

ORACLE

250+ PRACTICE QUESTIONS

OCA Java SE 7 Programmer I Study Guide (Exam 1Z0-803)

Complete Exam Preparation

Edward Finegan
OCA Java SE 7 Programmer

Robert Liguori
OCA Java SE 7 Programmer

ORACLE
Certified Associate
Java SE 7 Programmer



ORIGINAL • AUTHENTIC
Oracle Press
ONLY FROM MCGRAW-HILL



*Oracle Press*TM

OCA Java SE 7 Programmer I Study Guide

(Exam IZO-803)



Oracle Press™

OCA Java SE 7 Programmer I Study Guide

(Exam IZO-803)

Edward Finegan
Robert Liguori

McGraw-Hill is an independent entity from Oracle Corporation. This publication and digital content may be used in assisting students to prepare for the Oracle Certified Associate Java™ SE 7 Programmer I exam. Neither Oracle Corporation nor The McGraw-Hill Companies warrant that use of this publication will ensure passing the relevant exam. Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.



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

Copyright © 2013 by The McGraw-Hill Companies. All rights reserved. Except as permitted under the United States Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher, with the exception that the program listings may be entered, stored, and executed in a computer system, but they may not be reproduced for publication.

ISBN: 978-0-07-178944-8

MHID: 0-07-178944-8

The material in this eBook also appears in the print version of this title: ISBN: 978-0-07-178942-4, MHID: 0-07-178942-1.

McGraw-Hill eBooks are available at special quantity discounts to use as premiums and sales promotions, or for use in corporate training programs. To contact a representative please e-mail us at bulksales@mcgraw-hill.com.

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

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

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

Sponsoring Editor

Timothy Green

Editorial Supervisor

Jody McKenzie

Project Manager

Tania Andrabi, Cenveo Publisher Services

Acquisitions Coordinator

Stephanie Evans

Technical Editor

Ryan Cuprak

Copy Editor

Lisa Theobald

Proofreader

Carol Shields

Indexer

Rebecca Plunkett

Production Supervisor

James Kussow

Composition

Cenveo Publisher Services

Illustration

Cenveo Publisher Services

Art Director, Cover

Jeff Weeks

Cover Design

Pattie Lee

Information has been obtained by McGraw-Hill from sources believed to be reliable. However, because of the possibility of human or mechanical error by our sources, McGraw-Hill, or others, McGraw-Hill does not guarantee the accuracy, adequacy, or completeness of any information and is not responsible for any errors or omissions or the results obtained from the use of such information.

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

TERMS OF USE

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

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

To Adalyn Irene, my beautiful and amazing new daughter, and my wife, Shannon. Without their love and support, this project would not have been possible.

—Edward G. Finegan

To Ashleigh, Patti, my family, and friends

—Robert J. Liguori



Edward Finegan is the founder of Dryrain Technologies. His company specializes in mobile software development supported by enterprise Java backends. He has previously worked in the casino gaming industry, where he designed and implemented software for gaming machines. He also has previous experience in air traffic management systems and radar protocols.

Finegan has a bachelor's degree in computer science from Rowan University and a master's degree in computer science from Virginia Commonwealth University. His thesis, entitled *Intelligent Autonomous Data Categorization*, examined the possibility of using machine-learning algorithms to categorize data intelligently and autonomously.

Finegan is an avid Philadelphia sports fan. He enjoys spending time with his family, especially with his wife, Shannon, and new daughter, Adalyn. He spends his free time partaking in outdoor activities and home-improvement projects, as well as tinkering with the latest technologies.

He can be contacted at edward@ocajexam.com.



Robert Liguori is a computer scientist and has been developing, maintaining, and testing air traffic management systems since 1996. He is currently providing web services and test assistance for air traffic management systems in support of the Federal Aviation Administration. Liguori is a software/test engineer with STG Technologies.

Liguori has a bachelor's degree in computer science and information technology from Richard Stockton College of New Jersey. He holds various Oracle certifications.

Liguori enjoys toying around with creative endeavors such as Capo Critters, unique little critters that sit on top of guitar capos.

In 2010 Liguori teamed up with Ryan Cuprak to produce the book, *NetBeans IDE Programmer Certified Expert Exam Guide (Exam 310-045)* (McGraw-Hill, 2010). The book targets preparation for the NetBeans IDE exam.

In 2008 Liguori worked closely with Edward Finegan to produce the predecessor to this OCA book, *SCJA Sun Certified Java Associate Study Guide (CX-310-019)* (McGraw-Hill, 2009). The book targets preparation for the SCJA exam.

In 2007 Liguori spent his free time with his wife, Patricia Liguori, co-authoring a handy Java

reference guide: *Java Pocket Guide* (O'Reilly Media Inc., 2008). The book captures Java's fundamentals in a companion-size book.

Liguori enjoys spending time with his family, as well as surf fishing for rockfish (striped bass), tautog, kingfish, bluefish, redfish, and flounder along the East Coast of the United States.

He can be contacted at robert@ocajexam.com.

About the Technical Editor



Ryan Cuprak is an e-Formulation Analyst at Enginuity PLM and president of the Connecticut Java Users Group, which he has run since 2003. At Enginuity PLM, he is focused on developing data integrations to convert client's data and also user interface development. Prior to joining Enginuity, he worked for a start-up distributed computing company TurboWorx and Eastman Kodak's Molecular Imaging Systems group, now part of Carestream Health. At TurboWorx he was a Java developer and also a technical sales engineer supporting both pre-sales and professional services. Cuprak has earned a bachelor of science degree in computer science and biology from Loyola University Chicago. He is a Sun Certified NetBeans IDE Specialist. He can be contacted at ryan@ocajexam.com.

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.

[1 Packaging, Compiling, and Interpreting Java Code](#)

[2 Programming with Java Statements](#)

[3 Programming with Java Operators and Strings](#)

[4 Working with Basic Classes and Variables](#)

[5 Understanding Methods and Variable Scope](#)

[6 Programming with Arrays](#)

[7 Understanding Class Inheritance](#)

[8 Understanding Polymorphism and Casts](#)

[9 Handling Exceptions](#)

[10 Working with Classes and Their Relationships](#)

[A Java Platforms](#)

[B Java SE 7 Packages](#)

[C Java Keywords](#)

[D Bracket Conventions](#)

[E Unicode Standard](#)

[F Pseudo-code Algorithms](#)

[G Unified Modeling Language](#)

[H Practice Exam](#)

[I About the Download](#)

[Glossary](#)

[Index](#)

[Acknowledgments](#)[Preface](#)[Introduction](#)

1 Packaging, Compiling, and Interpreting Java Code

[Understand Packages](#)[Package Design](#)[package and import Statements](#)[Exercise 1-1: Replacing Implicit import Statements with Explicit import Statements](#)[Understand Package-Derived Classes](#)[Java Utilities API](#)[Java Basic Input/Output API](#)[The Java Networking API](#)[Java Abstract Window Toolkit API](#)[Java Swing API](#)[Exercise I -2: Understanding Extended Functionality of the Java Utilities API](#)[Understand Class Structure](#)[Naming Conventions](#)[Separators and Other Java Source Symbols](#)[Java Class Structure](#)[Compile and Interpret Java Code](#)[Java Compiler](#)[Java Interpreter](#)[Exercise 1-3: Compiling and Interpreting Packaged Software](#) [Two-Minute Drill](#)[Q&A](#)[Self Test](#)[Self Test Answers](#)

2 Programming with Java Statements

[Understand Assignment Statements](#)[The Assignment Expression Statement](#)[Create and Use Conditional Statements](#)[The if Conditional Statement](#)[The if-then Conditional Statement](#)[The if-then-else Conditional Statement](#)[The switch Conditional Statement](#)[Exercise 2-1: Evaluating the String Class in the switch Statement](#)[Create and Use Iteration Statements](#)[The for Loop Iteration Statement](#)[The Enhanced for Loop Iteration Statement](#)[Exercise 2-2: Iterating Through an ArrayList While Applying Conditions](#)[The while Iteration Statement](#)

[The do-while Iteration Statement](#)

[Exercise 2-3: Performing Code Refactoring](#)

[Exercise 2-4: Knowing Your Statement-Related Keywords](#)

[Create and Use Transfer of Control Statements](#)

[The break Transfer of Control Statement](#)

[The continue Transfer of Control Statement](#)

[The return Transfer of Control Statement](#)

[The labeled Statement](#)

[Two-Minute Drill](#)

[Q&A](#) [Self Test](#)

[Self Test Answers](#)

3 Programming with Java Operators and Strings

[Understand Fundamental Operators](#)

[Assignment Operators](#)

[Exercise 3-1: Using Compound Assignment Operators](#)

[Arithmetic Operators](#)

[Relational Operators](#)

[Logical Operators](#)

[Understand Operator Precedence](#)

[Operator Precedence](#)

[Overriding Operator Precedence](#)

[Use String Objects and Their Methods](#)

[Strings](#)

[The String Concatenation Operator](#)

[Exercise 3-2: Uncovering Bugs that Your Compiler May Not Find](#)

[Methods of the String Class](#)

[Use StringBuilder Objects and Their Methods](#)

[Methods of the StringBuilder Class](#)

[Exercise 3-3: Using Constructors of the StringBuilder Class](#)

[Test Equality Between Strings and other Objects equals Method of the String Class](#)

[Exercise 3-4: Working with the compare To Method of the String Class](#)

[Two-Minute Drill](#)

[Q&A](#) [Self Test](#)

[Self Test Answers](#)

4 Working with Basic Classes and Variables

[Understand Primitives, Enumerations, and Objects](#)

[Primitive Variables](#)

[Objects](#)

[Exercise 4-1: Compile and Run an Object](#)

[Arrays](#)

[Enumerations](#)

[Java Is Strongly Typed](#)

[Naming Conventions](#)

[Use Primitives, Enumerations, and Objects](#)

[Literals](#)

[Examples of Primitives, Enumerations, and Objects](#)

[Exercise 4-2: Creating Getters and Setters.](#)

 [Two-Minute Drill](#)

[Q&A](#)

[Self Test](#)

[Self Test Answers](#)

[5 Understanding Methods and Variable Scope](#)

[Create and Use Methods](#)

[Using Method Syntax](#)

[Making and Calling a Method](#)

[Overloading a Method](#)

[Pass Objects by Reference and Value](#)

[Passing Primitives by Value to Methods](#)

[Passing Objects by Reference to Methods](#)

[Understand Variable Scope](#)

[Local Variables](#)

[Method Parameters](#)

[Instance Variables](#)

[An Object's Lifecycle](#)

[Create and Use Constructors](#)

[Making a Constructor](#)

[Overloading a Constructor](#)

[Using the Default Constructor](#)

[Use this and super Keywords](#)

[The this Keyword](#)

[The super Keyword](#)

[Create Static Methods and Instance Variables](#)

[Static Methods](#)

[Static Variables](#)

[Constants](#)

 [Two-Minute Drill](#)

[Q&A](#)

[Self Test](#)

[Self Test Answers](#)

[6 Programming with Arrays](#)

[Work with Java Arrays](#)

[One-Dimensional Arrays](#)

[Multi-Dimensional Arrays](#)

[Work with ArrayList Objects and Their Methods](#)

[Using the ArrayList Class](#)

[ArrayList vs. Standard Arrays](#)

[Exercise 6-1: Implement an ArrayList and Standard Array](#)

 [Two-Minute Drill](#)

[Q&A](#)

[Self Test](#)

[Self Test Answers](#)

[7 Understanding Class Inheritance](#)

Implement and Use Inheritance and Class Types

[Inheritance](#)

[Overriding Methods](#)

[Abstract Classes](#)

[Interfaces](#)

[Advanced Concepts of Inheritance](#)

Understand Encapsulation Principles

[Good Design with Encapsulation](#)

[Access Modifiers](#)

[Setters and Getters](#)

Advanced Use of Classes with Inheritance and Encapsulation

[Java Access Modifiers Example](#)

[Inheritance with Concrete Classes Examples](#)

[Inheritance with Abstract Classes Examples](#)

[**Exercise 7-1:** Add Functionality to the Plant Simulator](#)

[Interface Example](#)

[Two-Minute Drill](#)

Q&A

[Self Test](#)

[Self Test Answers](#)

8 Understanding Polymorphism and Casts

Understand Polymorphism

[Concepts of Polymorphism](#)

[Practical Examples of Polymorphism](#)

[**Exercise 8-1:** Add Functionality to the Describable Example](#)

Understand Casting

[When Casting Is Needed](#)

[Two-Minute Drill](#)

Q&A

[Self Test](#)

[Self Test Answers](#)

9 Handling Exceptions

Understand the Rationale and Types of Exceptions

[Exception Hierarchy in Java](#)

[Checked Exceptions](#)

[Unchecked Exceptions](#)

[\(Unchecked\) Errors](#)

[**Exercise 9-1:** Determining When to Use Assertions in Place of Exceptions](#)

[**Exercise 9-2:** Analyzing the Source Code of Java Exceptions](#)

Understand the Nature of Exceptions

[Defining Exceptions](#)

[Throwing Exceptions](#)

[Propagating Exceptions](#)

[**Exercise 9-3:** Creating a Custom Exception Class](#)

Alter the Program Flow

[The try-catch Statement](#)

[The try-finally Statement](#)

[The try-catch-finally Statement](#)
[The try-with-resources Statement](#)
[The multi-catch Clause](#)

[**Exercise 9-4:** Using NetBeans Code Templates for Exception Handling Elements](#)

[Recognize Common Exceptions](#)

[Common Checked Exceptions](#)
[Common Unchecked Exceptions](#)
[Common Errors](#)

[**Exercise 9-5:** Creating an Error Condition](#)

 [Two-Minute Drill](#)

[Q&A](#) [Self Test](#)

[Self Test Answers](#)

10 Working with Classes and Their Relationships

[Understand Class Compositions and Associations](#)

[Class Compositions and Associations](#)
[Class Relationships](#)
[Multiplicities](#)
[Association Navigation](#)

[Class Compositions and Associations in Practice](#)

[Examples of Class Association Relationships](#)
[Examples of Class Composition Relationships](#)
[Examples of Association Navigation](#)

 [Two-Minute Drill](#)

[Q&A](#) [Self Test](#)

[Self Test Answers](#)

A Java Platforms

[Java Platform, Standard Edition](#)

[The Java SE API](#)
[The Java Runtime Environment](#)
[The Java Development Kit](#)
[Supported Operating Systems](#)

[Java 2 Platform, Micro Edition](#)

[Configurations](#)
[Profiles](#)
[Optional Packages](#)
[Squawk](#)

[Java Platform, Enterprise Edition](#)

B Java SE 7 Packages

[Core Packages](#)
[Integration Packages](#)
[User Interface Packages](#)
[Security Packages](#)
[XML-based Packages](#)

[C Java Keywords](#)

[D Bracket Conventions](#)

[Java Bracket Conventions](#)
[Miscellaneous Bracket Conventions](#)

[E Unicode Standard](#)

[F Pseudo-code Algorithms](#)

[Implementing Statement-Related Algorithms from Pseudo-code](#)
[Pseudo-code Algorithms](#)
[Pseudo-code Algorithms and Java](#)

[G Unified Modeling Language](#)

[Recognizing Representations of Significant UML Elements](#)
[Classes, Abstract Classes, and Interface Diagrams](#)
[Attributes and Operations](#)
[Visibility Modifiers](#)
[Recognizing Representations of UML Associations](#)
[Graphic Paths](#)
[Relationship Specifiers](#)

[H Practice Exam](#)

[Answers](#)

[I About the Download](#)

[Steps to Download MasterExam](#)
[System Requirements](#)
[MasterExam](#)
[Enterprise Architect Project File](#)
[Help](#)
[Removing Installation](#)
[Technical Support](#)
[LearnKey Technical Support](#)

[Glossary](#)

[Index](#)

The OCA exam covers a significant amount of information detailing areas from Java fundamentals to object-oriented concepts. To complete a voluminous project like this book, covering all of the related objectives, the authors decided to take the divide-and-conquer approach by splitting up the chapters based on their individual expertise. Finegan focused on the chapters covering object-oriented basic concepts. Liguori focused on the core Java fundamentals chapters.

OCA Java SE 7 Programmer Study Guide Project Team Acknowledgments

The McGraw-Hill and Support Team: Timothy Green, Stephanie Evans, Jody McKenzie, Jim Kussow, Melinda Lytle, Tania Andrabi, Vasundhara Sawhney, Lisa Theobald, Carol Shields, and Rebecca Plunkett

Waterside Productions, Inc.: Carole Jelen McClendon

Technical Editor: Ryan Cuprak

Informal Reviewers: Shannon Reilly Finegan, Richard Tkatch, Wayne Smith (SCJA version)

Personal Acknowledgments

Thank you to all of my family and friends. A project like this has a way of consuming more time than you ever expect. Their patience and encouragement has kept me motivated to accomplish this venture. I would like to give a special thanks to my wife, Shannon. She gave birth to our first child, Adalyn, in the middle of this project and helped me balance the time between family and my professional endeavors. Without her support, this book would have never been completed.



I would also like to express my gratitude to my co-author, Robert Liguori. It has been an enriching experience collaborating with him on this project. Robert is a talented professional, and I sincerely thank him for keeping me on track. His enthusiasm and dedication to the Java community is truly remarkable and is evident in his work.

—Edward G. Finegan

Thank you, Patti, my dear wife and friend. I would also like to acknowledge my parents, family, and friends for always being there for me. I also extend a special call out to my daughter, Ashleigh Liguori, and my niece, Ryley Wilson, as they have recently tried to find me under a mountain of books, articles, papers, laptops, and the like...all at the dining room table. I would also like to thank

my brother Michael for painting this pleasant picture below that I have set as my desktop background throughout the development of the book.



Finally, I would sincerely like to thank my co-author Edward Finegan, who took over as lead author for the update of this book. As such, I sincerely feel that the readers of this book will greatly benefit from the efforts Ed and I put into making the book as complete and effective as possible. So, dear readers, good luck on your exam and thank you for making this book your test preparation resource.

—Robert J. Liguori

The purpose of this study guide is to prepare you for the OCA Java SE 7 Programmer I (IZO-803) exam to earn your Oracle Certified Associate, Java SE 7 Programmer (OCA Java SE 7 Programmer) certification. This preparation will be accomplished by familiarizing you with the necessary knowledge related to Java fundamentals, concepts, tools, and technologies that will be represented on the exam. In short, the scope of this book is to help you pass the exam. As such, objective-specific areas are detailed throughout this book. Peripheral information, which is not needed to pass the exam, may not be included, or it may be presented in a limited fashion. Since this book covers a lot of information on the fundamentals of Java and related technologies, you may also want to use it as a general reference guide away from the certification process.

Achieving the OCA Java SE 7 Programmer certification will solidify your knowledge of the Java programming language, set the foundation for your evolution through the related technologies, and identify you as a true Java professional. We strongly recommend the OCA Java SE 7 Programmer certification to matured programmers and software developers.

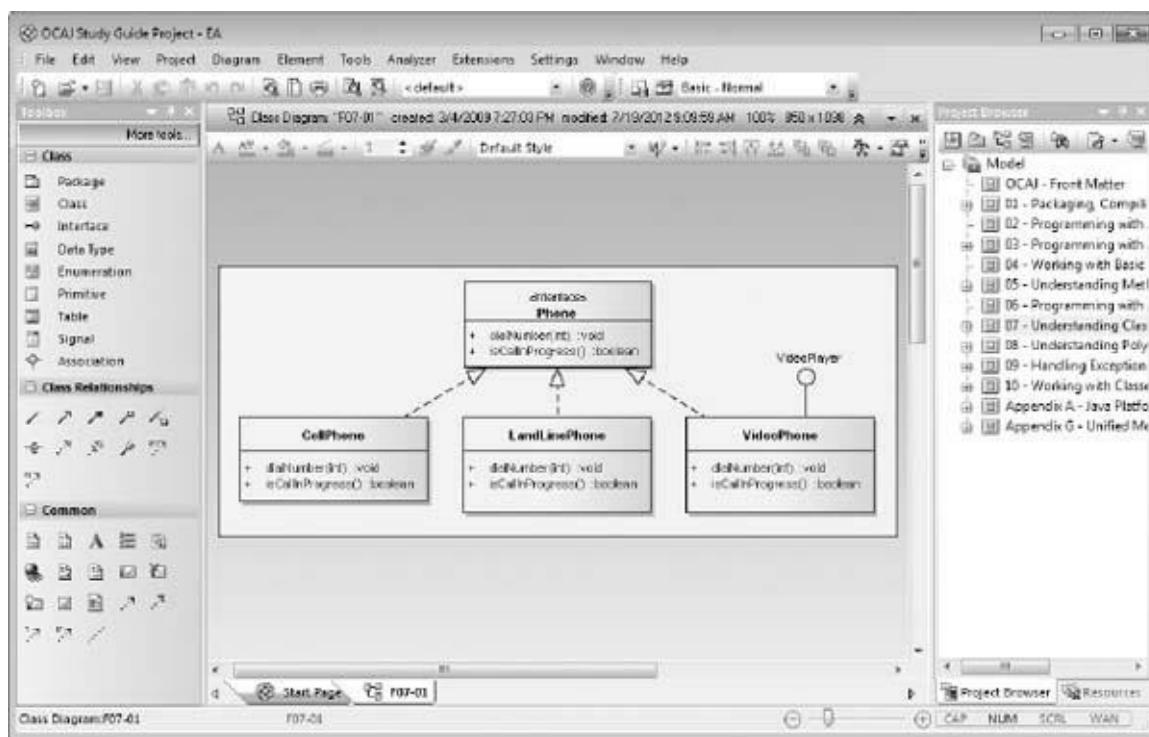
The Oracle certification series for Java has various exams for both Java SE and Java EE. This study guide focuses on the first step of the Java SE-related exams.

Passing the Java SE 7 Programmer I (IZO-803) exam allows you to achieve the Oracle Certified Associated, Java SE 7 Programmer Certification. Once you've achieved this, you can take the Java SE 7 Programmer II (IZO-804) to earn the Oracle Certified Professional, Java SE 7 Certification. If you already have prior Java certification, you may opt for the Upgrade to Java SE 7 Programmer (IZO-805) exam to go directly for the Oracle Certified Professional, Java SE 7 certification.

In This Book

This book covers Java fundamentals including development tools, basic constructs, operators, and strings. This book additionally covers object-oriented concepts including classes, class relationships, and object-oriented principles. A set of appendices covers Java keywords, bracket conventions, the Unicode standard, pseudo-code algorithms, Java SE Packages, and Unified Modeling Language (UML). A useful glossary is also provided. Enjoy.

FIGURE 1 Enterprise Architect CASE tool



Electronic Content

Available for download from the McGraw-Hill Media Center are a practice exam, select source code represented in the book, and the Enterprise Architect project file containing the UML diagrams that were rendered and used as draft images for the book (shown in [Figure 1](#)). For more information, please see [Appendix I](#), “About the Download,” at the back of the book.

Exam Readiness Checklist

At the end of the “Introduction,” you will find an “Exam Readiness Checklist.” This table has been constructed to allow you to cross-reference the official exam objectives with the objectives as they are presented and covered in this book. The checklist also lets you gauge your level of expertise on each objective at the outset of your studies. This should allow you to check your progress and make sure you spend the time you need on more difficult or unfamiliar sections. References have been provided for the objective exactly as the vendor presents it, the section of the study guide that covers that objective, and a chapter and page reference.

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 **Certification Objectives**—what you need to know to pass the section on the exam dealing with the chapter topic. The 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. These helpful hints are written by authors who have taken the exams and received their certification. Who better to tell you what to worry about? They know what you’re about to go through!

■ **Step-by-Step Exercises** are interspersed throughout the chapters. These are typically designed as hands-on exercises that allow you to get a feel for the real-world experience you'll need to pass the exams. 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.



■ **On the Job** notes describe the issues that come up most often in real-world settings. They provide a valuable perspective on certification- and product-related topics. They point out common mistakes and address questions that have arisen from on-the-job discussions and experience.

■ **Inside the Exam** sidebars highlight some of the most common and confusing problems that students encounter when taking a live exam. Designed to anticipate what the exam will emphasize, getting inside the exam will help ensure you know what you need to know to pass the exam. You can get a leg up on how to respond to those difficult-to-understand questions by focusing extra attention on these sidebars.

■ **Scenario & Solution** sections lay out potential problems and solutions in a quick-to-read format.

| SCENARIO & SOLUTION | |
|--|---|
| You want to use an AND operator that evaluates the second operand whether the first operand evaluates to true or false. Which would you use? | Boolean AND (<code>&</code>) |
| You want to use an OR operator that evaluates the second operand whether the first operand evaluates to true or false. Which would you use? | Boolean OR (<code> </code>) |
| You want to use an AND operator that evaluates the second operand only when the first operand evaluates to true. Which would you use? | Logical AND (<code>&&</code>) |
| You want to use an OR operator that evaluates the second operand only when the first operand evaluates to false. Which would you use? | Logical OR (<code> </code>) |

■ The **Certification Summary** is a succinct review of the chapter and a restatement of salient points regarding the exam.

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

Q&A

■ The **Self Test** offers questions similar to those found on the certification exams. The answers to these questions, as well as explanations of the answers, can be found at the end of each 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.

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:

■ *Reread 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.

- *Reread all the Exam Watch notes and Inside the Exam elements.* Remember that these notes are written by authors who have taken the exam and passed. They know what you should expect—and what you should be on the lookout for.
- *Review all the Scenario & Solution sections* for quick problem-solving.
- *Retake 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 consider all the questions in the book in one sitting. Pretend that you're taking the live exam. When you go through the questions the first time, you should mark your answers on a separate piece of paper. That way, you can run through the questions as many times as you need until you feel comfortable with the material.
- *Complete the exercises.* Did you do the exercises when you read through each chapter? If not, do them! These 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 you are not clear on a particular topic, reread that section in the chapter.

This OCA Java SE 7 Associate Study Guide has been designed to assist you in preparation of passing the Oracle's Java SE 7 Programmer I exam to achieve the OCA Java SE 7 Programmer certification. The information in this book is presented through textual content, coding examples, exercises, and more. To the best extent possible, all code examples have been validated on Macintosh OS X and Windows computers. Information is covered in detail for all exam objectives. The main areas covered in this book are as listed:

- The Java SE platform
- Java development and support tools
- Java fundamentals including statements, variables, method primitives, and operators
- String and StringBuilder class methods and functionality
- Basic Java elements including primitives, arrays, enumerations, and objects
- Classes and interfaces including class relationships
- Object-oriented principles
- Exception handling

Various appendices are included as well to assist you with your studies.

Specifics About the OCA Java SE 7 Programmer Certification Exam

Specifics of the OCA Java SE 7 Programmer exam objectives are detailed on Oracle Certification's web site at http://education.oracle.com/pls/web_prod-plq-dad/db_pages.getpage?page_id=41&p_org_id=1001&lang=US&p_exam_id=1Z0_803. Specifics about the exam registration process are supplied by Pearson VUE when you enroll for the exam. However, we do detail in the following sections the most important information you will need to know to enroll for and take the exam. Please visit the Oracle Certification web site for the most current information on exam objectives.

Referring to the Exam

The formal name for this exam is the Java SE 7 Programmer I (IZO-803) exam. The certification you will receive, if you pass the exam, is the Oracle Certified Associate, Java SE 7 Programmer certification.

Googling online, you will see that people and resources are calling out this IZO-803 exam as the OCA exam, the OCAJ exam, the OCAJP exam, and the OCAJP7. This would correlate to the OCA certification, OCAJ certification, OCAJP certification, and OCAJP7 certification.

For simplicity, we have made the best effort to refer to the exam as the OCA *exam* and the certification as the OCA *certification*.

Dynamics of the Java SE 7 Programmer I (IZO-803) Exam

This exam is geared toward matured Java programmers and developers who want to achieve foundational Java certification. The prerequisite-free exam consists of 90 questions. A passing percentage of 75 percent is necessary—in other words, at least 68 of 90 questions must be answered correctly. The designated time limit is 150 minutes (that is, 2 hours and 30 minutes).

The current U.S. price to take the exam is \$300. If you are employed by a technical organization, you can check to see if your company has an educational assistance policy.

Dynamics of the Java Standard Edition 5 and 6, Certified Associate (1Z0-850) Exam

The legacy SCJA (CX-310-019) exam has been updated to the Java Standard Edition 5 and 6, Certified Associate Exam which will achieve the passing candidate the Oracle Certified Associate, Java SE 5/SE 6 certification. The new exam number is 1Z0-850 and is targeted to the following certification candidates:

- Entry-level and junior programmers wanting to start and/or continue down the path of using Java technologies
- Software developers and technical leads wanting to solidify their Java-related skillsets
- Project and program managers wanting to gain a true perspective of the work and challenge of their teams, complementary to their planning and direction
- Computer science and information system students wanting to complement their education
- IT job seekers looking for a well-deserved edge
- Certification seekers wanting to round out their resume or curriculum vitae

If you wish to target this Java SE 5/6 version of the exam versus the current Java SE 7 version, the predecessor to this book, *SCJA Sun Certified Java Associate Study Guide (CX-310-019)* (McGraw-Hill, 2009), is an excellent resource to help you achieve your certification.

Scheduling the OCA Exam

The exam must be given at a proctored Pearson VUE testing location. You may schedule your appointment for the exam via one of three methods:

- You may schedule online.
- You may schedule by phone.
- You may schedule your exam through the test center.

You will find all of the information you need for these above options at the Pearson VUE web site (www.pearsonvue.com/oracle/). As a quick look, [Figure 2](#) shows the scheduling process on the Pearson VUE web site.

Pearson VUE Test Vouchers

Test Vouchers (prepaid exam certificates) may be available for purchase for the OCAJP7 exam. For more information about purchasing test vouchers, visit the voucher information page on the Pearson VUE web site at www1.pearsonvue.com/vouchers/, or visit the Pearson VUE voucher store at www1.pearsonvue.com/contact/voucherstore/.

Pearson VUE Test Center Locator

If you are interested in finding a testing facility near you, without registering, you can use Pearson VUE's Test Center Locator at www.vue.com/vtclocator/.

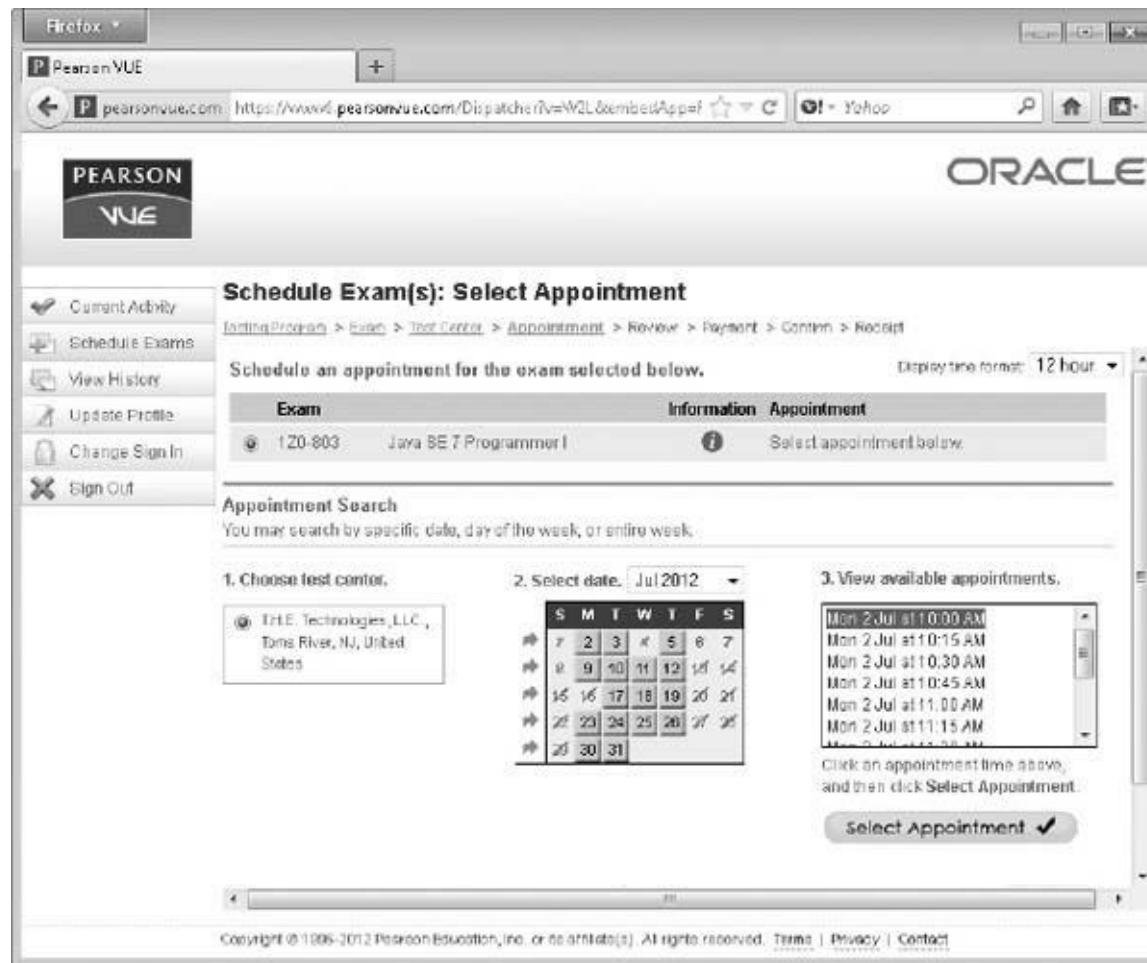
Preparing for the OCA Exam

Getting a good night's rest before the exam and eating a healthy breakfast will contribute to good test marks, but you should already know that. Don't cram for the exam the night before. If you find you

need to cram, you should reschedule the exam since you are not ready for it.

You will need to bring a few things with you for the exam, *and* you'll be leaving a few things behind. Let's take a look at some do's and don'ts.

FIGURE 2 Scheduling an exam



Do's

- **Do bring (at least) two forms of identification.** Valid identification includes a valid passport, current driver's license, government-issued identification, credit cards, or check-cashing cards. Two items must include your signature and at least one item must include your photograph. Make sure you don't shorthand your signature when you sign in since it must match your identification.
- **Do show up early.** Plan to arrive 15 minutes to a half hour before your exam's scheduled start time. You may find that the center needs to set up several people for exams. Getting in early may get you set up early, or at the least it will ensure that you don't start late.
- **Do use the restroom ahead of time.** The exam will take close to 2 hours to complete. Breaks are frowned upon and are most likely disallowed. In addition, you can't pause the time you have allocated if you take a break.
- **Do print out directions to the test facility.** Or get the address and punch it into your GPS navigator or directions app if you have one.

Don'ts

- **Don't bring your laptop, tablets, phone, or pager into the exam.** In addition, some facilities may ask that you do not enter the test area with your watch or wallet.

- *Don't bring books, notes, or writing supplies.* You may be given an erasable noteboard to use. Do not use the noteboard until the exam has started.
- *Don't bring large items.* Large storage may not be available for book bags and jackets. However, the testing facility may have small securable lockers for your usage. Before storing your cell phone or other electronic device, turn it off.
- *Don't bring drinks or snacks to the exam.* But feel free to enjoy refreshments while you are studying for the exam.
- *Don't study in the test center.*

Taking the OCA Exam

Just prior to taking the exam, the proctor may take your photograph. Bring your best smile. You may be asked to sign in and wait while the proctor sets up your exam on a PC. In my case, the proctor adjusted the web cam in the room so it was facing me. There may be other people in the room taking other tests, so bring ear plugs if you are sensitive to the mouse clicks and general noises or people around you. The proctor should make sure that you get to the first question, and then he or she will leave room. Good luck!

