching for class files, and you use the classpath described above, which directories will be searched?

`dirA? dirB? dirC?`

Too easy? How about the same question if the current directory is the root (/)? When the current directory is `dirA`, then `dirB` and `dirC` will be searched, but not

dirA (remember, we didn't specify the current directory by adding a dot (.) to the classpath). When the current directory is root, since dirB is not a direct subdirectory of root, no directories will be searched. Okay, how about if the current directory is dirB? Again, no directories will be searched! This is because dirB doesn't have a subdirectory named dirB. In other words, Java will look in dirB for a directory named dirB (which it won't find), without realizing that it's already in dirB.

Let's use the same directory structure and a different classpath:

```
/ (root)
      |
      |--dirA
            |
            |-- dirB
                  |
                  |--dirC

-cp /dirB:/dirA/dirB/dirC
```

In this case, what directories will be searched if the current directory is dirA? How about if the current directory is root? How about if the current directory is dirB? In this case, both paths in the classpath are absolute. It doesn't matter what the current directory is; since absolute paths are specified the search results will always be the same. Specifically, only dirC will be searched, regardless of the current directory. The first path (/dirB) is invalid since dirB is not a direct subdirectory of root, so dirB will never be searched. And, one more time, for emphasis, since dot (.) is not in the classpath, the current directory will only be searched if it happens to be described elsewhere in the classpath (in this case, dirC).

CERTIFICATION OBJECTIVE

JAR Files (Objective 7.5)

7.5 Given the fully-qualified name of a class that is deployed inside and/or outside a JAR file, construct the appropriate directory structure for that class. Given a code example and a classpath, determine whether the classpath will allow the code to compile successfully.

JAR Files and Searching

Once you've built and tested your application, you might want to bundle it up so that it's easy to distribute and easy for other people to install. One mechanism that Java provides for these purposes is a JAR file. JAR stands for Java Archive. JAR files are used to compress data (similar to ZIP files) and to archive data.

Here's an application with classes in different packages:

```
test
   |--UseStuff.java
   |--ws
       |--(create MyJar.jar here)
       |--myApp
             |--utils
             |      |--Dates.class       (package myApp.utils;)
             |--engine
                    |--rete.class        (package myApp.engine;)
                    |--minmax.class         "          "
```

You can create a single JAR file that contains all of the files in `myApp`, and also maintains `myApp`'s directory structure. Once this JAR file is created, it can be moved from place to place, and from machine to machine, and all of the classes in the JAR file can be accessed, via classpaths, by `java` and `javac`, without ever unJARing the JAR file. Although you won't need to know how to make JAR files for the exam, let's make the current directory `ws`, and then make a JAR file called `MyJar.jar`:

```
cd ws
jar -cf MyJar.jar myApp
```

The `jar` command will create a JAR file called `MyJar.jar` and it will contain the `myApp` directory and `myApp`'s entire subdirectory tree and files. You can look at the contents of the JAR file with the next command (this isn't on the exam either):

```
jar -tf MyJar.jar
```

(which produces something like)

```
META-INF/
META-INF/MANIFEST.MF
myApp/
myApp/.DS_Store
myApp/utils/
```

```
myApp/utils/Dates.class
myApp/engine/
myApp/engine/rete.class
myApp/engine/minmax.class
```

exam Watch

Here are some rules concerning the structure of JAR files:

- ***The jar command creates the*** `META-INF` ***directory automatically.***
- ***The jar command creates the*** `MANIFEST.MF` ***file automatically.***
- ***The jar command won't place any of your files in*** `META-INF/`***.***
- ***As you can see above, the exact tree structure is represented.***
- `java` ***and*** `javac` ***will use the JAR like a normal directory tree.***

Back to exam stuff. Finding a JAR file using a classpath is similar to finding a package file in a classpath. The difference is that when you specify a path for a JAR file, *you must include the name of the JAR file at the end of the path*. Let's say you want to compile `UseStuff.java` in the `test` directory, and `UseStuff.java` needs access to a class contained in `myApp.jar`. To compile `UseStuff.java` say

```
cd test
javac -classpath ws/myApp.jar UseStuff.java
```

Compare the use of the JAR file to using a class in a package. If `UseStuff.java` needed to use classes in the `myApp.utils` package, and the class was not in a JAR, you would say

```
cd test
javac -classpath ws UseStuff.java
```

Remember when using a classpath, the last directory in the path must be the super-directory of the *root* directory for the package. (In the preceding example, `myApp` is the root directory of the package `myApp.utils`.) Notice that `myApp` can be the root directory for more than one package (`myApp.utils` and `myApp.engine`), and the `java` and `javac` commands can find what they need across multiple *peer* packages like this. So, if `ws` is on the classpath and `ws` is the super-directory of `myApp`, then classes in both the `myApp.utils` and `myApp.engine` packages will be found.

exam Watch

When you use an import statement you are declaring only one package. When you say `import java.util.*;` ***you are saying "Use the short name for all of the classes in the*** `java.util` ***package." You're NOT getting the*** `java.util.jar` ***classes or*** `java.util.regex` ***packages! Those packages are totally independent of each other; the only thing they share is the same "root" directory, but they are not the same packages. As a corollary, you can't say*** `import java.*;` ***in the hopes of importing multiple packages—just remember, an*** `import` ***statement can import only a single package.***

Using .../jre/lib/ext with JAR files

When you install Java, you end up with a huge directory tree of Java-related stuff, including the JAR files that contain the classes that come standard with J2SE. As we discussed earlier, `java` and `javac` have a list of places that they access when searching for class files. Buried deep inside of your Java directory tree is a subdirectory tree named `jre/lib/ext`. If you put JAR files into the `ext` subdirectory, `java` and `javac` can find them, and use the class files they contain. You don't have to mention these subdirectories in a classpath statement—searching this directory is a function that's built right into Java. Sun recommends, however, that you use this feature only for your own internal testing and development, and not for software that you intend to distribute.

exam Watch

It's possible to create environment variables that provide an alias for long classpaths. The classpath for some of the JAR files in J2SE can be quite long, and so it's common for such an alias to be used when defining a classpath. If you see something like `JAVA_HOME` ***or*** `$JAVA_HOME` ***in an exam question it just means "That part of the absolute classpath up to the directories we're specifying explicitly." You can assume that the*** `JAVA_HOME` ***literal means this, and is pre-pended to the partial classpath you see.***

CERTIFICATION OBJECTIVE

Using Static Imports (Exam Objective 7.1)

7.1 Given a code example and a scenario, write code that uses the appropriate access modifiers, package declarations, and import statements to interact with (through access or inheritance) the code in the example.

Note: In Chapter 1 we covered most of what's defined in this objective, but we saved static imports for this chapter.

Static Imports

We've been using `import` statements throughout the book. Ultimately, the only value `import` statements have is that they save typing and they can make your code easier to read. In Java 5, the `import` statement was enhanced to provide even greater keystroke-reduction capabilities…although some would argue that this comes at the expense of readability. This new feature is known as *static imports*. Static imports can be used when you want to use a class's `static` members. (You can use this feature on classes in the API and on your own classes.) Here's a "before and after" example:

Before static imports:

```
public class TestStatic {
  public static void main(String[] args) {
    System.out.println(Integer.MAX_VALUE);
    System.out.println(Integer.toHexString(42));
  }
}
```

After static imports:

```
import static java.lang.System.out;              // 1
import static java.lang.Integer.*;               // 2
public class TestStaticImport {
  public static void main(String[] args)  {
    out.println(MAX_VALUE);                      // 3
    out.println(toHexString(42));                // 4
  }
}
```

Both classes produce the same output:

```
2147483647
2a
```

Let's look at what's happening in the code that's using the static import feature:

1. Even though the feature is commonly called "static import" the syntax MUST be `import static` followed by the fully qualified name of the `static` member you want to import, or a wildcard. In this case we're doing a static import on the `System` class `out` object.
2. In this case we might want to use several of the `static` members of the `java.lang.Integer` class. This static import statement uses the wildcard to say, "I want to do static imports of ALL the `static` members in this class."
3. Now we're finally seeing the *benefit* of the static import feature! We didn't have to type the `System` in `System.out.println`! Wow! Second, we didn't have to type the `Integer` in `Integer.MAX_VALUE`. So in this line of code we were able to use a shortcut for a `static` method AND a constant.
4. Finally, we do one more shortcut, this time for a method in the `Integer` class.

We've been a little sarcastic about this feature, but we're not the only ones. We're not convinced that saving a few keystrokes is worth possibly making the code a little harder to read, but enough developers requested it that it was added to the language.

Here are a couple of rules for using static imports:

- You must say `import static`; you can't say `static import`.
- Watch out for ambiguously named `static` members. For instance, if you do a static import for both the `Integer` class and the `Long` class, referring to `MAX_VALUE` will cause a compiler error, since both `Integer` and `Long` have a `MAX_VALUE` constant, and Java won't know which `MAX_VALUE` you're referring to.
- You can do a static import on `static` object references, constants (remember they're `static` and `final`), and `static` methods.

CERTIFICATION SUMMARY

We started by exploring the `javac` command more deeply. The `-d` option allows you to put class files generated by compilation into whatever directory you want to. The `-d` option lets you specify the destination of newly created class files.

Next we talked about some of the options available through the `java` application launcher. We discussed the ordering of the arguments `java` can take, including `[options] class [args]`. We learned how to query and update system properties in code and at the command line using the `-D` option.

The next topic was handling command-line arguments. The key concepts are that these arguments are put into a String array, and that the first argument goes into array element 0, the second argument into array element 1, and so on.

We turned to the important topic of how `java` and `javac` search for other class files when they need them, and how they use the same algorithm to find these classes. There are search locations predefined by Sun, and additional search locations, called *classpaths* that are user defined. The syntax for Unix classpaths is different than the syntax for Windows classpaths, and the exam will tend to use Unix syntax.

The topic of packages came next. Remember that once you put a class into a package, its name is atomic—in other words, it can't be split up. There is a tight relationship between a class's fully qualified package name and the directory structure in which the class resides.

JAR files were discussed next. JAR files are used to compress and archive data. They can be used to archive entire directory tree structures into a single JAR file. JAR files can be searched by `java` and `javac`.

We finished the chapter by discussing a new Java 5 feature, static imports. This is a convenience-only feature that reduces keying long names for `static` members in the classes you use in your programs.

TWO-MINUTE DRILL

Here are the key points from this chapter.

Using javac and java (Objective 7.2)

- Use `-d` to change the destination of a class file when it's first generated by the `javac` command.
- The `-d` option can build package-dependent destination classes on-the-fly if the *root* package directory already exists.
- Use the `-D` option in conjunction with the `java` command when you want to set a system property.
- System properties consist of name=value pairs that must be appended directly behind the `-D`, for example, `java -Dmyproperty=myvalue.`
- Command-line arguments are always treated as Strings.
- The `java` command-line argument 1 is put into array element 0, argument 2 is put into element 1, and so on.

Searching with java and javac (Objective 7.5)

- Both `java` and `javac` use the same algorithms to search for classes.
- Searching begins in the locations that contain the classes that come standard with J2SE.
- Users can define secondary search locations using classpaths.
- Default classpaths can be defined by using OS environment variables.
- A classpath can be declared at the command line, and it overrides the default classpath.
- A single classpath can define many different search locations.
- In Unix classpaths, forward slashes (`/`) are used to separate the directories that make up a path. In Windows, backslashes (`\`) are used.

- In Unix, colons (:) are used to separate the paths within a classpath. In Windows, semicolons (;) are used.
- In a classpath, to specify the current directory as a search location, use a dot (.)
- In a classpath, once a class is found, searching stops, so the order of locations to search is important.

Packages and Searching (Objective 7.5)

- When a class is put into a package, its fully qualified name must be used.
- An `import` statement provides an alias to a class's fully qualified name.
- In order for a class to be located, its fully qualified name must have a tight relationship with the directory structure in which it resides.
- A classpath can contain both relative and absolute paths.
- An absolute path starts with a `/` or a `\`.
- Only the final directory in a given path will be searched.