After the exam, you may be presented with an optional computer-based survey. This survey will ask you about your technical background and other related information. A common misconception is that the answering of the questions may be related to which exam questions you will be presented with; however, the survey is not related to the exam questions you will receive. The survey can take a few minutes to complete. The information gathered is important for those developing and refining future exams, so answer the questions honestly.

After you have completed the exam, your results will appear on the screen and will also be printed. Find the proctor (testing personnel) to retrieve your results from the printer and sign out. The point here is that *you should not leave* once you have completed the exam; stay and get your results.

Sharing Your Success

We would like to know how you did. You can send us an e-mail at results@ocajexam.com, or you can post your results on Java Ranch's Wall of Fame, <http://faq.javaranch.com/java/Ocjp7WallOfFame>.

Rescheduling the OCA Exam

If you need to reschedule (or cancel) your exam, do so outside of 24 hours of the start of the exam. Use Pearson VUE's Test Taker Services page (www.vue.com/programs/) to assist with the rescheduling or contact Pearson VUE directly. Rescheduling within 24 hours before the scheduled exam is subject to a same-day forfeit exam fee. Refunds are not granted if you don't show up for the exam (that is, if you're a no-show).

Additional OCA Resources

Numerous resources can supplement this book in assisting you with your goal of OCA certification. These resources include Java software utilities, Java community forums, language specifications and related documentation, OCA-related books, online and purchasable mock exams, and software tools (such as IDE's CASE tools, UML modeling tools, and so on). Although these peripheral tools and resources are highly beneficial and recommended, they are optional in regards to passing the OCA exam; this book attempts to cover all of the necessary material.

The following sections detail the previously mentioned resources.

Java Software

- Java Development Kits, www.oracle.com/technetwork/java/archive-139210.html
- Java Enterprise Editions (out of scope of exam, provided here just FYI),
www.oracle.com/technetwork/java/javaee/overview/index.html

Java Online Community Forums

- Oracle's technology forums, <https://forums.oracle.com/forums>
- Java Ranch's Big Moose Saloon Java technology forums, www.coderanch.com/forums.
- Java Programming forums, <http://javaprogrammingforums.com/>
- </dream.in.code>, www.dreamincode.net/forums/forum/32-java/
- IBM - Java Technology Forums, www.ibm.com/developerworks/forums/dw_jforums.jspa
- Tek Tips Java Forum, www.tek-tips.com/threadminder.cfm?pid=269
- Code Guru - Java Programming, <http://forums.codeguru.com/forumdisplay.php?f=67>
- Go4Expert, www.go4expert.com/forums/forumdisplay.php?f=21
- //javareference, www.javareference.com/
- Java User Groups, <http://java.sun.com/community/usergroups/>, <http://home.java.net/jugs/java-user-groups>

Java Tools and Technologies Specifications and Documentation

- The Java Tutorials, <http://docs.oracle.com/javase/tutorial/>
- Java Platform Standard Edition 7 Documentation, <http://docs.oracle.com/javase/7/docs/>
- Java Platform, Standard Edition 7 API Specification, <http://docs.oracle.com/javase/7/docs/api/>
- jDocs Java documentation repository, www.jdocs.com/
- UML specification, www.omg.org/spec/UML/Current/
- *The Java Language Specification: Java SE 7 Edition*,
<http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf>
- *The Java Language Specification, Third Edition*, <http://java.sun.com/docs/books/jls/>

Books Covering Material Found on the OCA Exam

Although the book you are holding sufficiently covers everything you need to know to pass the exam, supplemental reading can only help. Consider reviewing the following books to refine your skills:

- *Java Pocket Guide*, by Robert and Patricia Liguori (O'Reilly Media Inc., 2008)
- *NetBeans IDE Programmer Certified Expert Exam Guide (Exam-310-045)*, by Robert Liguori and Ryan Cuprak (McGraw-Hill, July 2010)
- *Java: The Complete Reference, Eighth Edition*, by Herbert Schildt (McGraw-Hill, June 2011)
- *UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd Edition)*, by Martin Fowler (Addison-Wesley, 2003)
- *SCJA Sun Certified Java Associate Study Guide for Test CX-310-019, 2nd Edition*, by Cameron W. McKenzie (Pulpjava, 2007)
- *SCJA Sun Certified Java Associate Mock Exam Questions*, by Cameron W. McKenzie (PulpJava, 2007)

OCA Mock Exams

In addition to the online mock exams associated with this book, various other free and commercial OCA and legacy SCJA mock exams exist. Various resources are listed here.

- Oracle Practice Exams, http://education.oracle.com/pls/web_prod-plq-dad/db_pages.getpage?page_id=208
- uCertify PrepKit and downloadable mock exam questions, www.ucertify.com/exams/Oracle/CX310-019.html
- Whizlabs SCJA Preparation Kit, www.whizlabs.com/scja/scja.html
- eJavaguru.com's online mock exam, www.ejavaguru.com/scjafreemockexam.php
- SCJA.de e-book and online mock exam questions, <http://scja.de/>
- ExamsExpert, www.examsexpert.com/oracle-certifications.html
- Transcender, www.selftestsoftware.com/certprep-materials/oracle.kap
- Self Test Software, www.transcender.com/

Integrated Development Environments

An *Integrated Development Environment (IDE)* is a development suite that allows developers to edit, compile, debug, connect to version control systems, collaborate, and do much more depending on the specific tool. Most modern IDEs have add-in capabilities for various software modules to enhance the IDE's capabilities. There is no reason not to use an IDE. We (the authors) recommend that you use the NetBeans IDE for your test preparation, because of its popularity, ease-of-use, and support from Oracle. However, any IDE from the following list would suffice:

- NetBeans IDE, www.netbeans.org
- Oracle JDeveloper IDE, www.oracle.com/technetwork/developer-tools/jdev/overview/
- IntelliJ IDEA, www.jetbrains.com
- Eclipse IDE, www.eclipse.org
- JCreator IDE, www.jcreator.com
- BlueJ, www.bluej.org

Tools with UML Modeling Features

Several tools and IDEs have UML features. Note that the predecessor exam to the OCA included questions on UML modeling. However, questions about UML are not included on the OCA. If you would like to explore UML modeling, following are a few tools you may look into—but, again, this is not on the exam. We do believe, however, that software programmers and developers should learn UML early in their careers.

- NetBeans IDE, <http://netbeans.org/features/uml/>
- JDeveloper IDE, www.oracle.com/technetwork/developer-tools/jdev/jdeveloper11g-datasheet-1-133040.pdf
- Enterprise Architect CASE tool, www.sparxsystems.com/products/ea/
- Visual Paradigm for UML (provides plug-ins for the popular IDEs), www.visual-paradigm.com/product/vpuml/

Miscellaneous Resources

Various other resources, such as the following, include games and Java news outlets that can assist you with getting high marks on the exam.

- Java Ranch Rules Round Up Game, www.javaranch.com/game/game2.jsp

No Yes

- DZone, <http://java.dzone.com>
- The Server Side, <http://theserverside.com>
- OCA FAQs on Java Ranch, www.coderanch.com/how-to/java/OcajpFaq
- The Java Tutorial (Programmer Level I Exam),
<http://docs.oracle.com/javase/tutorial/extra/certification/javase-7-programmer1.html>
- Java Language Specifications, <http://docs.oracle.com/javase/specs/>

Oracle's Certification Program in Java Technology

This section maps the exam's objectives to specific coverage in the study guide.

Exam 1Z0-803

| Exam Readiness Checklist | | | | |
|---|---------------------------------------|------|------|--------------|
| Official Objective | Study Guide Coverage | Ch # | Pg # | Beginner |
| | | | | Intermediate |
| 1.4 Import other Java packages to make them accessible in your code | Understand Packages | 1 | 2 | |
| Supplemental Coverage | Understand Package-Derived Classes | 1 | 10 | |
| 1.2 Define the structure of a Java class | Understand Class Structure | 1 | 19 | |
| 1.3 Create executable Java applications with a main method | Compile and Interpret Java Code | 1 | 23 | |
| Supplemental Coverage | Understand Assignment Statements | 2 | 50 | |
| 3.4 Create if and if/else constructs | Create and Use Conditional Statements | 2 | 52 | |
| 3.5 Use a switch statement | Create and Use Conditional Statements | 2 | 52 | |
| 5.2 Create and use for loops including the enhanced for loop | Create and Use Iteration Statements | 2 | 61 | |

Exam Readiness Checklist

| Official Objective | Study Guide Coverage | Ch # | Pg # | Beginner | Intermediate | Expert |
|--|--|------|------|----------|--------------|--------|
| 5.1 Create and use while loops | Create and Use Iteration Statements | 2 | 61 | | | |
| 5.3 Create and use do/while loops | Create and Use Iteration Statements | 2 | 61 | | | |
| 5.4 Compare loop constructs | Create and Use Iteration Statements | 2 | 61 | | | |
| 5.5 Use break and continue | Create and Use Transfer of Control Statements | 2 | 69 | | | |
| 3.1 Use Java operators | Understand Fundamental Operators | 3 | 86 | | | |
| 3.2 Overriding operator precedence | Understand Operator Precedence | 3 | 98 | | | |
| 2.7 Create and manipulate strings | Use String Objects and Their Methods | 3 | 100 | | | |
| 2.6 Manipulate data using the StringBuilder class and its methods | Use StringBuilder Objects and Their Methods | 3 | 114 | | | |
| 3.3 Test equality between strings and other objects using == and equals () | Test Equality between Strings and other Objects | 3 | 119 | | | |
| 2.1 Declare and initialize variables | Understand Primitives, Enumerations, and Objects | 4 | 144 | | | |
| 2.2 Differentiate between object reference variables and primitive variables | Use Primitives, Enumerations, and Objects | 4 | 161 | | | |
| 1.1 Define the scope of variables | Understand Variable Scope | 5 | 194 | | | |
| 2.4 Explain an object's lifecycle | Understand Variable Scope | 5 | 194 | | | |

Exam Readiness Checklist

| Official Objective | Study Guide Coverage | Ch # | Pg # | Beginner | Intermediate | Expert |
|---|---|------|------|----------|--------------|--------|
| 7.5 Use super and this to access objects and constructors | Use this and super Keywords | 5 | 203 | | | |
| 6.1 Create methods with arguments and return values | Create and Use Methods | 5 | 184 | | | |
| 6.5 Create and overload constructors | Create and Use Constructors | 5 | 200 | | | |
| 2.5 Call methods on objects | Create and Use Methods | 5 | 184 | | | |
| 6.2 Apply the static keyword to methods and fields | Create Static Methods and Instance Variables | 5 | 207 | | | |
| 6.3 Create an overloaded method | Create and Use Methods | 5 | 184 | | | |
| 6.4 Differentiate between default and user-defined constructors | Create and Use Constructors | 5 | 200 | | | |
| 6.8 Determine the effect upon object references and primitive values when they are passed into methods that change the values | Pass Objects by Reference and Value | 5 | 191 | | | |
| 4.1 Declare, instantiate, initialize and use a one-dimensional array | Work with Java Arrays | 6 | 234 | | | |
| 4.2 Declare, instantiate, initialize and use multi-dimensional array | Work with Java Arrays | 6 | 234 | | | |
| 4.3 Declare and use an <code>ArrayList</code> (new coverage) | Work with ArrayList Objects and Their Methods | 6 | 242 | | | |
| 7.1 Implement inheritance | Implement and Use Inheritance and Class Types | 7 | 264 | | | |
| 7.6 Use abstract classes and interfaces | Implement and Use Inheritance and Class Types | 7 | 264 | | | |

Exam Readiness Checklist

| Official Objective | Study Guide Coverage | Ch # | Pg # | Beginner | Intermediate | Expert |
|---|--|------|------|----------|--------------|--------|
| 6.6 Apply access modifiers | Understand Encapsulation Principles | 7 | 274 | | | |
| 2.3 Read or write to object fields | Advanced Use of Classes with Inheritance and Encapsulation | 7 | 280 | | | |
| 6.7 Apply encapsulation principles to a class | Understand Encapsulation Principles | 7 | 274 | | | |
| 7.2 Develop code that demonstrates the use of polymorphism | Understand Polymorphism | 8 | 308 | | | |
| 7.3 Differentiate between the type of a reference and the type of an object | Understand Polymorphism | 8 | 308 | | | |
| 7.4 Determine when casting is necessary | Understand Casting | 8 | 324 | | | |
| 8.3 Describe what exceptions are used for in Java | Understand the Rationale and Types of Exceptions | 9 | 346 | | | |
| 8.1 Differentiate among checked exceptions, RuntimeExceptions, and Errors | Understand the Rationale and Types of Exceptions | 9 | 346 | | | |
| 8.2 Create a try-catch block and determine how exceptions alter normal program flow | Alter the Program Flow | 9 | 353 | | | |
| 8.4 Invoke a method that throws an exception | Understand the Nature of Exceptions | 9 | 350 | | | |
| 8.5 Recognize common exception classes and categories | Recognize Common Exceptions | 9 | 361 | | | |
| Supplemental Coverage | Understand Class Compositions and Associations | 10 | 384 | | | |
| Supplemental Coverage | Class Compositions and Associations in Practice | 10 | 390 | | | |

To obtain material from the disc that accompanies the printed version of this eBook, please follow the instructions in [Appendix I About the Download](#).



1

Packaging, Compiling, and Interpreting Java Code

CERTIFICATION OBJECTIVES

- Understand Packages
- Understand Package-Derived Classes
- Understand Class Structure
- Compile and Interpret Java Code
- ✓ Two-Minute Drill

Q&A Self Test

Since you are holding this book, or reading an electronic version of it, you must have an affinity for Java. You must also have the desire to let everyone know through the Oracle Certified Associate, Java SE 7 Programmer (OCA) certification process that you are truly Java savvy. As such, you should either be—or have the desire to be—a Java programmer, and in the long term, a true Java developer. You may be or plan to be a project manager heading up a team of Java programmers and/or developers. In this case, you will need to acquire a basic understanding of the Java language and its technologies. In either case, this book is for you.

To start, you may be wondering about the core functional elements provided by the basic Java Standard Edition (SE) platform with regard to libraries and utilities, and how these elements are

organized. This chapter answers these questions by discussing Java packages and classes, along with their packaging, structuring, compilation, and interpretation processes.

When you have finished this chapter, you will have a firm understanding of packaging Java classes, high-level details of common Java SE packages, and the fundamentals of Java's compilation and interpretation tools.

CERTIFICATION OBJECTIVE

Understand Packages

Exam Objective 1.4 Import other Java packages to make them accessible in your code

Packaging is a common approach used to organize related classes and interfaces. Most reusable code is packaged. Unpackaged classes are commonly found in books and online tutorials, as well as software *applications* with a narrow focus. This section will show you how and when to package your Java classes and how to import external classes from your Java packages. The following topics will be covered:

- Package design
- Package and import statements

Package Design

Packages are thought of as containers for classes, but actually they define where classes will be located in the hierarchical directory structure. Packaging is encouraged by Java coding standards to decrease the likelihood of classes colliding. Packaging your classes also promotes code reuse, maintainability, and the object-oriented principle of encapsulation and modularity.

When you design Java packages, such as the grouping of classes, the key areas shown in [Table 1-1](#) should be considered.

TABLE 1-1 Package Attribute Considerations

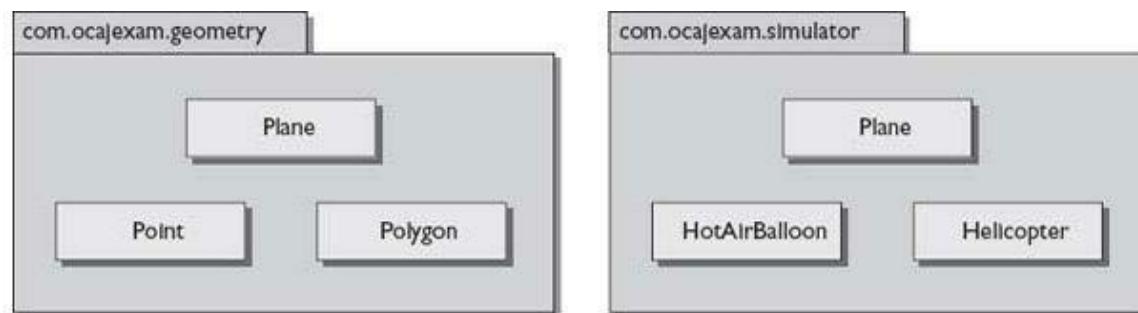
| Package Attribute | Benefits of Applying the Package Attribute |
|-------------------|--|
| Class coupling | Package dependencies are reduced with class coupling. |
| System coupling | Package dependencies are reduced with system coupling. |
| Package size | Typically, larger packages support reusability, whereas smaller packages support maintainability. |
| Maintainability | Often, software changes can be limited to a single package when the package houses focused functionality. |
| Naming | Consider conventions when naming your packages. Use a reverse domain name for the package structure. Use lowercase characters delimited with underscores to separate words in package names. |

Let's take a look at a real-world example. As program manager, suppose you need two sets of classes with unique functionality that will be used by the same end product. You task Developer A to build the first set and Developer B to build the second. You do not define the names of the classes, but

you do define the purpose of the package and what it must contain. Developer A is to create several geometry-based classes including a point class, a polygon class, and a plane class. Developer B is to build classes that will be included for simulation purposes, including objects such as hot air balloons, helicopters, and airplanes. You send them off to build their classes (without having them package their classes).

Come delivery time, they both give you a class named `Plane.java`—that is, one for the geometry plane class and one for the airplane class. Now you have a problem, because both of these source files (class files, too) cannot coexist in the same directory because they have the same name. The solution is packaging. If you had designated package names to the developers, this conflict never would have happened (as shown in [Figure 1-1](#)). The lesson learned is this: Always package your code, unless your coding project is trivial in nature.

FIGURE 1-1 Separate packaging of classes with the same names



package and import Statements

You should now have a general idea of when and why to package your source files. Next, you need to know exactly how to do this. To place a source file into a package, you use the `package` statement at the beginning of that file. You may use zero or one `package` statements per source file. To `import` classes from other packages into your source file, use the `import` statement. The `java.lang` package that houses the core language classes is imported by default.

The following code listing shows usage of the `package` and `import` statements. You can return to this listing as we discuss the `package` and `import` statements in detail throughout the chapter.

```

package com.ocaj.exam.tutorial; // Package statement
/* Imports class ArrayList from the java.util package */
import java.util.ArrayList;
/* Imports all classes from the java.io package */
import java.io.*;
public class MainClass {
    public static void main(String[] args) {
        /* Creates console from java.io package */
        Console console = System.console();
        String planet = console.readLine("\nEnter your favorite
planet: ");
        /* Creates list for planets */
        ArrayList planetList = new ArrayList();
        planetList.add(planet); // Adds users input to the list
        planetList.add("Gliese 581 c"); // Adds a string to the list
        System.out.println("\nTwo cool planets: " + planetList);
    }
}
$ Enter your favorite planet: Jupiter
$ Two cool planets: [Jupiter, Gliese 581 c]

```

The package Statement

The package statement includes the package keyword, followed by the package path delimited with periods. [Table 1-2](#) shows valid examples of package statements.

TABLE 1-2 Valid package Statements

| package Statement | Related Directory Structure |
|---------------------------------|--|
| package java.net; | [directory_path]\java\net\ |
| package com.ocajexam.utilities; | [directory_path]\com\ocajexam\utilities\ |
| package package_name; | [directory_path]\package_name\ |

package statements have the following attributes:

- They are optional.
- They are limited to one per source file.
- Standard coding convention for package statements reverses the domain name of the organization or group creating the package. For example, the owners of the domain name ocajexam.com may use the following package name for a utilities package: com.ocajexam.utilities.
- Package names equate to directory structures. The package name com.ocajexam.utils would equate to the directory com/ocajexam/utils. If a class includes a package statement that does not map to the relative directory structure, the class will not be usable.
- The package names beginning with java.* and javax.* are reserved.
- Package names should be lowercase. Individual words within the package name should be separated by underscores.

The Java SE API contains several packages. These packages are detailed in Oracle's Online Javadoc documentation at <http://docs.oracle.com/javase/7/docs/api/>.

Common packages you will see on the exam are packages for the Java Abstract Window Toolkit API, the Java Swing API, the Java Basic Input/Output API, the Java Networking API, the Java Utilities API, and the core Java Language API. You will need to know the basic functionality that each package/API contains.

The import Statement

An `import` statement allows you to include source code from other classes into a source file at compile time. The `import` statement includes the `import` keyword followed by the package path delimited with periods and ending with a class name or an asterisk, as shown in [Table 1-3](#). These `import` statements occur after the optional package statement and before the class definition. Each `import` statement can relate to one package only.

TABLE 1-3 Valid import Statements

| import Statement | Definition |
|--|---|
| <code>import java.net.*;</code> | Imports all the classes from the package <code>java.net</code> |
| <code>import java.net.URL;</code> | Imports only the <code>URL</code> class from the package <code>java.net</code> |
| <code>import static java.awt.Color.*;</code> | Imports all static members of the <code>Color</code> class of the package <code>java.awt</code> (J2SE 5.0 onward only) |
| <code>import static java.awt.color.ColorSpace .CS_GRAY;</code> | Imports the static member <code>CS_GRAY</code> of the <code>ColorSpace</code> class of the package <code>java.awt</code> (J2SE 5.0 onward only) |

SCENARIO & SOLUTION

| | |
|--|---|
| To paint basic graphics and images, which package should you use? | Use the Java AWT API package. <code>import java.awt.*;</code> |
| To create lightweight components for a GUI, which package should you use? | Use the Java Swing API package. <code>import javax.swing.*;</code> |
| To utilize data streams, which package should you use? | Use the Java Basic I/O package. <code>import java.io.*;</code> |
| To develop a networking application, which package should you use? | Use the Java Networking API package. <code>import java.net.*;</code> |
| To work with the collections framework, event model, and date/time facilities, which package should you use? | Use the Java Utilities API package. <code>import java.util.*;</code> |
| To utilize the core Java classes and interfaces, which package should you use? | Use the core Java Language package. <code>import java.lang.*;</code> |

For maintenance purposes, it is better to import your classes explicitly. This will allow the programmer to quickly determine which external classes are used throughout the class. As an example, rather than using `import java.util.`, use `import java.util.Vector`. In this real-world example, the coder would quickly see (with the latter approach) that the class imports only one class and it is a collection type. In this case, it is a legacy type and the determination to update the class with a*

newer collection type could be done quickly.

C and C++ programmers will see some look-and-feel similarities between Java's import statement and C/C++'s #include statement, even though there is no direct mapping in functionality.

The static import Statement

Static imports are a new feature to Java SE 5.0. (See more features here:

http://en.wikipedia.org/wiki/Java_version_history) Simply put, static imports allow you to import static members. The following example statements would be valid in Java SE 5.0 but would be invalid for J2SE 1.4:

```
/* Import static member ITALY */
import static java.util.Locale.ITALY;
...
System.out.println("Locale: " + ITALY); // Prints "Local: it_IT"
...

/* Imports all static members in class Locale */
import static java.util.Locale.*;
...
System.out.println("Locale: " + ITALY); // Prints "Local: it_IT"
System.out.println("Locale: " + GERMANY); // Prints "Local: de_DE"
System.out.println("Locale: " + JAPANESE); // Print "Local: ja"
...
```

Without the static imports shown in the example, the direct references to ITALY, GERMANY, and JAPANESE would be invalid and would cause compilation issues.

```
// import static java.util.Locale.ITALY;
...
System.out.println("Locale: " + ITALY); // Won't compile
```

EXERCISE 1-1

Replacing Implicit import Statements with Explicit import Statements

Consider the following sample application:

```

import java.io.*;
import java.text.*;
import java.util.*;
import java.util.logging.*;

public class TestClass {
    public static void main(String[] args) throws IOException {
        /* Ensure directory has been created */
        new File("logs").mkdir();
        /* Get the date to be used in the filename */
        DateFormat df = new SimpleDateFormat("yyyyMMddhhmmss");
        Date now = new Date();
        String date = df.format(now);
        /* Set up the filename in the logs directory */
        String logFileName = "logs\\testlog-" + date + ".txt";
        /* Set up Logger */
        FileHandler myFileHandler = new FileHandler(logFileName);
        myFileHandler.setFormatter(new SimpleFormatter());
        Logger ocajLogger = Logger.getLogger("OCAJ Logger");
        ocajLogger.setLevel(Level.ALL);
        ocajLogger.addHandler(myFileHandler);
        /* Log Message */
        ocajLogger.info("\nThis is a logged information message.");
        /* Close the file */
        myFileHandler.close();
    }
}

```

There can be **implicit imports** that allow all necessary classes of a package to be imported:

```
import java.io.*; // Implicit import example
```

There can be **explicit imports** that allow only the designated class or interface of a package to be imported:

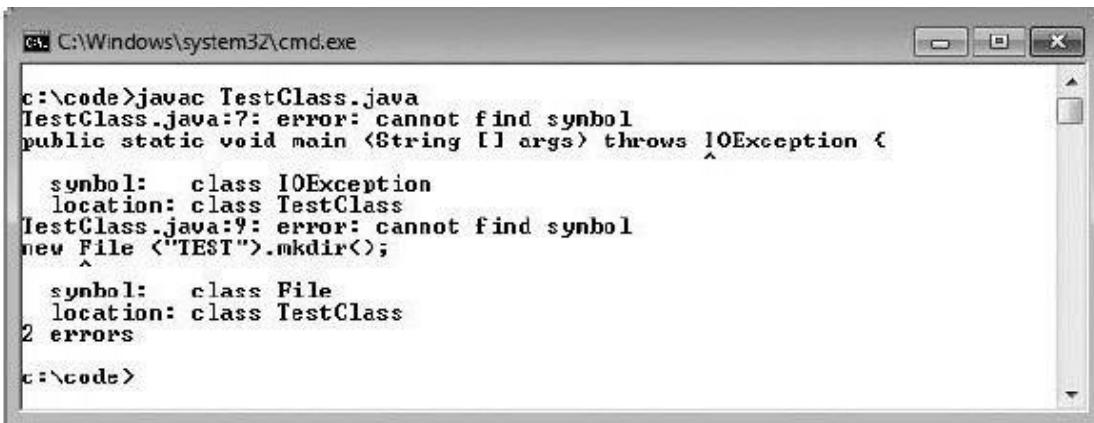
```
import java.io.File; // Explicit import example
```

This exercise will have you using explicit **import** statements in lieu of the **implicit import** statements for all of the necessary classes of the sample application. If you are unfamiliar with compiling and interpreting Java programs, finish reading this chapter and then come back to this exercise. Otherwise, let's begin.

1. Type the sample application into a new file and name it *TestClass.java*. Save the file.
2. Compile and run the application to ensure that you have created the file contents without error:
javac *TestClass.java* to compile, java *TestClass* to run. Verify that the log message prints to the screen. Also verify that a file has been created in the logs subdirectory with the same message in it.
3. Comment out all of the **import** statements:

```
//import java.io.*;  
//import java.text.*;  
//import java.util.*;  
//import java.util.logging.*;
```

4. Compile the application: `javac TestClass.java`. You will be presented with several compiler errors related to the missing class imports. As an example, the following illustration demonstrates the errors that are displayed when only the `java.io` package has been commented out:



The screenshot shows a Windows command prompt window titled "cmd C:\Windows\system32\cmd.exe". The command entered is `c:\code>javac TestClass.java`. The output shows two errors:

```
c:\code>javac TestClass.java  
TestClass.java:7: error: cannot find symbol  
public static void main (String [] args) throws IOException {  
          ^  
      symbol:   class IOException  
      location: class TestClass  
TestClass.java:9: error: cannot find symbol  
new File ("TEST").mkdir();  
          ^  
      symbol:   class File  
      location: class TestClass  
2 errors  
c:\code>
```

5. For each class that cannot be found, use the online Java Specification API to determine which package it belongs to and then update the source file with the necessary explicit `import` statement. Once completed, you will have replaced the four *implicit import* statements with nine *explicit import* statements.
6. Run the application again to ensure that the application works the same with the explicit imports as it did with the implicit import.
-

CERTIFICATION OBJECTIVE

Understand Package-Derived Classes

Oracle includes 209 packages in the Java SE 7 API. Each package has a specific focus. Fortunately, you need to be familiar with only a few of them for the OCA exam. These may include packages for Java utilities, basic input/output, networking, Abstract Window Toolkit (AWT), and Swing.

The following sections address these APIs:

- Java Utilities API
- Java Basic Input/Output API
- Java Networking API
- Java Abstract Window Toolkit API
- Java Swing API

Java Utilities API

The Java Utilities API is contained in the package `java.util`. This API provides functionality for a

variety of utility classes. The API's key classes and interfaces can be divided into several categories. Categories of classes that may be seen on the exam include the Java Collections Framework, date and time facilities, internationalization, and some miscellaneous utility classes.

Of these categories, the Java Collections Framework pulls the most weight since it is frequently used and provides the fundamental data structures necessary to build valuable Java applications. [Table 1-4](#) details the classes and interfaces of the Collections API that you may see referenced on the exam.

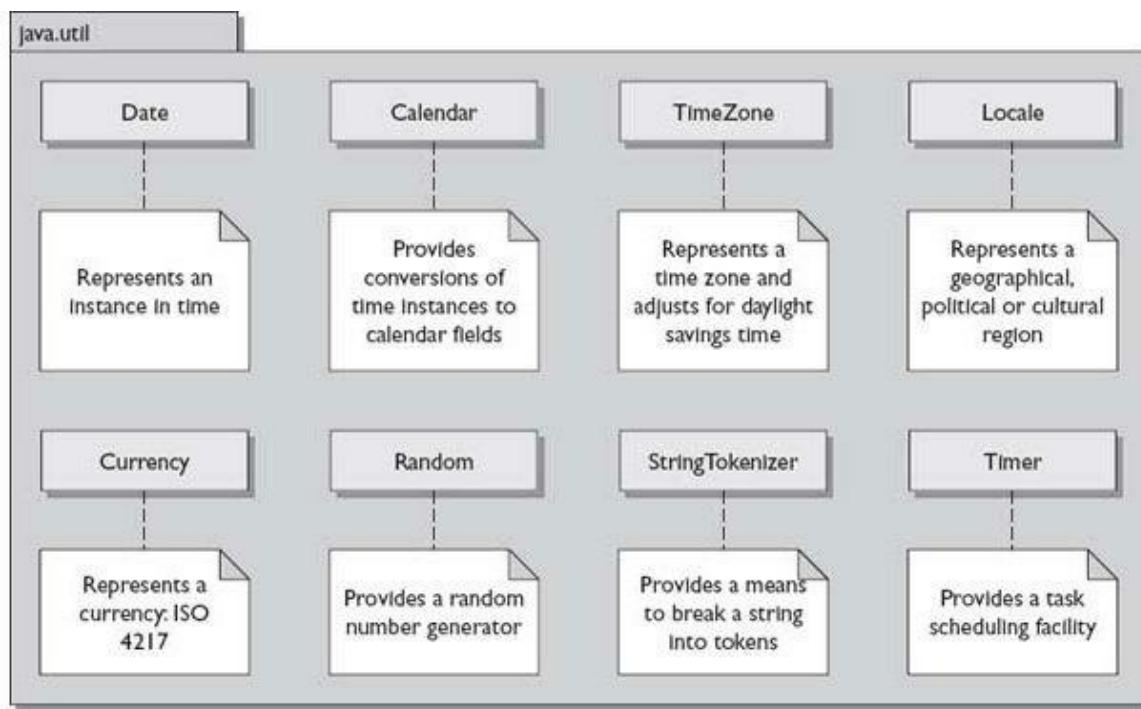
TABLE 1-4 Various Classes of the Java Collections Framework

| Interface | Implementations | Description |
|-----------|--|--|
| List | ArrayList, LinkedList, Vector | Data structures based on positional access. |
| Map | HashMap, Hashtable, LinkedHashMap, TreeMap | Data structures that map keys to values. |
| Set | HashSet, LinkedHashSet, TreeSet | Data structures based on element uniqueness. |
| Queue | PriorityQueue | Queues typically order elements in a first in, first out (FIFO) manner. Priority queues order elements according to a supplied comparator. |

To assist collections in sorting where the ordering is not natural, the Collections API provides the `Comparator` interface. Similarly, the `Comparable` interface that resides in the `java.lang` package is used to sort objects by their natural ordering.

Various other classes and interfaces reside in the `java.util` package. Date and time facilities are represented by the `Date`, `Calendar`, and `TimeZone` classes. Geographical regions are represented by the `Locale` class. The `Currency` class represents currencies per the ISO 4217 standard. A random number generator is provided by the `Random` class. And `StringTokenizer` breaks strings into tokens. Several other classes exist within `java.util`, but these (and the collection interfaces and classes) are classes that you may find yourself commonly using on the job. These utilities classes are represented in [Figure 1-2](#).

FIGURE 1-2 Various utility classes

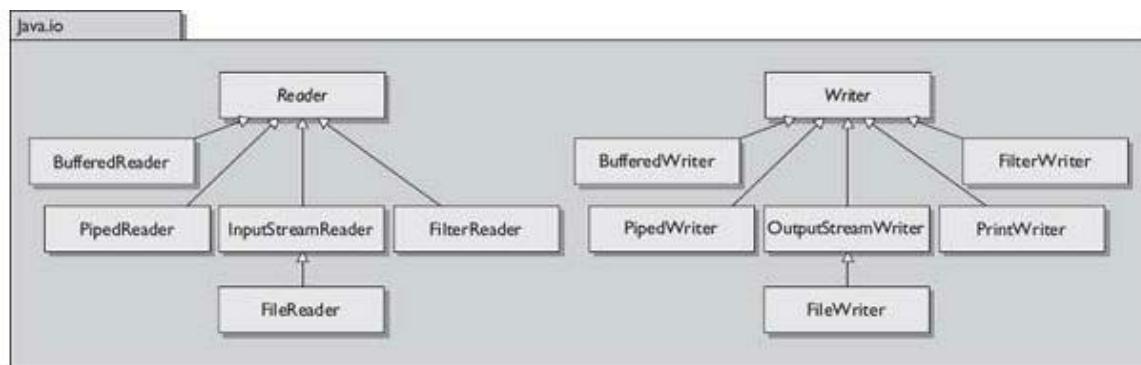


on the job *Many packages have related classes and interfaces that have unique functionality, so they are included in their own subpackages. For example, regular expressions are stored in a subpackage of the Java utilities (`java.util`) package. The subpackage is named `java.util.regex` and houses the `Matcher` and `Pattern` classes. Where needed, consider creating subpackages for your own projects.*

Java Basic Input/Output API

The Java Basic Input/Output API is contained in the package `java.io`. This API provides functionality for general system input and output in relationships to data streams, serialization, and the file system. Data stream classes include byte-stream subclasses of the `InputStream` and `OutputStream` classes. Data stream classes also include character-stream subclasses of the `Reader` and `Writer` classes. [Figure 1-3](#) depicts part of the class hierarchy for the `Reader` and `Writer` abstract classes.