JAR Files (Objective 7.5)

- An entire directory tree structure can be archived in a single JAR file.
- JAR files can be searched by `java` and `javac`.
- When you include a JAR file in a classpath, you must include not only the directory in which the JAR file is located, but the name of the JAR file too.
- For testing purposes, you can put JAR files into `.../jre/lib/ext`, which is somewhere inside the Java directory tree on your machine.

Static Imports (Objective 7.1)

- You must start a static import statement like this: `import static`
- You can use static imports to create shortcuts for `static` members (static variables, constants, and methods) of any class.

SELF TEST

1. Given:

```
1. // insert code here
2. class StatTest {
3.   public static void main(String[] args) {
4.     System.out.println(Integer.MAX_VALUE);
5.   }
6. }
```

Which, inserted independently at line 1, compiles? (Choose all that apply.)

A. `import static java.lang;`

B. `import static java.lang.Integer;`

C. `import static java.lang.Integer.*;`

D. `import static java.lang.Integer.*_VALUE;`

E. `import static java.lang.Integer.MAX_VALUE;`

F. None of the above statements are valid import syntax

2. Given:

```
import static java.lang.System.*;
class _ {
  static public void main(String... __A_V_) {
    String $ = "";
    for(int x=0; ++x < __A_V_.length; )
      $ += __A_V_[x];
    out.println($);
  }
}
```

And the command line:

```
java _ - A .
```

What is the result?

A. `-A`

B. `A.`

C. `-A.`

D. `_A.`

E. `_-A.`

F. Compilation fails

G. An exception is thrown at runtime

3. Given the default classpath:

```
/foo
```

And this directory structure:

```
foo
  |
  test
     |
     xcom
        |--A.class
        |--B.java
```

And these two files:

```
package xcom;
public class A { }

package xcom;
public class B extends A { }
```

Which allows `B.java` to compile? (Choose all that apply.)

A. Set the current directory to `xcom` then invoke
```
javac B.java
```

B. Set the current directory to `xcom` then invoke
```
javac -classpath . B.java
```

C. Set the current directory to `test` then invoke
```
javac -classpath . xcom/B.java
```

D. Set the current directory to `test` then invoke
```
javac -classpath xcom B.java
```

E. Set the current directory to `test` then invoke
```
javac -classpath xcom:. B.java
```

4. Given two files:

```
a=b.java
c_d.class
```

Are in the current directory, which command-line invocation(s) could complete without error? (Choose all that apply.)

A. `java -Da=b c_d`

B. `java -D a=b c_d`

C. `javac -Da=b c_d`

D. `javac -D a=b c_d`

5. If three versions of `MyClass.class` exist on a file system:

Version 1 is in `/foo/bar`
Version 2 is in `/foo/bar/baz`
Version 3 is in `/foo/bar/baz/bing`

And the system's classpath includes

```
/foo/bar/baz
```

And this command line is invoked from `/foo`

```
java -classpath /foo/bar/baz/bing:/foo/bar MyClass
```

Which version will be used by `java`?

A. `/foo/MyClass.class`

B. `/foo/bar/MyClass.class`

C. `/foo/bar/baz/MyClass.class`

D. `/foo/bar/baz/bing/MyClass.class`

E. The result is not predictable

6. Given two files:

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

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

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

A. `5 6 5 6`

B. `5 6` followed by an exception

C. Compilation fails with an error on line 6

D. Compilation fails with an error on line 7

E. Compilation fails with an error on line 8

F. Compilation fails with an error on line 9

7. Given:

```
3. import java.util.*;
4. public class Antique {
5.   public static void main(String[] args) {
6.     List<String> myList = new ArrayList<String>();
7.     assert (args.length > 0);
8.     System.out.println("still static");
9.   }
10. }
```

Which sets of commands (`javac` followed by `java`) will compile and run without exception or error? (Choose all that apply.)

A.
```
javac Antique.java
java Antique
```

B.
```
javac Antique.java
java -ea Antique
```

C.
```
javac -source 6 Antique.java
java Antique
```

D.
```
javac -source 1.4 Antique.java
java Antique
```

E.
```
javac -source 1.6 Antique.java
java -ea Antique
```

8. Given:

```
 3. import java.util.*;
 4. public class Values {
 5.   public static void main(String[] args) {
 6.     Properties p = System.getProperties();
 7.     p.setProperty("myProp", "myValue");
 8.     System.out.print(p.getProperty("cmdProp") + " ");
 9.     System.out.print(p.getProperty("myProp") + " ");
10.     System.out.print(p.getProperty("noProp") + " ");
11.     p.setProperty("cmdProp", "newValue");
12.     System.out.println(p.getProperty("cmdProp"));
13.   }
14. }
```

And given the command line invocation:

```
java -DcmdProp=cmdValue Values
```

What is the result?

A. `null myValue null null`

B. `cmdValue null null cmdValue`

C. `cmdValue null null newValue`

D. `cmdValue myValue null cmdValue`

E. `cmdValue myValue null newValue`

F. An exception is thrown at runtime

9. Given the following directory structure:

```
x-|
  |- FindBaz.class
  |
  |- test-|
          |- Baz.class
          |
          |- myApp-|
                   |- Baz.class
```

And given the contents of the related .java files:

```
1. public class FindBaz {
2.   public static void main(String[] args) { new Baz(); }
3. }
```

In the test directory:

```
1. public class Baz {
2.   static { System.out.println("test/Baz"); }
3. }
```

In the myApp directory:

```
1. public class Baz {
2.   static { System.out.println("myApp/Baz"); }
3. }
```

If the current directory is x, which invocations will produce the output `"test/Baz"`? (Choose all that apply.)

A. `java FindBaz`

B. `java -classpath test FindBaz`

C. `java -classpath .:test FindBaz`

D. `java -classpath .:test/myApp FindBaz`

E. `java -classpath test:test/myApp FindBaz`

F. `java -classpath test:test/myApp:. FindBaz`

G. `java -classpath test/myApp:test:. FindBaz`

10. Given the following directory structure:

```
test-|
     |- Test.java
     |
     |- myApp-|
              |- Foo.java
              |
              |- myAppSub-|
                          |- Bar.java
```

If the current directory is test, and you create a .jar file by invoking this,

```
jar -cf MyJar.jar myApp
```

then which path names will find a file in the .jar file? (Choose all that apply.)

A. `Foo.java`

B. `Test.java`

C. `myApp/Foo.java`

D. `myApp/Bar.java`

E. `META-INF/Foo.java`

F. `META-INF/myApp/Foo.java`

G. `myApp/myAppSub/Bar.java`

11. Given the following directory structure:

```
test-|
     |- GetJar.java
     |
     |- myApp-|
              |-Foo.java
```

And given the contents of GetJar.java and Foo.java:

```
3. public class GetJar {
4.   public static void main(String[] args) {
5.     System.out.println(myApp.Foo.d);
6.   }
7. }
```

```
3. package myApp;
4. public class Foo { public static int d = 8; }
```

If the current directory is "test", and myApp/Foo.class is placed in a JAR file called MyJar.jar located in test, which set(s) of commands will compile GetJar.java and produce the output `8`? (Choose all that apply.)

A.
```
javac -classpath MyJar.jar GetJar.java
java GetJar
```

B.
```
javac MyJar.jar GetJar.java
java GetJar
```

C.
```
javac -classpath MyJar.jar GetJar.java
java -classpath MyJar.jar GetJar
```

D.
```
javac MyJar.jar GetJar.java
java -classpath MyJar.jar GetJar
```

12. Given the following directory structure:

```
x-|
  |- GoDeep.class
  |
  |- test-|
          |- MyJar.jar
          |
          |- myApp-|
                   |-Foo.java
                   |-Foo.class
```

And given the contents of GoDeep.java and Foo.java:

```
3. public class GoDeep {
4.   public static void main(String[] args) {
5.     System.out.println(myApp.Foo.d);
6.   }
7. }
```

```
3. package myApp;
4. public class Foo { public static int d = 8; }
```

And MyJar.jar contains the following entry:

```
myApp/Foo.class
```

If the current directory is x, which commands will successfully execute GoDeep.class and produce the output `8`? (Choose all that apply.)

A. `java GoDeep`

B. `java -classpath . GoDeep`

C. `java -classpath test/MyJar.jar GoDeep`

D. `java GoDeep -classpath test/MyJar.jar`

E. `java GoDeep -classpath test/MyJar.jar:.`

F. `java -classpath .:test/MyJar.jar GoDeep`

G. `java -classpath test/MyJar.jar:. GoDeep`

SELF TEST ANSWERS

1. Given:

```
1. // insert code here
2. class StatTest {
3.   public static void main(String[] args) {
4.     System.out.println(Integer.MAX_VALUE);
5.   }
6. }
```

Which, inserted independently at line 1, compiles? (Choose all that apply.)

A. `import static java.lang;`

B. `import static java.lang.Integer;`

C. `import static java.lang.Integer.*;`

D. `import static java.lang.Integer.*_VALUE;`

E. `import static java.lang.Integer.MAX_VALUE;`

F. None of the above statements are valid import syntax

Answer:

☑ **C** and **E** are correct syntax for static imports. Line 4 isn't making use of `static imports`, so the code will also compile with none of the imports.

☒ **A, B, D,** and **F** are incorrect based on the above. (Objective 7.1)

2. Given:

```
import static java.lang.System.*;
class _ {
  static public void main(String... __A_V_) {
    String $ = "";
    for(int x=0; ++x < __A_V_.length; )
      $ += __A_V_[x];
    out.println($);
  }
}
```

And the command line:

```
java _ - A .
```

What is the result?

A. `-A`

B. `A.`

C. `-A.`

D. `_A.`

E. `_-A.`

F. Compilation fails

G. An exception is thrown at runtime

Answer:

☑ **B** is correct. This question is using valid (but inappropriate and weird) identifiers, static imports, var-args in `main()`, and pre-incrementing logic.

☒ **A, C, D, E, F,** and **G** are incorrect based on the above. (Objective 7.2)

3. Given the default classpath:

```
/foo
```

And this directory structure:

```
foo
  |
  test
     |
     xcom
        |--A.class
        |--B.java
```

And these two files:

```
package xcom;
public class A { }

package xcom;
public class B extends A { }
```

Which allows `B.java` to compile? (Choose all that apply.)

A. Set the current directory to `xcom` then invoke
```
javac B.java
```

B. Set the current directory to `xcom` then invoke
```
javac -classpath . B.java
```

C. Set the current directory to `test` then invoke
```
javac -classpath . xcom/B.java
```

D. Set the current directory to `test` then invoke
```
javac -classpath xcom B.java
```

E. Set the current directory to `test` then invoke
```
javac -classpath xcom:. B.java
```

Answer:

☑ **C** is correct. In order for `B.java` to compile, the compiler first needs to be able to find `B.java`. Once it's found `B.java` it needs to find `A.class`. Because `A.class` is in the `xcom` package the compiler won't find `A.class` if it's invoked from the `xcom` directory. Remember that the `-classpath` isn't looking for `B.java`, it's looking for whatever classes `B.java` needs (in this case `A.class`).

☒ **A, B,** and **D** are incorrect based on the above. **E** is incorrect because the compiler can't find `B.java`. (Objective 7.2)

4. Given two files:

```
a=b.java
c_d.class
```

Are in the current directory, which command-line invocation(s) could complete without error? (Choose all that apply.)

A. `java -Da=b c_d`

B. `java -D a=b c_d`

C. `javac -Da=b c_d`

D. `javac -D a=b c_d`

Answer:

☑ **A** is correct. The `-D` flag is NOT a compiler flag, and the name=value pair that is associated with the `-D` must follow the `-D` with no spaces.

☒ **B, C,** and **D** are incorrect based on the above. (Objective 7.2)

5. If three versions of `MyClass.class` exist on a file system:

Version 1 is in `/foo/bar`
Version 2 is in `/foo/bar/baz`
Version 3 is in `/foo/bar/baz/bing`

And the system's classpath includes

```
/foo/bar/baz
```

And this command line is invoked from `/foo`

```
java -classpath /foo/bar/baz/bing:/foo/bar MyClass
```

Which version will be used by `java`?