FIGURE 1-3 Reader and Writer class hierarchy



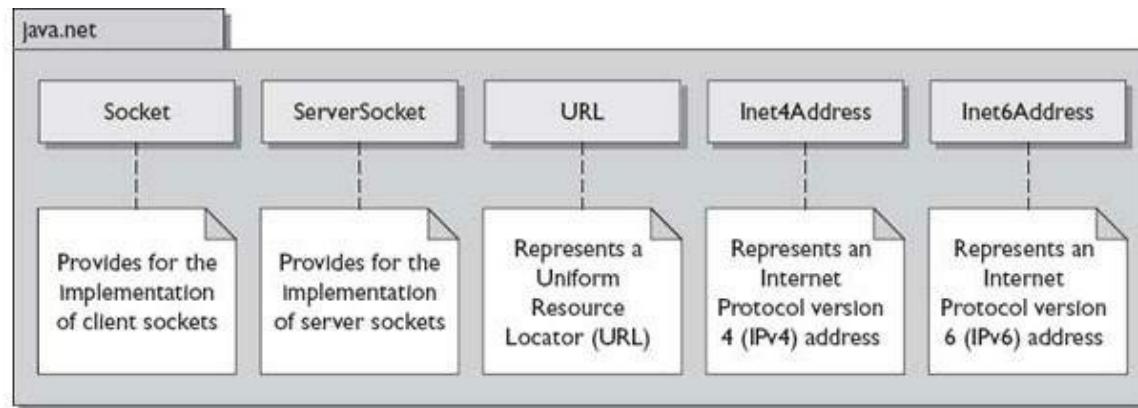
Other important `java.io` classes and interfaces include `File`, `FileDescriptor`, `FilenameFilter`, and `RandomAccessFile`. The `File` class provides a representation of file and directory pathnames. The `FileDescriptor` class provides a means to function as a handle for opening

files and sockets. The `FilenameFilter` interface, as its name implies, defines the functionality to filter filenames. The `RandomAccessFile` class allows for the reading and writing of files to specified locations.

The Java Networking API

The Java Networking API is contained in the package `java.net`. This API provides functionality in support of creating network applications. The API's key classes and interfaces are represented in [Figure 1-4](#). You probably will see few, if any, of these classes on the exam, but the figure will help you conceptualize what's in the `java.net` package. The improved performance New I/O API (`java.nio`) package, which provides for nonblocking networking and the socket factory support package (`javax.net`), is not on the exam.

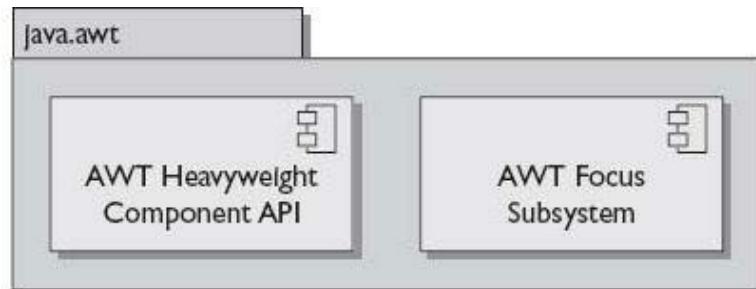
FIGURE 1-4 Various classes of the Networking API



Java Abstract Window Toolkit API

The Java Abstract Window Toolkit API is contained in the package `java.awt`. This API provides functionality for creating heavyweight components in regards to creating user interfaces and painting associated graphics and images. The AWT API was Java's original GUI API and has been superseded by the Swing API. Where Swing is now recommended, certain pieces of the AWT API still remain commonly used, such as the AWT Focus subsystem that was reworked in J2SE 1.4. The AWT Focus subsystem provides for navigation control between components. [Figure 1-5](#) depicts these major AWT elements.

FIGURE 1-5 AWT major elements



Java Swing API

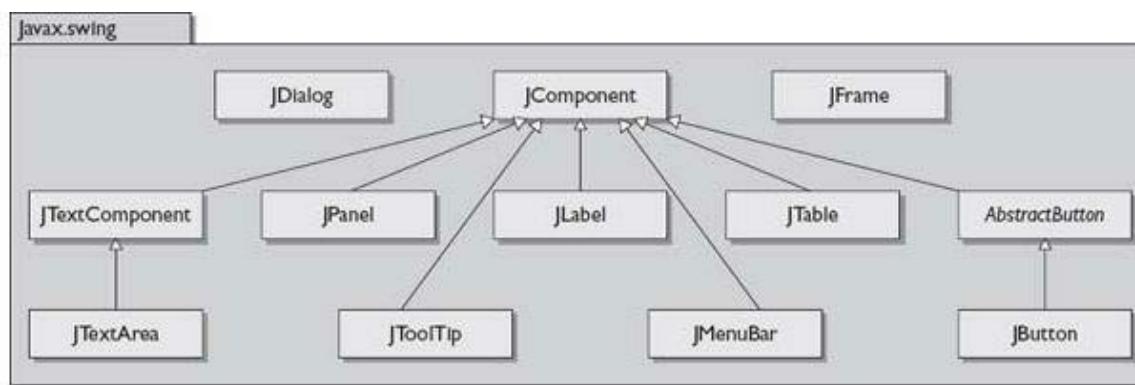
The Java Swing API is contained in the package `javax.swing`. This API provides functionality for

creating lightweight (pure-Java) containers and components. The Swing API, providing a more sophisticated set of GUI components, supersedes the AWT API. Many of the Swing classes are simply prefaced with the addition of “J” in contrast to the legacy AWT component equivalent. For example, Swing uses the class JButton to represent a button container, whereas AWT uses the class Button.

| SCENARIO & SOLUTION | |
|---|---|
| You need to create basic Java Swing components such as buttons, panes, and dialog boxes. Provide the code to import the necessary classes of a package. | // Java Swing API package import javax.swing.*; |
| You need to support text-related aspects of your Swing components. Provide the code to import the necessary classes of a package. | // Java Swing API text subpackage import javax.swing.text.*; |
| You need to implement and configure basic pluggable look-and-feel support. Provide the code to import the necessary classes of a package. | // Java Swing API plaf subpackage import javax.swing.plaf.*; |
| You need to use Swing event listeners and adapters. Provide the code to import the necessary classes of a package. | // Java Swing API event subpackage import javax.swing.event.*; |

Swing also provides look-and-feel support, allowing for universal style changes of the GUI’s components. Other features include tooltips, accessibility functionality, an event model, and enhanced components such as tables, trees, text components, sliders, and progress bars. Some of the Swing API’s key classes are represented in [Figure 1-6](#).

FIGURE 1-6 Various classes of the Swing API



The Swing API makes excellent use of subpackages, with 18 of them total in Java SE 7. As mentioned earlier, when common classes are separated into their own packages, code usability and maintainability is enhanced.

Swing takes advantage of the model-view-controller architecture (MVC). The model represents the current state of each component. The view is the representation of the components on the screen. The controller is the functionality that ties the UI components to events. Although understanding the underlying architecture of Swing is important, it’s not necessary for the exam. For comprehensive information on the Swing API, look to the book *Swing: A Beginner’s Guide*, by Herbert Schildt (McGraw-Hill Professional, 2007).

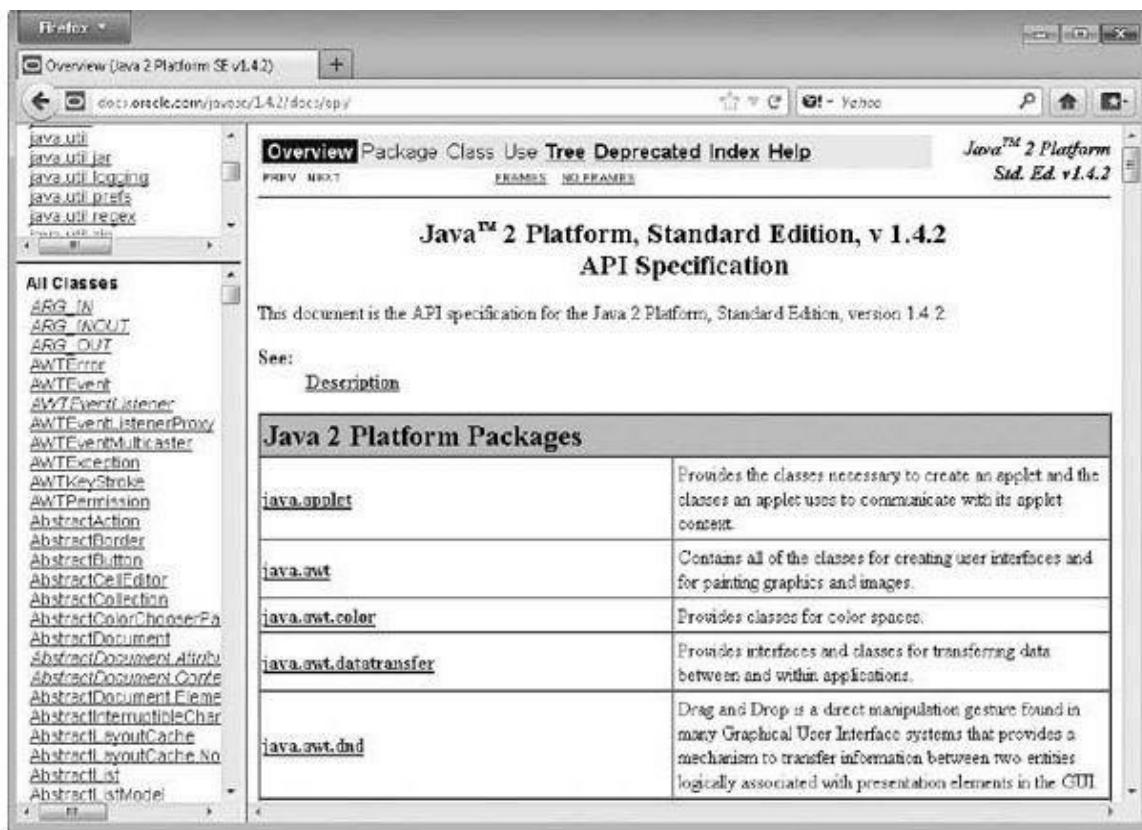
It's good to be familiar with the package prefixes java and javax. The prefix java is commonly used for the core packages. The prefix javax is commonly used for packages that comprise Java standard extensions. Take special notice of the prefix

EXERCISE 1-2

Understanding Extended Functionality of the Java Utilities API

In Java SE 6 and 7, a total of ten packages are in direct relationship to the Java Utilities API, with the base package being named `java.util`. J2SE 5.0 has only nine packages; J2SE 1.4, just six. This exercise will have you exploring the details of the Java Utilities API subpackages that were added in subsequent releases of the Java SE 1.4 platform.

1. Go to the online J2SE 1.4.2 API specification: <http://docs.oracle.com/javase/1.4.2/docs/api/>. It is shown in the following illustration.



2. Use the web browser's scroll bar to scroll down to the Java Utilities API packages.
3. Click the link for each related package. Explore the details of the classes and interfaces within each package.
4. Go to the online J2SE 5.0 API specification: <http://docs.oracle.com/javase/1.5.0/docs/api/>. It is shown in the following illustration.

Java™ 2 Platform Standard Edition 5.0
API Specification

This document is the API specification for the Java 2 Platform Standard Edition 5.0.

See: [Description](#)

Java 2 Platform Packages

| Package | Description |
|---------------------------------------|--|
| java.applet | Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context. |
| java.awt | Contains all of the classes for creating user interfaces and for painting graphics and images. |
| java.awt.color | Provides classes for color spaces. |
| java.awt.datatransfer | Provides interfaces and classes for transferring data between and within applications. |
| java.awt.dnd | Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between two entities logically associated with presentation elements in the GUI. |

5. Use the web browser's scroll bar to scroll down to the Java Utilities API packages.
6. Determine which three new subpackages were added to the Java Utilities API. Click the link for each of these new packages. Explore the details of the classes and interfaces within each package.
7. Go to the online Java SE 7 API specification: <http://docs.oracle.com/javase/7/docs/api/>. This is the API specification you should be referencing for the exam. It is shown in the following illustration.

Java™ Platform, Standard Edition 7
API Specification

This document is the API specification for the Java™ Platform, Standard Edition 7.

See: [Description](#)

Packages

| Package | Description |
|---------------------------------------|--|
| java.awt | Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context. |
| java.awt | Contains all of the classes for creating user interfaces and for painting graphics and images. |
| java.awt.color | Provides classes for color spaces. |
| java.awt.datatransfer | Provides interfaces and classes for transferring data between and within applications. |
| java.awt.dnd | Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between two entities logically associated with presentation elements in the GUI. |
| java.awt.event | Provides interfaces and classes for dealing with different types of events fired by AWT components. |
| java.awt.font | Provides classes and interface relating to fonts. |

8. Use the web browser's scroll bar to scroll down to the Java Utilities API packages.

9. Determine which new subpackage was added to the Java Utilities API since J2SE 5.0. Click the link for the new package. Explore the details of the classes within the package.

CERTIFICATION OBJECTIVE

Understand Class Structure

Exam Objective 1.2 Define the structure of a Java class

You must understand the structure of a Java class both to do well on the exam and to have a promising career with Java. It would help to have a fundamental knowledge of Java naming conventions as well as knowledge of the typical separators that are seen in Java source code (such as brackets for enclosing entities and also comment separators). These topics are covered in the following sections:

- Naming conventions
- Separators and other Java source symbols
- Java class structure

Naming Conventions

Naming conventions are rules for the usage/application of characters in creation of identifiers, methods, class names, and so forth throughout your code base. If some of your team members are not applying naming conventions to their code, you should encourage them to do so, for the good of the effort and for maintainability aspects for after the code is deployed.

The popular article, “How to Write Unmaintainable Code,” by Roedy Green, is worth reading (<http://thc.org/root/phun/unmaintain.html>). It brings to light, in a comical way, the challenges that can occur with maintaining code when there is a blatant or intentional disregard to software development best practices. On the flip side, “The Passionate Programmer, creating a remarkable career in software development,” by Chad Fowler, encourages the software developer to be the best that he or she can be.

on the job You may encounter people who will come up with their own naming conventions. Although this is better than not applying a convention, an outsider trying to maintain that person's code would need to learn the original convention and apply it as well for consistency. Fortunately, the Java community does subscribe to a shared thought on how naming conventions should be applied to the many different elements in Java. [Table 1-5](#) describes these conventions in a simple manner. When applying naming conventions, you should strive to use meaningful and unambiguous names. And remember that naming conventions are put into place for the primary goal of making Java programs more readable, and therefore maintainable. The practice of using camel case is in place with Java naming conventions. Camel case is the practice of using uppercase letters for the first characters in compound words.

TABLE 1-5 Java Naming Conventions

| Element | Lettering | Characteristic | Example |
|-------------------------------------|--|--|---|
| Class name | Begins uppercase, continues camel case | Noun | SpaceShip |
| Interface name | Begins uppercase, continues camel case | Adjective ending with “able” or “ible” when providing a capability, otherwise a noun | Dockable |
| Method name | Begins lowercase, continues camel case | Verb, may include adjective or noun | orbit |
| Instance and static variables names | Begins lowercase, continues camel case | Noun | moon |
| Parameters and local variables | Begins lowercase, continues camel case if multiple words are necessary | Single words, acronyms, or abbreviations | lop (i.e., line of position) |
| Generic type parameters | Single uppercase letter | The letter T is recommended | T |
| Constant | All uppercase letters | Multiple words separated by underscores | LEAGUE |
| Enumeration | Begins uppercase, continues camel case; the set of objects should be all uppercase | Noun | enum Occupation {MANNED, SEMI_MANNED, UNMANNED} |
| Package | All lowercase letters | Public packages should be the reversed domain name of the org | com.ocajexam.sim |

Separators and Other Java Source Symbols

The Java programming language makes use of several separators and symbols to aid in the structuring of the source code in a software program. [Table 1-6](#) details these separators and symbols.

TABLE 1-6 Symbols and Separators

| Symbol | Description | Purpose |
|--------|---|---|
| () | Parentheses | Encloses set of method arguments, encloses cast types, adjusts precedence in arithmetic expressions |
| { } | Braces | Encloses blocks of codes and also initializes arrays |
| [] | Box brackets | Declares array types and initializes arrays |
| < > | Angle brackets | Encloses generics |
| ; | Semicolon | Terminates statement at the end of a line |
| , | Comma | Separates identifiers in variable declarations, separates values, separates expressions in a for loop |
| . | Period | Delineates package names, selects an objects field or method, supports method chaining |
| : | Colon | Follows loop labels |
| ' ' | Single quotes | Defines a single character |
| " " | Double quotes | Defines a string of characters |
| // | Forward slashes | Indicates a single line comment |
| /* */ | Forward slashes with asterisks | Indicates a blocked comment for multiple lines |
| /** */ | Forward slashes with a double and a single asterisk | Indicates Javadoc comments |

Java Class Structure

Every Java program has at least one class. A Java class has a signature, optional constructors, optional data members (fields), and optional methods, as outlined here:

```
[modifiers] class classIdentifier [extends superClassIdentifier]
[implements interfaceIdentifier1, interfaceIdentifier2, etc.] {
    [data members]
    [constructors]
    [methods]
}
```

Each class may extend one and only one super class. Each class may implement one or more interfaces. Interfaces are separated by commas.

The following SpaceShip class shows typical elements annotated with comments. The file containing this SpaceShip class must be called SpaceShip.java .

```
package com.ocajexam.craft_simulator;

public class SpaceShip extends Ship implements Dockable {

    // Data Members
    public enum ShipType {
        FRIGATE, BATTLESHIP, MINELAYER, ESCORT, DEFENSE
    }
    ShipType shipType = ShipType.ESCORT;

    // Constructors
    public SpaceShip() {
        System.out.println("\nSpaceShip created with default ship
type.");
    }
    public SpaceShip(ShipType shipType) {
        System.out.println("\nSpaceShip created with specified ship
type.");
        this.shipType = shipType;
    }

    // Methods
    @Override
    public void dockShip () {
        // TODO
    }
    @Override
    public String toString() {
        String shipTypeRefined = this.shipType.name().toLowerCase();
        return "The pirate ship is a " + shipTypeRefined + " ship.";
    }
}
```

This SpaceShip class can be instantiated as demonstrated in the following code:

```
package com.ocajexam.craft_simulator;
import com.ocajexam.craft_simulator.PirateShip.ShipType;
public class SpaceShipSimulator {

    public static void main(String[] args) {

        // Create SpaceShip object with default ship type
        SpaceShip ship1 = new SpaceShip ();
        // Prints "The pirate ship is a caravel ship."
        System.out.println(ship1);

        // Create SpaceShip object with specified ship type
        SpaceShip ship2 = new SpaceShip (ShipType.FRIGATE);
        // Prints "The pirate ship is a galleon ship."
        System.out.println(ship2);
    }
}
```



The *override* annotation (that is, `@Override`) indicates that a method declaration intends on overriding a method declaration in the class's supertype.

[Chapter 4](#), “Working with Basic Classes and Variables”; [Chapter 5](#), “Understanding Variable Scope and Class Construction”; [Chapter 6](#), “Working with Classes and Their Relationships”; and [Chapter 7](#), “Understanding Class Inheritance” all go into more comprehensive details about creating and working with classes.

CERTIFICATION OBJECTIVE

Compile and Interpret Java Code

Exam Objective 1.3 Create executable Java applications with a main method

The Java Development Kit (JDK) includes several utilities for compiling, debugging, and running Java applications. This section details two utilities from the kit: the Java compiler and the Java interpreter. For more information on the JDK and its other utilities, see [Chapter 10](#).

Java Compiler

We will need a sample application to use for our Java compiler and interpreter exercises. We shall employ the simple `GreetingsUniverse.java` source file, shown in the following listing, throughout the section.

```
public class GreetingsUniverse {  
    public static void main(String[] args) {  
        System.out.println("Greetings, Universe!");  
    }  
}
```

Let’s take a look at compiling and interpreting simple Java programs along with their most basic command-line options.

Compiling Your Source Code

The Java compiler is only one of several tools in the JDK. When you have time, inspect the other tools resident in the JDK’s bin folder, as shown in [Figure 1-7](#). For the scope of the OCA exam, you will need to know the details surrounding only the compiler and interpreter.

FIGURE 1-7 Java Development Kit utilities

```
c:\Windows\system32\cmd.exe
c:\Program Files\Java\jdk1.7.0_04\bin>dir *.exe /w
Volume in drive C is 0S
Volume Serial Number is 6059-69EE

Directory of c:\Program Files\Java\jdk1.7.0_04\bin

appletviewer.exe    apt.exe          extcheck.exe      idlj.exe
jar.exe            jarsigner.exe   java-rmi.exe    java.exe
javac.exe          javadoc.exe    Javah.exe        javap.exe
javaws.exe         javaws.exe     Jcmd.exe        jconsole.exe
jdb.exe            jhat.exe       Jinfo.exe       jmap.exe
jps.exe            jrunscript.exe jsadebugd.exe  jstack.exe
jstat.exe          jstated.exe    Jvisualvm.exe keytool.exe
kinit.exe          klist.exe      Ktab.exe        native2ascii.exe
orbd.exe           pack200.exe   PolicyTool.exe rmid.exe
rmid.exe           rmiregistry.exe Schenagen.exe  serialver.exe
servertool.exe    tnameserv.exe unpack200.exe wsimport.exe
wsimport.exe       xjc.exe       46 File(s)      1,478,192 bytes
                           0 Dir(s)      265,837,461,504 bytes free

c:\Program Files\Java\jdk1.7.0_04\bin>
```

The Java compiler simply converts Java source files into bytecode. The Java compiler's usage is as follows:

```
javac [options] [source files]
```

The most straightforward way to compile a Java class is to preface the Java source files with the compiler utility from the command line: `javac.exe FileName.java`. The `.exe` is the standard executable file extension on Windows machines and is optional. The `.exe` extension is not present on executables on UNIX-like systems.

```
javac GreetingsUniverse.java
```

This will result in a bytecode file being produced with the same preface, such as `GreetingsUniverse.class`. This bytecode file will be placed into the same folder as the source code, unless the code is packaged and/or it's been told via a command-line option to be placed somewhere else.

You will find that many projects use Apache Ant and/or Maven build environments. Understanding the fundamentals of the command-line tools is necessary for writing/maintaining the scripts associated with these build products.

INSIDE THE EXAM

Command-Line Tools

Most projects use integrated development environments (IDEs) to compile and execute code. The clear benefit in using IDEs is that building and running code can be as easy as stepping few a couple of menu options or even just clicking a hot key. The disadvantage is that even though you may establish your settings through a configuration dialog box and see the commands and subsequent arguments in one of the workspace windows, you are not getting direct experience in repeatedly creating the complete structure of the commands and associated arguments by hand. The exam is

structured to validate that you have experience in scripting compiler and interpreter invocations. Do not take this prerequisite lightly. Take the exam only after you have mastered when and how to use the tools, switches, and associated arguments. At a later time, you can consider taking advantage of the “shortcut” features of popular IDEs such as those provided by NetBeans, Eclipse, IntelliJ IDEA, and JDeveloper.

Compiling Your Source Code with the -d Option

You may want to specify explicitly where you would like the compiled bytecode class files to go. You can accomplish this using the -d option:

```
javac -d classes GreetingsUniverse.java
```

This command-line structure will place the class file into the classes directory, and since the source code was packaged (that is, the source file included a package statement), the bytecode will be placed into the relative subdirectories.

```
[present working directory]\classes\com\ocajexam\tutorial\  
GreetingsUniverse.class
```

Compiling Your Code with the -classpath Option

If you want to compile your application with user-defined classes and packages, you may need to tell the JVM where to look by specifying them in the classpath. This classpath inclusion is accomplished by telling the compiler where the desired classes and packages are with the -cp or -classpath command-line option. In the following compiler invocation, the compiler includes in its compilation any source files that are located under the 3rdPartyCode\classes directory, as well as any classes located in the present working directory (the period). The -d option (again) will place the compiled bytecode into the classes directory.

```
javac -d classes -cp 3rdPartyCode\classes\; . GreetingsUniverse  
.java
```

Note that you do not need to include the classpath option if the classpath is defined with the CLASSPATH environment variable, or if the desired files are in the present working directory.

On Windows systems, classpath directories are delimited with backward slashes and paths are delimited with semicolons:

```
-classpath .; \dir_a\classes_a\; \dir_b\classes_b\
```

On POSIX-based systems, classpath directories are delimited with forward slashes and paths are delimited with colons:

```
-classpath .:/dir_a/classes_a/::/dir_b/classes_b/
```

Again, the period represents the present (or current) working directory.

Know your switches. The designers of the exam will try to throw you by presenting answers with mix-matching compiler and interpreter switches. You may even see some make-believe switches that do not exist anywhere. For additional preparation, query the commands' complete set of switches by typing `java -help` or `javac -help`. Switches are also known as command-line parameters, command-line switches, options, and flags.

Java Interpreter

Interpreting the Java files is the basis for creating the Java application, as shown in [Figure 1-8](#). Let's examine how to invoke the interpreter and its command-line options.

FIGURE 1-8 Bytecode conversion



```
java [-options] class [args...]
```

Interpreting Your Bytecode

The Java interpreter is invoked with the `java[.exe]` command. It is used to interpret bytecode and execute your program.

You can easily invoke the interpreter on a class that's not packaged as follows:

```
java MainClass
```

You can optionally start the program with the `javaw` command on Microsoft Windows to exclude the command window. This is a nice feature with GUI-based applications since the console window is often not necessary.

```
javaw.exe MainClass
```

Similarly, on POSIX-based systems, you can use the ampersand to run the application as a background process.

```
java MainClass &
```

Interpreting Your Code with the -classpath Option

When interpreting your code, you may need to define where certain classes and packages are located. You can find your classes at runtime when you include the `-cp` or `-classpath` option with the interpreter. If the classes you want to include are packaged, then you can start your application by pointing the full path of the application to the base directory of classes, as in the following interpreter

invocation:

```
java -cp classes com.ocajexam.tutorial.MainClass
```

The delimitation syntax is the same for the `-cp` and `-classpath` options, as defined earlier in the “Compiling Your Code with the `-classpath` Option” section.

Interpreting Your Bytecode with the `-D` Option

The `-D` command-line option allows for the setting of new property values. The usage is as follows:

```
java -D<name>=<value> class
```

The following single-file application comprising the `PropertiesManager` class prints out all of the system properties:

```
import java.util.Properties;
public class PropertiesManager {
    public static void main(String[] args) {
        if (args.length == 0) {System.exit(0);}
        Properties props = System.getProperties();
        /* New property example */

        props.setProperty("new_property2", "new_value2");
        switch (args[0]) {
            case "-list_all":
                props.list(System.out); // Lists all properties
                break;
            case "-list_prop":
                /* Lists value */
                System.out.println(props.getProperty(args[1]));
                break;
            default:
                System.out.println("Usage: java PropertiesManager
[-list_all]");
                System.out.println("      java PropertiesManager
[-list_prop [property]]");
                break;
        }
    }
}
```

Let's run this application while setting a new system property called `new_property1` to the value of `new_value1`:

```
java -Dnew_property1=new_value1 PropertiesManager -list_all
```

You'll see in the standard output that the listing of the system properties includes the new property that we set and its value:

```
new_property1=new_value1  
java.specification.name=Java Platform API Specification
```

Optionally, you can set a value by instantiating the `Properties` class and then setting a property and its value with the `setProperty` method.

To help you conceptualize system properties a little better, [Table 1-7](#) details a subset of the standard system properties.

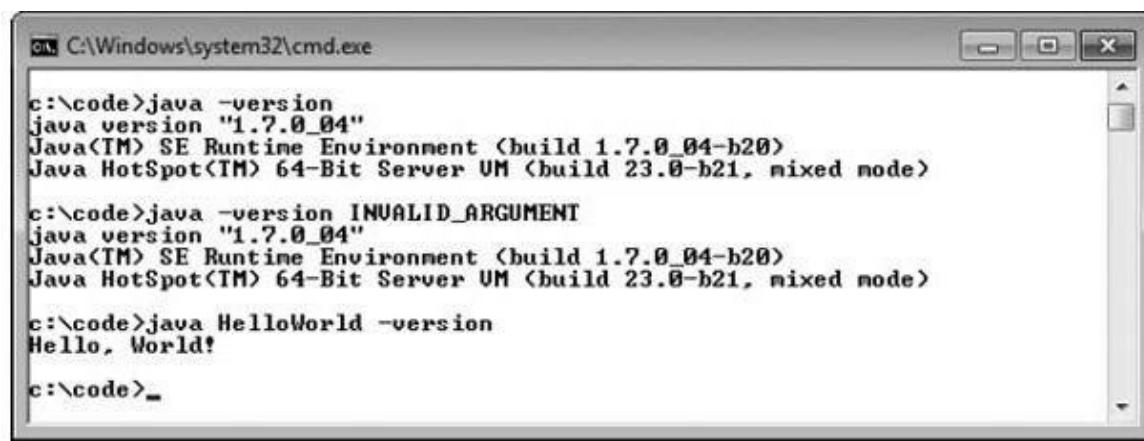
TABLE 1-7 Subset of System Properties

| System Property | Property Description |
|---------------------------------|---|
| <code>file.separator</code> | The platform specific file separator (/ for POSIX, \ for Windows) |
| <code>java.class.path</code> | The classpath as defined for the system's environment variable |
| <code>java.class.version</code> | The Java class version number |
| <code>java.home</code> | The directory of the Java installation |
| <code>java.vendor</code> | The vendor supplying the Java platform |
| <code>java.vendor.url</code> | The vendor's Uniform Resource Locator |
| <code>java.version</code> | The version of the Java Interpreter/JVM |
| <code>line.separator</code> | The platform-specific line separator (\r on Mac OS 9, \n for POSIX, \r\n for Microsoft Windows) |
| <code>os.arch</code> | The architecture of the operating system |
| <code>os.name</code> | The name of the operating system |
| <code>os.version</code> | The version of the operating system |
| <code>path.separator</code> | The platform-specific path separator (: for POSIX, ; for Windows) |
| <code>user.dir</code> | The current working directory of the user |
| <code>user.home</code> | The home directory of the user |
| <code>user.language</code> | The language code of the default locale |
| <code>user.name</code> | The username for the current user |
| <code>user.timezone</code> | The system's default time zone |

Retrieving the Version of the Interpreter with the `-version` Option

The `-version` command-line option is used with the Java interpreter to return the version of the JVM and exit. Don't take the simplicity of the command for granted, as the designers of the exam may try to trick you by including additional arguments after the command. Take the time to toy with the command by adding arguments and putting the `-version` option in various places. Do not make any assumptions about how you think the application will respond. [Figure 1-9](#) demonstrates varying results based on where the `-version` option is used.

FIGURE 1-9 The `-version` command-line option



```
c:\code>java -version
java version "1.7.0_04"
Java(TM) SE Runtime Environment (build 1.7.0_04-b20)
Java HotSpot(TM) 64-Bit Server VM (build 23.0-b21, mixed mode)

c:\code>java -version INVALID_ARGUMENT
java version "1.7.0_04"
Java(TM) SE Runtime Environment (build 1.7.0_04-b20)
Java HotSpot(TM) 64-Bit Server VM (build 23.0-b21, mixed mode)

c:\code>java HelloWorld -version
Hello, World!

c:\code>
```

on the Job Check out the other JDK utilities at your disposal. You can find them in the bin directory of your JDK. JConsole in particular is a valuable GUI-based tool that is used to monitor and manage Java applications. Among the many features, JConsole allows for viewing memory and thread usages. JConsole was released with J2SE 5.0.

EXERCISE 1-3

Compiling and Interpreting Packaged Software

When you compile and run packaged software from an IDE, the execution process can be as easy as clicking a run icon, as the IDE will maintain the classpath for you and will also let you know if anything is out of sorts. When you try to compile and interpret the code yourself from the command line, you will need to know exactly how to path your files. Consider our sample application that is placed in the com.ocajexam.tutorial package (that is, the com/ocajexam/tutorial directory).

```
package com.ocajexam.tutorial;
public class GreetingsUniverse {
    public static void main(String[] args) {
        System.out.println("Greetings, Universe!");
    }
}
```

This exercise will have you compiling and running the application with new classes created in a separate package.

1. Compile the program:

```
javac -d . GreetingsUniverse.java
```

2. Run the program to ensure it is error-free:

```
java -cp . com.ocajexam.tutorial.GreetingsUniverse
```

3. Create three classes named Earth, Mars, and Venus and place them in the com.ocajexam.tutorial.planets package. Create constructors that will print the names of the planets to standard out. The details for the Earth class are given here as an example of what you will need to do:

```
package com.ocajexam.tutorial.planets;
public class Earth {
    public Earth {
        System.out.println("Hello from Earth!");
    }
}
```

4. Instantiate each class from the main program, by adding the necessary code to the `GreetingsUniverse` class.

```
Earth e = new Earth();
```

5. Ensure that all of the source code is in the paths `src/com/ocajexam/tutorial/` and `src/com/ocajexam/tutorial/planets/`.

6. Determine the command-line arguments needed to compile the complete program. Compile the program, and debug where necessary.

7. Determine the command-line arguments needed to interpret the program. Run the program.
-

The standard output will read as follows:

```
$ Greetings, Universe!
Hello from Earth!
Hello from Mars!
Hello from Venus!
```

CERTIFICATION SUMMARY

This chapter discussed packaging, structuring, compiling, and interpreting Java code. The chapter started with a discussion on the importance of organizing your classes into packages as well as using the `package` and `import` statements to define and include different pieces of source code. Through the middle of the chapter, we discussed the key features of commonly used Java packages: `java.awt`, `javax.swing`, `java.net`, `java.io`, and `java.util`. We discussed the basic structure of a Java class. We then concluded the chapter by providing detailed information on how to compile and interpret Java source and class files and how to work with their command-line options. At this point, you should be able to (outside of an IDE) package, build, and run basic Java programs independently.

TWO-MINUTE DRILL

Understand Packages

- Packages are containers for classes.
- A package statement defines the directory path where files are stored.
- A package statement uses periods for delimitation.
- Package names should be lowercase and separated with underscores between words.
- Package names beginning with `java.*` and `javax.*` are reserved.
- There can be zero or one package statement per source file.
- An `import` statement is used to include source code from external classes.
- An `import` statement occurs after the optional package statement and before the class definition.

- An import statement can define a specific class name to import.
- An import statement can use an asterisk to include all classes within a given package.

Understand Package-Derived Classes

- The Java Abstract Window Toolkit API is included in the `java.awt` package and subpackages.
- The `java.awt` package includes GUI creation and painting graphics and images functionality.
- The Java Swing API is included in the `javax.swing` package and subpackages.
- The `javax.swing` package includes classes and interfaces that support lightweight GUI component functionality.
- The Java Basic Input/Output-related classes are contained in the `java.io` package.
- The `java.io` package includes classes and interfaces that support input/ output functionality of the file system, data streams, and serialization.'
- Java networking classes are included in the `java.net` package.
- The `java.net` package includes classes and interfaces that support basic networking functionality that is also extended by the `javax.net` package.
- Fundamental Java utilities are included in the `java.util` package.
- The `java.util` package and subpackages include classes and interfaces that support the Java Collections Framework, legacy collection classes, event model, date and time facilities, and internationalization functionality.