A. `/foo/MyClass.class`

B. `/foo/bar/MyClass.class`

C. `/foo/bar/baz/MyClass.class`

D. `/foo/bar/baz/bing/MyClass.class`

E. The result is not predictable.

Answer:

☒ **D** is correct. A -classpath included with a `java` invocation overrides a system classpath. When `java` is using any classpath, it reads the classpath from left to right, and uses the first match it finds.

☒ **A, B, C,** and **E** are incorrect based on the above. (Objective 7.5)

6. Given two files:

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

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

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

A. `5 6 5 6`

B. `5 6` followed by an exception

C. Compilation fails with an error on line 6

D. Compilation fails with an error on line 7

E. Compilation fails with an error on line 8

F. Compilation fails with an error on line 9

Answer:

☑ **C, D,** and **E** are correct. Variable `a` (default access) cannot be accessed from outside the package. Since variable `b` is protected, it can be accessed only through inheritance.

☒ **A, B,** and **F** are incorrect based on the above. (Objectives 1.1, 7.1)

7. Given:

```
3. import java.util.*;
4. public class Antique {
5.   public static void main(String[] args) {
6.     List<String> myList = new ArrayList<String>();
```

```
 7.     assert (args.length > 0);
 8.     System.out.println("still static");
 9.   }
10. }
```

Which sets of commands (`javac` followed by `java`) will compile and run without exception or error? (Choose all that apply.)

A.
```
javac Antique.java
java Antique
```

B.
```
javac Antique.java
java -ea Antique
```

C.
```
javac -source 6 Antique.java
java Antique
```

D.
```
javac -source 1.4 Antique.java
java Antique
```

E.
```
javac -source 1.6 Antique.java
java -ea Antique
```

Answer:

☑ **A** and **C** are correct. If assertions (which were first available in Java 1.4) are enabled, an AssertionError will be thrown at line 7.

☒ **D** is incorrect because the code uses generics, and generics weren't introduced until Java 5. **B** and **E** are incorrect based on the above. (Objective 7.2)

8. Given:

```
 3. import java.util.*;
 4. public class Values {
 5.   public static void main(String[] args) {
 6.     Properties p = System.getProperties();
 7.     p.setProperty("myProp", "myValue");
 8.     System.out.print(p.getProperty("cmdProp") + " ");
 9.     System.out.print(p.getProperty("myProp") + " ");
10.     System.out.print(p.getProperty("noProp") + " ");
11.     p.setProperty("cmdProp", "newValue");
12.     System.out.println(p.getProperty("cmdProp"));
13.   }
14. }
```

And given the command line invocation:

```
java -DcmdProp=cmdValue Values
```

What is the result?

A. `null myValue null null`

B. `cmdValue null null cmdValue`

C. `cmdValue null null newValue`

D. `cmdValue myValue null cmdValue`

E. `cmdValue myValue null newValue`

F. An exception is thrown at runtime

Answer:

☑ **E** is correct. System properties can be set at the command line, as indicated correctly in the example. System properties can also be set and overridden programmatically.

☒ **A, B, C, D,** and **F** are incorrect based on the above. (Objective 7.2)

9. Given the following directory structure:

```
x-|
  |- FindBaz.class
  |
  |- test-|
          |- Baz.class
          |
          |- myApp-|
                   |- Baz.class
```

And given the contents of the related .java files:

```
1. public class FindBaz {
2.   public static void main(String[] args) { new Baz(); }
3. }
```

In the test directory:

```
1. public class Baz {
2.   static { System.out.println("test/Baz"); }
3. }
```

In the myApp directory:

```
1. public class Baz {
2.   static { System.out.println("myApp/Baz"); }
3. }
```

If the current directory is x, which invocations will produce the output `"test/Baz"`? (Choose all that apply.)

A. `java FindBaz`

B. `java -classpath test FindBaz`

C. `java -classpath .:test FindBaz`

D. `java -classpath .:test/myApp FindBaz`

E. `java -classpath test:test/myApp FindBaz`

F. `java -classpath test:test/myApp:. FindBaz`

G. `java -classpath test/myApp:test:. FindBaz`

Answer:

☑ **C** and **F** are correct. The `java` command must find both `FindBaz` and the version of `Baz` located in the test directory. The `"."` finds `FindBaz`, and `"test"` must come before `"test/myApp"` or `java` will find the other version of `Baz`. Remember the real exam will default to using the Unix path separator.

☒ **A, B, D, E,** and **G** are incorrect based on the above. (Objective 7.2)

10. Given the following directory structure:

```
test-|
     |- Test.java
     |
     |- myApp-|
              |- Foo.java
              |
              |- myAppSub-|
                          |- Bar.java
```

If the current directory is test, and you create a .jar file by invoking this,

```
jar -cf MyJar.jar myApp
```

then which path names will find a file in the .jar file? (Choose all that apply.)

A. `Foo.java`

B. `Test.java`

C. `myApp/Foo.java`

D. `myApp/Bar.java`

E. `META-INF/Foo.java`

F. `META-INF/myApp/Foo.java`

G. `myApp/myAppSub/Bar.java`

Answer:

☑ **C** and **G** are correct. The files in a .jar file will exist within the same exact directory tree structure in which they existed when the .jar was created. Although a .jar file will contain a META-INF directory, none of your files will be in it. Finally, if any files exist in the directory from which the `jar` command was invoked, they won't be included in the .jar file by default.

☒ **A, B, D, E,** and **F** are incorrect based on the above. (Objective 7.5)

11. Given the following directory structure:

```
test-|
     |- GetJar.java
     |
     |- myApp-|
              |-Foo.java
```

And given the contents of GetJar.java and Foo.java:

```
3. public class GetJar {
4.   public static void main(String[] args) {
5.     System.out.println(myApp.Foo.d);
6.   }
7. }

3. package myApp;
4. public class Foo { public static int d = 8; }
```

If the current directory is "test", and myApp/Foo.class is placed in a JAR file called MyJar.jar located in test, which set(s) of commands will compile GetJar.java and produce the output `8`? (Choose all that apply.)

A.
```
javac -classpath MyJar.jar GetJar.java
java GetJar
```

B.
```
javac MyJar.jar GetJar.java
java GetJar
```

C.
```
javac -classpath MyJar.jar GetJar.java
java -classpath MyJar.jar GetJar
```

D.
```
javac MyJar.jar GetJar.java
java -classpath MyJar.jar GetJar
```

Answer:

☑ **A** is correct. Given the current directory and where the necessary files are located, these are the correct command line statements.

☒ **B** and **D** are wrong because javac MyJar.jar GetJar.java is incorrect syntax. **C** is wrong because the -classpath MyJar.java in the java invocation does not include the test directory. (Objective 7.5)

12. Given the following directory structure:

```
x-|
  |- GoDeep.class
  |
  |- test-|
          |- MyJar.jar
          |
          |- myApp-|
                   |-Foo.java
                   |-Foo.class
```

And given the contents of GoDeep.java and Foo.java:

```
3. public class GoDeep {
4.   public static void main(String[] args) {
5.     System.out.println(myApp.Foo.d);
6.   }
7. }
```

```
3. package myApp;
4. public class Foo { public static int d = 8; }
```

And MyJar.jar contains the following entry:

```
myApp/Foo.class
```

If the current directory is x, which commands will successfully execute GoDeep.class and produce the output `8`? (Choose all that apply.)

A. `java GoDeep`

B. `java -classpath . GoDeep`

C. `java -classpath test/MyJar.jar GoDeep`

D. `java GoDeep -classpath test/MyJar.jar`

E. `java GoDeep -classpath test/MyJar.jar:.`

F. `java -classpath .:test/MyJar.jar GoDeep`

G. `java -classpath test/MyJar.jar:. GoDeep`

Answer:

☑ **F** and **G** are correct. The `java` command must find both `GoDeep` and `Foo`, and the `-classpath` option must come before the class name. Note, the current directory (.), in the `classpath` can be searched first or last.

☒ **A, B, C, D,** and **E** are incorrect based on the above. (Objective 7.5)

A

About the CD

The CD-ROM included with this book comes complete with MasterExam and the electronic version of the book. The software is easy to install on any Windows 98/NT/2000/XP/Vista computer and must be installed to access the MasterExam feature. You may, however, browse the electronic book directly from the CD without installation. To register for a second bonus MasterExam, simply click the Bonus Material link on the Main Page and follow the directions to the free online registration.

System Requirements

Software requires Windows 98 or higher and Internet Explorer 5.0 or above and 20 MB of hard disk space for full installation. The Electronic book requires Adobe Acrobat Reader.

Installing and Running MasterExam

If your computer CD-ROM drive is configured to auto run, the CD will automatically start up upon inserting the disk. From the opening screen you may install MasterExam by pressing the MasterExam buttons. This will begin the installation process and create a program group named "LearnKey." To run MasterExam, choose Start | Programs | LearnKey. If the auto run feature did not launch your CD, browse to the CD and click RunInstall.

MasterExam

MasterExam provides you with a simulation of the actual exam. The number of questions, types of questions, and the time allowed are intended to be an accurate representation of the exam environment. You have the option to take an open-book exam, including hints, references, and answers; a closed-book exam; or the timed MasterExam simulation.

When you launch MasterExam, a digital clock display will appear in the upper-left corner of your screen. The clock will continue to count down to zero unless you choose to end the exam before the time expires. To register for a second bonus MasterExam, simply click the Bonus Material link on the Main Page and follow the directions to the free online registration.

Electronic Book

The entire contents of the Study Guide are provided in PDF format. Adobe's Acrobat Reader has been included on the CD.

Help

A help file is provided through the Help button on the main page in the lower-left corner. Individual help features are also available through MasterExam and LearnKey's Online Training.

Removing Installation(s)

MasterExam is installed on your hard drive. For best results for removal of programs use the Start | Programs | LearnKey | Uninstall options to remove MasterExam.

If you want to remove the Real Player, use the Add/Remove Programs icon from your Control Panel. You may also remove the LearnKey training program from this location.

Technical Support

For questions regarding the technical content of the electronic book, or MasterExam, please visit www.osborne.com or e-mail customer.service@mcgraw-hill.com. For customers outside the 50 United States, e-mail international_cs@mcgraw-hill.com.

LearnKey Technical Support

For technical problems with the software (installation, operation, removing installations), and for questions regarding any LearnKey Online Training content, please visit www.learnkey.com or e-mail techsupport@learnkey.com.

This page intentionally left blank

INDEX

-- (decrement) operator, 302–303
- (subtraction) operator, 306
! (boolean invert) logical operator, 309–310
!= (not equal to) operator, 292
% (remainder) operator, 299
& (non-short-circuit AND) operator, 288, 308
&& (short-circuit AND) operator, 306, 308, 548
* (multiplication) operator, 308
* quantifier, 495
. (dot) metacharacter, 497
. (dot operator), 25, 27
.java files, 791–792
/ (division) operator, 306
: (colons), 354
; (semicolons), 17, 41, 344–345, 675
? quantifier, 495
@argfiles command-line options, 791
\ (backslashes), 798
^ (exclusive-OR [XOR]) logical operator, 288, 309–310, 553
^ (regex carat) operator, 494
{} (curly braces), 329, 332
| (non-short-circuit OR) operator, 288, 308
|| (short-circuit OR) operator, 306, 308
+ (addition) operator, 288, 299, 307
+ quantifier, 495
++ (increment) operator, 302–303
+= (compound additive operator), 301
<? extends Animal> syntax, 618
<? super ...> syntax, 619
<> (angle brackets), 600
<?> wildcard, 620
<E> placeholder, 622
<Integer> type, 602
<JButton> type, 608
<Object> type, 600
<X> type declaration, 629
= (assignment) operators
  assigning one primitive variable to another, 198
  compound, 197, 289–290
  floating-point numbers, 196
  literals too large for variable, 196–197
  overview, 190–191, 288–289
  primitive casting, 193–195
  primitive variables, 191–193
  reference variable, 198–200
  variable scope, 200–202
== (equals) operator, 245–246, 292–294, 544

A

absolute paths, 801–802
abstract classes, 16–19, 20
abstract keyword, 15
abstract methods, 41–45
access control
  class access, 13
  default access, 13–14, 32–34, 36–38
  defined, 12
  local variables, 38–39
  modifiers, 24–26
  private members, 29–32
  protected members, 32–36
  public access, 14–15, 26–29
access modifiers
  declaring class members
    default, 32–34, 36–38
    local variables and, 38–39
    overview, 24–26
    private, 29–32
    protected, 32–36
    public, 26–29
  defined, 12
  method-local inner class, 682
add( ) method, 581, 620
addAll( ) method, 601, 603
addAnimals( ) method, 616–617
addition (+) operator, 288, 299, 307
addJob( ) method, 754–755
AND operators
  & non-short-circuit, 288, 308
  && short-circuit, 306, 308, 548
angle brackets (<>), 600
animate( ) method, 102
anonymous arrays, 228–230
anonymous inner classes
  argument-defined, 678–680
  overview, 673–678
anotherObject object, 749
append( ) method, 440
applications, launching with java command, 793–796
appropriate use of assertions, 392–394
arg_index format string element, 507
args index, 796
arguments. *See also* var-args
  anonymous inner classes defined by, 678–680
  command-line, 393, 795–796
  constructor, 238, 628
  defined, 46
  final, 41
  just-in-time array, 229–230
  using assertions to validate, 392–393
arithmetic operators
  decrement, 302–303
  increment, 302–303
  overview, 298
  remainder, 299
  string concatenation, 299–301
ArrayIndexOutOfBoundsException subclass, 224, 368, 382

ArrayList class
basics, 567–568
Collection Interface Concrete Implementation, 565
List interface implementation, 562
mixing generic and non-generic collections, 601–607
of Strings, 596
arrays
constructing
anonymous, 228–230
multidimensional, 223
on one line, 226–228
one-dimensional, 221–222
overview, 220–221
declarations, 55–57, 219–220, 226–228
enhanced for loop for, 350–352
initialization blocks, 234–237
initializing
and constructing anonymous, 228–230
declaring, and constructing on one line, 226–228
elements in loop, 225–226
legal element assignments, 230
multidimensional, 233–234
of object references, 231–232
one-dimensional, 232–233
overview, 224–225
primitive, 230–231
reference assignments for one-dimensional, 232–234
instance variables, 206
length attribute, 437
as objects, 297
primitive, 230–231
returning values, 128
shortcut syntax, 226–228
use of brackets, 56
Arrays class
asList ( ) method, 579
collections, 576
converting to Lists, 579
key methods, 593
searching, 576–578
sort( ) method, 576
asList ( ) method, 579
assert statements, 386–389, 394
AssertionError subtype, 375, 381–382, 385, 393
assertions
appropriate use of, 392–394
disabling
at runtime, 390
selective, 390–391
enabling
compiling assertion-aware code, 388–389
identifiers versus keywords, 387–388
at runtime, 389
selective, 390–391
expression rules, 385–387
overview, 328, 383–385
assignment (=) operators
assigning one primitive variable to another, 198
compound, 197, 289–290
floating-point numbers, 196
literals too large for variable, 196–197
overview, 190–191, 288–289
primitive casting, 193–195
primitive variables, 191–193
reference variable, 198–200
variable scope, 200–202
assignments. *See also* arrays; garbage collection; wrappers
assignment operators
assigning one primitive variable to another, 198
floating-point numbers, 196
literals too large for variable, 196–197
overview, 190–191
primitive casting, 193–195
primitive variables, 191–193
reference variable, 198–200
variable scope, 200–202
autoboxing
equals( ) method, 245–246
equals operator, 245–246
overview, 244–245
use of, 246–247
heap, 184–185
literal values for primitive types
boolean, 189
character, 189–190
floating-point, 188–189
integer, 186–188
overview, 186
string, 190
local variables
array references, 210
assigning one reference variable to another, 210–213
object references, 209–210
overview, 207
primitives, 207–209
overloading
with boxing and var-args, 249–250
in combination with var-args, 253–254
overview, 247–249
when combining widening and boxing, 251–253
widening reference variables, 250–251
passing variables into methods
object reference variables, 213–214
overview, 213
pass-by-value semantics, 214–215
primitive variables, 215–218
stack, 184–185
uninitialized variables
array, 206
object reference, 203–206
overview, 203
primitive, 203–204
atomic operations, 733–735
autoboxing
with collections, 568
equals( ) method, 245–246
equals operator, 245–246
overloading
with var-args and, 249–250
when combining widening and, 252–253
overview, 244–245
in switch statements, 337
use of, 246–247
automatic variables, 207. *See also* local variables

B

backed collections
  key methods, 593–595
  overview, 589–591
  using PriorityQueue class, 591–592
backslashes (\), 798
bar( ) method, 215
Bar object, 199, 679
behavior, 2. *See also* methods
binarySearch( ) method, 576
bitwise operators, 305
block variables, 201
blocked threads, 719–720
boolean add(element) interface method, 594
boolean containsKey(object key) interface method, 594
boolean contains(object) interface method, 594
boolean containsValue(object value) interface method, 594
boolean createNewFile( ) method, 446
boolean equals (Object obj) method, 543
boolean exists( ) method, 446
boolean hasNext( ) method, 580
boolean invert (!) logical operator, 309–310
Boolean wrapper, 239
booleans
  arguments in backed collections, 590–591
  assigning versus testing, 334
  literals, 189
  for loops, 346–347
  and relational operators, 290–291
boxing
  equals( ) method, 245–246
  equals operator, 245–246
  overloading
    with var-args and, 249–250
    when combining widening and, 252–253
  overview, 244–245
  in switch statements, 337
  use of, 246–247
brackets, array, 56
branching, if-else, 329–332
break statements
  labeled, 354–356
  in switch blocks, 338–340
  switch statements, 335
  unlabeled, 353–354
  use of, 352–353
BufferedReader class, 443
BufferedWriter class, 444
byte variables, 192

C

Calculator class, 754
Calendar class, 477–479
call stack
  constructors on, 133
  overloaded constructors on, 141
  propagating uncaught exceptions, 362–363
  unwinding, 367
camelCase format, 7–8
can't-be-overridden restriction, 40
carat (^) operator, 494
case constants
  default, 341–342
  evaluation of, 338
  legal expressions for, 335–338
casting
  primitive, 193–195
  reference variables, 116–119
catch clauses, 357–359, 362
catch keyword, 357–359
ceiling( ) method, 587
ceilingKey( ) method, 587
chained methods, 442
chaining
  combining I/O classes, 449–452
  constructors, 132–133
changeNum( ) method, 470
changeSize( ) method, 110
character literals, 189–190
characters, Unicode, 51, 189, 426–427
charAt( ) method, 435
ChatClient class, 663–664
checked exceptions, 373–375
checkup( ) method, 608
class files, 791–794, 805
class literals, 738
ClassCastException class, 382, 547
classes. *See also* dates; *exceptions; individual classes by name*; inner classes; member declarations
  cohesion, 3, 151–154
  collections, 557–561
  combining, 449–452
  declaration
    abstract classes, 16–19
    class access, 13
    default access, 13–14
    final classes, 15–16
    modifiers, 12–13, 15
    overview, 10–11
    public access, 13–15
    source file rules, 11–12
  defined, 2
  File, 443, 445–447
  finding other, 3–4
  generic, 623–627
  interfaces
    implementing, 122
    relationship with, 20
  naming standards, 7
  searching for
    classpaths, 797–798
    overview, 796–797
    packages, 799–801
    relative and absolute paths, 801–802
  thread-safe, 742–744
  wrapper, 237–238
classes directory, 791–793
-classpath option, javac command, 790–791, 794, 798
classpaths, 797–798, 808
close( ) method, 448–449
cmdProp=cmdVal property, 795
cohesion, 3, 151–154
Collection classes, 56
Collection interface, 557–559
collections. *See also* generics; hashCode( ) method
  ArrayList basics, 567–568
  autoboxing with, 568
  backed
    key methods, 593–595
    overview, 589–591
    using PriorityQueue class, 591–592
  classes, 557–561
  converting arrays to Lists to arrays, 579
  interfaces
    List, 561–562
    Map, 563–564
    overview, 557–561

collections. *See also* generics;
hashCode( ) method (*Cont.*)
Queue, 564–566
Set, 562–563
legacy code, 597–600
Lists, 580–581
Maps, 583–586
mixing generic and non-generic, 601–607
overriding Object methods
equals( ), 544–549
overview, 542
toString( ), 542–544
overview, 556–557
searching
arrays and, 576–578
TreeSets and TreeMaps, 586–588
Sets, 581–583
sorting
Arrays class, 576
Comparable interface, 571–573
Comparator interface, 574–575
overview, 569–571
Collections class, 593
Collections Framework, 556
Collections.sort( ) method, 571, 573–574, 576
Collections.synchronizedList( ) method, 743–744
colons (:), 354
combining I/O classes, 449–452
command-line arguments, 393, 795–796
commands
jar, 803–804
java command, 793–796
javac
assertion-aware code, 388–389
compiling with, 790–793
constructor code generated by, 137
failures, 350
JAR files, 804
warnings, 603
searching for other classes with
classpaths, 797–798
overview, 796–797
packages, 799–801
relative and absolute paths, 801–802
Comparable interface, 561, 566, 571–573
Comparator interface, 561, 574–575, 592
compare( ) method, 574
compareTo( ) method, 571–574
compiler
assertion-aware code, 388–389
constructor code generated by, 137
failures, 350
JAR files, 804
javac command, 790–793
searching for other classes
classpaths, 797–798
overview, 796–797
packages, 799–801
relative and absolute paths, 801–802
warnings, 603
compound additive operator (+=), 301
compound assignment operators, 197, 289–290
concat( ) method, 435–436
concatenation operator, string, 299–301
concrete classes, 42–44
conditional expressions, 346–347
conditional operators, 304–305
consistency of equals( ) method, 549
constant pool, String, 434
constant specific class bodies, 63, 65
constants
case
default, 341–342
evaluation of, 338
legal expressions for, 335–338
declaring interface, 22–23
enums, 63
MAX_VALUE, 807
values, 23
constructing arrays, 220–223
constructor arguments, 238, 628
constructors
chaining, 132–133
code causing to run, 222
compiler-generated, 137
declarations
in enums, 63–65
overview, 47–48
default, 135–139
overloaded, 139–145
overview, 130–132
rules for, 133–134
super, 136–139
wrapper, 239
continue statements
labeled, 354–356
overview, 352–353
unlabeled, 353–354
contracts
equals( ) method, 549
hashCode( ) method, 554–556
conversion format string element, 508
conversion utilities, wrapper
parseXxx( ) methods, 241
toString( ) method, 242
toXxxString( ) method, 242–243
valueOf( ) method, 241
xxxValue( ) methods, 240–241
copying reference variables, 213
correct use of assertions, 392–394
coupling, 151–153
covariant returns, 127–128
-cp abbreviation, java command, 798
CreateAnArrayList class, 628
curly braces ({}), 329, 332
currencies, 473–474, 482–487
-d option, javac command, 790–793
-D option, system properties, 794–795