Understand Class Structure

- Naming conventions are used to make Java programs more readable and maintainable.
- Naming conventions are applied to several Java elements including class names, interface names, method names, instance and static variable names, parameter and local variable names, generic type parameter names, constant names, enumeration names, and package names.
- The preferred order of presenting elements in a class is data members, followed by constructors, followed by methods. Note that the inclusion of each type of element is optional.

Compile and Interpret Java Code

- The Java compiler is invoked with the `javac[.exe]` command.
- The `.exe` extension is optional on Microsoft Windows machines and is not present on UNIX-like systems.
- The compiler's `-d` command-line option defines where compiled class files I should be placed.
- The compiler's `-d` command-line option will include the package location if the class has been declared with a package statement.
- The compiler's `-classpath` command-line option defines directory paths in search of classes.
- The Java interpreter is invoked with the `java[.exe]` command.
- The interpreter's `-classpath` switch defines directory paths to use at runtime.
- The interpreter's `-D` command-line option allows for the setting of system property values.
- The interpreter's syntax for the `-D` command-line option is `-Dproperty=value`.
- The interpreter's `-version` command-line option is used to return the version of the JVM and exit.
- The `-h` command-line option can be applied either to the compiler or the interpreter to print out the tools usage information.

SELF TEST

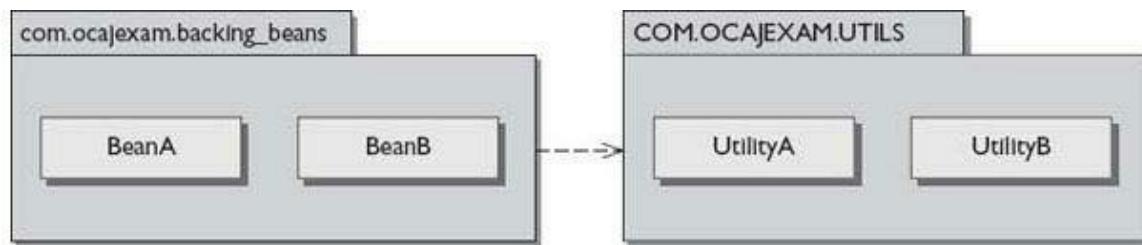
Understanding Packages

- 1.** Which two import statements will allow for the import of the `HashMap` class?
 - A. `import java.util.HashMap;`
 - B. `import Java.util.*;`
 - C. `import java.util.HashMap.*;`
 - D. `import java.util.hashMap;`
- 2.** Which statement would designate that your file belongs in the package `com.ocajexam.utilities`?
 - A. `pack com.ocajexam.utilities;`
 - B. `package Com.ocajexam.utilities.*`
 - C. `package com.ocajexam.utilities.*;`
 - D. `package com.ocajexam.utilities;`
- 3.** Which of the following is the only Java package that is imported by default?
 - A. `java.awt`
 - B. `java.lang`
 - C. `java.util`
 - D. `java.io`
- 4.** What Java-related features are new to J2SE 5.0?
 - A. Static imports
 - B. package and import statements
 - C. Autoboxing and unboxing
 - D. The enhanced for loop

Understand Package-Derived Classes

- 5.** The `JCheckBox` and `JComboBox` classes belong to which package?
 - A. `java.awt`
 - B. `javax.awt`
 - C. `java.swing`
 - D. `javax.swing`
- 6.** Which package contains the Java Collections Framework?
 - A. `java.io`
 - B. `java.net`
 - C. `java.util`
 - D. `java.utils`
- 7.** The Java Basic I/O API contains what types of classes and interfaces?
 - A. Internationalization
 - B. RMI, JDBC, and JNDI
 - C. Data streams, serialization, and file system
 - D. Collection API and data streams
- 8.** Which API provides a lightweight solution for GUI components?
 - A. AWT
 - B. Abstract Window Toolkit
 - C. Swing
 - D. AWT and Swing

- 9.** Consider the following illustration. What problem exists with the packaging? You may wish to reference [Appendix G](#) on the Unified Modeling Language (UML) for assistance.



- A. You can have only one class per package.
- B. Packages cannot have associations between them.
- C. Package com.ocajexam.backing_beans fails to meet the appropriate packaging naming conventions.
- D. Package COM.OCAJEXAM.UTILS fails to meet the appropriate packaging naming conventions.

Understand Class Structure

- 10.** When applying naming conventions, which Java elements should start with a capital letter and continue on using the camel case convention?
- A. Class names
 - B. Interface names
 - C. Constant names
 - D. Package names
 - E. All of the above
- 11.** When instantiating an object with generics, should angle brackets, box brackets, parentheses, or double-quotes be used to enclose the generic type? Select the appropriate answer.
- A. List<Integer> a = new ArrayList<Integer>();
 - B. List[Integer] a = new ArrayList[Integer]();
 - C. List{Integer} a = new ArrayList{Integer}();
 - D. List"Integer" a = new ArrayList"Integer"();
- 12.** When organizing the elements in a class, which order is preferred?
- A. Data members, methods, constructors
 - B. Data members, constructors, methods
 - C. Constructors, methods, data members
 - D. Constructors, data members, methods
 - E. Methods, constructors, data members

Compile and Interpret Java Code

- 13.** Which usage represents a valid way of compiling a Java class?
- A. java MainClass.class
 - B. javac MainClass
 - C. javac MainClass.source
 - D. javac MainClass.java
- 14.** Which two command-line invocations of the Java interpreter return the version of the interpreter?
- A. java -version
 - B. java --version

- C. `java -version ProgramName`
- D. `java ProgramName -version`

15. Which two command-line usages appropriately identify the classpath?

- A. `javac -cp /project/classes/ MainClass.java`
- B. `javac -sp /project/classes/ MainClass.java`
- C. `javac -classpath /project/classes/ MainClass.java`
- D. `javac -classpaths /project/classes/ MainClass.java`

16. Which command-line usages appropriately set a system property value?

- A. `java -Dcom.ocajexam.propertyValue=003 MainClass`
- B. `java -d com.ocajexam.propertyValue=003 MainClass`
- C. `java -prop com.ocajexam.propertyValue=003 MainClass`
- D. `java -D:com.ocajexam.propertyValue=003 MainClass`

SELF TEST ANSWERS

Understand Packages

1. Which two import statements will allow for the import of the `HashMap` class?

- A. `import java.util.HashMap;`
- B. `import Java.util.*;`
- C. `import java.util.HashMap.*;`
- D. `import java.util.hashMap;`

Answer:

- A and B. The `HashMap` class can be imported directly via `import java.util. HashMap` or with a wild card via `import java.util.*;`.
- C and D are incorrect. C is incorrect because the answer is a static import statement that imports static members of the `HashMap` class, and not the class itself. D is incorrect because class names are case-sensitive, so the class name `hashMap` does not equate to `HashMap`.

2. Which statement would designate that your file belongs in the package `com.ocajexam.utilities`?

- A. `pack com.ocajexam.utilities;`
- B. `package com.ocajexam.utilities.*`
- C. `package com.ocajexam.utilities.*;`
- D. `package com.ocajexam.utilities;`

Answer:

- D. The keyword `package` is appropriately used, followed by the package name delimited with periods and followed by a semicolon.
- A, B, and C are incorrect. A is incorrect because the word `pack` is not a valid keyword. B is incorrect because a package statement must end with a semicolon, and you cannot use asterisks in package statements. C is incorrect because you cannot use asterisks in package statements.

3. Which of the following is the only Java package that is imported by default?

- A. `java.awt`
- B. `java.lang`
- C. `java.util`
- D. `java.io`

Answer:

- B. The `java.lang` package is the only package that has all of its classes imported by default.
- A, C, and D are incorrect. The classes of packages `java.awt`, `java.util`, and `java.io` are not imported by default.

4. What Java-related features are new to J2SE 5.0?

- A. Static imports
- B. Package and `import` statements
- C. Autoboxing and unboxing
- D. The enhanced `for` loop

Answer:

- A, C, and D.** Static imports, autoboxing/unboxing, and the enhanced for loop are all new features of J2SE 5.0.
 - B** is incorrect because basic package and import statements are not new to J2SE 5.0.
-

Understand Package-Derived Classes

5. The `JCheckBox` and `JComboBox` classes belong to which package?

- A.** `java.awt`
 - B.** `javax.awt`
 - C.** `java.swing`
 - D.** `javax.swing`
-

Answer:

- D.** Components belonging to the Swing API are generally prefaced with a capital *J*. Therefore, `JCheckBox` and `JComboBox` would be part of the Java Swing API and not the Java AWT API. The Java Swing API base package is `javax.swing`.
 - A, B, and C** are incorrect. **A** is incorrect because the package `java.awt` does not include the `JCheckBox` and `JComboBox` classes since they belong to the Java Swing API. Note that the package `java.awt` includes the `CheckBox` class, as opposed to the `JCheckBox` class. **B** and **C** are incorrect because the package names `javax.awt` and `java.swing` do not exist.
-

6. Which package contains the Java Collections Framework?

- A.** `java.io`
 - B.** `java.net`
 - C.** `java.util`
 - D.** `java.utils`
-

Answer:

- C.** The Java Collections Framework is part of the Java Utilities API in the `java.util` package.
 - A, B, and D** are incorrect. **A** is incorrect because the Java Basic I/O API's base package is named `java.io` and does not contain the Java Collections Framework. **B** is incorrect because the Java Networking API's base package is named `java.net` and also does not contain the Collections Framework. **D** is incorrect because there is no package named `java.utils`.
-

7. The Java Basic I/O API contains what types of classes and interfaces?

- A.** Internationalization
 - B.** RMI, JDBC, and JNDI
 - C.** Data streams, serialization, and file system
 - D.** Collection API and data streams
-

Answer:

- C.** The Java Basic I/O API contains classes and interfaces for data streams, serialization, and the file system.
- A, B, and D** are incorrect because internationalization (i18n), RMI, JDBC, JNDI, and the

8. Which API provides a lightweight solution for GUI components?

- A. AWT
 - B. Abstract Window Toolkit
 - C. Swing
 - D. AWT and Swing
-

Answer:

- C. The Swing API provides a lightweight solution for GUI components, meaning that the Swing API's classes are built from pure Java code.
 - A, B, and D are incorrect. AWT and the Abstract Window Toolkit are one and the same and provide a heavyweight solution for GUI components.
-

9. Consider the following illustration. What problem exists with the packaging? You may wish to reference [Appendix G](#) on the Unified Modeling Language (UML) for assistance.

- A. You can have only one class per package.
 - B. Packages cannot have associations between them.
 - C. Package com.ocajexam.backing_beans fails to meet the appropriate packaging naming conventions.
 - D. Package com.OCAJEXAM.UTILS fails to meet the appropriate packaging naming conventions.
-

Answer:

- D. com.OCAJEXAM.UTILS fails to meet the appropriate packaging naming conventions. Package names should be lowercase. Note that package names should also have an underscore between words. However, the words in ocajexam are joined in the URL; therefore, excluding the underscore here is acceptable. The package name should read com.ocajexam.utils.
 - A, B, and C are incorrect. A is incorrect because being restricted to having one class in a package is ludicrous. There is no limit. B is incorrect because packages can and frequently do have associations with other packages. C is incorrect because com.ocajexam.backing_beans meets appropriate packaging naming conventions.
-

Understand Class Structure

10. When applying naming conventions, which Java elements should start with a capital letter and continue on using the camel case convention?

- A. Class names
 - B. Interface names
 - C. Constant names
 - D. Package names
 - E. All of the above
-

Answer:

- A and B. Class names and interface names should start with a capital letter and continue on using the camel case convention.
- C and D are incorrect. C is incorrect because constant names should be all capital letters

separated by underscores. **D** is incorrect because package names do not include capital letters, nor do they subscribe to the camel case convention.

11. When instantiating an object with generics, should angle brackets, box brackets, parentheses or double-quotes be used to enclose the generic type? Select the appropriate answer.

- A. `List<Integer> a = new ArrayList<Integer>();`
 - B. `List[Integer] a = new ArrayList[Integer]();`
 - C. `List{Integer} a = new ArrayList{Integer}();`
 - D. `List"Integer" a = new ArrayList"Integer"();`
-

Answer:

- A. Generics use angle brackets.
 - B, C, and D are incorrect. Box brackets, parentheses, and double quotes are not used to enclose the generic type.
-

12. When organizing the elements in a class, which order is preferred?

- A. Data members, methods, constructors
 - B. Data members, constructors, methods
 - C. Constructors, methods, data members
 - D. Constructors, data members, methods
 - E. Methods, constructors, data members
-

Answer:

- B. The preferred order in presenting elements in a class is to present the data members first, followed by constructors, followed by methods.
 - A, C, D, and E are incorrect. While ordering the elements in these manners will not cause any functional or compilation errors, it is not preferred.
-

Compile and Interpret Java Code

13. Which usage represents a valid way of compiling a Java class?

- A. `java MainClass.class`
 - B. `javac MainClass`
 - C. `javac MainClass.source`
 - D. `javac MainClass.java`
-

Answer:

- D. The compiler is invoked by the `javac` command. When compiling a Java class, you must include the filename, which houses the main classes including the `.java` extension.
 - A, B, and C are incorrect. A is incorrect because `MainClass.class` is bytecode that is already compiled. B is incorrect because `MainClass` is missing the `.java` extension. C is incorrect because `MainClass.source` is not a valid name for any type of Java file.
-

14. Which two command-line invocations of the Java interpreter return the version of the interpreter?

- A. `java -version`
- B. `java --version`

- C. `java -version ProgramName`
 - D. `java ProgramName -version`
-

Answer:

- A and C. The `-version` flag should be used as the first argument. The application will return the appropriate strings to standard output with the version information and then immediately exit. The second argument is ignored.
 - B and D are incorrect. B is incorrect because the `version` flag does not allow double dashes. You may see double dashes for flags in utilities, especially those following the GNU license. However, the double dashes do not apply to the `version` flag of the Java interpreter. D is incorrect because the `version` flag must be used as the first argument or its functionality will be ignored.
-

15. Which two command-line usages appropriately identify the classpath?

- A. `javac -cp /project/classes/ MainClass.java`
 - B. `javac -sp /project/classes/ MainClass.java`
 - C. `javac -classpath /project/classes/ MainClass.java`
 - D. `javac -classpaths /project/classes/ MainClass.java`
-

Answer:

- A and C. The option flag that is used to specify the classpath is `-cp` or `-classpath`.
 - B and D are incorrect because the option flags `-sp` and `-classpaths` are invalid.
-

16. Which command-line usages appropriately set a system property value?

- A. `java -Dcom.ocajexam.propertyValue=003 MainClass`
 - B. `java -d com.ocajexam.propertyValue=003 MainClass`
 - C. `java -prop com.ocajexam.propertyValue=003 MainClass`
 - D. `java -D:com.ocajexam.propertyValue=003 MainClass`
-

Answer:

- A. The property setting is used with the interpreter, not the compiler. The property name must be sandwiched between the `-D` flag and the equal sign. The desired value should immediately follow the equal sign.
 - B, C, and D are incorrect because `-d`, `-prop`, and `-D:` are invalid ways to designate a system property.
-



2

Programming with Java Statements

CERTIFICATION OBJECTIVES

- Understand Assignment Statements
- Create and Use Conditional Statements
- Create and Use Iteration Statements
- Create and Use Transfer of Control Statements
- ✓ Two-Minute Drill

Q&A Self Test

The language statements within software applications allow the proper sequence of execution and associated functionality to occur. The more statement types a software language includes, the more effective the language can be. With Java, for example, the ability to programmatically allowing a system to stay up and running by supporting the code with exception handling statements is an effective benefit. [Table 2-1](#) provides short definitions of the Java statement types defined in *The Java Language Specification, Third Edition*, by James Gosling, Bill Joy, Guy Steele, and Gilad Bracha (Addison-Wesley, 2005). The statements covered on the exam and in this chapter are accompanied by a checkmark. You can refer to the language specification for more details on the statements that are not on the exam.

TABLE 2-1 Java Statements

| Statement Name | Definition | On the Exam |
|--|--|-------------|
| The <code>assert</code> statement | Used to determine whether code is functioning as expected. When its expression is evaluated to false, an exception is thrown. | |
| The <code>break</code> statement | Used to exit the body of a <code>switch</code> statement or loop. | ✓ |
| The <code>case</code> statement | Used as part of the <code>switch</code> statement to execute statements when its value matches the <code>switch</code> statement's conditional value. | ✓ |
| The <code>continue</code> statement | Used to terminate the current iteration of a <code>do-while</code> , <code>while</code> , or <code>for</code> loop and continue with the next iteration. | ✓ |
| The <code>while</code> statement | Used for iteration based on a condition. | ✓ |
| The <code>do-while</code> statement | Used for iteration based on a condition. The body of the <code>do-while</code> statement is executed at least once. | ✓ |
| The <code>empty</code> statement | Used for trivial purposes where no functionality is needed. It is represented by a single semicolon. | |
| The expression statements | Used to evaluate expressions. See Table 2-2. | ✓ |
| The <code>for</code> loop statement | Used for iteration. Main components are an initialization part, an expression part, and an update part. | ✓ |
| The enhanced <code>for</code> loop statement | Used for iteration through an iterable object or array. | ✓ |
| The <code>if</code> statement | Used for the conditional execution of statements. | ✓ |
| The <code>if-then</code> statement | Used for the conditional execution of statements by providing multiple conditions. | ✓ |
| The <code>if-then-else</code> statement | Used for the conditional execution of statements by providing multiple conditions and fall-through when no conditions are met. | ✓ |
| The labeled statement | Used to give a statement a prefixed label. | |
| The <code>return</code> statement | Used to exit a method and return a specified value. | ✓ |
| The <code>switch</code> statement | Used for branching code based on conditions. | ✓ |
| The <code>synchronized</code> statement | Used for access control of threads. | |
| The <code>throw</code> statement | Used to throw an exception. | Chapter 9 |
| The <code>try-catch-finally</code> statement | Used for exception handling. | Chapter 9 |

To be an effective Java programmer, you must master the basic statements. Oracle knows this and has included complete coverage of the basic statements on the exam. This chapter will teach you how to recognize and code Java statements.

We begin by understanding fundamental statements. The Java programming language contains a variety of statement types. Even though the various statement types serve different purposes, those covered in this chapter can be grouped into four main categories: expression statements, conditional statements, iteration statements, and transfer of control statements.

Expression statements are used for the evaluation of expressions. The only expression statement required for this exam is the *assignment statement*. Assignment statements allow assignments to be performed on variables. *Conditional statements*, also known as decision statements, assist in directing the flow of control when a decision needs to be made. Conditional statements include the `if`, `if-then`, `if-then-else`, and `switch` statements. *Iteration statements* provide support in looping through blocks of code. Iteration statements include the `for` loop, the enhanced `for` loop, the `while` statement, and the `do-while` statement. *Transfer of control statements* provide a means of stopping or

interrupting the normal flow of control. Transfer of control statements include the `continue`, `break`, and `return` statements. Transfer of control statements are always seen within other types of statements.

The goal of this chapter is for you to gain the knowledge of when and how to use all of the necessary types of Java statements that will be included in the OCA Java Associate SE 7 Programmer (OCA) exam.

CERTIFICATION OBJECTIVE

Understand Assignment Statements

An assignment statement sets a value within a variable. All assignment statements are considered to be expression statements. And although no explicit exam objective exists for this, you'll need to have a basic knowledge of expression statements, and assignment statements in particular.

Let's start with the expression statement. Expression statements essentially work with expressions. Expressions in Java are anything that has a value or is reduced to a value. Typically, expressions evaluate to primitive types, such as in the case of adding two numbers—for example, `(1+2)`. Concatenating strings together with the concatenation `(+)` operator results in a string and is also considered an expression. All expressions can be used as statements; the only requirement is that they end with a semicolon. [Table 2-2](#) shows examples of some typical expression statements and where they are discussed in this book.

TABLE 2-2 Expression Statements

| Expression Statement | Expression Statement Example | Coverage |
|----------------------|--------------------------------|-----------|
| Assignment | <code>variableName = 7;</code> | Chapter 2 |
| Pre-increment | <code>++variableName;</code> | Chapter 4 |
| Pre-decrement | <code>--variableName;</code> | Chapter 4 |
| Post-increment | <code>variableName++;</code> | Chapter 4 |
| Post-decrement | <code>variableName--;</code> | Chapter 4 |
| Method invocation | <code>performMethod();</code> | Chapter 5 |
| Object creation | <code>new ClassName();</code> | Chapter 4 |

The Assignment Expression Statement

Assignment expression statements, commonly known simply as assignment statements, are designed to assign values to variables. All assignment statements must be terminated with a semicolon. The statement's ability to store information in variables provides the main characteristic of usefulness to computer applications.

Here is the general usage of the assignment statement:

variable = value;

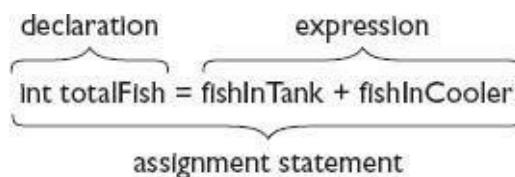
Given the declaration of an integer primitive, let's look at an assignment in its most basic form. The assignment statement comprises three key elements. On the left is the variable that will be associated

with the memory and type necessary to store the value. On the right is a literal value. If an expression is on the right, such as (1+2), it must be evaluated down to its literal value before it can be assigned. Lastly, an equal sign resides between the variable and value of an assignment statement. Here are some example statements:

```
int variableName; // Declaration of an integer  
variableName = 100; // Assignment expression statement
```

As long as the application is running and the object in which the variable exists is still alive (that is, available in memory), the value for variableName will remain the assigned value, unless it is explicitly changed with another expression statement. The statement, illustrated in [Figure 2-1](#), combines a declaration, an expression, and an assignment statement. In addition, it uses the values stored from previous assignment statements.

FIGURE 2-1 Combined statements



```
int fishInTank = 100; int fishInCooler = 50;  
int totalFish = fishInTank + fishInCooler;
```

Trying to save an invalid literal to a declared primitive type variable will result in a compiler error. For example, the compilation error “Exception in thread “xxxx” java.lang.RuntimeException: Uncompilable source code - incompatible types...” would appear for the following code:

```
int totalFish = "INVALID_USE_OF_A_STRING";
```

For more information about working with primitives, see [Chapter 4](#).

CERTIFICATION OBJECTIVE

Create and Use Conditional Statements

Exam Objective 3.4 Create if and if/else constructs

Exam Objective 3.5 Use a switch statement

Conditional statements are used when there is a need for determining the direction of flow based on conditions. Conditional statements include the `if`, `if-then`, `if-then-else`, and `switch` statements. The conditional statements represented in [Table 2-3](#) will appear on the exam.

TABLE 2-3 Conditional Statements

| Formal Name | Keywords | Expression Types | Example |
|--------------|--|--|--|
| if | if, else (optional) | boolean | if (value == 0) {} |
| if-then | if, else if, else if (optional) | boolean | if (value == 0) {} else if (value == 1) {} else if (value >= 2) {} |
| if-then-else | if, else if, else if (optional), else | boolean | if (value == 0) {} else if (value >=1) {} else {} |
| switch | switch, case, default (optional), break (optional) | char, byte, short, int, Character, Byte, Short, String, Integer, enumeration types | switch (100) { case 100: break; case 200: break; case 300: break; default: break; } |

The if Conditional Statement

The `if` statement is designed to conditionally execute a statement or conditionally decide between a choice of statements. The `if` statement will execute only one statement upon the condition, unless braces are supplied. Braces, also known as curly brackets, allow for multiple enclosed statements to be executed. This group of statements is also known as a *block*. The expression that is evaluated within `if` statements must evaluate to a boolean value, or the application will not compile. The `else` clause is optional and may be omitted.

INSIDE THE EXAM

The if, if-then, and if-then-else Statements

The distinction between the `if`, `if-then`, and `if-then-else` statements may seem blurred. This is partially because the `then` keyword used in some other programming languages is not used in Java, even though the Java constructs are formally known as `if-then` and `if-then-else`. Let's clarify some confusing points about the `if`-related statements by providing some facts.

- The `if` statement allows for the optional use of the `else` branch. This may be a little confusing since you may expect the `if` statement to stand alone without any branches.
- The `if-then` statement must have at least one `else if` branch. Optionally, an unlimited amount of `else if` branches may be included. You cannot use an `else` statement in an `if-then` statement or the statement would be considered an `if-then-else` statement.
- The `if-then-else` statement must have at least one `else if` branch. The `else if` branch is not optional, because if it were not present, the statement would be considered to be an `if` statement that includes the optional `else` branch.

Here's the general usage of the `if` statement:

```
if (expression)
    statementA;
else
    statementB;
```

In the following example, we look at the most basic structure of an if statement. Here, we check to see if a person (*isFisherman*) is a fisherman, and if so, the expression associated with the if statement would evaluate to true. Because it is true, the example's fishing trip value (*isFishingTrip*) is modified to true. No action would be taken if the *isFisherman* expression evaluated to false.

```
boolean isFisherman = true;
boolean isFishingTrip = false;
if (isFisherman)
    isFishingTrip = true;
```

Let's change the code a little bit. Next, you will see that a fishing trip will occur only if there are one or more fishermen, as the expression reads (*fishermen >= 1*). See [Chapter 3](#) for more details on relationship operators (for example, *<*, *<=*, *>*, *>=*, *==*, *!=*). You also see that when "one or more fishermen" evaluates to true, a block of statements will be executed.

```
int fishermen = 2;
boolean isFishingTrip = false;
if (fishermen >= 1) {
    isFishingTrip = true;
    System.out.print("Going Fishing!");
}
$ Going Fishing!
```

Executing statements in relationship to false conditions is also common in programming. In the following example, when the expression evaluates to false, the statement associated with the else part of the if statement is executed:

```
boolean isFisherman = false;
if (isFisherman) System.out.println("Going fishing!");
else System.out.println("I'm doing anything but fishing!");
$ I'm doing anything but fishing!
```

The if-then Conditional Statement

The if-then conditional statement—also known as the if else if statement—is used when multiple conditions need to flow through a decision-based scenario.

Here's the general usage of the if-then statement:

```
if (expressionA)
    statementA;
else if (expressionB)
    statementB;
```

The expressions must evaluate to boolean values. Each statement may optionally be a group of statements enclosed in braces.

INSIDE THE EXAM

The if Statement

The most important thing you need to remember about the expression in the `if` statement is that it can accept any expression that returns a boolean value, and once the expression evaluates to `true` all subsequent `else` statements are skipped. Note, too, that even though relational operators (such as `>=`) are commonly used, assignment statements are always allowed.

Review and understand the following code examples:

```
boolean b;
boolean bValue = (b = true); // Evaluates to true
if (bValue) System.out.println("TRUE");
else System.out.println("FALSE");
if (bValue = false) System.out.println("TRUE");
else System.out.println("FALSE");
if (bValue == false) System.out.println("TRUE");
else System.out.println("FALSE");
$ TRUE
$ FALSE
$ TRUE
```

You also need to know that the assignment statements of all primitives will return their primitive values. So, if it's not an assignment of a boolean type, the return value will not be boolean. As such, the following code will not compile:

```
int i; // Valid declaration
int iValue = (i=1); // Valid evaluation to int
/* Fails here as boolean value is
expected in the expression */
if (iValue) {};
```

Similarly, this code will not compile:

```
/* Fails here as boolean value is
expected in the expression */
if (i=1) {};
```

The compile error will look like this:

```
Error: incompatible types; found:
int, required: boolean
```

Let's look at another example. (For those not familiar with surf fishing, when fishing off the beach, a lead pyramid-shaped sinker is used to keep the line on the bottom of the ocean.) In the following code segment, conditions are evaluated matching the appropriate `pyramidSinker` by weight against

the necessary tide:

```
int pyramidSinker = 3;
System.out.print("A pyramid sinker that weighs " + pyramidSinker
    + "ounces is ");
if (pyramidSinker == 2)
    System.out.print("used for a slow moving tide. ");
else if (pyramidSinker == 3)
    System.out.print("used for a moderate moving tide. ");
else if (pyramidSinker == 4)
    System.out.print("used for a fast moving tide. ");
$ A pyramid sinker that weighs 3 ounces is used for a moderate
moving tide.
```

We used the string concatenation (+) operator in this example. While the functionality is straightforward, you'll find more information about its behavior in [Chapter 3](#).

e x a m watch

The if family of statements evaluate expressions that must result in a boolean type where the value is true or false. Be aware that an object from the Boolean wrapper class is also allowed, because it will go through unboxing in order to return the expected type. Unboxing is the automatic production of its primitive value in cases where it is needed. The following code demonstrates the use of a Boolean wrapper class object within the expression of an if statement:

```
Boolean wrapperBoolean = new Boolean ("true");
/* Valid */
boolean primitiveBoolean1 = wrapperBoolean.booleanValue();
/* Valid because of unboxing */
boolean primitiveBoolean2 = wrapperBoolean;
if (wrapperBoolean)
    System.out.println("Works because of unboxing");
```

For more information on autoboxing and unboxing, see [Chapter 4](#).

The if-then-else Conditional Statement

As with the if and if-then statements, all expressions must evaluate to true or false as the expected primitive type is boolean. The main difference in the if-then-else statement is that the code will fall through to the final stand-alone else when the expression fails to return true for any condition. Each statement may optionally be a group of statements enclosed in braces. There is no limit to the number of else if clauses.

Here's the general usage of the if-then-else statement:

```
if (expressionA)
    statementA;
else if (expressionB)
    statementB;
else if (expressionC)
    statementC;
...
else
    statementZZ;
```

In the following code listing, the method `getCastResult()` represents the efforts of a fisherman casting his line out into the ocean. The return value will be a `String` of value “fish,” “shark,” or “skate” and in this application the value is stored into the `resultOfCast` variable. This `String` value is evaluated against the stipulated string passed into the `equals` method. If the criteria are met for any `if` or `else if` condition, the associated block of code is executed; otherwise the code related to the final `else` is executed. This code clearly demonstrates a complete `if-then-else` scenario.

```
...
private FishingSession fishingSession = new FishingSession();
...
public void castForFish() {
    fishingSession.setCatch();
    String resultOfCast = fishingSession.getCastResult();
    if (resultOfCast.equals("fish")) {
        Fish keeperFish = new Fish();
        keeperFish = fishingSession.getFishResult();
        String type = keeperFish.getTypeOfFish();
        System.out.println("Wahoo! Keeper fish: " + type);
    } else if (resultOfCast.equals("shark")) {
        System.out.println("Need to throw this one back!");
    } else if (resultOfCast.equals("skate")) {
        System.out.println("Yuck, Leo can take this one off the
            hook!");
    } else {
        System.out.println("Darn, no catch!");
    }
}
...
$ Wahoo! Keeper fish: Striped Bass
```

Note that the `Fish` class and associated methods are deliberately not shown since the scope of this example is the `if-then-else` scenario only.

on the job *If abrupt termination occurs (for example, due to an overflow error) during the evaluation of the conditional expression within an `if` statement, then all subsequent `if-then` (that is, `else if`) and `if-then-else` (that is, `else`) statements will end abruptly as well.*

The switch Conditional Statement

The `switch` conditional statement is used to match the value from a `switch` statement expression against a value associated with a `case` keyword. Once matched, the enclosed statement(s) associated

with the matching case value are executed and subsequent case statements are executed, unless a break statement is encountered. The break statements are optional and will cause the immediate termination of the switch conditional statement.



When two case statements within the same switch statement have the same value, a compiler error will be thrown.

```
switch (intValue) {
    case 200: System.out.println("Case 1");
    /* Compiler error, Error: duplicate case label */
    case 200: System.out.println("Case 2");
}
```

The expression of the switch statement must evaluate to byte, short, int, or char. Wrapper classes of type Byte, Short, Int, and Character are also allowed since they are automatically unboxed to primitive types. Enumerated types (that is, enum) are permitted as well. Additionally, Java SE 7 added support for evaluation of the String object in the expression.

Here's the general usage of the switch statement:

```
switch (expression) {
    case valueA:
        // Sequences of statements
        break;
    case valueB:
        // Sequences of statements
        break;
    default:
        // Sequences of statements
    ...
}
```

Let's take a look at a complete switch conditional statement example. In the following generateRandomFish method, we use a random number generator to produce a value that will be used in the switch expression. The number generated will be 0, 1, 2, or 3. The switch statement will use the value to match it to the value of a case statement. In the example, a String with the name randomFish will be set depending on the case matched. The only possible value that does not have a matching case statement is the number 3. Therefore, this condition will be handled by the default statement. Whenever a break statement is hit, it will cause immediate termination of the switch statement.

```
public String generateRandomFish() {  
    String randomFish;  
    Random randomObject = new Random();  
    int randomNumber = randomObject.nextInt(4);  
    switch (randomNumber) {  
        case 0:  
            randomFish = "Blue Fish";  
            break;  
        case 1:  
            randomFish = "Red Drum";  
            break;  
        case 2:  
            randomFish = "Striped Bass";  
            break;  
        default:  
            randomFish = "Unknown Fish Type";  
            break;  
    }  
    return randomFish;  
}
```

The case statements can be organized in any manner. The default case is often listed last for code readability. Remember that without break statements, the switch block will continue with its fall-through, from the point that the condition has been met. The following code is a valid switch conditional statement that uses an enumeration type for its expression value:

```
private enum ClamBait {FRESH, SALTED, ARTIFICIAL}  
...  
ClamBait bait = ClamBait.SALTED;  
switch (bait) {  
default:  
    System.out.println("No bait");  
    break;  
case FRESH:  
    System.out.println("Fresh clams");  
    break;  
case SALTED:  
    System.out.println("Salted clams");  
    break;  
case ARTIFICIAL:  
    System.out.println("Artificial clams");  
    break;  
}
```

Knowing what you can and cannot do with switch statements will help expedite your development efforts.

SCENARIO & SOLUTION

| | |
|---|--|
| To ensure your statement is bug-free, which type of statements should you include within the switch? | Both <code>break</code> statements and the <code>default</code> statement are commonly used in the switch. Forgetting these statements can lead to improper fall-throughs or unhandled conditions. Note that many bug-finding tools will flag missing <code>default</code> statements. |
| You want to use a range in a <code>case</code> statement (for instance, <code>case 7-35</code>). Is this a valid feature in Java, as it is with other languages? | Ranges in <code>case</code> statements are <i>not</i> allowed. Consider setting up a condition in an <code>if</code> statement. For example: <code>if (x >=7 && x <=35) {}</code> |
| You want to use the <code>switch</code> statement, using <code>String</code> values where the expression is expected, as is possible with other languages. Is this a valid feature in Java? | Strings are not valid at the decision point for <code>switch</code> statements prior to Java SE 7. For Java SE 6 and earlier, consider using an <code>if</code> statement instead. For example: <code>if (strValue.equals("S1")) {}</code> |

EXERCISE 2-1

Evaluating the String Class in the switch Statement

Build a small program that demonstrates the use of the `String` class being evaluated in a `switch` statement. Follow the model that is used for the `switch` statement with the other data types, and your application will run just fine. Make sure you are using Java SE 7.

CERTIFICATION OBJECTIVE

Create and Use Iteration Statements

Exam Objective 5.4 Compare loop constructs

Exam Objective 5.2 Create and use for loops including the enhanced for loop

Exam Objective 5.1 Create and use while loops

Exam Objective 5.3 Create and use do/while loops

Iteration statements are used when there is a need to iterate through pieces of code. Iteration statements include the `for` loop, enhanced `for` loop, and the `while` and `do-while` statements. The `break` statement is used to exit the body of any iteration statement. The `continue` statement is used to terminate the current iteration and continue with the next iteration. The iteration statements detailed/compared in [Table 2-4](#) will appear on the exam.

TABLE 2-4 Iteration Statements

| Formal Name | Keywords | Main Expression Components | Example |
|-------------------|--|--|-------------------------------|
| for loop | for, break (optional), continue (optional) | Initializer, expression, update mechanism | for (i=0; i<j; i++) {} |
| Enhanced for loop | for, break (optional), continue (optional) | Element, array, or collection | for (Fish f : listOfFish) {}; |
| while | while, break (optional), continue (optional) | Boolean expression | while (value == 1) { } |
| do-while | do,while, break (optional), continue (optional) | Boolean expression | do { } while (value == 1); |

The for Loop Iteration Statement

The `for` loop statement is designed to iterate through code. It has main parts that include an initialization part, an expression part, and an iteration part. The initialization does not need to declare a variable as long as the variable is declared before the `for` statement. So `int x = 0;` and `x=0;` are both acceptable in the initialization part. Be aware, though, that the scope of the variable declared within the initialization part of the `for` loop ends once the `for` loop terminates. The expression within the `for` loop statement must evaluate to a boolean value. The iteration, also known as the update part, provides the mechanism that will allow the iteration to occur. A basic update part is represented as `i++;`.

Here's the general usage of the `for` statement:

```
for ( initialization; expression; iteration) {
    // Sequence of statements
}
```

The following is an example of a basic `for` loop where the initialization variable is declared outside the `for` loop statement:

```
int m;
for (m = 1; m < 5; m++) {
    System.out.print("Marker " + m + ", ");
}
System.out.print("Last Marker " + m + "\n");
$ Marker 0, Marker 1, Marker 2, Marker 3, Marker 4, Last Marker 5
```

The following is a similar example, but with the variable declared in the `for` loop:

```
for (int m = 1; m < 5; m++) {
    System.out.print("Marker " + m + ", ");
}
System.out.print("Last Marker " + m + "\n");
```

Declaring the initialize variable in the `for` loop is allowed and is the common approach. However,

you can't use the variable once you have exited the loop. The following will result in a compilation error:

```
for (int m = 1; m < 5; m++) {  
    System.out.print("Marker " + m + ", ");  
}  
System.out.print("Last Marker " + m + "\n"); // m is out of  
scope  
# Error: variable m not found in class [ClassName].
```

INSIDE THE EXAM

Exposing Corner Cases with Your Compiler

The exam designers were not satisfied with simply validating your knowledge of the fundamental Java material. They took the time to work in corner cases as well as modify the structure of the code in such a slight manner that it appears to be correct but is not. When you work through the examples in this book, take the time to modify things a bit, intentionally introducing errors, to see how the compiler reacts. Your ability to think like the compiler will help you score higher on the exam.

Third-party developers of Java development kits can define their own text for compiler error messages. Where they will likely try to model the messages provided by Oracle's JDK, sometimes care will be taken to make the messages more precise. Consider generating compiler errors with the latest Oracle JDK compiler, as well as a compiler provided by an integrated development environment (IDE) such as the specific Eclipse SDK. Compare the similarities and differences.

The Enhanced for Loop Iteration Statement

The enhanced for loop is used to iterate through an array, a collection, or an object that implements the interface iterable. The enhanced for loop is also commonly known as the for each loop and the for in loop. Iteration occurs for each element in the array or iterable class. Remember that the loop can be terminated at any time by the inclusion of a break statement. And as with the other iteration statements, the continue statement will terminate the current iteration and start with the next iteration.

Here's the general usage of the for statement:

for (type variable : collection) statement-sequence

The following code segment demonstrates how a for loop can easily dump out the contents of an array. Here, the enhanced for loop iterates over each hook integer in the array hookSizes. For each iteration, the hook size is printed out.

```
int hookSizes[] = { 1, 1, 1, 2, 2, 4, 5, 5, 5, 6, 7, 8, 8, 9 };  
for (int hook: hookSizes) System.out.print(hook + " ");  
$ 1 1 1 2 2 4 5 5 5 6 7 8 8 9
```

The enhanced `for` loop is frequently used for searching through items in a collection. Here, the enhanced `for` loop iterates over each hook `Integer` in the collection `hookSizesList`. For each iteration, the hook size is printed out. This example demonstrates the use of collections and generics.

```
Integer hookSizeList;  
ayList<Integer> hookSizesList = new ArrayList<Integer>();  
hookSizesList.add(1);  
hookSizesList.add(4);  
hookSizesList.add(5);  
for (Integer hook : hookSizesList) System.out.print(hook + " ");  
$ 1 4 5
```

See *Java Generics and Collections* by Maurice Naftalin and Philip Wadler (O'Reilly, 2006) for comprehensive coverage of the Generics and Collections frameworks.

EXERCISE 2-2

Iterating Through an `ArrayList` While Applying Conditions

This exercise will have you iterating through an `ArrayList` of floats. Specifically, this exercise will have you printing out only the legal sizes of keeper fish.

1. Create an `ArrayList` of floats called `fishLengthList`. This list will represent the sizes of a few striped bass.
2. Add the following floats to the list: 10.0, 15.5, 18.0, 29.5, 45.5. These numbers represent the length in inches of the bass.
3. Iterate through the list, printing out only the numbers larger than the required length. Assume the required length is 28 inches.

To gain more knowledge on the `ArrayList` class, see [Chapter 4](#).

on the job *Most IDEs support customizable formatting that can often be applied by selecting a format option from a menu. Using an IDE to ensure formatting is properly and consistently applied is a good idea. A popular Java code beautifier that is available as a plug-in to many tools is Jalopy: <http://jalopy.sourceforge.net/jalopy/manual.html>.*

The `while` Iteration Statement

The `while` statement is designed to iterate through code. The `while` loop statement evaluates an expression and executes the `while` loop body only if the expression evaluates to true. There is typically an expression within the body that will affect the result of the expression.

Here's the general usage of the `while` statement:

```
while (expression) {  
    // Sequences of statements  
}
```

The following code example demonstrates the use of the `while` statement. Here, a fisherman will continue fishing until his fish limit has been reached. Specifically, when the `fishLimit` variable within the body of the `while` statement reaches 10, the fisherman's session will be set to inactive. Since the `while` statement demands that the session be active, its loop will terminate upon the change.

```
fishingSession.setSession("active");
/* WHILE STATEMENT */
while (fishingSession.getSession().equals("active")) {
    castForFish(); // Updates fishLimit instance variable
    if (fishLimit == 10) {
        fishingSession.setSession("inactive");
    }
}
```

 **Various formatting styles can be followed when formatting your code. Formatting considerations include indentation, white space usage, line wrapping, code separation, and braces handling. You should select a style and maintain it throughout your code. For demonstration purposes, here are two distinct ways that braces are handled:**

K&R style braces handling:

```
while (x==y) {
    performSomeMethod();
}
```

Allman style brace handling:

```
while (x==y)
{
    performSomeMethod();
}
```

The do-while Iteration Statement

The `do-while` statement is designed to iterate through code. It is very similar to the `while` loop statement, except that it always executes the body at least once. The `do-while` loop evaluates an expression and continues to execute the body only if it evaluates to true. Typically, an expression within the body will affect the result of the expression.

Here's the general usage of the `do-while` statement:

```
do {
    // Sequence of statements
} while (expression)
```

EXERCISE 2-3

Performing Code Refactoring

In the following code example, we want to make sure the fisherman gets at least one cast in. Although this appears to make logical sense, you always need to think about corner cases. What if a fox steals the fisherman's bait before he gets a chance to cast? In this case, the `piecesOfBait` variable would equal zero, but the fisherman would still cast as the body of the `do-while` loop is guaranteed at least one iteration. See if you can refactor this code with a `while` statement to avoid the possible condition of casting with no bait.

```
fishngSession.setSession("active");
int piecesOfBait = 5;
piecesOfBait = 0; // Fox steals the bait!
do {
    castForFish();
    /* Check to see if bait is available */
    if (fishngSession.isBaitAvailable() == false) {
        /* Place a new piece of bait on the hook */
        fishngSession.setBaitAvailable(true);
        piecesOfBait--;
    }
} while (piecesOfBait != 0);
```

SCENARIO & SOLUTION

| | |
|--|--|
| You want to iterate though a collection. Which iteration statement would be the best choice? | You will need to use the enhanced <code>for</code> loop statement. |
| You want to execute a statement based on the result of a <code>boolean</code> expression. Which conditional statement would be the best choice? | You will need to use the <code>if</code> statement. |
| You want to provide conditional cases in relationship to enumeration values. What conditional statement would be your only choice? | You will need to use the <code>switch</code> statement. |
| You want to execute a block of statements and then iterate through the block based on a condition. What iteration statement would be your only choice? | You will need to use the <code>do-while</code> statement. |
| You want to exit a case statement permanently. What transfer of control statement would you choose? | You will need to use the <code>break</code> statement. |

Selecting the right statement types during development can make coding your algorithms easier. Proper statement selection will also promote the ease of software maintenance efforts if the code ever needs to be modified. It's important to realize that statements are used for different purposes, and one particular type of statement cannot solve all development needs. You will find it not uncommon to use a combination of statement types to implement the code for many algorithms. Having a strong foundation of what the main purposes are of the different types of statements will assist you when you need to use them together.

EXERCISE 2-4

Knowing Your Statement-Related Keywords

[Table 2-5](#) represents all of the statement-related Java keywords. This exercise will allow you to use the table to assist you in deducing the keywords you might see while using the various types of statements.

TABLE 2-5 Java EE 5 Keywords

| Java Keywords | | | | |
|---------------|----------|---------|--------|-------|
| break | continue | else | if | throw |
| case | default | finally | return | try |
| catch | Do | for | switch | while |

Let's start the exercise.

1. List the primary keywords you may see in conditional statements.
2. List the primary keywords you may see in iteration statements.
3. List the primary keywords you may see in transfer of control statements.
4. Bonus: List the primary keywords you may see in exception handling statements.

CERTIFICATION OBJECTIVE

Create and Use Transfer of Control Statements

Exam Objective 5.5 Use break and continue

Transfer of control statements include the `break`, `continue`, and `return` statements. We've mentioned these statements in the previous sections, but we cover them directly here. In short, transfer of control statements provide a means to stop or interrupt the normal flow of control. They are always used within other types of statements. A transfer of control statement always work with the labeled statement. We'll examine working with the labeled statement after first looking at the most common way of using `break`, `continue`, and `return` statements.

The `break` Transfer of Control Statement

The `break` statement is used to exit or force an abrupt termination of the body of the `switch` conditional statement as well as the body of the `do`, `for` loop; enhanced `for` loop; and `while` and `do-while` iteration statements.

The general usage of the `break` statement is simple:

```
break;
```

In the following example, the `for` loop is completely exited when the `break` statement is called.

The break statement is called when the hours allowed fishing exceed the total hours fishing. In short, the method prints a statement for every hour allowed to be fished in a one-day period.

```
public void fishingByHour() {  
    int totalHoursFishing = 0;  
    int hoursAllowedFishing = 4;  
    for (int i = 1; i < 25; ++i) {  
        totalHoursFishing = ++totalHoursFishing;  
        if (totalHoursFishing > hoursAllowedFishing)  
            break;  
        System.out.println("Fishing for hour" + i + ".");  
    }  
}
```

Here's the result of executing this method:

```
Fishing for hour 1.  
Fishing for hour 2.  
Fishing for hour 3.  
Fishing for hour 4.
```

The continue Transfer of Control Statement

The continue statement is used to terminate the current iteration of a do, for loop; enhanced for loop; or while or do-while loop and continue with the next iteration.

The general usage of the continue statement is also simple:

```
continue;
```

In the following example, the continue statement (when reached) immediately changes the flow to the next iteration of the for statement. The method prints a statement for each day allowed to go camping or camping and fishing. The fishing days start from day one and are defined by daysAllowedFishing. The continue statement is reached on days 1, 2, and 3, causing the rest of the iteration in the for loop to be skipped. On subsequently days, the continue statement is not reached, allowing the rest of the for loop to be exercised. The print statements are applied as reached.

```
public void activitiesByDay() {  
    totalDaysCamping = 0;  
    int daysAllowedFishing = 5;  
    for (int i = 1; i < 8; ++i) {  
        System.out.print("\nDay " + i + ": camping ");  
        totalDaysCamping = ++totalDaysCamping;  
        if (totalDaysCamping > daysAllowedFishing)  
            continue;  
        System.out.print("and fishing")  
    }  
}
```

Here's the result of executing this method:

```
Day 1: camping and fishing
Day 2: camping and fishing
Day 3: camping and fishing
Day 4: camping and fishing
Day 5: camping and fishing
Day 6: camping
Day 7: camping
```

The return Transfer of Control Statement

The `return` statement is used to exit a method and optionally return a specified value as an expression. A `return` statement with no `return` value must have the keyword `void` in the method declaration.

Here's the general usage of the `return` statement:

```
return [expression];
```

In the following example, the `getTotalFishType` method returns the value of the `fishTypesTotal` int primitive that was computed in the methods assignment statement. The `getTotalCaughtFish` method returns an int as well; however, in this case, the expression is inline. In both cases, the `int` keyword is provided in the method declaration, as necessary.

```
public int getTotalFishTypes
    (int saltWaterFishTotal, int freshWaterFishTotal, int brack-
ishFishTotal) {
    int fishTypesTotal = saltWaterFishTotal + freshWaterFishTotal
        + brackishFishTotal;
    return fishTypesTotal;
}

public int getTotalCaughtFish (int keeperFish, int throwBack-
Fish) {
    return keeperFish + throwBackFish;
}
```

on the job *If a `return` statement is the last statement in a method, and the method doesn't return anything, the `return` statement is optional. In this optional case, the `return` statement is typically not used.*

The labeled Statement

The labeled statement is used to give a statement a prefixed label. It can be used in conjunction with the `continue` and `break` statements. Use labeled statements sparingly; you should use them over other approaches only on a few occasions.

The general usage of the labeled statement is the addition of a label followed by a colon with the appropriate statement immediately following it:

```
labelIdentifier:
```

Statement (such as a for loop)

Here are the general usages of the `continue` and `break` statements in conjunction with the labeled statement:

`break labelIdentifier;`

and

`continue labelIdentifier;`

Let's look at a simple example of each. In the following example, the flow is transferred from the labeled `break` statement to the end of the labeled `myBreakLabel` outer loop.

```
public void labeledBreakTest() {  
    myBreakLabel:  
    while (true) {  
        System.out.println("While loop 1");  
        while (true) {  
            System.out.println("While loop 2");  
            while (true) {  
                System.out.println("While loop 3");  
                break myBreakLabel;  
            }  
        }  
    }  
}
```

Here's the result of executing this method:

```
While loop 1  
While loop 2  
While loop 3
```

In the following example, the flow is transferred from the labeled `continue` statement to the labeled `myContinueLabel` outer loop. Note that the fourth `while` loop is not reachable due to the `continue` statement.

```

public void labeledContinueTest() {
    myContinueLabel:
    while (true) {
        System.out.println("While loop 1");
        while (true) {
            System.out.println("While loop 2");
            while (true) {
                System.out.println("While loop 3");
                continue myContinueLabel;
                while (true)
                    System.out.println("While loop 4");
            }
        }
    }
}

```

The result of executing this method is the printing of the first three statements continuously repeated, as shown next:

```

While loop 1
While loop 2
While loop 3
While loop 1
While loop 2
While loop 3
While loop 1
While loop 2
While loop 3
...

```

exam watch

It is unlikely that you will see labeled statements on the exam. This coverage was simply intended for completeness of the transfer of control statement features.

CERTIFICATION SUMMARY

This chapter on fundamental statements discussed details related to the fundamental statement types. By studying this chapter, you should now be able to recognize and develop the following types of statements:

- Expression statements, with a focus on the assignment statement
- Conditional statements (`if`, `if-then`, `if-then-else`, and `switch`)
- Iteration statements (`for`, enhanced `for`, `while`, and `do-while`)
- Transfer of control statements (`continue`, `break`, and `return`)

It's important to note that throughout a Java developer's career, each one of these statement types will be seen and used quite frequently. At this point, you should be well prepared for exam questions covering Java statements.

TWO-MINUTE DRILL

Understand Assignment Statements

- Assignment statements assign values to variables.
- Assignment statements that do not return boolean types will cause the compiler to report an error when used as the expression in an `if` statement.
- Trying to save an invalid literal to a declared primitive type variable will result in a compiler error.

Create and Use Conditional Statements

- Conditional statements are used for determining the direction of flow based on conditions.
- Types of conditional statements include the `if`, `if-then`, `if-then-else`, and `switch` statements.
- The default case statement can be placed anywhere in the body of the `switch` statement.
- The expressions used in `if` statements must evaluate to boolean values, or the application will fail to compile.
- Boolean wrapper classes are allowed as expressions in `if` statements because they are unboxed. Remember that unboxing is the automatic production of primitive values from their related wrapper classes when the primitive value is required.

Create and Use Iteration Statements

- Iteration statements are designed for iterating through pieces of code.
- Iteration statements include the `for` loop, enhanced `for` loop, and the `while` and `do-while` statements.
- The `for` loop statement has main components that include an initialization part, an expression part, and an update part.
- The enhanced `for` loop statement is used for iteration through an iterable object or array.
- The `while` loop statement is used for iteration based on a condition.
- The `do-while` statement is used for iteration based on a condition. The body of this statement is always executed at least once.

Create and Use Transfer of Control Statements

- Transfer of control statements interrupt or stop the flow of execution.
- Transfer of control statements include `continue`, `break`, and `return` statements.
- The `continue` statement is used to terminate the current iteration of a `do`, `for` loop; enhanced `for` loop; and `while` or `do-while` loop and continue with the next iteration.
- The `break` statement is used to exit or force an abrupt termination of the body of the `switch` conditional statement as well as the body of the `do`, `for` loop, enhanced `for` loop, `while` and `do-while` iteration statements.
- The `return` statement is used to exit a method and may return a specified value.

- The labeled statement is used to give a statement a prefixed label. It is used with the continue and break statements.
- A block is a sequence of statements within braces—for example, {int x=0; int y=1}.

SELF TEST

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

Understand Assignment Statements

- 1.** Which is not a type of statement?
 - Conditional statement
 - Assignment statement
 - Iteration statement
 - Propagation statement
- 2.** What type of statement is the following equation: $y = (m*x) + b$?
 - Conditional statement
 - Assignment statement
 - Assertion statement
 - Transfer of control statement
- 3.** Which statements correctly declare boolean variables?
 - Boolean isValid = true;
 - boolean isValid = TRUE;
 - boolean isValid = new Boolean (true);
 - boolean isValid = 1;

Create and Use Conditional Statements

- 4.** Given x is declared with a valid integer, which conditional statement will not compile?
 - if (x == 0) {System.out.println("True Statement");}
 - if (x == 0) {System.out.println("False Statement");}
 - if (x == 0) {} elseif (x == 1) {System.out.println("Valid Statement");}
 - if (x == 0) ; else if (x == 1){} else {}
- 5.** A switch statement works with which wrapper class/reference type(s)?
 - Character
 - Byte
 - Short
 - Int
- 6.** Which of the following statements will not compile?
 - if (true) ;
 - if (true) {}
 - if (true) {};
 - if (true) {;;}
 - if (true) ;{};
 - All statements will compile.

7. Given:

```
public class Dinner {  
    public static void main (String[] args)  
    {  
        boolean isKeeperFish = false;  
        if (isKeeperFish = true) {  
            System.out.println("Fish for dinner");  
        } else {  
            System.out.println("Take out for dinner");  
        }  
    }  
}
```

What will be the result of the application's execution?

- A. Fish for dinner will be printed.
- B. Take out for dinner will be printed.
- C. A compilation error will occur.

Create and Use Iteration Statements

8. You need to update a value of a hash table (that is, `HashMap`) where the primary key must equal a specified string. Which statements would you need to use in the implementation of this algorithm?
- A. Iteration statement
 - B. Expression statement
 - C. Conditional statement
 - D. Transfer of control statement
9. The `for` loop has been enhanced in Java 5.0. Which is *not* a common term for the improved `for` loop?
- A. The `for in` loop
 - B. The specialized `for` loop
 - C. The `for each` loop
 - D. The enhanced `for` loop

Create and Use Transfer of Control Statements

10. Which keyword is part of a transfer of control statement?
- A. `if`
 - B. `return`
 - C. `do`
 - D. `assert`

SELF TEST ANSWERS

Understand Assignment Statements

1. Which is not a type of statement?

- A. Conditional statement
- B. Assignment statement
- C. Iteration statement
- D. Propagation statement

Answer:

- D. There is no such thing as a propagation statement.
 - A, B, and C are incorrect. Conditional, assignment, and iteration are all types of statements.
-

2. What type of statement is the following equation: $y = (m*x) + b$?

- A. Conditional statement
- B. Assignment statement
- C. Assertion statement
- D. Transfer of control statement

Answer:

- B. An assignment statement would be used to code the given example of $y = (m*x) + b$.
 - A, C, and D are incorrect. The conditional, assertion, and transfer of control statements are not used to perform assignments.
-

3. Which statements correctly declare boolean variables?

- A. Boolean isValid = true;
- B. boolean isValid = TRUE;
- C. boolean isValid = new Boolean (true);
- D. boolean isValid = 1;

Answer:

- A and C. These statements properly declare boolean variables. Remember that the only valid literal values for the boolean primitives are true and false.
 - B and D are incorrect. B is incorrect because TRUE is not a valid literal value. D is incorrect because you cannot assign the value 1 to a boolean variable.
-

Create and Use Conditional Statements

4. Given x is declared with a valid integer, which conditional statement will not compile?

- A. if (x == 0) {System.out.println("True Statement");}
- B. if (x == 0) {System.out.println("False Statement");}
- C. if (x == 0) {} elseif (x == 1) {System.out.println("Valid Statement");}
- D. if (x == 0) ; else if (x == 1){} else {}

Answer:

- C. The statement will not compile. Without a space between the `else` and `if` keywords, the compiler will be thrown an error similar to “Error: method elseif (boolean) not found....”
 A, B, and D are incorrect. All of these conditional statements will compile successfully.
-

5. A switch statement works with which wrapper class/reference type(s)?

- A. Character
- B. Byte
- C. Short
- D. Int

Answer:

- A, B, and C. The switch statements work with `Character`, `Byte`, and `Short` wrapper classes as well as the `Integer` wrapper class.
 D is incorrect. There is no such thing as an `Int` wrapper type. This was a trick question. The switch statement works with either the `int` primitive or the `Integer` wrapper type.
-

6. Which of the following statements will not compile?

- A. `if (true) ;`
- B. `if (true) {}`
- C. `if (true) {};`
- D. `if (true) {{}}`
- E. `if (true) ;{};`
- F. All statements will compile.

Answer:

- F. All of the statements will compile.
-

7. Given:

```
public class Dinner {  
    public static void main (String[] args)  
    {  
        boolean isKeeperFish = false;  
        if (isKeeperFish = true) {  
            System.out.println("Fish for dinner");  
        } else {  
            System.out.println("Take out for dinner");  
        }  
    }  
}
```

What will be the result of the application’s execution?

- A. Fish for dinner will be printed.
- B. Take out for dinner will be printed.
- C. A compilation error will occur.

Answer:

- A. Since only one equal sign (that is, assignment statement) was used in the `if` statement, the

`isKeeperFish` variable was assigned the value of `true`.

B and **C** are incorrect.

Create and Use Iteration Statements

- 8.** You need to update a value of a hash table (that is, `HashMap`) where the primary key must equal a specified string. Which statements would you need to use in the implementation of this algorithm?

- A.** Iteration statement
 - B.** Expression statement
 - C.** Conditional statement
 - D.** Transfer of control statement
-

Answer:

A, B, and C. Iteration, expression, and conditional statements would be used to implement the algorithm. The following code segment demonstrates the use of these statements by programmatically replacing the ring on the little finger of a person's left hand. The statements are prefaced by comments that identify their types.

```
import java.util.HashMap;
public class HashMapExample {
    public static void main(String[] args) {
        HashMap<String, String> leftHand = new HashMap<String, String>();
        leftHand.put("Thumb", null);
        leftHand.put("Index finger", "Puzzle Ring");
        leftHand.put("Middle finger", null);
        leftHand.put("Ring finger", "Engagement Ring");
        leftHand.put("Little finger", "Pinky Ring");
        // Iteration statement
        for (String s : leftHand.keySet()) {
            // Conditional statement
            if (s.equals("Little finger")) {
                System.out.println(s + " had a " + leftHand.get(s));
                // Expression Statement
                leftHand.put("Little finger", "Engineer's Ring");
                System.out.println(s + " has an " + leftHand.get(s));
            }
        }
    }
}
$ Little finger had a Pinky Ring
$ Little finger has an Engineer's Ring
```

D is incorrect. There is no transfer of control statement in the algorithm.

- 9.** The `for` loop has been enhanced in Java 5.0. Which is *not* a common term for the improved for loop?

- A.** The `for in` loop
- B.** The specialized `for` loop
- C.** The `for each` loop
- D.** The enhanced `for` loop

Answer:

- B.** The enhanced `for` loop is not commonly referenced as a specialized `for` loop.
 A, C, and D are incorrect. The enhanced `for` loop is also commonly referenced as the `for in` loop and the `for each` loop.
-

Create and Use Transfer of Control Statements

10. Which keyword is part of a transfer of control statement?

- A.** `if`
 - B.** `return`
 - C.** `do`
 - D.** `assert`
-

Answer:

- B.** The keyword `return` is used as part of a transfer of control statement.
 A, C, and D are incorrect. The keywords `if`, `do`, and `assert` are not part of any transfer of control statements.
-



3

Programming with Java Operators and Strings

CERTIFICATION OBJECTIVES

- Understand Fundamental Operators
- Understand Operator Precedence
- Use String Objects and Their Methods
- Use StringBuilder Objects and Their Methods
- Test Equality Between Strings and other Objects

✓ Two-Minute Drill

Q&A Self Test

Two of the most fundamental elements of the Java programming language are Java operators and strings. This chapter discusses Java operators and how they manipulate their operands. You will need a full understanding of the different types and groupings of operators and their precedence to score well on the exam. This chapter provides you with all of the operator-related information you need to know.

Strings are commonly used in Java, so they will also be present throughout the exam. This chapter details the `String` and `StringBuilder` classes and their related functionality. Topics include the

string concatenation operator and the `toString` method, as well as a discussion of valuable methods from the `String` and `StringBuilder` classes. We wrap up the chapter discussing the testing of equality between strings.

After completing this chapter, you will have all the knowledge necessary to score well on the operator- and string-related questions on the exam.

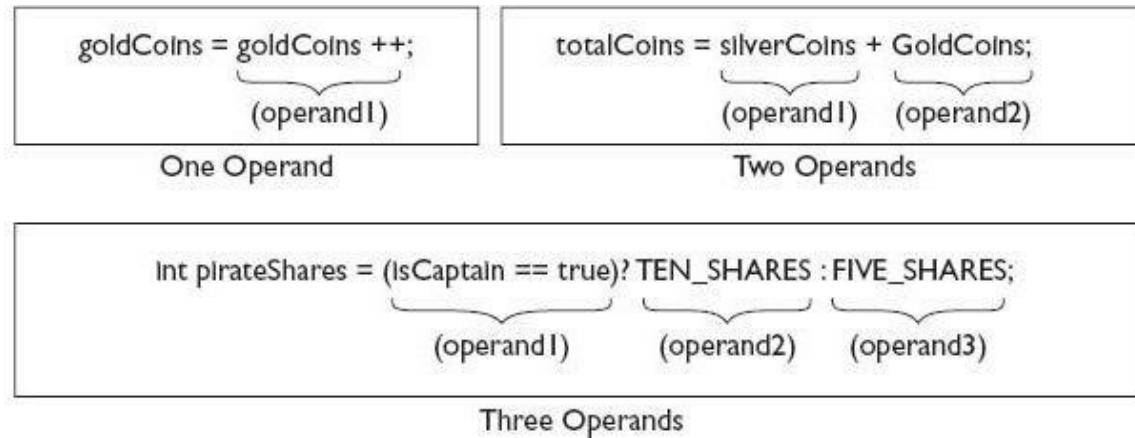
CERTIFICATION OBJECTIVE

Understand Fundamental Operators

Exam Objective 3.1 Use Java operators

Java operators are used to return a result from an expression using one, two, or three operands. Operands are the values placed to the right or left side of the operators. Prefix/postfix-increment/decrement operators use one operand. The conditional ternary operator (`?:`) uses three operands. All other operators use two operands. Examples of operand use are shown in [Figure 3-1](#). Note that the result of evaluating operands is typically a primitive value.

FIGURE 3-1 Operands



The following topics will be covered in these pages:

- Assignment operators
- Arithmetic operators
- Relational operators
- Logical operators

Assignment Operators

Assignment operators are used to assign values to variables.

= Assignment operator

The assignment operator by itself is the equal sign (`=`). (Chapter 2 discusses assignment statements, and [Chapter 4](#) discusses the assignment of literals into primitive data types and the creation of reference type variables.) At its simplest, the assignment operator moves valid literals into variables.

Assignment operators cause compiler errors when the literals are not valid for the variable (that is, its associated type) to which they are assigned. The following are valid assignment statements using the assignment operator:

```
boolean hasTreasureChestKey = true;
byte shipmates = 20;
PirateShip ship = new PirateShip();
```

The following are invalid assignments and will cause compiler errors:

```
/* Invalid literal, TRUE must be lower case */
boolean hasTreasureChestKey = TRUE;
/* Invalid literal, byte value cannot exceed 127 */
byte shipmates = 500;
/* Invalid constructor */
PirateShip ship = new PirateShip(UNEXPECTED_ARG);
```

Compound Assignment Operators

A variety of compound assignment operators exist. The exam covers only the addition and subtraction compound assignment operators.

| | |
|---|------------------------------------|
| = | Assignment by addition operator |
| - | Assignment by subtraction operator |

Consider the following two assignment statements:

```
goldCoins = goldCoins + 99;
pirateShips = pirateShips - 1;
```

The following two statements (with the same meaning and results as the preceding examples) are written with compound assignment operators:

```
goldCoins += 99;
pirateShips -= 1;
```

on the job While the use of compound assignment operators cuts down on keystrokes, it is generally good practice to use the “longhand” approach since the code is clearly more readable.

EXERCISE 3-1

Using Compound Assignment Operators

This exercise will clear up any confusion you may have about compound assignment operators. The following application will be used for the exercise. (Don’t run it until after step 3.)

```
public class Main {  
    public static void main(String[] args) {  
        byte a;  
        a = 10;  
        System.out.println(a += 3);  
        a = 15;  
        System.out.println(a -= 3);  
        a = 20;  
        System.out.println(a *= 3);  
        a = 25;  
        System.out.println(a /= 3);  
        a = 30;  
        System.out.println(a %= 3);  
        a = 35;  
        // Optional as outside the scope of the exam  
        System.out.println(a &= 3);  
        a = 40;  
        System.out.println(a ^= 3);  
        a = 45;  
        System.out.println(a |= 3);  
        a = 50;  
        System.out.println(a <<= 3);  
        a = 55;  
        System.out.println(a >>= 3);  
        a = 60;  
        System.out.println(a >>>= 3);  
        // End optional  
    }  
}
```

1. Grab a pencil and a piece of paper. Optionally, you can use [Table 3-1](#) as your worksheet.

TABLE 3-1 Refactoring Compound Assignment Statements

| Assigned Value of a | Compound Assignment | Refactored Statement | New Value of a |
|---------------------|---------------------|----------------------|----------------|
| a = 10; | a += 3; | a = 10 + 3; | 13 |
| a = 15; | a -= 3; | | |
| a = 20; | a *= 3; | | |
| a = 25; | a /= 3; | | |
| a = 30; | a %= 3; | | |
| a = 35; | a &= 3; | | |
| a = 40; | a ^= 3; | | |
| a = 45; | a = 3; | | |
| a = 50; | a <<= 3; | | |
| a = 55; | a >>= 3; | | |
| a = 60; | a >>>= 3; | | |

2. For each statement that has a compound assignment operator, rewrite the statement without the compound assignment operator and replace the variable with its associated value. For example, let's take the assignment statement with the addition compound assignment operator:

```
a = 5;
System.out.println(a += 3);
```

It would be rewritten as (a = a + 3)—specifically (a = 5 + 3);.

3. Evaluate the expressions, without using a computer.
 4. Compile and run the given application. Compare your results.
-

(Note that many of these operators do not appear on the exam. The point of the exercise is to get you properly acquainted with compound assignment operators, by repetition.)

on the job *It is common to represent assignments in pseudo-code with the colon and equal sign characters (for example, A := 20). Notice that := looks similar to +=, -=, and other Java assignment operators such as *=, /=, and %=. Be aware, however, that the pseudo-code assignment representation (:=) is not a Java assignment operator, and if you see it in any Java code, it will not compile.*

Arithmetic Operators

The exam will include nine arithmetic operators. Five of these operators are used for basic operations (addition, subtraction, multiplication, division, and modulus). The other four operators are used for incrementing and decrementing a value. We'll examine the five operators used for basic operations first.

Basic Arithmetic Operators

The five basic arithmetic operators are

| | |
|---|-----------------------------------|
| + | Addition (sum) operator |
| - | Subtraction (difference) operator |
| * | Multiplication (product) operator |
| / | Division (quotient) operator |
| % | Modulus (remainder) operator |

Adding, subtracting, multiplying, dividing, and producing remainders with operators is straightforward. The following examples demonstrate this:

```
/* Addition (+) operator example */
int greyCannonBalls = 50;
int blackCannonBalls = 50;
int totalCannonBalls = greyCannonBalls + blackCannonBalls; // 100
/* Subtraction (-) operator example */
int firedCannonBalls = 10;
totalCannonBalls = totalCannonBalls - firedCannonBalls; // 90
/* Multiplication (*) operator example */
int matches = 20;
int matchboxes = 20;
int totalMatches = matches * matchboxes; // 400
/* Division (/) operator example */
int pirates = 104;
int pirateShips = 3;
int assignedPiratesPerShip = pirates / pirateShips; // 34
/* Remainder (modulus) (%) operator example */
int pirateRemainder = pirates % pirateShips; // 2 (remainder)
```

Prefix Increment, Postfix Increment, Prefix Decrement, and Postfix Decrement Operators

Four operators allow decrementing or incrementing of variables:

| | |
|-----|----------------------------|
| ++x | Prefix increment operator |
| --x | Prefix decrement operator |
| x++ | Postfix increment operator |
| x-- | Postfix decrement operator |

Prefix increment and prefix decrement operators provide a shorthand way of incrementing and decrementing the variable by 1, respectively. Rather than creating an expression as $y=x+1$, you could write $y=++x$. Similarly, you could replace the expression $y=x-1$ with $y=--x$. This works because the execution of the prefix operators occurs on the operand prior to the evaluation of the whole expression. Postfix increment and postfix decrement characters execute the postfix operators after the expression has been evaluated. Therefore, $y = x++$ would equate to $y=x$ followed by $x=x+1$. And $y = x--$ would equate to $y=x$ followed by $x=x-1$.