D

daemon threads, 704
DataInputStream class, 473
DataOutputStream class, 473
Date class, 476–477
DateFormat class, 480–481, 484
dates
Calendar class, 477–479
Date class, 476–477
DateFormat class, 480–481
Locale class, 482–485
orchestrating classes related to, 474–475
overview, 473–474
dead thread state, 716, 720
deadlocks, thread, 745–746
"Deadly Diamond of Death" scenario, 100

decimal literals, 186
decision statements
  if
    if-else branching, 329–332
    legal expressions for, 332–334
    overview, 328
  switch
    break statement in, 335, 338–340
    default case, 341–342
    fall-through in, 338–340
    legal expressions for, 335–338
    overview, 328, 334–335
declaration
  access modifiers
    default, 32–34, 36–38
    local variables and, 38–39
    overview, 24–26
    private, 29–32
    protected, 32–36
    public, 26–29
  array, 219–220, 226–228
  class
    abstract, 16–19
    access, 13
    default access, 13–14
    final, 15–16
    modifiers, 12–13, 15
    overview, 10–11
    public access, 13–15
    source file rules, 11–12
  classpath, 797–798
  constructor, 47–48
  enum
    constructors, methods, and variables in, 63–65
    overview, 60–63
  exception, 371–376
  generics
    classes, 623–627
    methods, 627–630
    overview, 622–623
  interface
    constants, 22–23
    overview, 19–22
  for loop, 346, 351
  nonaccess modifiers
    abstract methods, 41–45
    final arguments, 41
    final methods, 40
    methods with var-args, 46–47
    native methods, 46
    overview, 39
    strictfp methods, 46
    synchronized methods, 45
  var-arg rule, 47
  variable
    array, 55–57
    in enums, 63–65
    final, 57–58
    generics and polymorphism, 608
    instance, 51–53
    local, 53–55
    overview, 49
    primitive, 49–51
    reference, 51
    static, 59–60
    transient, 59
    volatile, 59
declared return types, 129
decoupling reference variables, 258
decrement (--) operator, 302–303
default access classes, 13–14
default access control type, 24
default constructors, 135–139
default hashcode method, 552
default keyword, 341–343
default members, 32–34, 36–38
default primitive and reference type values, 203
default priority, thread, 725
defining threads, 705–706
delete( ) method, 440–441
delimiters, 502
descending order, 588
development
  compiling with javac command, 790–793
  Java Archive files, 802–805
  launching applications with java command, 793–796
  overview, 789–790
  searching for other classes
    classpaths, 797–798
    overview, 796–797
    packages, 799–801
    relative and absolute paths, 801–802
  static imports, 806–807
direct subdirectories, 802
directories
  classpaths, 797–798
  javac command, 791–793
  META-INF, 804
  myApp, 803
  relative and absolute paths, 801–802
  root, 804
  source, 792
  working with files and, 452–457
disabling assertions
  at runtime, 390
  selective, 390–391
division (/) operator, 306
do loops, 344–345
doInsert( ) method, 620
doStuff( ) method, 114, 215, 670, 679
dot (.) metacharacter, 497
dot operator (.), 25, 27
downcasting, 117–118

E

early loop termination, 348
else if statement, 330–331
else statement, 329–332
enabling assertions
  compiling assertion-aware code, 388–389
  identifiers versus keywords, 387–388
  javac, 388
  at runtime, 389
  selective, 390–391
encapsulation, 86–89
enclosing class, 682
engines, regex, 488
enhanced for loops, 350–352
entry points, 338
enums
  declaring constructors, methods, and variables in, 63–65
  equality operators for, 294–295
  maps, 584
  overview, 60–63
EOFException subclass, 370, 373
equal (=) sign, 191
equal to (==) operator, 245–246, 292–294, 544

equality operators
for enums, 294–295
overview, 292
for primitives, 292–293
for reference variables, 293–294
equals (==) operator, 245–246, 292–294, 544
equals( ) method
collections, 562–563
hashCode( ), 548–549
hashcodes, 586
inheritance, 91
maps, 583
overriding, 544–549
wrappers, 245–246
equalsIgnoreCase( ) method, 435–436
equals(java.lang.Object) method, 554
equals(Object) method, 554
Error class, 366–368, 373–375
error detection and handling. *See* exceptions
errors, scoping, 201–202
escape codes, 190, 498
events, 9
Exception class, 366–368, 374
ExceptionInInitializerError, 382
exceptions
catch keyword, 357–359
checked, 373–375
declaration and public interface, 371–376
defined, 357, 365–366
finally block, 359–362
hierarchy of, 366–369
JVM thrown, 379–380
matching, 369–371
overview, 328, 356–357
programmatic
defined, 379
list of, 382
programmatically thrown, 380–381
propagating uncaught, 362–365
rethrowing same, 376–378
try keyword, 357–359
unchecked, 373, 375
exclusive-OR (XOR) logical operator, 288, 309–310, 553
execution, threads of, 702, 709, 757
existingDir subdirectory, 454
explicit casts, 193, 197
expressions
assertions, 385–387
for case constant, 335–338
conditional, 346–347
for if statements, 332–334
iteration, 347–348
for statement, 351
for switch statements, 335–338
extends keyword
generic methods, 618
interfaces, 122–124
java.lang.Thread, 705–706

F

fall-through in switch blocks, 338–340
FIFO (first-in, first-out), 564
File class, 443, 445–447
file navigation
combining I/O classes, 449–452
creating files using class File, 445–447
directories, 452–457
FileWriter and FileReader classes, 447–449
overview, 443–445
FileNotFoundException subclass, 370–371
FileReader class, 443, 447–449
FileWriter class, 444, 447–449
final arguments, 41
final classes, 15–16
final keyword, 15, 40
final methods, 40, 103
final variables, 57–58, 303, 336
final void notify( ) method, 543
final void notifyAll( ) method, 543
final void wait( ) method, 543
finalize( ) method, 263, 542
finally clause, 359–362
find( ) method, 499
first-in, first-out (FIFO), 564
flags, 507
floating points, 15, 46, 188–189
floor( ) method, 587
floorKey( ) method, 587
flow control
assertions
enabling, 387–391
expression rules, 385–387
overview, 383–385
using appropriately, 392–394
exceptions
catch keyword, 357–359
declaration of, 371–376
defined, 365–366
finally block, 359–362
hierarchy of, 366–369
JVM thrown, 379–380
matching, 369–371
overview, 356–357
programmatically thrown, 380–381
propagating uncaught, 362–365
public interface and, 371–376
rethrowing same, 376–378
try keyword, 357–359
if statements
if-else branching, 329–334
legal expressions for, 332–334
overview, 328
loops
break statement, 352–353
continue statement, 352–353
do, 344–345
for, 345–352
labeled statements, 354–356
unlabeled statements, 353–354
while, 343–344
overview, 327–328
switch statements
break statement in, 338–340
default case, 341–342
fall-through in, 338–340
legal expressions for, 335–338
overview, 328, 334–335
flush( ) method, 448–449
Foo class
argument-defined anonymous inner classes, 679–680
compiler-generated constructor code, 137
natural order in, 561
overloaded methods, 114
reference variable assignments, 199
shadowing instance variables, 217–218
for loops
conditional expressions, 346–347
declaration, 346

enhanced, 350–352
initialization, 208, 346
issues with, 348–350
iteration expressions, 347–348
legacy collections, 598
overview, 345–346
PriorityQueue class, 592
threads, 714
forced exits, 347–348
forcing garbage collection, 260–262
for-each. *See* for loops
for-in. *See* for loops
format( ) method, 506–508
formatting
dates and numbers
Calendar class, 477–479
Date class, 476–477
DateFormat class, 480–481
Locale class, 482–485
NumberFormat class, 485–487
orchestrating classes, 474–475
overview, 473
tokenizing
and delimiters, 502
format( ) method, 506–508
overview, 501–502
printf( ) method, 506–508
Scanner class, 504–506
String.split( ) method, 502–504

G

garbage collection
code making objects eligible for
finalize( ) method, 263
forcing, 260–262
isolating references, 259–260
nulling references, 257–258
overview, 257
reassigning reference variables, 258–259
in Java, 255–257
memory management and, 254–255
generics
declaring
classes, 623–627
methods, 627–630
overview, 622–623
legacy code, 597–600
methods, 609–622
mixing with non-generic collections, 601–607
overview, 541–542, 595–597
polymorphism and, 607–609
get( ) method, 581, 584, 586, 599, 606
getDateInstance( ) method, 480–481
getInstance( ) method, 477, 480
getName( ) method, 710–711
getProperty( ) method, 795
getState( ) method, 708
getter methods, 8–9, 88
greedy quantifiers, 495–497
group( ) method, 499

H

HardToRead class, 543
HAS-A relationships, 96–98
hashCode( ) method
equals( ) method, 548–549
HashSet, 562
Maps, 583
overriding
contract, 554–556
implementing, 552–554
overview, 549–552
real-life hashing, 551
HashMap class
Collection interface, 557, 565
collections, 566
hashCode( ) method, 555–556
LinkedHashMap, 564
Map interface implementation, 563
overriding equals( ), 545
use of, 583
HashSet class, 562, 565
Hashtable class, 557, 560, 564–565
headMap( ) method, 590
headSet( ) method, 590
heap, 220–221, 255
hexadecimal literals, 187–188
hierarchy of exceptions, 366–369
high cohesion, 152–153
higher( ) method, 587
higherKey( ) method, 587
higher-level classes, 460
highest-priority threads, 724

I

IDE (Integrated Development Environment) tool, 8
identifiers
versus keywords, 387–388
legal, 5–6
Map interface, 563
overview, 2–4
IEEE 754 standard, 15
if statements
if-else branching, 329–332
initialization, 208
legal expressions for, 332–334
overview, 328
illegal overrides, 109, 114
IllegalArgumentException, 381–382, 393
IllegalMonitorStateException, 749–750
IllegalStateException, 382
immutability of strings, 426–433
implementation classes, 121–122, 560, 680
implementers, interface, 677–678
implementing
equals( ) method, 546–549
hashCode( ) method, 552–554
interfaces, 120–125
java.lang.Runnable, 706
implicit casts, 193
import statements, 3–4, 11, 799–800, 805–806
imports, static, 806–807
increment (++) operator, 302–303
increment expression, 350
indenting, 331–332
indexOf( ) method, 561, 581
IndexOutOfBoundsException, 368
indirect implementations, 296
inheritance
versus dot operator for member access, 27
HAS-A relationship, 96–98
IS-A relationship, 94–95
overview, 3, 90–94
and serialization, 468–472
and subclasses, 25–26
initialization blocks, 234–237

initializing
arrays
anonymous, 228–230
elements in loop, 225–226
legal element assignments, 230
multidimensional, 233–234
of object references, 231–232
on one line, 226–228
one-dimensional, 232–233
overview, 224–225
primitive, 230–231
reference assignments for one-dimensional, 232–234
local variables, 54
for loops, 346
inner classes
anonymous
argument-defined, 678–680
overview, 673–678
coding regular, 664–668
instances, 664, 682
method-local, 670–672
overview, 661–664
referencing inner or outer instance from within, 668–670
static nested classes, 680–681
insert( ) method, 441, 602–603
insertion points, 576
instance methods, 102
instance variables
array, 206
defined, 51–53
hashCode( ) method, 552–553
object reference, 203–206
primitive, 203–204
protecting, 31
scope of, 201
uninitialized
array, 206
object reference, 203–206
overview, 203
primitive, 203–204
instanceof operator
compiler error, 297–298
equals( ) method, 547
inheritance, 90
IS-A relationship, 233
overview, 295–298
instances
initialization blocks, 234–235, 237
for java.text and java.util classes, 487
static nested classes, 680
instantiating
constructors
chaining, 132–133
default, 135–139
overloaded, 139–145
overview, 130–132
rules for, 133–134
inner classes, 666–668
outer class, 667
static nested classes, 681
threads, 706–708
int hashCode( ) method, 543
int indexOf(object) interface method, 594
int size( ) interface method, 594
int variable, 192
Integer class, 242, 807
integer literals
decimal, 186
hexadecimal, 187–188
octal, 187
Integrated Development Environment (IDE) tool, 8
interactions, thread
notifyAll( ), 752–757
overview, 746–751
interfaces
as array types, 231
collections
List, 561–562
Map, 563–564
overview, 557–561
Queue, 564–566
Set, 562–563
declaring
constants, 22–23
overview, 19–22
implementers, 677–678
implementing, 120–125
naming standards, 7
overview, 3
relationship with classes, 20
invoking
overloaded methods, 111–113
superclass version of overridden methods, 107–108
I/O
class File, 445–447
combining classes, 449–452
FileReader, 447–449
files and directories, 452–457
FileWriter, 447–449
java.io.Console class, 457–459
overview, 443–459
IOException class, 370–371, 373
IS-A relationship, 94–95, 233, 251, 547
isAlive( ) method, 40, 708
isEmpty( ) method, 755
islands of isolation, 259
isolating references, 259–260
iteration
defined, 560
expressions and for loops, 347–348
Iterators, 580, 594