It's important to note that $y=++x$ is not exactly equivalent to $y=x+1$, because the value of x changes in the former but not in the latter. This is the same for $y=--x$ and $y=x-1$.

The prefix increment operator increments a value by 1 before an expression has been evaluated.

```
int x = 10;  
int y = ++x ;  
System.out.println("x=" + x + " , y=" + y); // x= 11, y= 11
```

The postfix increment operator increments a value by 1 after an expression has been evaluated.

```
int x = 10;  
int y = x++ ;  
System.out.println("x=" + x + " , y=" + y); // x= 11, y= 10
```

The prefix decrement operator decrements a value by 1 before an expression has been evaluated.

```
int x = 10;  
int y = --x ;  
System.out.println("x=" + x + " , y=" + y); // x= 9, y= 9
```

The postfix decrement operator decrements a value by 1 after an expression has been evaluated.

```
int x = 10;  
int y = x-- ;  
System.out.println("x=" + x + " , y=" + y); // x= 9, y= 10
```

Relational Operators

Relational operators return Boolean values in relationship to the evaluation of their left and right operands. The six most common relational operators are on the exam. Four of them equate to the greater than and less than comparisons. Two are strictly related to equality, as we will discuss at the end of this section.

Basic Relational Operators

| | |
|----|-----------------------------------|
| < | Less than operator |
| <= | Less than or equal to operator |
| > | Greater than operator |
| >= | Greater than or equal to operator |

The less than, less than or equal to, greater than, and greater than or equal to operators are used to compare integers, floating points, and characters. When the expression used with the relational operators is true, the Boolean value of true is returned; otherwise, false is returned. Here's an example:

```
/* returns true as 1 is less than 2 */  
boolean b1 = 1 < 2;  
/* returns false as 3 is not less than 2 */  
boolean b2 = 3 < 2;
```

```

/* returns true as 3 is greater than 2 */
boolean b3 = 3 > 2;
/* returns false as 1 is not greater than 2 */
boolean b4 = 1 > 2;
/* returns true as 2 is less than or equal to 2 */
boolean b5 = 2 <= 2;
/* returns false as 3 is not less than or equal to 2 */
boolean b6 = 3 <= 2;
/* returns true as 3 is greater than or equal to 3 */
boolean b7 = 3 >= 3;
/* returns false as 2 is not greater than or equal to 3 */
boolean b8 = 2 >= 3;

```

So far, we've examined only the relationship of int primitives. Let's take a look at the various ways char primitives can be evaluated with relational operators, specifically the less than operator for these examples. Remember that characters (that is, char primitives) accept integers (within the valid 16-bit unsigned range), hexadecimal, octal, and character literals. Each literal in the following examples represents the letters "A" and "B." The left operands are character "A" and the right operands are character "B." Since each expression is essentially the same, they all evaluate to true.

```

boolean b1 = 'A' < 'B'; // Character literals
boolean b2 = '\u0041' < '\u0042'; // Unicode literals
boolean b3 = 0x0041 < 0x0042; // Hexadecimal literals
boolean b4 = 65 < 66; // Integer literals that fit in a char
boolean b5 = 0101 < 0102; // Octal literals
boolean b6 = '\101' < '\102'; // Octal literals
boolean b7 = 'A' < 0102; // Character and Octal literals

```

As mentioned, you can also test the relationship between floating points. The following are a few examples:

```

boolean b1 = 9.00D < 9.50D; // Floating points with D postfixes
boolean b2 = 9.00d < 9.50d; // Floating points with d postfixes
boolean b3 = 9.00F < 9.50F; // Floating points with F postfixes
boolean b4 = 9.0f < 9.50f; // Floating points with f postfixes
boolean b5 = (double)9 < (double)10; // Integers with explicit casts
boolean b6 = (float)9 < (float)10; // Integers with explicit casts
boolean b7 = 9 < 10; // Integers that fit into floating points
boolean b8 = (9d < 10f);
boolean b9 = (float)11 < 12;

```

Equality Operators

Relational operators that directly compare the equality of primitives (numbers, characters, Booleans) and object reference variables are considered equality operators.

| | |
|-----------------|-----------------------|
| <code>==</code> | Equal to operator |
| <code>!=</code> | Not equal to operator |

Comparing primitives of the same type is straightforward. If the right and left operands of the equal to operator are equal, the Boolean value of true is returned; otherwise false is returned. If the right and

left operands of the not equal to operator are not equal, the Boolean value of true is returned; otherwise false is returned. The following code examples compare all eight primitives to values of the same type:

```
int value = 12;
/* boolean comparison, prints true */
System.out.println(true == true);
/* char comparison, prints false */
System.out.println('a' != 'a');
/* byte comparison, prints true */
System.out.println((byte)value == (byte)value);
/* short comparison, prints false */
System.out.println((short)value != (short)value);
/* integer comparison, prints true */
System.out.println(value == value);
/* float comparison, prints true */
System.out.println(12F == 12f);
/* double comparison, prints false */
System.out.println(12D != 12d);
```

You should actually use an epsilon when comparing floating-point numbers. See “Comparing Floating Point Numbers, 2012 Edition” for more information on comparing floating points for equality: <http://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/>.



Reference values of objects can also be compared. Consider the following code:

```
Object a = new Object();
Object b = new Object();
Object c = b;
```

The reference variables are a, b, and c. As shown, reference variables a and b are unique. Reference variable c refers to reference variable b, so for equality purposes, they are the same. The following code shows the results of comparing these variables.

```
/* Prints false, different references */
System.out.println(a == b);
/* Prints false, different references */
System.out.println(a == c);
/* Prints true, same references */
System.out.println(b == c);
```

The following are similar statements, but they use the not equal to operator:

```
System.out.println(a != b); // Prints true, different references
System.out.println(a != c); // Prints true, different references
System.out.println(b != c); // Prints false, same references
```

Numeric Promotion of Binary Values By this point, you may be wondering what the compiler does with the operands when they are of different primitive types. Numeric promotion rules are applied on binary values for the additive (+, -), multiplicative (*, /, %), comparison (<, <=, >, >=), equality (==, !=), bitwise (&, ^, |), and conditional (?:) operators. See [Table 3-2](#).

TABLE 3-2 Numeric Promotion of Binary Values

| Binary Numeric Promotion | |
|--------------------------|--|
| Check 1 | Check if one and only one operand is a double primitive. If so, convert the non-double primitive to a double, and stop checks. |
| Check 2 | Check if one and only one operand is a float primitive. If so, convert the non-float primitive to a float, and stop checks. |
| Check 3 | Check if one and only one operand is a long primitive. If so, convert the non-long primitive to a long, and stop checks. |
| Check 4 | Convert both operands to int. |

Logical Operators

Logical operators return Boolean values. Three are logical operators on the exam: logical AND, logical OR, and logical negation.

Logical (Conditional) Operators

Logical (conditional) operators evaluate a pair of Boolean operands. Understanding their short-circuit principle is necessary for the exam.

| | |
|----|--|
| && | Logical AND (conditional AND) operator |
| | Logical OR (conditional OR) operator |

The logical AND operator evaluates the left and right operands. If both values of the operands have a value of true, then a value of true is returned. The logical AND is considered a short-circuit operator. If the left operand returns false, then there is no need to check the right operator since both would need to be true to return true; thus, it short-circuits. Therefore, whenever the left operand returns false, the expression terminates and returns a value of false.

The following code demonstrates the usage of the logical AND operator.

```
/* Assigns true */
boolean and1 = true && true;
/* Assigns false */
boolean and2 = true && false;
/* Assigns false, right operand not evaluated */
boolean and3 = false && true;
/* Assigns false, right operand not evaluated */
boolean and4 = false && false;
```

The logical OR operator evaluates the left and right operands. If either value of the operands has a value of true, a value of true is returned. The logical OR is considered a short-circuit operator. If the

left operand returns true, there is no need to check the right operator, since either needs to be true to return true; thus, it short-circuits. Again, whenever the left operand returns true, the expression terminates and returns a value of true.

The following code demonstrates the usage of the logical OR operator.

```
/* Assigns true, right operand not evaluated */
boolean or1 = true || true;
/* Assigns true, right operand not evaluated */
boolean or2 = true || false;
/* Assigns true */
boolean or3 = false || true;
/* Assigns false */
boolean or4 = false || false;
```

Logical Negation Operator

The logical negation operator is also known as the inversion operator or Boolean invert operator. This is a simple operator, but don't take it lightly... you may see it quite often on the exam.

!

Logical negation (inversion) operator

The logical negation operator returns the opposite of a Boolean value.

The following code demonstrates the usage of the logical Negation operator.

```
System.out.println(!false); // Prints true
System.out.println(!true); // Prints false
System.out.println (!!true); // Prints true
System.out.println (!!true); // Prints false
System.out.println (!!!true); // Prints true
```

Expect to see the logical negation operator used in conjunction with any method or expression that returns a Boolean value. The following list details some of these expressions that return Boolean values:

- Expressions with relational operators return Boolean values.
- Expressions with logical (conditional) operators return Boolean values.
- The `equals` method of the `Object` class returns Boolean values.
- The `String` methods `startsWith` and `endsWith` return Boolean values.

The following are some examples of statements that include the logical negation operator:

```

/* Example with relational expression */
iVar1 = 0;
iVar2 = 1;
if (!(iVar1 <= iVar2)) {}

/* Example with logical expressions */
boolean bVar1 = false; boolean bVar2 = true;
if ((bVar1 && bVar2) || (!(bVar1 && bVar2))) {}

/* Example with equals method */
if ("NAME".equals("NAME")) {}

/* Example with the String class's startsWith method */
String s = "Captain Jack";
System.out.println(!s.startsWith("Captain"));

```

The logical inversion operator cannot be used on a non-Boolean value. The following code will not compile:

```

!10; // compiler error, integer use is illegal
!"STRING"; // compiler error, string use is illegal

```

Logical AND and logical OR are on the exam. Boolean AND and Boolean OR, along with bitwise AND and bitwise OR, are not on the exam. You might, for example, want to use the nonlogical expressions associated with the right operand if a change occurs to a variable where the new result is used later in your code. The following Scenario & Solution details the specifics of this scenario.

| SCENARIO & SOLUTION | |
|--|---|
| You want to use an AND operator that evaluates the second operand whether the first operand evaluates to true or false. Which would you use? | Boolean AND (<code>&</code>) |
| You want to use an OR operator that evaluates the second operand whether the first operand evaluates to true or false. Which would you use? | Boolean OR (<code> </code>) |
| You want to use an AND operator that evaluates the second operand only when the first operand evaluates to true. Which would you use? | Logical AND (<code>&&</code>) |
| You want to use an OR operator that evaluates the second operand only when the first operand evaluates to false. Which would you use? | Logical OR (<code> </code>) |

CERTIFICATION OBJECTIVE

Understand Operator Precedence

Exam Objective 3.2 Overriding Operator Precedence

Operator precedence is the order in which operators will be evaluated when several operators are included in an expression. Operator precedence can be overridden using parentheses. Know the basics surrounding operator precedence and you'll do well on the exam in this area.

The following topics will be covered in these pages:

- Operator precedence
- Overriding operator precedence

Operator Precedence

Operators with a higher precedence are evaluated before operators with a lower precedence. [Table 3-3](#) lists the Java operator from the highest precedence to the lowest precedence and their associations. The association (for example, left to right) defines which operand will be used (or evaluated) first.

TABLE 3-3 Java Operators on the OCA Exam

| Relative Precedence | Operator | Description | Association |
|---------------------|--|--|---------------|
| 1 | [] | Array index | Left to right |
| | () | Method call | |
| | . | Member access | |
| 2 | ++, -- | Postfix increment, postfix decrement | Right to left |
| | ++, -- | Prefix increment, prefix decrement | Right to left |
| 3 | ! | Boolean (logical) NOT | |
| | ~ | Bitwise NOT | Right to left |
| 4 | (type) | Type cast | |
| | new | Object creation | Right to left |
| 5 | *, /, % | Multiplication, division, remainder (modulus) | Left to right |
| 6 | +, - | Addition, subtraction | Left to right |
| | + | String concatenation | |
| 7 | <, <=, >, >= | Less than, less than or equal to, greater than, greater than or equal to | Left to right |
| | instanceof | reference test | |
| 8 | ==, != | Value equality and inequality / reference equality and inequality | Left to right |
| | ==, != | Value equality and inequality / reference equality and inequality | |
| 9 | & | Bitwise AND / Boolean AND | |
| 10 | | Bitwise XOR (Exclusive OR) / Boolean XOR | Left to right |
| 11 | ^ | Bitwise OR (Inclusive OR) / Boolean OR | Left to right |
| 12 | && | Conditional AND | Left to right |
| 13 | | Conditional OR | Left to right |
| 14 | ? : | Conditional (ternary) | Right to left |
| 15 | =, *=, /=, +=, -=, %=, <=>, >>=, >>>=, &=, ^=, = | Assignment and compound assignments | Right to left |
| | | | |

Overriding Operator Precedence

Operator precedence can be overridden by the use of parentheses. When multiple sets of parentheses are present, the innermost set is evaluated first. Let's take a look at some basic code examples exercising operator precedence.

When operators have the same precedence, they are evaluated from left to right:

```
int p1 = 1; int p2 = 5; int p3 = 10;  
/* Same precedence */  
System.out.println(p1 + p2 - p3); // -4
```

When operators do not have the same precedence, the operator with the higher precedence is evaluated first:

```
int p1 = 1; int p2 = 5; int p3 = 10;  
/* Lower, followed by higher precedence */  
System.out.println(p1 + p2 * p3); // 51
```

When an expression includes parentheses, operator precedence is overridden:

```
int p1 = 1; int p2 = 5; int p3 = 10;  
/* Parentheses overriding precedence */  
System.out.println((p1 + p2) * p3); // 60
```

When an expression has multiple sets of parentheses, the operator associated with the innermost parentheses is evaluated first:

```
int p1 = 1; int p2 = 5; int p3 = 10; int p4 = 25;  
/* Using innermost parentheses first */  
System.out.println((p1 * (p2 + p3)) - p4); // -10
```

CERTIFICATION OBJECTIVE

Use String Objects and Their Methods

Exam Objective 2.7 Create and manipulate strings

Strings are commonly used in the Java programming language. This section discusses what strings are and how to concatenate separate strings, and then details the methods of the `String` class. When you have completed this section, which covers the following topics, you should fully understand what strings are and how to use them:

- Strings
- String concatenation operator
- Methods of the `String` class

Strings

String objects are used to represent 16-bit Unicode character strings. Consider the 16 bits `000001011001` followed by `000001101111`. These bits in Unicode are represented as `\u0059` and `\u006F`. `\u0059` is mapped to the character “Y” and `\u006F` is mapped to the character “o”. An easy way of concatenating 16-bit Unicode character strings together in a reusable element is by declaring the data within a string:

```
String exclamation = "Yo"; // 000001011001 and 000001101111
```

See [Appendix C](#) for more information on the Unicode standard.

Strings are *immutable* objects, meaning their values never change. For example, the following text, “Dead Men Tell No Tales”, can be created as a string:

```
String quote = "Dead Men Tell No Tales";
```

In the following example, the value of the string does not change after a `String` method returns a modified value. Remember that strings are immutable. Here, we invoke the `replace` method on the string. Again, the new string is returned but will not change the value.

```
quote.replace("No Tales", "Tales"); // Returns new value
System.out.println(quote); // Prints the original value
$ Dead Men Tell No Tales
```

We can create strings in several ways. As with instantiating any object, you need to construct an object and assign it to a reference variable. As a reminder, a *reference variable* holds the value’s address. Let’s look at some of the things you can do with strings.

You can create a string without an assigned string object. Make sure you eventually give it a value, or you’ll get a compiler error.

```
String quote1; // quote1 is a reference variable with no
assigned string object
quote1 = "Ahoy matey"; // Assigns string object to the reference
```

You can use a couple of basic approaches to create a string object with an empty string representation:

```
String quote2a = new String(); // quote2a is a reference variable
String quote2b = new String(""); // Equivalent statement
```

You can create a string object without using a constructor:

```
String quote3 = "The existence of the sea means the existence"
+ " of Pirates! -- Malayan proverb";
```

SCENARIO & SOLUTION

| | |
|--|--------------------------------------|
| You want to use an object that represents an immutable character string. Which class will you use to create the object? | The <code>String</code> class |
| You want to use an object that represents a mutable character string. Which class will you use to create the object? | The <code>StringBuilder</code> class |
| You want to use an object that represents a thread-safe mutable character string. Which class will you use to create the object? | The <code>StringBuffer</code> class |

You can create a string object while using a constructor:

```
/* quote4 is a reference variable to the new string object */
String quote4 = new String("Yo ho ho!");
```

You can create a reference variable that refers to a separate reference variable of a string object:

```
String quote5 = "You're welcome to my gold. -- William Kidd";
String quote6 = quote5; // quote6 refers to the quote5 reference
```

You can assign a new string object to an existing string reference variable:

```
String quote7 = "The treasure is in the sand."; Assigns string
object to the reference variable /* and move to a previous
line.
quote7 = "The treasure is between the rails."; /* Assigns new
string to the same reference variable */ and move to a new
previous line.
```

If you want to use a mutable character string, consider `StringBuffer` or `StringBuilder` as represented in the preceding Scenario & Solution.

The String Concatenation Operator

The string concatenation operator concatenates (joins) strings together. The operator is denoted with the + sign.

| | |
|---|-------------------------------|
| + | String concatenation operator |
|---|-------------------------------|

If you have been programming for at least six months, odds are you have glued two strings together at some time. Java's string concatenation operator makes the act of joining two strings very easy. For example, "doub" + "loon" equates to "doublloon". Let's look at some more complete code:

```
String item = "doublloon";
String question = "What is a " + item + "? ";
System.out.println ("Question: " + question);
```

Line 2 replaces the `item` variable with its contents, "doublloon", and so the question string becomes

What is a doublloon?

Notice that the question mark was appended as well.

Line 3 replaces the `question` variable with its contents and so the following string is returned:

```
$ Question: What is a doublloon?
```

It is that simple. But wait! What happens when primitives are added to the concatenation? The Java Language specification reads, "The + operator is syntactically left-associative, no matter whether it is determined by type analysis to represent string concatenation or numeric addition." We can examine this behavior in the following examples:

```

float reale = .007812f; // percent of one gold doubloon
float escudo = .125f; // percent of one gold doubloon

/* Prints "0.132812% of one gold doubloon" */
System.out.println
    (reale + escudo + "% of one gold doubloon"); // values added

/* Prints "0.132812% of one gold doubloon" */
System.out.println
    ((reale + escudo) + "% of one gold doubloon"); // includes parentheses

/* Prints "% of one gold doubloon: 0.132812" */
System.out.println
    ("% of one gold doubloon: " + (reale + escudo)); // includes parentheses

/* Prints "Coin values concatenated: 0.0078120.125" */
System.out.println
    ("Coin values concatenated:" + reale + escudo); // values not added

```

The `toString` Method

The `Object` class has a method that returns a string representation of objects. This method is appropriately named the `toString` method. All classes in Java extend the `Object` class by default; therefore, every class inherits this method. When you're creating classes, it is common practice to override the `toString` method to return the data that best represents the state of the object. The `toString` method makes common use of the string concatenation operator.

Let's take a look at the `TreasureMap` class with the `toString` method overridden.

```

public class TreasureMap {
    private String owner = "Blackbeard";
    private String location = "Outer Banks";
    public String toString () {
        return "Map Owner: " + this.owner + ", treasure location: "
            + this.location;
    }
}

```

Here, the `toString` method returns the contents of the class's instance variables. Now let's print out the representation of a `TreasureMap` object:

```

TreasureMap t = new TreasureMap();
System.out.println(t);
$ Map Owner: Blackbeard, treasure location: Outer Banks

```

Concatenation results may be unexpected if you are including variables that are not initially strings. You should become very comfortable with working with the string concatenation operator, so let's take another look at it.

Consider a string and two integers:

```

String title1 = " shovels.";
String title2 = "Shovels: ";
int flatShovels = 5;
int roundPointShovels = 6;

```

The compiler performs left-to-right association for the additive and string concatenation operators.

For the following two statements, the first two integers are added together. Next, the concatenation operator takes the `toString` representation of the result and concatenates it with the other string:

```
/* Prints '11 shovels' */
System.out.println(flatShovels + roundPointShovels + title1);

/* Prints '11 shovels' */
System.out.println((flatShovels + roundPointShovels) + title1);
```

Next, moving from left to right, the compiler takes the `title2` string and joins it with the string representation of the `flatShovels` integer variable. The result is a string. Now this result string is joined to the string representation of the `roundPointShovels` variable. Note that the `toString` method is used to return the string.

```
/* Prints 'Shovels: 56' */
System.out.println(title2 + flatShovels + roundPointShovels);
```

Parentheses take precedence, so you can join the sum of the integer values with the string if you code it as follows:

```
/* Prints 'Shovels: 11' */
System.out.println(title2 + (flatShovels + roundPointShovels));
```

EXERCISE 3-2

Uncovering Bugs that Your Compiler May Not Find

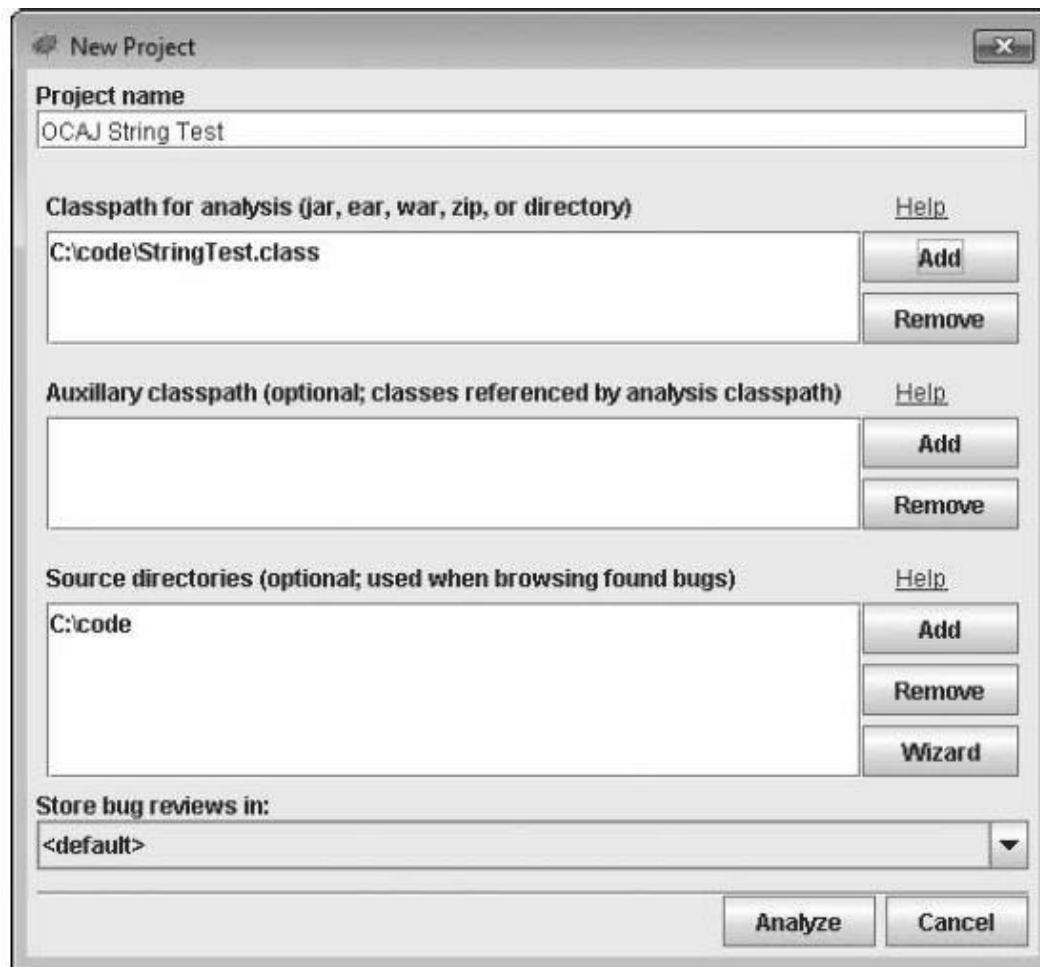
Consider the strings in the following application:

```
public class StringTest {
    public static void main(String[] args) {
        String s1 = new String ("String one");
        String s2 = "String two";
        String s3 = "String " + "three";
    }
}
```

One of the strings is constructed in an inefficient manner. Do you know which one? Let's find out using the FindBugs application from the University of Maryland.

1. Create a directory named “code” somewhere on your PC (for example, C:\ code).
2. Create the `StringTest.java` source file.
3. Compile the `StringTest.java` source file: `javac StringTest.java`.
4. Download the FindBugs software from <http://findbugs.sourceforge.net/>.
5. Extract, install, and run the FindBugs application. Note that the Eclipse and NetBeans IDEs have plug-ins for the FindBugs tool as well as other software quality tools such as PMD and Checkstyle.

6. Create a new project in FindBugs by choosing File | New Project.
7. Add the project name (for instance, OCA String Test).
8. Click the Add button for the text area associated with the Class Archives And Directories To Analyze. Find and select the StringTest.class file under the C:\code directory and click Choose.
9. Click the Add button for the text area associated with the Source Directories. Find and select the C:\code directory (not the source file) and then click Choose.
10. The New Project dialog box will look similar to the following illustration, with the exception of your personal directory locations. Click Finish.



11. You will see that two bugs are returned. We are concerned with the first one. Drill down in the window that shows the bugs (Bugs | Performance | [...]). The application will detail the warning and show you the source code with the line in error highlighted.
 12. Fix the bug by not calling the constructor.
 13. Rerun the test.
-

Methods of the String Class

Several methods of the String class are commonly used, such as the following: `charAt`, `indexOf`, `length`, `concat`, `replace`, `startsWith`, `endsWith`, `substring`, `trim`, `toLowerCase`, `toUpperCase`, and `ensureIgnoreCase()`. These methods are detailed in [Figure 3-2](#) and in the following section.

FIGURE 3-2 Commonly used methods of the String class

| String |
|--------------------------------------|
| + charAt(int) : char |
| + concat(String) : String |
| + endsWith(String) : boolean |
| + equalsIgnoreCase(String) : boolean |
| + indexOf(int) : int |
| + indexOf(int, int) : int |
| + indexOf(String) : int |
| + indexOf(String, int) : int |
| + length() : int |
| + startsWith(String, int) : boolean |
| + startsWith(String) : boolean |
| + toLowerCase(Locale) : String |
| + toLowerCase() : String |
| + toString() : String |
| + toUpperCase(Locale) : String |
| + toUpperCase() : String |
| + trim() : String |

In following sections, you'll find a description of each method, followed by the method declarations and associated examples.

First, consider the following string:

```
String pirateMessage = " Buried Treasure Chest! ";
```

The string has two leading blank spaces and one trailing blank space. This is important in relationship to the upcoming examples. The string is shown again in [Figure 3-3](#) with the index values shown in relationship to the individual characters.

FIGURE 3-3 OCA string detailing blanks

| | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|
| B | u | r | i | e | d | T | r | e | a | s | u | r | e | C | h | e | s | t | ! | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |

Let's use each method to perform some action with the string `pirateMessage`.

The `charAt` Method

The `String` class's `charAt` method returns a primitive `char` value from a specified `int` index value in relationship to the referenced string object.

There is one `charAt` method declaration:

```
public char charAt(int index) {...}
```

Here are some examples:

```
/* Returns the 'blank space' character from location 0 */
char c1 = pirateMessage.charAt(0);
/* Returns the character 'B' from location 2 */
char c2 = pirateMessage.charAt(2);
/* Returns the character '!' from location 23 */
char c3 = pirateMessage.charAt(23);
/* Returns the 'blank space' character from location 24 */
char c4 = pirateMessage.charAt(24);
/* Throws a StringIndexOutOfBoundsException exception*/
char c5 = pirateMessage.charAt(25);
```

The indexOf Method

The `String` class's `indexOf` method returns primitive `int` values representing the index of a character or string in relationship to the referenced string object.

Four public `indexOf` method declarations exist:

```
public int indexOf(int ch) {...}
public int indexOf(int ch, int fromIndex) {...}
public int indexOf(String str) {...}
public int indexOf(String str, int fromIndex) {...}
```

Here are some examples:

```
/* Returns the integer 3 as it is the first 'u' in the string. */
int i1 = pirateMessage.indexOf('u'); // 3
/* Returns the integer 14 as it is the first 'u' in the string past
 * location 9.
 */
int i2 = pirateMessage.indexOf('u', 9); // 14
/* Returns the integer 13 as it starts at location 13 in the
 * string.
 *
int i3 = pirateMessage.indexOf("sure"); // 13
/* Returns the integer -1 as there is no Treasure string on or
 *past location 10
 */
int i4 = pirateMessage.indexOf("Treasure", 10); // -1!
/* Returns the integer -1 as there is no character u on or past
 * location 100
 */
int i5 = pirateMessage.indexOf("u", 100); // -1!
```

The length Method

The `String` class's `length` method returns a primitive `int` value representing the length of the referenced string object.

There is one `length` method declaration:

```
public int length() {...}
```

Here are some examples:

```
/* Returns the string's length of 25 */
int i = pirateMessage.length(); // 25
// Use of String's length method
String string = "box";
int value1 = string.length(); // 3
// Use of array's length attribute
String[] stringArray = new String[3];
int value2 = stringArray.length; // 3
```



The String class uses the length method (for example, `string.length()`). Arrays reference an instance variable in their state (for example, `array.length`). Therefore, the string methods use the set of parentheses to return their length and arrays do not. This is a gotcha that you will want to look for on the exam.

The concat Method

The String class's concat method concatenates the specified string to the end of the original string.

The sole concat method declaration is

```
public String concat(String str) {...}
```

Here's an example:

```
/* Returns the concatenated string
 *" Buried Treasure Chest! Weigh anchor!"
 */
String c = pirateMessage.concat ("Weigh anchor!");
```

The replace Method

The String class's replace method returns strings, replacing all characters or strings in relationship to the referenced string object. The CharSequence interface allows for the use of either a String, StringBuffer, or StringBuilder object.

Two replace method declarations can be used:

```
public String replace(char oldChar, char newChar) {...}
public String replace(CharSequence target, CharSequence
replacement) {...}
```

Here are some examples:

```

/* Returns the string with all characters 'B' replaced with 'J'. */
String s1 = pirateMessage.replace
    ('B', 'J'); // Juried Treasure Chest!
/* Returns the string with all blank characters ' ' replaced
 * with 'X'.
*/
String s2 = pirateMessage.replace
    (' ', 'X'); // XXBuriedXTreasureXChest!X
/* Returns the string with all strings 'Chest' replaced
 * with 'Coins'.
*/
String s3 = pirateMessage.replace
    ("Chest", "Coins"); // Buried Treasure Coins!

```

The startWith Method

The `String` class's `startsWith` method returns a primitive boolean value representing the results of a test to see if the supplied prefix starts the referenced `String` object.

Two `startsWith` method declarations can be used:

```

public boolean startsWith(String prefix, int toffset) {...}
public boolean startsWith(String prefix) {...}

```

Here are some examples:

```

/* Returns true as the referenced string starts with the
compared string. */
boolean b1 = pirateMessage.startsWith
    (" Buried Treasure"); // true
/* Returns false as the referenced string does not start with
 * the compared string.
*/
boolean b2 = pirateMessage.startsWith(" Discovered"); // false
/* Returns false as the referenced string does not start with
 * the compared string at location 8.
*/
boolean b3 = pirateMessage.startsWith("Treasure", 8); // false
/* Returns true as the referenced string does start with
 * the compared string at location 9.
*/
boolean b4 = pirateMessage.startsWith("Treasure", 9); // true

```

The endsWith Method

The `String` class's `endsWith` method returns a primitive boolean value representing the results of a test to see if the supplied suffix ends the referenced `String` object.

There is one `endsWith` method declaration:

```

public boolean endsWith(String suffix) {...}

```

Here are some examples:

```
/* Returns true as the referenced string ends with the compared
 * string.
 */
boolean e1 = pirateMessage.endsWith("Treasure Chest! "); // true
/* Returns false as the referenced string does not end with the
 * compared string.
 */
boolean e2 = pirateMessage.endsWith("Treasure Chest "); // false
```

The substring Method

The `String` class's `substring` method returns new strings that are substrings of the referenced string object.

Two `substring` method declarations exist:

```
public String substring(int beginIndex) {...}
public String substring(int beginIndex, int endIndex) {
```

Here are some examples:

```
/* Returns the entire string starting at index 9. */
String ss1 = pirateMessage.substring(9); // Treasure Chest!
/* Returns the string at index 9. */
String ss2 = pirateMessage.substring(9, 10); // T
/* Returns the string at index 9 and ending at index 23. */
String ss3 = pirateMessage.substring(9, 23); // Treasure Chest
/* Produces runtime error. */
String ss4 = pirateMessage.substring(9, 8); // out of range
/* Returns a blank */
String ss5 = pirateMessage.substring(9, 9); // Blank
```

The trim Method

The `String` class's `trim` method returns the entire string minus leading and trailing whitespace characters in relationship to the referenced string object. The whitespace character corresponds to the Unicode character \u0020.

```
System.out.println(" ".equals("\u0020")); // true
```

The sole `trim` method declaration is

```
public String trim() {...}
```

Here is an example:

```
/* "Buried Treasure Chest!" with no leading or trailing
 * white spaces
 */
String t = pirateMessage.trim();
```

The toLowerCase Method

The String class's toLowerCase method returns the entire string as lowercase characters.

Two toLowerCase method declarations can be used:

```
public String toLowerCase () {...}
public String toLowerCase (Locale locale) {...}
```

Here is an example:

```
/* Returns all lower case characters " buried treasure chest! " */
String l1 = pirateMessage.toLowerCase();
```

The toUpperCase Method

The String class's toUpperCase method returns the entire string as uppercase characters.

Two toUpperCase method declarations can be used:

```
public String toUpperCase () {...}
public String toUpperCase (Locale locale) {...}
```

Here is an example:

```
/* Returns all uppercase characters " BURIED TREASURE CHEST! " */
String u1 = pirateMessage.toUpperCase();
```

The equalsIgnoreCase Method

The String class's equalsIgnoreCase method returns a boolean value after comparing two strings while ignoring case consideration.

The equalsIgnoreCase method declaration is

```
public boolean equalsIgnoreCase (String str) {...}
```

Here are some examples:

```
/* Compares " Buried Treasure Chest! " with
 * " Buried TREASURE Chest! "
 */
Boolean b1 = pirateMessage.equalsIgnoreCase
    (" Buried TREASURE Chest! "); // true
/* Compares " Buried Treasure Chest! " with
 * " Buried XXXXXXXX Chest! "
 */
Boolean b2 = pirateMessage.equalsIgnoreCase
    (" Buried XXXXXXXX Chest! "); // false
```

www.jdocs.com/. JDocs provides an interface to the source code of several Java-based projects, including the Java platform Standard Edition. If you are using NetBeans, it is just a matter of configuring the IDE to use it, if it doesn't already use it. As an example, if your focus is in the source editor of NetBeans and you **CTRL**-click the word **String** in a **String** declaration, the source code will open in a new window for the **String** class.

INSIDE THE EXAM

Chaining

Java allows for methods to be chained together. Consider the following message from the captain of a pirate ship:

```
String msg = " Maroon the First Mate  
with a flagon of water and a  
pistol! ";
```

We want to change the message to read, “Maroon the Quartermaster with a flagon of water.” Three changes need to be made to adjust the string as desired:

1. Trim the leading and trailing whitespace.
2. Replace the substring **First Mate** with **Quartermaster**.
3. Remove **and a pistol!**
4. Add a period at the end of the sentence.

A variety of methods and utilities can be used to make these changes. We will use the **trim**, **replace**, and **substring** methods, in this order:

```
msg = msg.trim(); // Trims whitespace  
msg = msg.replace("First Mate",  
"Quartermaster"); // Replaces text  
msg = msg.substring(0,47); // Returns  
first 48 characters.
```

Rather than writing these assignments individually, we can have one assignment statement with all of the methods chained. For simplicity, we also add the period with the string concatenation operator:

```
msg = msg.trim().replace("First Mate",  
"Quartermaster").substring(0,47) + ". ";
```

Whether methods are invoked separately or chained together, the end result is the same:

```
System.out.println (msg);  
$ Maroon the Quartermaster with a flagon  
of water.
```

Look for chaining on the exam.

Use StringBuilder Objects and Their Methods

Exam Objective 2.6 Manipulate data using the StringBuilder class and its methods

An object of the `StringBuilder` class represents a mutable character string, whereas an object of the `StringBuffer` class represents a thread-safe mutable character string. Remember that an object of the `String` class represents a immutable character string. In regards to the `StringBuilder` class and the exam, you need to familiarize yourself with the class's most common methods and constructors.

Methods of the `StringBuilder` Class

Several methods of the `StringBuilder` class are commonly used: `append`, `insert`, `delete`, `deleteCharAt`, and `reverse`. These methods are included in [Figure 3-4](#) and discussed in the following section.

FIGURE 3-4 Commonly used methods of the `StringBuilder` class

| StringBuilder | |
|--|--|
| + append(Object) :StringBuilder | |
| + append(String) :StringBuilder | |
| - append(StringBuilder) :StringBuilder | |
| + append(StringBuffer) :StringBuilder | |
| + append(CharSequence) :StringBuilder | |
| + append(CharSequence, int, int) :StringBuilder | |
| + append(char[]) :StringBuilder | |
| + append(char[], int, int) :StringBuilder | |
| + append(boolean) :StringBuilder | |
| + append(char) :StringBuilder | |
| + append(int) :StringBuilder | |
| + append(long) :StringBuilder | |
| + append(float) :StringBuilder | |
| + append(double) :StringBuilder | |
| + appendCodePoint(int) :StringBuilder | |
| + delete(int, int) :StringBuilder | |
| + deleteCharAt(int) :StringBuilder | |
| + indexOf(String) :int | |
| + indexOf(String, int) :int | |
| + insert(int, char[], int, int) :StringBuilder | |
| + insert(int, Object) :StringBuilder | |
| + insert(int, String) :StringBuilder | |
| + insert(int, char[]) :StringBuilder | |
| + insert(int, CharSequence) :StringBuilder | |
| + insert(int, CharSequence, int, int) :StringBuilder | |
| + insert(int, boolean) :StringBuilder | |
| + insert(int, char) :StringBuilder | |
| + insert(int, int) :StringBuilder | |
| + insert(int, long) :StringBuilder | |
| + insert(int, float) :StringBuilder | |
| + insert(int, double) :StringBuilder | |
| + lastIndexOf(String) :int | |
| + lastIndexOf(String, int) :int | |
| - readObject(java.io.ObjectInputStream) :void | |
| + replace(int, int, String) :StringBuilder | |
| + reverse() :StringBuilder | |
| + StringBuilder() | |
| + StringBuilder(int) | |
| + StringBuilder(String) | |
| + StringBuilder(CharSequence) | |
| + toString() :String | |
| - writeObject(java.io.ObjectOutputStream) :void | |

Coming up, you'll see a description of each method, followed by the method declarations and associated examples.

First, consider the following string:

```
StringBuilder mateyMessage = new StringBuilder ("Shiver Me Timbers!");
```

The string is shown again in [Figure 3-5](#) with the index values shown in relationship to the individual characters. This message is used for many of the examples throughout this section.

FIGURE 3-5 OCA StringBuilder message

| | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|
| S | h | i | v | e | r | M | e | T | i | m | b | e | r | s | ! | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |

Let's now take a look at some of the `StringBuilder` methods.

The append Method

The `StringBuilder` class's `append` method appends the supplied data as a character string.

There are 13 `append` method declarations. These overloaded methods are shown here in order to have coverage of the different Java types.

```
public StringBuilder append(Object o) {...}
public StringBuilder append(String str) {...}
public StringBuilder append(StringBuffer sb) {...}
public StringBuilder append(CharSequence s) {...}
public StringBuilder append(CharSequence s, int start, int end) {...}
public StringBuilder append(char[] str) {...}
public StringBuilder append(char[] str, int offset, int len) {...}
public StringBuilder append(boolean b) {...}
public StringBuilder append(char c) {...}
public StringBuilder append(int i) {...}
public StringBuilder append(long l) {...}
public StringBuilder append(float f) {...}
public StringBuilder append(double d) {...}
```

Here are some examples:

```
/* Appends the supplied character sequence to the string */
StringBuilder mateyMessage;
mateyMessage = new StringBuilder ("Shiver Me Tim");
/* Prints out "Shivers! Bad Storm!" */
System.out.println(mateyMessage.append(" Bad Storm!"));

StringBuilder e = new StringBuilder ("Examples:");
e.append(" ").append("1"); // String
e.append(" ").append(new StringBuffer("2"));
e.append(" ").append('\u0031'); // char
e.append(" ").append((int)2); // int
e.append(" ").append(1L); // long
e.append(" ").append(2F); // float
e.append(" ").append(1D); // double
e.append(" ").append(true); // true

/* Prints out "Examples: 1 2 1 2 1 2.0 1.0 true" */
System.out.println(e);
```

The insert Method

The `StringBuilder` class's `insert` method inserts the string representation of the supplied data starting at the specified location.

There are 11 `insert` method declarations, though you need to be familiar with only the most basic declarations:

```
public StringBuilder insert
    (int index, char[] str, int offset, int len) {...}
public StringBuilder insert(int offset, Object obj) {...}
public StringBuilder insert(int offset, String str) {...}
public StringBuilder insert(int offset, char [] str) {...}
public StringBuilder insert(int dstOffset, CharSequence) {...}
public StringBuilder insert
    (int dstOffset, CharSequence s, int start, int end) {...}
public StringBuilder insert(int offset, boolean b) {...}
public StringBuilder insert(int offset, char c) {...}
public StringBuilder insert(int offset, int i) {...}
public StringBuilder insert(int offset float f) {...}
public StringBuilder insert(int offset, double d) {...}
```

Here is an example:

```
StringBuilder mateyMessage;
mateyMessage = new StringBuilder ("Shiver Me Tim");
/* Prints out "Shiver Me Timbers and Bricks! */
System.out.println(mateyMessage.insert(17, " and Bricks"));
```

The delete Method

The `StringBuilder` class's `delete` method removes characters in a substring of the `StringBuilder` object. The substring begins with the value in the `start` argument and ends with the `end - 1` argument.

There is one `delete` method declaration:

```
public StringBuilder delete(int start, int end) {...}
```

Here is an example:

```
StringBuilder mateyMessage;
mateyMessage = new StringBuilder ("Shiver Me Tim");
/* Prints out "Shivers!"
System.out.println(mateyMessage.delete(6,16));
```

The deleteCharAt Method

The `StringBuilder` class's `deleteCharAt` method removes the character from the specified index.

There is one `deleteCharAt` method declaration:

```
public StringBuilder deleteCharAt (int index) {...}
```

Here is an example:

```
StringBuilder mateyMessage;
mateyMessage = new StringBuilder ("Shiver Me Tim");
/* Removes the '!' and prints out "Shiver Me Timbers"
System.out.println(mateyMessage.deleteCharAt(17));
```

The reverse Method

The `StringBuilder` class's `reverse` method reverses the order of the character sequence.

There is one `reverse` method declaration:

```
public StringBuilder reverse () {...}
```

Here is an example:

```
StringBuilder r = new StringBuilder ("part");
r.reverse();
/* Prints out "It's a trap!" */
System.out.println("It's a " + r + "!");
```

EXERCISE 3-3

Using Constructors of the `StringBuilder` Class

It's a good idea to get familiar with the different constructors of the `StringBuilder` class. Review the Javadoc documentation to determine when you would use the different constructors and perform the following steps:

1. Create `StringBuilder` instances with the `StringBuilder ()` constructor.
2. Create `StringBuilder` instances with the `StringBuilder (CharSequence seq)` constructor.
3. Create `StringBuilder` instances with the `StringBuilder (int capacity)` constructor.
4. Create `StringBuilder` instances with the `StringBuilder (String str)` constructor.

CERTIFICATION OBJECTIVE

Test Equality Between Strings and other Objects

Exam Objective 3.3 Test equality between strings and other objects using == and equals ()

One way to compare objects in Java is by performing a comparison by using the `equals` method. Note that the `equals` method in a class that will be used in a comparison must override the `equals` method of the `Object` class. The `hashCode` method also needs to be overridden, but this method is outside the scope of the test. The `String` class overrides the `equals` method as is shown in the following code sample:

```

/** The value is used for character storage. */
private final char value[];
/** The offset is the first index of the
storage that is used. */
private final int offset;
...
public boolean equals(Object anObject) {
if (this == anObject) {
    return true;
}
if (anObject instanceof String) {
    String anotherString = (String) anObject;
    int n = count;
if (n == anotherString.count) {
    char v1[] = value;
    char v2[] = anotherString.value;
    int i = offset;
    int j = anotherString.offset;
        while (n-- != 0) {
            if (v1[i++] != v2[j++])
                return false;
        }
        return true;
    }
}
return false;
}

```

(This code snippet was taken from jDocs, at www.jdocs.com/javase/7.b12/java/lang/String.html, and was provided here to stress the importance of using the `equals` method when comparing the textual values of strings.)

For this objective, we'll explore the `equals` method and the `==` operator a little further.

equals Method of the String Class

When comparing character strings among two separate strings, the `equals` method should be used:

```

String msg1 = "WALK THE PLANK!";
String msg2 = "WALK THE PLANK!";
String msg3 = ("WALK THE PLANK!");
String msg4 = new String ("WALK THE PLANK!");

System.out.println(msg1.equals(msg2)); // true
System.out.println(msg1.equals(msg3)); // true
System.out.println(msg1.equals(msg4)); // true
System.out.println(msg2.equals(msg3)); // true
System.out.println(msg3.equals(msg4)); // true

```

When comparing object references, the `==` operator should be used:

```
String cmd = "Set Sail!";
String command = cmd;
System.out.println(cmd == command); // true
```

Do not attempt to compare character sequence values of strings with the `==` operator. Even though the result may appear to be correct, this is an inappropriate use of the operator, because the `==` operator is designed to check whether two object references refer to the same instance of an object, and not the character sequences of strings. Here's an example that does return true, but this equality-testing approach should not be practiced:

```
String interjection1 = "Arrgh!";
String interjection2 = "Arrgh!";
System.out.println(interjection1 == interjection2); // Bad
```

If you do make this mistake, modern IDEs will typically flag the issue as a warning and give you the option to refactor the code to use the `equals` method, as shown here:

```
System.out.println(interjection1.equals(interjection2));
```

In addition, the IDE, such as NetBeans, may give you the option of refactoring in the `equals` method with nulls checks while using the ternary operator.

```
System.out.println((interjection1 == null ? interjection2 == null : interjection1.equals(interjection2)));
```

EXERCISE 3-4

Working with the `compareTo` Method of the `String` Class

Given the following code that prints, “0”:

```
String eggs1 = "Cackle fruit";
String eggs2 = "Cackle fruit";
System.out.println(eggs1.compareTo(eggs2));
```

1. Modify the string(s) so that a value greater than 0 is printed, and explain why this happens.
2. Modify the string(s) so that a value less than 0 is printed, and explain why this happens.

CERTIFICATION SUMMARY

This chapter discussed everything you need to know about operators and strings for the exam.

Operators of type assignment, arithmetic, relational, and logical were all presented in detail. Assignment operators included the general assignment, assignment by addition, and assignment by subtraction operators. Arithmetic operators included the addition, subtraction, multiplication, division, and remainder (modulus) operators, as well as the prefix increment, prefix decrement,

postfix increment, and postfix decrement operators. Relational operators included the less than, less than or equal to, greater than, greater than or equal to, equal to, and not equal to operators. Logical operators included the logical negation, logical AND, and logical OR operators.

We explored operator precedence and overriding operators. The concepts here were rather simple, and knowing the precedence of the common operators and the purposing of the parentheses in overriding precedence is all you'll need to retain.

Strings were discussed in three main areas: creating strings, the string concatenation operator, and methods of the `String` class. The following methods of the `String` class were covered: `charAt`, `indexOf`, `length`, `concat`, `replace`, `startsWith`, `endsWith`, `substring`, `trim`, `toLowerCase`, `toUpperCase`, and `equalsIgnoreCase`. The `StringBuilder` class was discussed in comparison to the `String` class. The following methods of the `StringBuilder` class were covered: `append`, `insert`, `delete`, `deleteCharAt`, and `reverse`.

Testing object and `String` equality using the `equals` method was also discussed. The `==` operator was also discussed and we explained when to and when not to use it.

Knowing the fine details of these core areas related to operators and strings is necessary for the exam.

TWO-MINUTE DRILL

Understand Fundamental Operators

- The exam covers the following assignment and compound assignment operators: `=`, `+=`, and `-=`.
- The assignment operator (`=`) assigns values to variables.
- The additional compound assignment operator is used for shorthand. As such, `a=a+b` is written `a+=b`.
- The subtraction compound assignment operator is used for shorthand. As such, `a=a-b` is written `a-=b`.
- The exam covers the following arithmetic operators: `+`, `-`, `*`, `/`, `%`, `++`, and `--`.
- The addition operation (`+`) is used to add two operands together.
- The subtraction operator (`-`) is used to subtract the right operand from the left operand.
- The multiplication operator (`*`) is used to multiply two operands together.
- The divisor operator (`/`) is used to divide the right operand into the left operand.
- The modulus operator (`%`) returns the remainder of a division.
- The prefix increment (`++`) and prefix decrement (`--`) operators are used to increment or decrement a value before it is used in an expression.
- The postfix increment (`++`) and postfix decrement (`--`) operators are used to increment or decrement a value after it is used in an expression.
- The exam covers the following relational operators: `<`, `<=`, `>`, `>=`, `==`, and `!=`.
- The less than operator (`<`) returns true if the left operand is less than the right operand.
- The less than or equal to operator (`<=`) returns true if the left operand is less than or equal to the right operand.
- The greater than operator (`>`) returns true if the right operand is less than the left operand.
- The greater than or equal to operator (`>=`) returns true if the right operand is less than or equal to the left operand.
- The equal to equality operator (`==`) returns true if the left operand is equal to the right operand.
- The not equal to equality operator (`!=`) returns true if the left operand is not equal to the right operand.

operand.

- Equality operators can test numbers, characters, Booleans, and reference variables.
- The exam covers the following logical operators: !, &&, and ||.
- The logical negation (inversion) operator (!) negates the value of the boolean operand.
- The logical AND (conditional AND) operator (&&) returns true if both operands are true.
- The logical AND operator is known as a short-circuit operator because it does not evaluate the right operand if the left operand is false.
- The logical OR (conditional OR) operator (||) returns true if either operand is true.
- The conditional OR operator is known as a short-circuit operator because it does not evaluate the right operand if the left operand is true.

Understand Operator Precedence

- Operators with a higher precedence are evaluated before operators with a lower precedence.
- Operator precedence is overridden by the use of parentheses.
- When multiple sets of parentheses are present, relative to operator precedence, the innermost set is evaluated first.
- Operators in an expression that have the same precedence will be evaluated from left to right.

Use String Objects and Their Methods

- An object of the `String` class represents an immutable character string.
- Mutable* means changeable. Note that Java variables such as primitives are mutable by default and can be made immutable by using the `final` keyword.
- The `CharSequence` interface is implemented by the `String` classes as well as the `StringBuilder` and `StringBuffer` classes. This interface can be used as an argument in the `String` class's `replace` method.
- The string concatenation operator (+) joins two strings together and creates a new string.
- The string concatenation operator will join two operands together, as long as one or both of them are strings.
- The `String` class's `charAt` method returns a primitive `char` value from a specified `int` index value in relationship to the referenced string.
- The `String` class's `indexOf` method returns a primitive `int` value representing the index of a character or string in relationship to the referenced string.
- The `String` class's `length` method returns a primitive `int` value representing the length of the referenced string.
- The `String` class's `concat` method concatenates the specified string to the end of the original string.
- The `String` class's `replace` method returns strings replacing all characters or strings in relationship to the referenced string.
- The `String` class's `startsWith` method returns a primitive `boolean` value representing the results of a test to see if the supplied prefix starts the referenced string.
- The `String` class's `endsWith` method returns a primitive `boolean` value representing the results of a test to see if the supplied suffix ends the referenced string.
- The `String` class's `substring` methods return new strings that are substrings of the referenced string.
- The `String` class's `trim` method returns the entire string minus leading and trailing whitespace characters in relationship to the referenced string.
- The `String` class's `toLowerCase` method returns the entire string as lower case characters.

- ❑ The `String` class's `toUpperCase` method returns the entire string as uppercase characters.
- ❑ The `String` class's `equalsIgnoreCase` method returns a boolean value after comparing two string while ignoring case consideration.

Use `StringBuilder` Objects and Their Methods

- ❑ An object of the `StringBuilder` class represents a mutable character string.
- ❑ An object of the `StringBuffer` class represents a thread-safe mutable character string.
- ❑ The `StringBuilder` class makes use of the following methods that the `String` class has also declared: `charAt`, `indexOf`, `length`, `replace`, `startWith`, `endsWith`, and `substring`.
- ❑ The `StringBuilder` class's `append` method appends the supplied data as a character string.
- ❑ The `StringBuilder` class's `insert` method inserts the string representation of the supplied data starting at the specified location.
- ❑ The `StringBuilder` class's `delete` method removes characters in a substring of the `StringBuilder` object.
- ❑ The `StringBuilder` class's `deleteCharAt` method removes the character from the specified index.
- ❑ The `StringBuilder` class's `reverse` method reverses the order of the character sequence.

Test Equality Between Strings and other Objects

- ❑ Use the `equals` method of the `String` class (overridden from the `Object` class) to test the equality of the character sequence values of string objects.
- ❑ Use the `==` operator to test to see if the object references (for example, the memory addresses) of strings are equal.
- ❑ Use the `==` operator to test the equality of primitives.

SELF TEST

Understand Fundamental Operators

1. Given:

```
public class ArithmeticResultsOutput {  
    public static void main (String[] args) {  
        int i = 0;  
        int j = 0;  
        if (i++ == ++j) {  
            System.out.println("True: i=" + i + ", j=" + j);  
        } else {  
            System.out.println("False: i=" + i + ", j=" + j);  
        }  
    }  
}
```

What will be printed to standard out?

- A. True: i=0, j=1
- B. True: i=1, j=1
- C. False: i=0, j=1
- D. False: i=1, j=1

2. Which set of operators represents the complete set of valid Java assignment operators?

- A. `%=`, `&=`, `* =`, `$ =`, `:=`, `/ =`, `^ =`, `| =`, `+ =`, `<<=`, `=`, `-=`, `>>=`, `>>>=`
- B. `%=`, `&=`, `*=`, `/=`, `*=`, `| =`, `+=`, `<<=`, `<<<=`, `=`, `-=`, `>>=`, `>>>=`

- C. $\%=$, $\&=$, $*=$, $/=$, $\|=$, $+=$, $<<=$, $=$, $-=$, $>>=$, $>>>=$
- D. $\%=$, $\&=$, $*=$, $\$=$, $/=$, $*=$, $\|=$, $+=$, $<<=$, $<<<=$, $=$, $-=$, $>>=$, $>>>=$

3. Given the following Java code segment, what will be printed, considering the usage of the modulus operators?

```
System.out.print(49 % 26 % 5 % 1);
```

- A. 23
- B. 3
- C. 1
- D. 0

4. Given:

```
public class BooleanResultsOutput {  
    public static void main (String[] args) {  
        boolean booleanValue1 = true;  
        boolean booleanValue2 = false;  
        System.out.print(! (booleanValue1 & !booleanValue2) + ", ");  
        System.out.print(! (booleanValue1 | !booleanValue2) + ", ");  
        System.out.print(! (booleanValue1 ^ !booleanValue2));  
    }  
}
```

What will be printed, considering the usage of the logical Boolean operators?

- A. false, false, true
- B. false, true, true
- C. true, false, true
- D. true, true, true

5. Given:

```

public class ArithmeticResultsOutput {
    public static void main (String[] args) {
        int i1 = 100; int j1 = 200;
        if ((i1 == 99) & (--j1 == 199)) {
            System.out.print("Value1: " + (i1 + j1) + " ");
        } else {
            System.out.print("Value2: " + (i1 + j1) + " ");
        }
        int i2 = 100; int j2 = 200;
        if ((i2 == 99) && (--j2 == 199)) {
            System.out.print("Value1: " + (i2 + j2) + " ");
        } else {
            System.out.print("Value2: " + (i2 + j2) + " ");
        }
        int i3 = 100; int j3 = 200;
        if ((i3 == 100) | (--j3 == 200)) {
            System.out.print("Value1: " + (i3 + j3) + " ");
        } else {
            System.out.print("Value2: " + (i3 + j3) + " ");
        }
        int i4 = 100; int j4 = 200;
        if ((i4 == 100) || (--j4 == 200)) {
            System.out.print("Value1: " + (i4 + j4) + " ");
        } else {
            System.out.print("Value2: " + (i4 + j4) + " ");
        }
    }
}

```

What will be printed to standard out?

- A. Value2: 300 Value2: 300 Value1: 300 Value1: 300
- B. Value2: 299 Value2: 300 Value1: 299 Value1: 300
- C. Value1: 299 Value1: 300 Value2: 299 Value2: 300
- D. Value1: 300 Value1: 299 Value2: 300 Value2: 299

6. Given the following code segment:

```

public void validatePrime() {
    long p = 17496; // 'prime number' candidate
    Double primeSquareRoot = Math.sqrt(p);
    boolean isPrime = true;
    for (long j = 2; j <= primeSquareRoot.longValue(); j++) {
        if (p % j == 0) {
            // Print divisors
            System.out.println(j + "x" + p / j);
            isPrime = false;
        }
    }
    System.out.println("Prime number: " + isPrime);
}

```

Which of the following is true? Hint: 17496 is not a prime number.

- A. The code will not compile due to a syntactical error somewhere in the code.
- B. The code will not compile since the expression ($p \% j == 0$) should be written as (($p \% j$) == 0).
- C. Divisors will be printed to standard out (for example, 2x8478, and so on), along with Prime number: false as the final output.
- D. Divisors will be printed to standard out (for example, 2x8478, and so on), along with “Prime number: 0” as the final output.

7. Given:

```
public class EqualityTests {  
    public static void main (String[] args) {  
        Integer value1 = new Integer("312");  
        Integer value2 = new Integer("312");  
        Object object1 = new Object();  
        Object object2 = new Object();  
        Object object3 = value1;  
    }  
}
```

Which expressions evaluate to true?

- A. value1.equals(value2)
- B. value1.equals(object1)
- C. value1.equals(object3)
- D. object1.equals(object2)

Understand Operator Precedence

8. Given:

```
System.out.print( true | false & true + "," );  
System.out.println( false & true | true );
```

What will be printed to standard out?

- A. true, true
- B. true, false
- C. false, true
- D. false, false
- E. Compilation error

Use String Objects and Their Methods

9. Given:

```
System.out.print(3 + 3 + "3");  
System.out.print(" and ");  
System.out.println("3" + 3 + 3);
```

What will be printed to standard out?

- A. 333 and 333
- B. 63 and 63
- C. 333 and 63
- D. 63 and 333

10. Consider the interface `CharSequence` that is a required argument in one of the `replace` method declarations:

```
public String replace(CharSequence target, CharSequence  
replacement) {  
    ...  
}
```

This `CharSequence` interface is a super interface to which concrete classes?

- A. `String`
- B. `StringBoxer`
- C. `StringBuffer`
- D. `StringBuilder`

11. Which statement is false about the `toString` method?

- A. The `toString` method is a method of the `Object` class.
- B. The `toString` method returns a string representation of the object.
- C. The `toString` method must return the object's state information in the form of a string.
- D. The `toString` method is commonly overridden.

12. Which `indexOf` method declaration is invalid?

- A. `indexOf(int ch)`
- B. `indexOf(int ch, int fromIndex)`
- C. `indexOf(String str, int fromIndex)`
- D. `indexOf(CharSequence str, int fromIndex)`

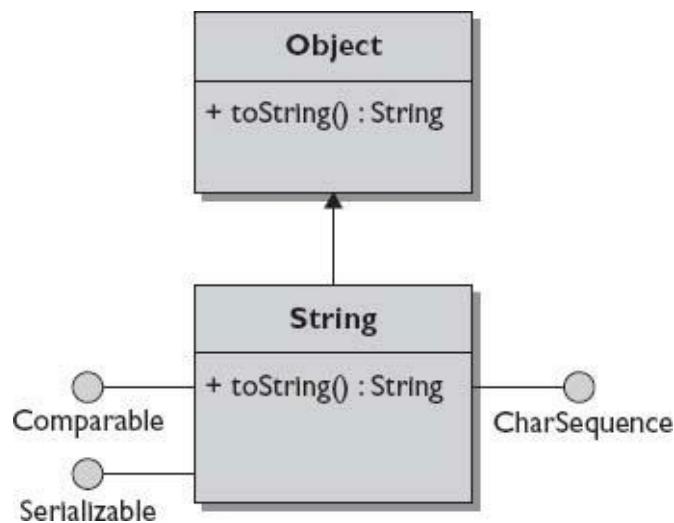
13. Given:

```
String tenCharString = "AAAAAAAAAA";  
System.out.println(tenCharString.replace("AAA", "LLL"));
```

What is printed to the standard out?

- A. AAAAAAAAAA
- B. LLLAAAAAAA
- C. LLLLLLLLLL
- D. LLLLLLLLLL

14. Consider the following illustration. Which statements, also represented in the illustration, are true?



- A. The String class implements the Object interface.
- B. The String class implements the Comparable, Serializable, and CharSequence interfaces.
- C. The `toString` method overrides the `toString` method of the Object class, allowing the String object to return its own string.
- D. The `toString` method is publicly accessible.

Use StringBuilder Objects and Their Methods

15. Which declaration of the `StringBuilder` class exists?

- A. `public StringBuilder reverse (String str) {...}`
- B. `public StringBuilder reverse (int index, String str) {...}`
- C. `public StringBuilder reverse () {...}`
- D. All of the above

Test Equality Between Strings and other Objects

16. Given:

```

String name1 = new String ("Benjamin");
StringBuilder name2 = new StringBuilder ("Benjamin");
System.out.println(name2.equals(name1));
  
```

Are the `String` and `StringBuilder` classes of comparable types? Select the correct statement.

- A. The `String` and `StringBuilder` classes are comparable types.
- B. The `String` and `StringBuilder` classes are incomparable types.

SELF TEST ANSWERS

Understand Fundamental Operators

1. Given:

```
public class ArithmeticResultsOutput {  
    public static void main (String[] args) {  
        int i = 0;  
        int j = 0;  
        if (i++ == ++j) {  
            System.out.println("True: i=" + i + ", j=" + j);  
        } else {  
            System.out.println("False: i=" + i + ", j=" + j);  
        }  
    }  
}
```

What will be printed to standard out?

- A. True: i=0, j=1
- B. True: i=1, j=1
- C. False: i=0, j=1
- D. False: i=1, j=1

Answer:

D. The value of j is prefix incremented before the evaluation; however, the value of i is not. Therefore, the expression is evaluated with a boolean value of false as a result, since 0 does not equal 1 (that is, i=0 and j=1). After the expression has been evaluated, but before the associated print statement is executed, the value of i is postfix incremented (that is, (i=1)). Therefore, the correct answer is False: i=1, j=1.

A, B, and C are incorrect answers as justified by the correct answer's explanation.

2. Which set of operators represents the complete set of valid Java assignment operators?

- A. %=, &=, * =, \$ =, : =, /=, *=, |=, +=, <<=, =, -=, >>=, >>>=
- B. %=, &=, *=, /=, ^=, |=, +=, <<=, <<<=, =, -=, >>=, >>>=
- C. %=, &=, *=, /=, *=, |=, +=, <<=, =, -=, >>=, >>>=
- D. %=, &=, *=, \$=, /=, *=, |=, +=, <<=, <<<=, =, -=, >>=, >>>=

Answer:

C. The complete set of valid Java assignment operators is represented.
 A, B, and D are incorrect answers. **A** is incorrect because \$= and := are not valid Java assignment operators. **B** is incorrect because <<<= is not a valid Java assignment operator. **D** is incorrect because \$= and <<<= are not valid Java assignment operators.

3. Given the following Java code segment, what will be printed, considering the usage of the modulus operators?

```
System.out.print(49 % 26 % 5 % 1);
```

- B. 3
C. 1
D. 0
-

Answer:

- D. The remainder of 49/26 is 23. The remainder of 23/5 is 3. The remainder of 3/1 is 0. The answer is 0.
 A, B, and C are incorrect answers as justified by the correct answer's explanation.
-

4. Given:

```
public class BooleanResultsOutput {  
    public static void main (String[] args) {  
        boolean booleanValue1 = true;  
        boolean booleanValue2 = false;  
        System.out.print (! (booleanValue1 & !booleanValue2) + ", ");  
        System.out.print (! (booleanValue1 | !booleanValue2) + ", ");  
        System.out.print (! (booleanValue1 ^ !booleanValue2));  
    }  
}
```

What will be printed, considering the usage of the logical Boolean operators?

- A. false, false, true
B. false, true, true
C. true, false, true
D. true, true, true
-

Answer:

- A. The first expression statement (`!(true & !(false))`) evaluates to `false`. Here, the right operand is negated to `true` by the (Boolean invert) operator, the Boolean AND operator equates the expression of the two operands to `true`, and the (Boolean invert) operator equates the resultant value to `false`. The second expression statement (`!(true | !(false))`) evaluates to `false`. Here, the right operand is negated to `true` by the (Boolean invert) operator, the Boolean OR operator equates the expression of the two operands to `true`, and the (Boolean invert) operator equates the resultant value to `false`. The third expression statement (`!(true ^ !(false))`) evaluates to `true`. Here, the right operand is negated to `true` by the (Boolean invert) operator, the Boolean XOR operator equates the expression of the two operands to `false`, and the (Boolean invert) operator equates the resultant value to `true`.

- B, C, and D are incorrect answers as justified by the correct answer's explanation.
-

5. Given:

```

public class ArithmeticResultsOutput {
    public static void main (String[] args) {
        int i1 = 100; int j1 = 200;
        if ((i1 == 99) & (--j1 == 199)) {
            System.out.print("Value1: " + (i1 + j1) + " ");
        } else {
            System.out.print("Value2: " + (i1 + j1) + " ");
        }
        int i2 = 100; int j2 = 200;
        if ((i2 == 99) && (--j2 == 199)) {
            System.out.print("Value1: " + (i2 + j2) + " ");
        } else {
            System.out.print("Value2: " + (i2 + j2) + " ");
        }
        int i3 = 100; int j3 = 200;
        if ((i3 == 100) | (--j3 == 200)) {
            System.out.print("Value1: " + (i3 + j3) + " ");
        } else {
            System.out.print("Value2: " + (i3 + j3) + " ");
        }
        int i4 = 100; int j4 = 200;
        if ((i4 == 100) || (--j4 == 200)) {
            System.out.print("Value1: " + (i4 + j4) + " ");
        } else {
            System.out.print("Value2: " + (i4 + j4) + " ");
        }
    }
}

```

What will be printed to standard out?

- A. Value2: 300 Value2: 300 Value1: 300 Value1: 300
- B. Value2: 299 Value2: 300 Value1: 299 Value1: 300
- C. Value1: 299 Value1: 300 Value2: 299 Value2: 300
- D. Value1: 300 Value1: 299 Value2: 300 Value2: 299

Answer:

B is the correct because Value2: 299 Value2: 300 Value1: 299 Value1: 300 will be printed to the standard out. Note that `&&` and `||` are short-circuit operators. So... When the first operand of a conditional AND (`&&`) expression evaluates to `false`, the second operand is not evaluated. When the first operand of a conditional OR (`||`) expression evaluates to `true`, the second operand is not evaluated. Thus, for the second and fourth `if` statements, the second operand isn't evaluated. Therefore, the prefix increment operators are never executed and do not affect the values of the `j[x]` variables.

A, C, and D are incorrect answers as justified by the correct answer's explanation.

6. Given the following code segment:

```

public void validatePrime() {
    long p = 17496; // 'prime number' candidate
    Double primeSquareRoot = Math.sqrt(p);
    boolean isPrime = true;
    for (long j = 2; j <= primeSquareRoot.longValue(); j++) {
        if (p % j == 0) {
            // Print divisors
            System.out.println(j + "x" + p / j);
            isPrime = false;
        }
    }
    System.out.println("Prime number: " + isPrime);
}

```

Which of the following is true? Hint: 17496 is not a prime number.

- A. The code will not compile due to a syntactical error somewhere in the code.
 - B. The code will not compile since the expression ($p \% j == 0$) should be written as (($p \% j$) == 0).
 - C. Divisors will be printed to standard out (for example, 2x8478, and so on), along with "Prime number: false" as the final output.
 - D. Divisors will be printed to standard out (for example, 2x8478, and so on), along with Prime number: 0 as the final output.
-

Answer:

- C. Divisors will be printed to standard out followed by Prime number: false. For those who are curious, the complete list of divisors printed are 2x8748, 3x5832, 4x4374, 6x2916, 8x2187, 9x1944, 12x1458, 18x972, 24x729, 27x648, 36x486, 54x324, 72x243, 81x216, and 108x162.
 - A, B, and D are incorrect. A is incorrect because there are no syntactical errors in the code. B is incorrect because a set of parentheses around $p \% j$ is not required. Answer D is incorrect because the code does not print out the character 0, it prints out the boolean literal value false.
-

7. Given:

```

public class EqualityTests {
    public static void main (String[] args) {
        Integer value1 = new Integer("312");
        Integer value2 = new Integer("312");
        Object object1 = new Object();
        Object object2 = new Object();
        Object object3 = value1;
    }
}

```

Which expressions evaluate to true?

- A. `value1.equals(value2)`
- B. `value1.equals(object1)`
- C. `value1.equals(object3)`

D. object1.equals(object2)

Answer:

- A** and **C**. **A** is correct because the class Integer implements the Comparable interface, allowing use of the equals method. **C** is correct because the Integer object was used to create the Object reference.
- B** and **D** are incorrect because the code cannot equate two objects with different references.
-

Understand Operator Precedence

8. Given:

```
System.out.print( true | false & true + ", " );
System.out.println( false & true | true );
```

What will be printed to standard out?