J

JAR (Java Archive) files, 802–805, 808
jar command, 803–804
Java 6 compiler. *See* javac command
Java Archive (JAR) files, 802–805, 808
Java Code Conventions, 4, 6–8
java command
launching applications with, 793–796
searching for other classes
classpaths, 797–798
overview, 796–797
packages, 799–801
relative and absolute paths, 801–802
Java Virtual Machine (JVM)
exceptions, 379–380, 382
thread scheduler, 716
JavaBeans standards, 4, 8–10
javac command
assertion-aware code, 388–389
compiling with, 790–793
constructor code generated by, 137
failures, 350

JAR files, 804
searching for other classes
classpaths, 797–798
overview, 796–797
packages, 799–801
relative and absolute paths, 801–802
warnings, 603
java.io Mini API, 450
java.io.Console class, 457–459
java.lang.Class instances, 737
java.lang.Enum class, 584
java.lang.Object class, 718
java.lang.Runnable interface, 706
java.lang.Thread class, 705–706, 717
java.text class, 487
java.text.DateFormat, 474
java.text.NumberFormat, 474
java.util class, 487, 796
java.util package, 566, 805
java.util.ArrayList class, 567
java.util.Calendar, 474
java.util.Collections class, 558, 569
java.util.Date class, 474
java.util.HashMap class, 796
java.util.jar package, 805
java.util.Locale, 474
java.util.NavigableMap interface, 586
java.util.NavigableSet interface, 586
java.util.Properties class, 794
java.util.regex package, 805
java.utils.Arrays.sort( ) method, 571
JButton class, 608
jobs list, 755–756
join( ) method, 716–717, 726–728
jre/lib/ext subdirectory tree, 805
just-in-time array arguments, 229–230
JVM (Java Virtual Machine)
exceptions, 379–380, 382
thread scheduler, 716

K

keys, 795
keywords
abstract, 15
catch, 357–359
chart, 6
default, 341–343
extends
generic methods, 618
interfaces, 122–124
java.lang.Thread, 705–706
final, 15, 40
versus identifiers, 387–388
new, 222
overview, 3
strictfp, 15, 46
super, 619
throw, 367
try
exceptions, 357–359
with finally block, 359–361
wait( ) method, 750
without catch block, 361
without finally block, 361

L

labeled statements, 354–356
launching applications with java command, 793–796
legacy code, 597–602
legal array element assignments, 230
legal expressions
for case constant, 335–338
for if statements, 332–334
for switch statements, 335–338
legal identifiers, 4
legal overloaded methods, 110
legal overridden methods, 109
legal return types
covariant, 127–128
on overloaded methods, 126–127
on overridden methods, 127–128
returning values, 128–130
length attributes, array, 437
length( ) method, 435–436
length variable, 225–226
LinkedHashMap class, 557, 564–566
LinkedHashSet class, 560, 563, 565
LinkedList class, 562, 564–565, 580
List class, 608
List interface
collections, 561–562
in Collections Framework, 557
converting to arrays, 579
generic declarations, 622
key methods, 594–595
use of, 580–581
List<?> syntax, 620
List<Integer> syntax, 621
List<Object> syntax, 620–621
list.add( ) method, 620
listeners, 9
lists, enumerated. *See* enums
literals
boolean, 189
character, 189–190
floating-point, 188–189
integer
decimal, 186
hexadecimal, 187–188
octal, 187
string, 190, 434
live objects, 257
local arrays, 210
local object references, 209–210
local primitives, 207–209
local variables
access member modifiers and, 38–39
array references, 210
assigning one reference variable to another, 210–213
defined, 53–55
inner classes, 671
object references, 209–210
overview, 207
primitives, 207–209
scope of, 201
uninitialized, 203
Locale class, 482–485
locks, 735–737, 739–740
logical operators
bitwise, 305
boolean invert, 309–310
non-short-circuit, 308–309
short-circuit, 306–308
XOR, 309–310
Long class, 242, 807
loops
break statement, 352–353
continue statement, 352–353

loops (*Cont.*)
do, 344–345
for
conditional expression, 346–347
declaration, 346
enhanced, 350–352
initialization, 346
issues with, 348–350
iteration expression, 347–348
legacy collections, 598
overview, 345–346
PriorityQueue class, 592
threads, 714
initializing elements in, 225–226
labeled statements, 354–356
unlabeled statements, 353–354
wait( ) in, 753–757
while, 343–344
loose coupling, 152
lower( ) method, 587
lowerKey( ) method, 587
lower-level classes, 460
lower-priority threads, 724

M

Machine class, 754
main( ) method, 546, 665, 702, 710, 713
makeArrayList( ) method, 627
makeInner( ) method, 666
makeWithdrawal( ) method, 734
MANIFEST.MF file, 804
Map interface
collections, 563–564
key methods, 594–595
use of, 583–586
MapTest.main( ) method, 585
marker interfaces, 461, 562
Matcher class, 498–501
matching, exception, 369–371
MAX_VALUE constant, 807
meaningfully equivalent instances, 245–246
Meeks, Jonathan, 97
member classes, 662
member declarations
access modifiers
default members, 32–34, 36–38
local variables and, 38–39
overview, 24–26
private members, 29–32
protected members, 32–36
public members, 26–29
constructor, 47–48
enum
constructors, methods, and variables in, 63–65
overview, 60–63
nonaccess modifiers
abstract methods, 41–45
final arguments, 41
final methods, 40
methods with var-args, 46–47
native methods, 46
overview, 39
strictfp methods, 46
synchronized methods, 45
variable
array, 55–57
final, 57–58
instance, 51–53
local, 53–55
overview, 49
primitive, 49–51
reference, 51
static, 59–60
transient, 59
volatile, 59
visibility and access, 39
member methods, 662
member modifiers applied to inner classes, 670
member variables, 662. *See also* instance variables
memory
garbage collection, 257
management and garbage collection, 254–255
and String class, 433–434
metacharacters
and searches, 490–492
and strings, 497–498
META-INF directory, 804
method-local inner classes, 670–672
methods. *See also* hashCode( ) method; *individual methods by name*
backed collections, 593–595
chained, 442
comparison of modifiers on, 53
declarations, 41
declaring in enums, 63–65
defined, 2
effects of static on, 150
generic, 609–622, 627–630
in interfaces, 20–21
JavaBean signatures for, 10
naming standards, 7
overriding Object
equals( ), 544–549
toString( ), 542–544
overriding rules, 106–107
passing variables into
object reference, 213–214
overview, 213
pass-by-value semantics, 214–215
primitive, 215–218
String class, 434–438
StringBuffer, 440–442
StringBuilder, 440–442
synchronizing static, 737–739
thread scheduler, 717–718
with var-args, 46–47
mkdir( ) method, 453
modifiers. *See also individual modifiers by name*
applied to inner classes, 670
class access
default, 13–14
public, 14–15
class nonaccess
abstract, 16–19
final, 15–16
member access
default members, 32–34, 36–38
local variables and, 38–39
overview, 24–26
private members, 29–32
protected members, 32–36
public members, 26–29
member nonaccess
abstract methods, 41–45
final arguments, 41
final methods, 40
methods with var-args, 46–47
native methods, 46
overview, 39
strictfp methods, 46
synchronized methods, 45
overview, 12–13
variables and methods, 53

Moof class, 546
moofValue instance variable, 546–547
multidimensional arrays, 57, 223, 233–234
multiple inheritance, 100
multiplication (*) operator, 308
multithreading, 702, 713–716
myApp directory, 803
myApp.engine package, 804
myApp.utils package, 804
MyClass class, 738, 791
MyInner class, 666
MyOuter class, 666
MyOuter2 class, 670

N

name variable, 586
names.get(0) method, 743
names.remove(0) method, 744
names.size( ) method, 743–744
naming standards
  classes, 7
  interfaces, 7
  JavaBean listeners, 9
  JavaBean properties, 8–9
  methods, 7
  variables, 7
narrowing, 193
native methods, 46
natural order, 561
NavigableMap interface, 566
NavigableSet interface, 566
navigating
  files
    combining I/O classes, 449–452
    creating files using class File, 445–447
    directories, 452–457
    FileWriter and FileReader classes, 447–449
    overview, 443–445
  methods relating to, 588
  TreeSets and TreeMaps, 586–588
nested classes, static, 680–681. *See also* inner classes
new keyword, 222
new threads, 716, 718
next( ) method, 581
nextXxx( ) methods, 505–506
no-arg constructors, 133–134
NoClassDefFoundError exception, 382
nonabstract methods, 17–18
nonaccess class modifiers
  abstract classes, 16–19
  defined, 12
  final classes, 15–16
nonaccess member modifiers
  abstract methods, 41–45
  final arguments, 41
  final methods, 40
  methods with variable argument lists (var-args), 46–47
  native methods, 46
  overview, 39
  strictfp methods, 46
  synchronized methods, 45
non-final local variables, 682
non-generic collections, 597, 601–607
non-serializable elements, 472
non-short-circuit AND (&) operator, 288, 308
non-short-circuit logical operators, 308–309
non-short-circuit OR (|) operator, 288, 308
non-static inner class instantiation code, 667
non-static methods, 735, 740–741
non-static synchronized methods, 739, 741
nonstatic variables, 147–148
non-synchronized methods, 736–737
non-transient variables, 556
non-type safe code, 602–607
NoSuchElementException class, 743
not equal to (!=) operator, 292
notify( ) method, 542, 746, 748, 752–756
notifyAll( ) method, 542, 746, 752–757
null references, 205, 209, 224, 257–258, 563–564
NullPointerException class, 224, 247, 373, 380, 382
NumberFormat class, 484–487
NumberFormatException class, 381–382
numbers, formatting, 473–487
numeric primitives, 50

O

Object class. *See also* hashCode( ) method
  arrays, 579
  influencing thread scheduler, 718
  overriding methods
    equals( ), 544–549
    toString( ), 542–544
  overview, 542
object get( ) interface methods, 594
object graphs, 461–465
object next( ) method, 580
object orientation
  cohesion, 151–154
  constructors
    chaining, 132–133
    default, 135–139
    overloaded, 139–145
    overview, 130–132
    rules for, 133–134
  coupling, 151–153
  encapsulation, 86–89
  implementing interfaces, 120–125
  inheritance
    HAS-A relationship, 96–98
    IS-A relationship, 94–95
    overview, 90–94
  legal return types
    covariant returns, 127–128
    on overloaded methods, 126–127
    on overridden methods, 127–128
    returning values, 128–130
  overloaded methods
    invoking, 111–113
    legal, 110
    overview, 109–110
    polymorphism in, 113–115
  overridden methods
    examples of legal and illegal, 109
    invoking superclass version of, 107–108
    overview, 103–107
  polymorphism, 98–102
  reference variable casting, 116–119
  statics
    accessing, 148–151
    overview, 145–148

object reference instance variables, 203–206
object reference variables
  arrays of, 231–232
  assigning null to, 199
  passing into methods, 213–214
Object type, 572
Object[] toArray( ) interface method, 594
ObjectInputStream method, 460–461
object-oriented design, 97
ObjectOutputStream method, 460–461
objects
  arrays as, 297
  defined, 2
  uneligiblizing for garbage collection, 263
  wrapper, 239–240
octal literals, 187
offer( ) method, 562, 591–592
one-dimensional arrays, 221–222, 232–233
operands, 298
operators. *See also individual operators by name*
  arithmetic
    decrement, 302–303
    increment, 302–303
    overview, 298
    remainder, 299
    string concatenation, 299–301
  assignment
    compound, 197, 289–290
    overview, 288–290
  conditional, 304–305
  instanceof
    compiler error, 297–298
    overview, 295–297
  logical
    bitwise, 305
    boolean invert, 309–310
    non-short-circuit, 308–309
    short-circuit, 306–308
    XOR, 309–310
  precedence, 290
  relational
    equality operators, 292–295
    overview, 290–291
OR operator
  non-short-circuit, 288, 308
  short-circuit, 306, 308
ordered collections, 560
outer classes
  instantiating inner class from within, 666–668
  overview, 664
  referencing from within inner classes, 668–670
  static members of, 681
OutOfMemoryError subtype, 375
outside outer class instance code, 666–668
overloaded constructors, 139–145
overloading methods
  with boxing and var-args, 249–250
  in combination with var-args, 253–254
  invoking, 111–113
  legal, 110
  legal return types on, 126–127
  versus overridden methods, 115
  overview, 109–110, 247–249
  polymorphism in, 113–115
  when combining widening and boxing, 251–253
  widening reference variables, 250–251
overridden methods. *See also* hashCode( ) method
  invoking superclass version of, 107–108
  legal and illegal, 109
  legal return types on, 127–128
  Object
    equals( ), 544–549
    toString( ), 542–544
  versus overloaded methods, 115
  overview, 103–107
  polymorphism in, 113–115
  versus static methods, 151

P

packages, 3, 11–12, 799–801
parameterized type, 598, 626–628
parameters, 46
parentheses, 300
parse( ) method, 481
parseXxx( ) methods, 241
parsing
  locating data via pattern matching, 498–501
  overview, 487–488
  searches
    greedy quantifiers, 495–497
    overview, 488–489
    predefined dot, 495
    simple, 489–490
    strings and metacharacters, 497–498
    using metacharacters, 490–492
    using quantifiers, 492–495
Part class, 566
pass-by-value semantics, 214–215
passing variables into methods
  object reference variables, 213–214
  overview, 213
  pass-by-value semantics, 214–215
  primitive variables, 215–218
Pattern class, 498–501
pattern matching, 498–501
peek( ) method, 562, 591
poll( ) method, 562, 588, 591–592
pollFirst( ) method, 588
pollFirstEntry( ) method, 588
pollFirstXxx( ) method, 591
polling, 588
pollLast( ) method, 588
pollLastEntry( ) method, 588
polymorphism
  anonymous inner classes, 675
  defined, 92
  generics and, 607–610
  in overloaded methods, 113–115
  in overridden methods, 113–115
  overriden methods, 108
  overview, 98–102
pool, String constant, 434
populateList( ) method, 570
precedence, 299
precision format string element, 507
predefined dot, 495
preventing thread execution, 720–721
primitive arrays, 230–231
primitive instance variables, 203–204
primitive literals
  boolean, 189
  character, 189–190
  defined, 186
  floating-point, 188–189
  integer, 186–188
  string, 190

primitive return types, 128–129
primitive variables
assignments, 191–193
casting
assigning floating-point numbers, 196
assigning literals too large for, 196–197
assigning one primitive to another, 198
overview, 193–195
declaration, 49–51
defined, 49
equality operators for, 292–293
passing into methods, 215–218
ranges, 49–51
printf( ) method, 506–508
PrintWriter class, 444
priorities, thread, 724–728
PriorityQueue class
backed collections, 591–592
Collection interface, 565
methods, 594
overview, 564–566
private access, 32
private constructors, 133
private members, 29–32
private method-local inner class, 672
private methods, 11, 393
private variables, 733
programmatic exceptions
defined, 379
list of, 382
programmatically thrown exceptions, 380–381
propagating uncaught exceptions, 362–365
properties, system, 794–795
protected access, 37
protected members, 32–36
protected method-local inner class, 672
public access
classes, 14–15
effects of, 32
public boolean equalsIgnoreCase(String s) method, 436
public char charAt(int index) method, 435
public int length( ) method, 436
public interface, 371–376
public members, 26–29
public methods, 11, 392–393, 672
public modifier, 21
public static String toString( ) methods, 593
public static void sort( ) method, 593
public String concat(String s) method, 435–436
public String replace(char old, char new) method, 436
public String substring( ) methods, 436–437
public String toLowerCase( ) method, 437
public String toString( ) method, 438, 441–442
public String toUpperCase( ) method, 438
public String trim( ) method, 438
public StringBuilder delete(int start, int end) method, 440–441
public StringBuilder insert(int offset, String s) method, 441
public synchronized StringBuffer append(String s) method, 440
public synchronized StringBuffer reverse( ) method, 441
put(key, value) interface method, 594

Q

quantifiers
greedy, 495–497
and searches, 492–495
Queue interface, 557, 564–566, 594–595
queues, 562, 591–592, 717

R

race conditions, 733
RandomAccess interface, 562
ranges, primitive variable, 49–51
reachable references, 257
read( ) method, 746
readLine( ) method, 455, 457
readObject( ) method, 465–468
readPassword method, 457
reassigning reference variables, 258–259
Red-Black tree structure, 563
reference type, 113
reference variables
anonymous inner classes, 674, 676
assignments, 198–200
casting, 116–119
declaring, 51
defined, 49, 191
equality operators for, 293–294
isolating, 259–260
nulling, 257–258
polymorphism, 99
reassigning, 258–259
and string objects, 429
widening, 250–251
references, reachable, 257
referencing instances from within inner classes, 668–670
reflexiveness of equals( ) method, 549
regex (regular expressions)
engines, 488
Matcher class, 498–501
overview, 488, 510
Pattern class, 498–501
searches
dot metacharacter, 495
greedy quantifiers, 495–497
overview, 488–489
simple, 489–490
string and metacharacters, 497–498
using metacharacters, 490–492
using quantifiers, 492–495
tokenizing, 501–506
regions, 500
regular expressions (regex)
engines, 488
Matcher class, 498–501
overview, 488, 510
Pattern class, 498–501
searches
dot metacharacter, 495
greedy quantifiers, 495–497
overview, 488–489
simple, 489–490
string and metacharacters, 497–498
using metacharacters, 490–492
using quantifiers, 492–495
tokenizing, 501–506

regular inner classes, 664–668
relational operators
  equality
    for enums, 294–295
    overview, 292
    for primitives, 292–293
    for reference variables, 293–294
  overview, 290–291
relative paths, 801–802
remainder (%) operator, 299
removeFirst( ) method, 743–744
remove(index) interface method, 594
remove(key) interface method, 594
remove(object) interface method, 594
replace( ) method, 435–436
rethrowing exceptions, 376–378
return types, legal
  covariant returns, 127–128
  on overloaded methods, 126–127
  on overridden methods, 127–128
  returning values, 128–130
reverse( ) method, 441, 593
roll( ) method, 479
root directory, 804
rules for constructors, 133–134
run( ) method
  Thread class
    completion, 716
    instantiating, 708
    leaving running state, 728
    making thread, 704–705
    order of actions, 714
    overview, 710
  using wait( ) in loop, 755–757
Runnable interface, 678, 705–709
runnable threads, 716–719, 726
running multiple threads, 713–716
running threads, 719, 733
runtime
  disabling assertions at, 390
  enabling assertions at, 389
Runtime class, 261
RuntimeException class, 366–368, 373–374

S

s variable, 200
Scanner class, 501, 504–506
scheduler, thread, 716–718, 728
scope, variable, 200–202, 349
searching
  arrays, 576–578
  Java Archive files and, 803–805
  for other classes, 796–802
    classpaths, 797–798
    overview, 796–797
    packages, 799–801
    relative and absolute paths, 801–802
  simple searches, 489–490
  TreeSets and TreeMaps, 586–588
  using metacharacters, 490–492
  using quantifiers, 492–495
seeOuter( ) method, 670
selective enabling and disabling of assertions, 390–391
semantics, pass-by-value, 214–215
semicolons (;), 17, 41, 344–345, 675
serial numbers, 566
serialization
  and inheritance, 468–472
  object graphs, 461–465
  ObjectInputStream methods, 460–461
  ObjectOutputStream method, 460–461
  overview, 459–460
  readObject method, 465–468
  and statics, 472–473
  transient variables, 59
  writeObject method, 465–468
Set interface
  implementations for collections, 562–563
  key methods, 594–595
  use of, 581–583
Set keySet( ) interface method, 594
setParseIntegerOnly( ) method, 486
setPriority( ) method, 725
setProperty method, 795
setter methods, 8–9, 88
shadowing variables, 54–55, 201, 216–218
short-circuit AND (&&) operator, 306, 308, 548
short-circuit logical operators, 306–308
short-circuit OR (||) operator, 306, 308
shortcut syntax, array, 226–228
side effects, assertion, 394
signed number types, 49
simple searches, 489–490
size( ) method, 581, 743
sleep( ) method, 717, 721, 723, 727, 757
sleeping threads, 719–724
sort( ) method
  ArrayList class, 569
  Arrays class, 576, 593
  Collections class, 571–572, 574
SortedMap class, 591
SortedSet class, 591
sorting collections
  Arrays class, 576
  Comparable interface, 571–573
  Comparator interface, 574–575
  defined, 561
  overview, 569–571
source directories, 792
source file declaration rules, 11–12
-source option, javac command, 790
split( ) method, 502–504
spontaneous wakeup, 755
square brackets, 219–220
StackOverflowError subtype, 380, 382
standards, JavaBeans, 4, 8–10
start( ) method
  Matcher class, 499
  threads, 708–709, 716, 718, 757
starting threads
  overview, 709–712
  and running multiple, 713–716
  thread scheduler, 716–718
statements. *See also* decision statements
  assert, 386–389, 394
  break
    labeled, 354–356
    in switch blocks, 338–340
    switch statements, 335
    unlabeled, 353–354
    use of, 352–353
  continue
    labeled, 354–356
    overview, 352–353
    unlabeled, 353–354
  else, 329–332
  else if, 330–331
  import, 3–4, 11, 799–800, 805–806
  labeled, 354–356
  unlabeled, 353–354

states
defined, 2
thread
overview, 718–720
preventing execution, 720–721
priorities and yield( ), 724–728
sleeping, 721–724
static boolean equals( ) method, 593
static Comparator reverseOrder( ) method, 593
static compile( ) method, 499
static imports, 806–807
static initialization blocks, 234–235
static inner classes, 680
static int binarySearch( ) method, 593
static List asList( ) method, 593
static methods
accessing, 148–151
blocked threads, 720
defined, 59–60
inner classes, 667–668
local classes declared in, 672
overview, 145–151
sort( ), 576
synchronizing, 737–741
in Thread class, 756
Thread.sleep( ), 721
static modifier, 45, 59–60, 145, 682
static nested classes, 680–682
static synchronized methods, 739, 741
static Thread.currentThread( ) method, 711
static Thread.yield( ) method, 726
static variables
accessing, 148–151
defined, 59–60
overview, 145–151
scope of, 201
and serialization, 472–473
static void reverse( ) method, 593
static void sort( ) method, 593
stop( ) method, 720
Stream classes, 444
strictfp keyword, 15, 46
String class
creating new strings, 434
immutability, 426–433
important methods, 434–438
and memory, 433–434
overriding equals( ), 546
overview, 426
reference variables, 211–213
split( ) method, 502–504
toString( ) method, 543
string concatenation operator, 299–301
String constant pool, 434
StringBuffer class
important methods, 440–442
overview, 438–440
synchronizing code, 738–739
thread-safe, 742
StringBuilder class
important methods, 440–442
overview, 438–440
thread-safe, 742
StringIndexOutOfBoundsException subclass, 368
strings
collections, 566
generics, 596
literals, 190, 434
Maps, 584
and metacharacters, 497–498
overview, 426
reference variables, 211–213
String class
creating new, 434
immutability, 426–433
important methods, 434–438
and memory, 433–434
overview, 426
StringBuffer class
important methods, 440–442
overview, 438–440
StringBuilder class
important methods, 440–442
overview, 438–440
synchronizing code, 738
subclasses
and inheritance, 3, 25–26
inner classes, 677
versus interfaces, 122
protected and default members, 32–34
Thread class, 705
subMap( ) method, 589–590
subSet( ) method, 590
substring( ) method, 435–437
subtraction (-) operator, 306
subXxx( ) method, 591
Sun's Java Code Conventions, 4
super constructors, 136–139
super keyword, 619
superclasses, 3
suspend( ) method, 720
switch statements
break statement in, 335, 338–340
default case, 341–342
fall-through in, 338–340
legal expressions for, 335–338
overview, 328, 334–335
symmetry of equals( ) method, 549
synchronized methods
in code synchronization, 734–740
defined, 45
thread interaction, 751
Vector, 562
wait( ) in loops, 756
synchronizedList( ) method, 744
synchronizing code
atomic operations, 733–735
deadlock, 745–746
if thread can't get lock, 739–740
locks, 735–737
overview, 728–733
static methods, 737–739
thread-safe classes, 742–744
when synchronization is needed, 740–742
syntax
array shortcut, 226–228
generics, 598
system properties, 794–795
System.gc( ) method, 261
System.out.println( ) method, 504, 542, 630

T

T[ ] toArray(T[ ]) interface method, 594
tailMap( ) method, 590
tailSet( ) method, 590
tailXxx( ) method, 591
target Runnables, 707
ternary operators, 304
test expressions, 347
TestClass class, 800
testIt( ) method, 33
this reference, 29, 668

Thread class
extending, 705–706
final methods, 40
influencing thread scheduler, 717
instantiating thread, 706–708
methods, 704–705, 720
one-dimensional arrays, 222
run( ) method completion, 716
starting thread, 709–712
thread scheduler, 716–718, 728
threads
blocked, 719–720
dead, 716, 720
defining, 705–706
of execution, 702, 709, 757
instantiating, 706–708
interaction
notifyAll( ), 752–757
overview, 746–751
making, 704–705
new, 716, 718
overview, 701–704
runnable, 716–719, 726
running, 719, 733
sleeping, 719–724
starting
overview, 709–712
and running multiple, 713–716
thread scheduler, 716–718
states and transitions
overview, 718–720
preventing execution, 720–721
priorities, 724–728
sleeping, 721–724
yield( ) method, 726
synchronizing code
atomic operations, 733–735
deadlock, 745–746
if thread can't get lock, 739–740
locks, 735–737
overview, 728–733
static methods, 737–739
thread-safe classes, 742–744
when needed, 740–742
waiting, 719–720
thread-safe classes, 742–744
Thread.sleep( ) method, 721
throw keyword, 367
Throwable class, 366–368
thrown exceptions, 357
tight coupling, 152
toArray( ) method, 579, 581
tokenizing
and delimiters, 502
format( ) method, 506–508
overview, 501–502
printf( ) method, 506–508
Scanner class, 504–506
String.split( ) method, 502–504
toLowerCase( ) method, 437
toString( ) method
overriding, 542–544, 570
overriding equals( ), 548
public string, 438
in String class, 435
StringBuffer class, 441–442
wrapper conversion, 242
toUpperCase( ) method, 438
toXxxString( ) method, 242–243
transient method-local inner class, 672
transient modifier, 465
transient variables, 59, 460, 470, 555–556
transitions, thread
overview, 718–720
preventing execution, 720–721
priorities and yield( ), 724–728
sleeping, 721–724
yield( ), 724–728
transitivity of equals( ) method, 549
TreeMap class
backed collections, 589–590
collections, 557, 565–566
methods, 588, 590
overview, 564
searching, 586–588
subMap( ), 589
TreeSet class
backed collections, 590
Collection interface, 565
methods, 588, 590
overview, 563, 582
searching, 586–588
trim( ) method, 438
try keyword
exceptions, 357–359
with finally block, 359–361
wait( ) method, 750
without catch block, 361
without finally block, 361
t.start( ) method, 713
two-dimensional arrays, 223
type erasure, 604–605
type parameters, 598, 626–628
type safe code, 602–606

U

UML (Unified Modeling Language), 97
unboxing
equals( ) method, 245–246
equals operator, 245–246
overloading
with var-args and, 249–250
when combining widening and, 252–253
overview, 244–245
in switch statements, 337
use of, 246–247
uncaught exceptions, propagating, 362–365
unchecked exceptions, 373, 375
Unicode characters, 51, 189, 426–427
Unified Modeling Language (UML), 97
uninitialized variables
array instance, 206
object reference instance, 203–206
overview, 203
primitive instance, 203–204
unique hashcodes, 585
unique identifiers, 563
unlabeled statements, 353–354
unwinding stack, 367
upcasting, 118
use cases, date and number, 475

V

valueOf( ) methods, 240–241
values, constant, 23
var-args (variable argument lists)
declaration rules for, 47
methods with, 46–47
overloading, 249–250, 253–254
syntax, 796
variable scope, 200–202, 349

variables. *See also individual variables by name*
assignment operators
assigning one primitive to another, 198
floating-point numbers, 196
literals too large for, 196–197
overview, 190–191
primitive, 191–193
primitive casting, 193–195
reference, 198–200
scope, 200–202
comparison of modifiers on, 53
declarations
arrays, 55–57
in enums, 63–65
final, 57–58
generics and polymorphism, 608
instance, 51–53
local, 53–55
overview, 49
primitive, 49–51
reference, 51
static, 59–60
transient, 59
volatile, 59
effects of static on, 150
in interfaces, 20–21
local
access member modifiers and, 38–39
array references, 210
assigning one reference to another, 210–213
defined, 53–55
inner classes, 671
object references, 209–210
overview, 207
primitives, 207–209
scope of, 201
uninitialized, 203
naming standards, 7
passing into methods
object reference variables, 213–214
overview, 213
pass-by-value semantics, 214–215
primitive variables, 215–218
uninitialized
array instance, 206
object reference instance, 203–206
overview, 203
primitive instance, 203–204
Vector class, 562, 564–565
VirtualMachineError subtype, 375
void finalize( ) method, 543
void return types, 129
volatile variables, 59

W

wait( ) method
lock status, 740
in loops, 753–757
overview, 718
thread interaction, 746–752
threads leaving running state, 728
waiting threads, 719–720
wakeup, spontaneous, 755
warnings, compiler, 603
while loops, 343–344, 756
widening
overloading when combining boxing and, 251–253
reference variables, 250–251
wrapper classes, 251
width format string element, 507
wrappers
autoboxing
equals( ) method, 245–246
equals operator, 245–246
overview, 244–245
use of, 246–247
classes
overriding equals( ), 546
overview, 237–238
and widening, 251
constructors, 239
conversion methods, 243
conversion utilities
parseXxx( ) methods, 241
toString( ) method, 242
toXxxString( ) method, 242–243
valueOf( ) method, 241
xxxValue( ) methods, 240–241
NullPointerException, 247
valueOf( ) methods, 240
wrapping I/O classes, 449–452
writeObject( ) method, 465–468

X

x variable, 200
x2 variable, 200
x3 variable, 200
-Xlint:unchecked flag, 605
XOR (exclusive-OR) logical operator, 288, 309–310, 553
xxxValue( ) methods, 240–241

Y

y variable, 200
yield( ) method, 724–728

Z

z variable, 200

LICENSE AGREEMENT

THIS PRODUCT (THE "PRODUCT") CONTAINS PROPRIETARY SOFTWARE, DATA AND INFORMATION (INCLUDING DOCUMENTATION) OWNED BY THE McGRAW-HILL COMPANIES, INC. ("McGRAW-HILL") AND ITS LICENSORS. YOUR RIGHT TO USE THE PRODUCT IS GOVERNED BY THE TERMS AND CONDITIONS OF THIS AGREEMENT.

LICENSE: Throughout this License Agreement, "you" shall mean either the individual or the entity whose agent opens this package. You are granted a non-exclusive and non-transferable license to use the Product subject to the following terms:
(i) If you have licensed a single user version of the Product, the Product may only be used on a single computer (i.e., a single CPU). If you licensed and paid the fee applicable to a local area network or wide area network version of the Product, you are subject to the terms of the following subparagraph (ii).
(ii) If you have licensed a local area network version, you may use the Product on unlimited workstations located in one single building selected by you that is served by such local area network. If you have licensed a wide area network version, you may use the Product on unlimited workstations located in multiple buildings on the same site selected by you that is served by such wide area network; provided, however, that any building will not be considered located in the same site if it is more than five (5) miles away from any building included in such site. In addition, you may only use a local area or wide area network version of the Product on one single server. If you wish to use the Product on more than one server, you must obtain written authorization from McGraw-Hill and pay additional fees.
(iii) You may make one copy of the Product for back-up purposes only and you must maintain an accurate record as to the location of the back-up at all times.

COPYRIGHT; RESTRICTIONS ON USE AND TRANSFER: All rights (including copyright) in and to the Product are owned by McGraw-Hill and its licensors. You are the owner of the enclosed disc on which the Product is recorded. You may not use, copy, decompile, disassemble, reverse engineer, modify, reproduce, create derivative works, transmit, distribute, sublicense, store in a database or retrieval system of any kind, rent or transfer the Product, or any portion thereof, in any form or by any means (including electronically or otherwise) except as expressly provided for in this License Agreement. You must reproduce the copyright notices, trademark notices, legends and logos of McGraw-Hill and its licensors that appear on the Product on the back-up copy of the Product which you are permitted to make hereunder. All rights in the Product not expressly granted herein are reserved by McGraw-Hill and its licensors.

TERM: This License Agreement is effective until terminated. It will terminate if you fail to comply with any term or condition of this License Agreement. Upon termination, you are obligated to return to McGraw-Hill the Product together with all copies thereof and to purge all copies of the Product included in any and all servers and computer facilities.

DISCLAIMER OF WARRANTY: THE PRODUCT AND THE BACK-UP COPY ARE LICENSED "AS IS." McGRAW-HILL, ITS LICENSORS AND THE AUTHORS MAKE NO WARRANTIES, EXPRESS OR IMPLIED, AS TO THE RESULTS TO BE OBTAINED BY ANY PERSON OR ENTITY FROM USE OF THE PRODUCT, ANY INFORMATION OR DATA INCLUDED THEREIN AND/OR ANY TECHNICAL SUPPORT SERVICES PROVIDED HEREUNDER, IF ANY ("TECHNICAL SUPPORT SERVICES"). McGRAW-HILL, ITS LICENSORS AND THE AUTHORS MAKE NO EXPRESS OR IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE OR USE WITH RESPECT TO THE PRODUCT. McGRAW-HILL, ITS LICENSORS, AND THE AUTHORS MAKE NO GUARANTEE THAT YOU WILL PASS ANY CERTIFICATION EXAM WHATSOEVER BY USING THIS PRODUCT. NEITHER McGRAW-HILL, ANY OF ITS LICENSORS NOR THE AUTHORS WARRANT THAT THE FUNCTIONS CONTAINED IN THE PRODUCT WILL MEET YOUR REQUIREMENTS OR THAT THE OPERATION OF THE PRODUCT WILL BE UNINTERRUPTED OR ERROR FREE. YOU ASSUME THE ENTIRE RISK WITH RESPECT TO THE QUALITY AND PERFORMANCE OF THE PRODUCT.

LIMITED WARRANTY FOR DISC: To the original licensee only, McGraw-Hill warrants that the enclosed disc on which the Product is recorded is free from defects in materials and workmanship under normal use and service for a period of ninety (90) days from the date of purchase. In the event of a defect in the disc covered by the foregoing warranty, McGraw-Hill will replace the disc.

LIMITATION OF LIABILITY: NEITHER McGRAW-HILL, ITS LICENSORS NOR THE AUTHORS SHALL BE LIABLE FOR ANY INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES, SUCH AS BUT NOT LIMITED TO, LOSS OF ANTICIPATED PROFITS OR BENEFITS, RESULTING FROM THE USE OR INABILITY TO USE THE PRODUCT 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. Some states do not allow the exclusion or limitation of indirect, special or consequential damages, so the above limitation may not apply to you.

U.S. GOVERNMENT RESTRICTED RIGHTS: Any software included in the Product is provided with restricted rights subject to subparagraphs (c), (1) and (2) of the Commercial Computer Software-Restricted Rights clause at 48 C.F.R. 52.227-19. The terms of this Agreement applicable to the use of the data in the Product are those under which the data are generally made available to the general public by McGraw-Hill. Except as provided herein, no reproduction, use, or disclosure rights are granted with respect to the data included in the Product and no right to modify or create derivative works from any such data is hereby granted.

GENERAL: This License Agreement constitutes the entire agreement between the parties relating to the Product. The terms of any Purchase Order shall have no effect on the terms of this License Agreement. Failure of McGraw-Hill to insist at any time on strict compliance with this License Agreement shall not constitute a waiver of any rights under this License Agreement. This License Agreement shall be construed and governed in accordance with the laws of the State of New York. If any provision of this License Agreement is held to be contrary to law, that provision will be enforced to the maximum extent permissible and the remaining provisions will remain in full force and effect.