- A.** true, true
 - B.** true, false
 - C.** false, true
 - D.** false, false
 - E.** Compilation error
-

Answer:

- E.** Concatenation of a Boolean and a string is not allowed in the first print statement. Therefore, the first line will not compile. Changing the statement by adding parentheses around the Boolean evaluations would allow the line to compile:

```
System.out.print( (true | false & true) + ", " );
```

Once the line was compilable, the correct answer would have been A, considering operating precedence with the Boolean AND (&) operator having a higher precedence over the Boolean OR () operator.

- A, B, C, and D** are incorrect. **A** is incorrect because the code will not compile. **B, C, and D** are incorrect because if the code is refined to be compilable, the answer would have been A.
-

Use String Objects and Their Methods

9. Given:

```
System.out.print(3 + 3 + "3");
System.out.print(" and ");
System.out.println("3" + 3 + 3);
```

What will be printed to standard out?

- A.** 333 and 333
- B.** 63 and 63
- C.** 333 and 63

D. 63 and 333

Answer:

- D.** The (+) operators have left-to-right association. The first two operands of the first statement are numeric, so the addition (+) operator is used. Therefore, $3 + 3 = 6$. Since 6 + "3" uses a string as an operand, the string concatenation (+) operator is used. Therefore, concatenating the strings 6 and 3 renders the string 63. The last statement is handled a little differently. The first operand is a String; therefore the string concatenation operator is used with the other operands. Thus, concatenating strings "3" + "3" + "3" renders the string 333. The correct answer is 63 and 333.
- A, B, and C** incorrect. Note that changing ("3" + 3 + 3) to ("3" + (3 + 3)) would have rendered 36.
-

10. Consider the interface CharSequence that is a required argument in one of the replace method declarations:

```
public String replace(CharSequence target, CharSequence  
replacement) {  
    ...  
}
```

This CharSequence interface is a super interface to which concrete classes?

- A.** String
 - B.** StringBoxer
 - C.** StringBuffer
 - D.** StringBuilder
-

Answer:

- A, C, and D.** The concrete classes String, StringBuffer, and StringBuilder all implement the interface CharSequence. These classes can all be used in a polymorphic manner in regards to CharSequence being an expected argument in one of the String class's replace methods.
- B** is incorrect. There is no such thing as a StringBoxer class.
-

11. Which statement is false about the `toString` method?

- A.** The `toString` method is a method of the `Object` class.
 - B.** The `toString` method returns a string representation of the object.
 - C.** The `toString` method must return the object's state information in the form of a string.
 - D.** The `toString` method is commonly overridden.
-

Answer:

- C.** While the `toString` method is commonly used to return the object's state information, any information that can be gathered may be returned in the string.
- A, B, and D** are incorrect answers since they all represent true statements. **A** is incorrect because the `toString` method is a method of the `Object` class. **B** is incorrect because the `toString` method returns a string representation of the object. **D** is incorrect because the `toString` method is also commonly overridden.
-

12. Which indexOf method declaration is invalid?
- A. indexOf(int ch)
 - B. indexOf(int ch, int fromIndex)
 - C. indexOf(String str, int fromIndex)
 - D. indexOf(CharSequence str, int fromIndex)

Answer:

- D. The method declaration including indexOf(CharSequence str, int fromIndex) is invalid. CharSequence is not used as an argument in any indexOf method. Note that String, StringBuffer, and StringBuilder all declare their own indexOf methods.
- A, B, and C are incorrect because they are all valid method declarations.

13. Given:

```
String tenCharString = "AAAAAAAAAA";
System.out.println(tenCharString.replace("AAA", "LLL"));
```

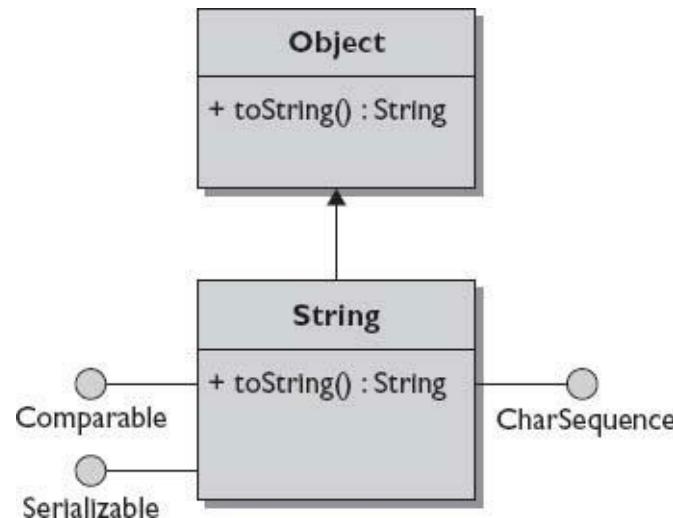
What is printed to the standard out?

- A. AAAAAAAA
- B. LLLAAAAAA
- C. LLLLLLLLA
- D. LLLLLLLLLL

Answer:

- C. The replace method of the String class replaces all instances of the specified string. The first three instances of AAA are replaced by LLL, making LLLLLLLA correct.
- A, B, and D are incorrect answers as justified by the correct answer's explanation.

14. Consider the following illustration. Which statements, also represented in the illustration, are true?



- A. The String class implements the Object interface.
- B. The String class implements the Comparable, Serializable, and CharSequence interfaces.

- C. The `toString` method overrides the `toString` method of the `Object` class, allowing the `String` object to return its own string.
 - D. The `toString` method is publicly accessible.
-

Answer:

- B, C, and D** are all correct because they represent true statements. **B** is correct because the `String` class implements the `Comparable`, `Serializable`, and `CharSequence` interfaces. **C** is correct because the `toString` method overrides the `toString` method of the `Object` class, allowing the `String` object to return its own string. **D** is correct because the `toString` method is also publicly accessible.
 - A** is incorrect. The `Object` class is a concrete class. Therefore, the `String` class does not implement an `Object` interface since there is no such thing as an `Object` interface. The `String` class actually extends an `Object` concrete class.
-

Use `StringBuilder` Objects and Their Methods

15. Which declaration of the `StringBuilder` class exists?

- A. `public StringBuilder reverse (String str) {...}`
 - B. `public StringBuilder reverse (int index, String str) {...}`
 - C. `public StringBuilder reverse () {...}`
 - D. All of the above
-

Answer:

- C.** The `reverse` method of the `StringBuilder` class does not have any arguments.
 - A, B, and D** are incorrect answers as justified by the correct answer's explanation.
-

Test Equality Between Strings and other Objects

16. Given:

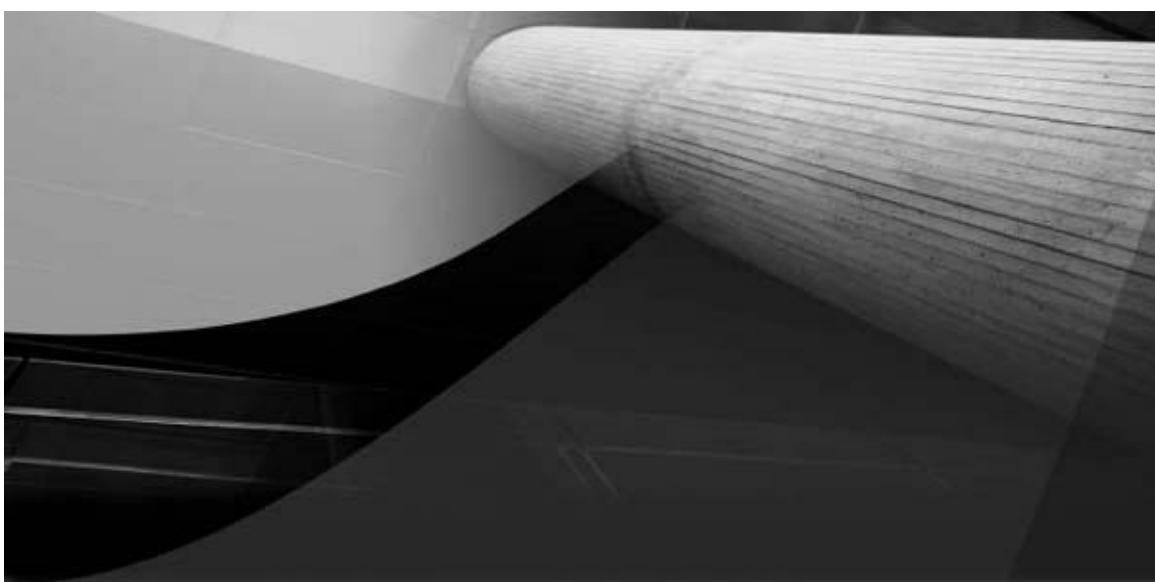
```
String name1 = new String ("Benjamin");
StringBuilder name2 = new StringBuilder ("Benjamin");
System.out.println(name2.equals(name1));
```

Are the `String` and `StringBuilder` classes of comparable types? Select the correct statement.

- A. The `String` and `StringBuilder` classes are comparable types.
 - B. The `String` and `StringBuilder` classes are incomparable types.
-

Answer:

- B.** The `String` and `StringBuilder` classes are incomparable types.
- A** is incorrect as the `String` and `StringBuilder` classes are incomparable types.



4

Working with Basic Classes and Variables

CERTIFICATION OBJECTIVES

- Understand Primitives, Enumerations, and Objects
 - Use Primitives, Enumerations, and Objects
- ✓ Two-Minute Drill

Q&A Self Test

This chapter will cover the basics of primitive variables and objects. The OCA will have questions that will require the test taker to understand the difference between primitives and objects and how each is used. The next few chapters will build off this foundation. The following topics will be covered in this chapter:

- Primitives
- Objects
- Arrays
- Enumerations
- Java's strongly typed nature
- Naming conventions

Understand Primitives, Enumerations, and Objects

Exam Objective 2.1 Declare and initialize variables

Exam Objective 2.2 Differentiate between object reference variables and primitive variables.

An application is made up of *variables* that store data and code that manipulates the data. Java uses *primitive* variables to store its most basic data. These primitives are then used to create more advanced data types called *objects*. This is what makes Java an object-oriented language: it allows the developer to store related code and data together in discrete objects.

This is a very important and fundamental concept that must be understood to understand how the Java language truly works.

Primitive Variables

Java primitives are a special subset of Java data types. They are the foundation of storing data in a program. It is important to understand what a primitive is and what it is not for the OCA exam. The Java language has eight primitives. The important thing to remember about each primitive is what kind of value you would store in it. The size in memory and minimum/maximum value sizes are good to know, but you will not be required to memorize them for the exam.

What Is a Primitive?

A primitive is the most basic form of data in a Java program, hence the name primitive. In fact, all advanced data types such as objects can be broken down into their primitive parts. When a primitive is declared, it reserves a certain number of bits in memory. The size of the memory allocation is dependent on the type of primitive. Each primitive data type has a set size that it will always occupy in memory.

The eight primitive data types are

- boolean (boolean)
- char (character)
- byte (byte)
- short (short integer)
- int (integer)
- long (long integer)
- float (floating point)
- double (double precision floating point)

Remember that something represented in code as an `Integer` (this refers to an object) is different from an `int` (which refers to a primitive containing an integer value).

While working with a primitive variable, you may only set it or read it. Calculations performed with primitives are much faster than calculations performed with similar objects.

Declaring and Initializing Primitives

Like all variables in Java, primitives need to be declared before they can be used. When a variable is declared, its type and name are set. The following code shows a primitive `int` being declared:

```
int gallons;
```

The `gallons` variable is now declared as a `int`. This variable can store only an integer and cannot be broken down into any smaller elements. Now that the variable is declared, it can only be set or read—that is all. No methods can be called using this variable because it is a primitive. This will be discussed in more depth later in the chapter when objects are explored. Primitives that are declared as instance variables get assigned a default value of 0 or false if they are a `boolean`. When primitives are used in the body of a method, they must be assigned a value before being used. If they are not assigned a value, a compile time error will be generated.

This code uses the new integer:

```
int gallons = 13;  
int totalGallons += gallons;
```

boolean Primitive

A `boolean` primitive is used to store a value of `true` or `false`. They store a 1-bit value and will default to `false` as an instance variable. Although they represent only 1 bit of information, their exact size is not defined in the Java standard and may occupy more space on different platforms. Valid literals include `true` and `false`.

```
boolean hasTurboCharger = true;  
hasTurboCharger = false;
```

char Primitive

The `char` primitive is used to store a single 16-bit Unicode character and requires 16 bits of memory. The range of a `char` corresponds to the minimum and maximum as defined by the Unicode specification '`\u0000`' (or 0) and '`\uffff`' (or 65,535 inclusive), respectively. When a `char` is set with a character in code, single quotes should be used—'`'Y'`', for example. A `char` has a default value of '`\u0000`', or 0, when used as an instance variable. This is the only Java primitive that is unsigned.

The following code segments demonstrate the uses of `char`. Valid literals include individual characters, special characters, Unicode characters, and hexadecimal and octal representations:

```
char c1 = 'S'; // S as a character  
char c2 = '\u0068'; // h in Unicode  
char c3 = 0x0065; // e in hexadecimal  
char c4 = 0154; // l in octal  
char c5 = (char) 131170; // b, casted (131170-131072=121)  
char c6 = (char) 131193; // y, casted (131193-131072=121)  
char c7 = '\''; // ' apostrophe special character  
char c8 = 's'; // s as a character  
char[] autoDesignerArray = {c1, c2, c3, c4, c5, c6, c7, c8};  
System.out.println(autoDesignerArray + "Mustang"); // Shelby's  
Mustang
```

See [Table E-1](#) in Appendix E on the Unicode standard for representation of printable and nonprintable ASCII characters as Unicode values.

byte Primitive

A byte is a Java primitive that is normally used to store small, signed numbers up to a byte in size. As an instance variable, it has a default value of 0. A byte occupies 8 bits of memory and can store an 8-bit signed two's complement non-floating point number. It has a maximum value of 127 and a minimum value of -128 inclusive. An integer is explicitly cast to 1 byte when an explicit cast is not performed. The following code segments demonstrate the uses of byte:

```
byte passengers = 4; // implicit cast from integer to byte
byte doors = (byte) 2; // explicit cast from integer to byte
```

short Primitive

A short is a Java primitive used for small numbers. It is most commonly used when the developer wants to save memory space over an int. As an instance variable, it has a default value of 0. A short occupies 16 bits of memory and can store a 16-bit signed two's complement non-floating point number. It has a maximum value of 32,767 and a minimum value of -32,768 inclusive. The following code segments demonstrate the uses of short:

```
short unladenWeightInLbs = 2350; // implicit cast to two bytes
short capacityInCu = (short) 427; // explicit cast to two bytes
```

int Primitive

An int is the most commonly used Java primitive. An int is used to store most whole numbers. As an instance variable, it has a default value of 0. An int occupies 32 bits of memory and can store a 32-bit signed two's complement non-floating point number. It has a maximum value of 2,147,483,647 and a minimum value of -2,147,483,648 inclusive. The following code segments demonstrate the uses of int:

```
int auctionPrice = 7800000;

char cylinders = '\u0008';
int cyl = cylinders; // implicit cast from char to integer

byte wheelbase = 90;
int wBase = wheelbase; // implicit cast from byte to integer

short horsepower = 250;
int hPower = horsepower; // implicit cast from short to integer

int length = (int) 151.5F; // floats must be explicitly casted
int powerToWeightRatio = (int) 405.1D; // doubles must be
explicitly casted
```

long Primitive

A long is a Java primitive used for whole numbers that are larger than an int can store. As an instance variable, it has a default value of 0L. The L appended to this number indicates that it is a long and not an int. A long occupies 64 bits of memory and can store a 64-bit signed two's complement non-floating point number. It has a maximum value of 9,223,372,036,854,775,807 and a minimum value of -9,223,372,036,854,775,808 inclusive. The following code segments demonstrate

the uses of long:

```
long mustangBingResults = 146000000L;
long mustangGoogleResults = 405000001;
/* explicit cast to long */
long mustangAmazonBookResults = (long) 5774;
/* implicit cast to long */
long mustangAmazonManualResults = 2380;
```

float Primitive

A float is a primitive that is used to store decimal values. It has a default value of 0.0f when used as an instance variable. This value is equal to zero, but the f or F appended to the end indicates that this value is a float, not a double. A float requires 32 bits of memory and may contain a 32-bit value with a maximum of $3.4e^{+38}$ and a minimum positive nonzero value of $1.4e^{-45}$ inclusive. (These values are rounded for simplicity. The exact size of a float is a formula that can be found in section 4.2.3 of “The Java Language Specification,” but this is beyond the scope of this book and the OCA exam.) The following code segments demonstrate the uses of float (note the use of f or F to denote that the number is a float):

```
float currentBid = 80100.99F;
float openingBid = 20000.00f;
float reservePrice = (float) 92000;
float myBid = 36000; // implicit cast from integer to float
```

double Primitive

A double is a primitive that is used to store large decimal values. The double primitive is the default primitive that most developers will use to store floating-point numbers. It has a default value of 0.0 as an instance variable. A double occupies 64 bits of memory and may contain a 64-bit value with a maximum of $1.8e^{+308}$ and a minimum positive nonzero value of $5e^{-324}$ inclusive. (These values are rounded for simplicity. The exact size of a double is a formula that can be found in section 4.2.3 of “The Java Language Specification,” but this is beyond the scope of this book and the OCA exam.) The following code segments demonstrate the uses of double:

```
double leafSpringCobraEngine = 4.7D;
double chrysler331Engine = 5.4d;
double ford427Engine = (double)7;
double ford428Engine = 7.01;
double fordV8Engine = 5; // implicit cast from integer to double
```

Why So Many Primitives?

There are eight Java primitives. The astute reader will likely be wondering why Java has so many, and why most seem to store the same data. This is a very good observation. The primitives byte, short, int, and long all store signed whole numbers. These primitives are listed in size order from smallest to largest. If a number such as 32 were to be stored, all four primitives would work without issue, since 32 is small enough to fit into a byte, which is the smallest type. The only negative to using a larger type such as a short, int, or long would be the space in memory that it occupies.

The primitives for storing floating-point numbers are `float` and `double`. A small number such as 5 can be stored in either with the only difference being the memory that the primitive consumes. However, as a floating-point number grows, it can lose precision. A `double` has double the precision of a `float`.

The `char` and `boolean` primitives are each unique. A `boolean` is the only primitive that can store a true or false value. And a `char` is the only primitive that stores an unsigned whole number.

Floating-Point Math

As a developer, you must be careful when using floating-point numbers. A floating-point number will lose precision as the number becomes larger. The following code is a simple example of two `floats` being added and printed to standard output:

```
float a = 19801216.0f;
float b = 20120307.12f;
float c = a + b;
//Format the float into US Currency format.
String d = NumberFormat.getCurrencyInstance().format(c);
System.out.println(d);
//Print the number directly as it is stored.
System.out.println(c);
```

It is easy to expect the output of this code segment to be \$39,921,523.12 after the `NumberFormat` class formats the number as U.S. currency. However, this is not the correct output. The code will instead output the following:

```
$39,921,524.00
3.9921524E7
```

The output is off by 0.88. This is an example of the `float` primitive losing the precision of a number. Although the output is not correct, it is expected based on how `floats` work. If the code segment is changed to use the `double` primitive, the correct output will be displayed. The following code segments use `double`, which increases the precision of the numbers:

```
double x = 19801216.0;
double y = 20120307.12;
double z = x + y;
//Format the double into US Currency format.
String s = NumberFormat.getCurrencyInstance().format(z);
System.out.println(s);
//Print the number directly as it is stored.
System.out.println(z);
```

This code segment will output the following:

```
$39,921,523.12
3.9921523120000005E7
```

This is the correct output. Notice that the raw `double`, `3.9921523120000005E7`, is much more precise than the raw `float`, `3.9921524E7`. However, it is important for you to understand that the `double` does not solve all problems like this. In this case, the precision needed was more than the

`float` was able to provide, but it was within the bounds of a double. If the required precision grew, or the number got larger, a double would have the same problem. This example illustrates why currency should never be stored as a double or `float`. The proper way to store currency, or any other floating-point number that needs guaranteed precision, is to use the Java class `BigDecimal`. `BigDecimal` can store any number, with the only limit being the memory your application is able to use. `BigDecimal` is not as fast to work with as primitives are, but it provides accuracy that primitives cannot provide. For more information on how `BigDecimal` works, see the Java Platform, Standard Edition 7 API Specification.

The following Scenario & Solution details each of the primitives that will appear on the OCA exam. It is important that you understand this content.

| SCENARIO & SOLUTION | |
|---|----------------------|
| What primitive would you use to store a value that will be true or false? | <code>boolean</code> |
| What primitive would you use to store a value that will be a whole number? | <code>int</code> |
| What primitive would you use to store a Unicode value? | <code>char</code> |
| What primitive would you use to store a value that may not be a whole number if you are concerned with memory size? | <code>float</code> |
| What primitive would you use to store large or high precision floating-point numbers? This also tends to be the default primitive for floating-point numbers. | <code>double</code> |
| What primitive would you use to store a very large whole number? | <code>long</code> |
| What primitive would you use to store a byte of data? | <code>byte</code> |
| What primitive would you use to store a value of 3000, without using any more memory than needed? | <code>short</code> |

Primitives vs. Their Wrapper Class

You know that primitives are the basic building blocks in Java and are one of the few things that are not objects. Java has a built-in wrapper class for each primitive that can convert a primitive to an object. The wrapper classes are `Integer`, `Double`, `Boolean`, `Character`, `Short`, `Byte`, `Long`, and `Float`. Notice that each of these classes starts with a *capital* letter, whereas a primitive begins with a *lowercase* letter. If you see a data type `Float`, it is referring to the object, while the data type `float` refers to the primitive. As of J2SE 5.0, a primitive will automatically be converted in either direction between its wrapper class and associated primitive. This feature is called *autoboxing* and *unboxing*.

The following is an example of an `Integer` object being initialized:

```
// An Integer is created and initialized to 5
Integer valueA = new Integer(5);
/*A primitive int is set to the int
value stored in the Integer object*/
int num = number.intValue();
// Autoboxing is used to convert an int to an Integer
Integer valueB = num;
```

Primitives and their equivalent objects can, in most cases, be used interchangeably. However, this is a bad coding practice. When performing math operations, using primitives will result in much faster calculations. Primitives also consume a smaller memory footprint.

When reading Java code that other developers have created, you will mostly see the primitives.

on the job *primitives int, double, and boolean. These three primitives cover most standard cases and are the standard default choice of most developers. Developers tend to worry only about optimizing the size of the primitive when they are used repeatedly in arrays or in data structures that remain persistent for a long period of time.*

Reviewing All Primitives

[Table 4-1](#) details all eight Java primitives. For the OCA exam, it is most important that you remember what data type you would use for the data you are storing. The size and range is nice to know but not required for the exam.

TABLE 4-1 Java Primitive Data Types

| Data Type | Used For | Size | Range |
|-----------|-------------------|---------|---------------------------------------|
| boolean | true or false | 1 bit | N/A |
| char | Unicode character | 16 bits | \u0000 to \xFFFF (0 to 65,535) |
| byte | integer | 8 bits | -128 to 127 |
| short | integer | 16 bits | -32,768 to 32,767 |
| int | integer | 32 bits | -2,147,483,648 to 2,147,483,647 |
| long | integer | 64 bits | - 2^{63} to $2^{63}-1$ |
| float | floating point | 32 bits | positive $1.4e^{-45}$ to $3.4e^{+38}$ |
| double | floating point | 64 bits | positive $5e^{-324}$ to $1.8e^{+308}$ |

Objects

Java is an object-oriented language, and as the name implies, your understanding what objects are and how they work is fundamental and very important. Almost everything you work with in Java is an object. Primitives are one of the few exceptions to this rule. You will learn what is stored in objects and how they help keep code organized.

Understanding Objects

Objects are a more advanced data type than primitives. They internally use primitives and other objects to store their data and contain related code. The data is used to maintain the state of the object, while the code is organized into methods that perform actions on this data. A well-designed class should be clearly defined and easily reused in different applications. This is the fundamental philosophy of an object-oriented language.

Objects vs. Classes and the New Operator

The distinction between objects and classes is important for you to understand. When a developer writes code, he or she is creating or modifying a class. A class is the file containing the code that the developer writes. It is a tangible item. A class is like a blueprint to tell the Java virtual machine (JVM) how to create an object at runtime. The new operator tells the JVM to create a new instance of this class, the result of which is an object. Many objects can be built from one class. The following is an example of a class. This class is employed to create an object used to represent a car.

```
public class Car {  
    int topSpeed;  
    boolean running;  
    Car(int topSpeed, boolean running) {  
        this.running = running;  
        this.topSpeed = topSpeed;  
    }  
    boolean isRunning() {  
        return running;  
    }  
}
```

The preceding class can be used to represent a car object. The class can store a boolean value that represents whether the car is running and an int value that represents the top speed. From this class, the JVM can create one or many instances of the car class. Each instance will become its own car object.

In the following code segment, two car objects are created:

```
Car fastCar = new Car(200,true);  
Car slowCar = new Car(100,true);
```

Both fastCar and slowCar are instances of the car class. Just like the primitives, each variable is declared as a car object. However, unlike primitives, the JVM does not automatically initialize an object. To initialize an object, the new operator must be used. The new operator tells the JVM that it needs to create a car object with the arguments given to the constructor. The new operator will always return a new and independent instance of the class.

Initializing Objects

Earlier in the chapter, you learned how each primitive has a finite predetermined size. Unlike a primitive, an object's size is not clearly defined. An object's size depends on the size of all the primitives and objects that it stores. Because objects store other objects, their size must also be considered. When an object is initialized, the JVM makes a reference in memory to the location of the object. Objects also have the ability to change in size as the objects they store grow or shrink. An object is declared in the same manner as a primitive, but it cannot be used until it has been initialized with the new operator or set equal to an existing object.

In the following example, we use the car class again:

```

/* This is legal. You can use the method isRunning because
the object has been initialized. */
Car bigCar;
bigCar = new Car(125,true);
boolean running = bigCar.isRunning();

/* This is legal. You can use the method isRunning because
the object smallCar has been set to the same initialized
object as bigCar. This will make smallCar and bigCar the same
object. */
Car smallCar;
smallCar = bigCar;
boolean running = smallCar.isRunning();

/* This is an illegal example. You cannot use a method on an
uninitialized object. */
Car oldCar;
boolean running = oldCar.isRunning();

```

Notice that, unlike a primitive, an object must be initialized with the `new` operator. Before initialization, an object is set to `null` by default. If a `null` object is used, it will throw a null pointer exception.

When to Use Objects

Primitives are used to store simple values. Integers or floating-point numbers within the bounds of a primitive data type are easy to store. Unfortunately, not all applications deal with values that fit neatly in the bounds of a primitive. For example, if an integer value is very large and has the potential to become even larger, the primitive `int` or `long` may not be appropriate. Instead, the developer may have to use one of the classes from the built-in Java packages for handling large numbers. These classes are `BigInteger` and `BigDecimal`. `BigInteger` is able to store a whole number and is limited in size only by the memory that the application can use. `BigDecimal` is similar to `BigInteger` except that it can store a floating-point number.

Objects can and should be created to store data that is similar. Remember that it is good object-oriented design to group together like code and data in a distinct class. Objects should be used to store complex related data.

A `NullPointerException` is a runtime exception that is thrown when an object is used that is `null`. This is a very common runtime error and can cause an application to exit. It can be very frustrating and time-consuming to track down. The best way to prevent this is to be careful, initialize your objects, and use a condition statement to check objects that may have been set to `null` elsewhere in code.

Here is an example of how to check whether an object is `null`:

```

DataObject x;
if(x != null){
    x.doAction();
}

```

Here, a `DataStore` object is declared and called `x`. The variable `x` is then checked to determine whether it is `null`. If it is not `null`, it proceeds inside of the `if` statement and executes the `doAction` method. In this case, `x` is never initialized so it would not pass the condition of the `if` statement. The

flow of execution would skip the body of the `if` statement and `doAction` would never be called. This prevents a `NullPointerException` from being thrown.

It is a good habit to check to see if unknown variables are `null` before they are used. Even if your application is in an error state, checking the variable will allow a better log message to be recorded before the application exits.

EXERCISE 4-1

Compile and Run an Object

This exercise will get you more familiar with objects. You will use the `Car` class and then add more functionality to it.

1. Copy the `Car` class into a text file and save it as `Car.java`.
2. Create a new text file and name it `Main.java`. This will be your main class. Copy the following code into this file:

```
public class Main {  
    public static void main(String[] args) {  
        // Your code goes here  
    }  
}
```

3. Use the following code to create a `Car` object:

```
Car yourCar = new Car(230,true);
```

4. Use the following code to display to the user whether or not the car is running:

```
System.out.println(yourCar.isRunning());
```
 5. Go back to the `Car` class and add a method to get the car's top speed.
 6. Add a line to your `Main.java` file that will display the car's top speed.
-

Arrays

An array is a series of variables of the same type that can be accessed by an index. Arrays are useful when all the variables in a set of data will be used in a similar way. Arrays can be made from primitive data types or objects. Even if an array is made of primitive data types, the `new` operator must be used.

The following is an example of an array made up of the `int` data type:

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

In this example, we declare a variable named `testScore` to be an integer array. This is done by adding box brackets to the end of the data type: `int[]` is the result. The `[]` after the data type means it will be an array. The box brackets should follow the data type, but it is valid for them to follow the variable name instead. Standard Java coding conventions suggest they should be used only with the data type. Regardless of whether the new array is of primitives or objects, it must be initialized with `new` and then with the data type. The number in brackets indicates the size of the array. In this

example, the array has three items. Each item is of type `int`. Individual elements in the array can be accessed or modified; they can also be placed in another `int`. The index for an array is zero-based. This means that the first element has an index of zero.

The example that follows demonstrates how an array can be used:

```
int[] testScore = new int[3];
testScore[0] = 98;
testScore[1] = 100;
testScore[2] = 72;
int shannonsTestScore = testScore[1];
```

Arrays are useful in loops. It is very common to access an array in a loop with a variable as the index. This variable would be incremented each time through the loop until the end of the array. The developer must take caution not to use an index that is out of bounds. An out of bounds index will cause the JVM to throw an exception at runtime. Once the size of an array is set, it cannot be changed. This makes arrays less useful for situations in which the data set may grow. [Figure 4-1](#) shows a basic array declaration.

FIGURE 4-1 Array declaration

```
int[] intArray = new int[4]


|        |        |        |        |
|--------|--------|--------|--------|
| int[0] | int[1] | int[2] | int[3] |
|--------|--------|--------|--------|


```

Arrays can also be multidimensional. A multidimensional array has more than one index. Arrays and their uses will be discussed in more detail in [Chapter 6](#).

Enumerations

Enumerations are a special data type in Java that allows for a variable to be set to predefined constants. The variable must equal one of the values that have been predefined for it. An enumeration is useful when there is a limited set of options that a variable can equal and it is restricted to these known values. For example, a deck of playing cards will always have four suits: clubs, diamonds, hearts, and spades. If a developer wanted to represent a card, an enumeration could be used to represent the suit.

```
enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }
```

This is an example of an enumeration that would be used to store the suit of a playing card. It is defined with the keyword `enum`. The `enum` keyword is used in the same manner as the `class` keyword. It can be defined in its own Java file or embedded in a class.

The next example demonstrates the use of an enumeration. The variable `card` is declared as a `Suit`. `Suit` was defined earlier as an enumeration. The `cardSuit` variable can now be used to store one of the four predefined suit values.

```
Suit cardSuit;
cardSuit = Suit.CLUBS;
if(cardSuit == Suit.CLUBS) {
    System.out.println("The suit of this card is clubs.");
}
```

Benefits of Using Enumerations

An object can be created to work in the same manner as an enumeration. In fact, enumerations were not even included in the Java language until version 5.0. However, enumerations make code more readable and provide less room for programmer error.

Java Is Strongly Typed

Java is a *strongly typed* programming language. Java requires that a developer declare the data type of each variable used. Once a variable is declared as one type, all data stored in it must be compatible with that type. Think back to the primitive data types we reviewed earlier in the chapter. For example, once a variable is declared as an `int`, only `int` data can be stored within it. Data can be converted from one type to another, as discussed in the following sections.

Understanding Strongly Typed Variables

Strongly typed variables help to create more reliable code. In most cases, the Java compiler will not allow the developer to store mismatched data types inadvertently. Only variables with the same data type are compatible with each other. For example, a `float` cannot be stored in any other data type other than a `float`. The same is true for all primitives and objects. The JVM will perform some automatic conversions for the developer. It is important that you understand that the types are not compatible and that the code will work only because of the conversion that is happening.

An example of a conversion would be going from an `int` to a `float`. The JVM will allow for an `int` to be placed into a `float` because it can convert an `int` to a `float` without losing precision. The converse is not true, however. A `float` cannot be converted to an `int` without the loss of precision.

Casting Variables to Different Types

Java does allow a variable to be cast to a different type. Casting tells the compiler that a variable of one type is intended to be used as a different type. To cast a variable, place the new data type in parentheses in front of the data or variable. Data can be cast only to types with which it is compatible. If data is illegally cast, the program will throw an exception at runtime.

An object can be cast to any parent or child object if the object was initialized as that child object. (This is an advanced concept of object-oriented languages and will be discussed in more detail in later chapters.) Primitives can also be cast to other primitives or compatible objects. For example, a `float` can be cast to an `int`. In this scenario, the cast would truncate the `float` to a whole number. Following are some examples of casting variables and data:

```
float floatNum = 1.5f;
int wasFloat = (int) floatNum;
```

The variable `wasFloat` would be equal to 1 since the .5 would be truncated to make the data compatible.

on the job *Casting variables is something that a developer should use lightly. There are times that variables must be cast, and there are even advanced programming techniques that rely on it. However, casting variables just adds unneeded complexity to the code. Cast data only when you have a good reason to do so.*

Naming Conventions

Using the correct naming conventions while creating a Java application is a critical step in creating easily readable and maintainable code. Java does not have many restrictions on how classes and objects can be named. However, nearly every experienced Java developer uses a single naming convention suggested by Oracle. For all the conventions, see Oracle's "Code Conventions for the Java Programming Language" specification. This section will cover the common conventions that will be seen on the OCA exam.

When you are creating a class, the class name should be a noun. The first letter should be capitalized along with each internal word after the first. Names should be short, yet descriptive. Shown next are some examples of good class names following the naming convention:

```
class SportCar {...}  
class BaseballPlayer {...}  
class Channel {...}
```

Variables should also have short but meaningful names. However, it is okay to use one-letter names for temporary variables. A variable's name should give an outside observer some insight as to what the variable is used for. The name should start with a lowercase letter, but each sequential internal word should be capitalized. Shown next is a sample of some variables named following the convention:

```
int milesPerGallon;  
float price;  
int i;  
Car raceCar;
```

e x a m watch

The OCA exam will ask questions about which variable is a primitive data type and which is an object. If you don't have a firm understanding of each, you may find the answers confusing. An important rule to remember is the OCA exam will always follow proper Java naming conventions and start object data types with a capital letter and begin primitive data types with a lowercase letter. For example, a float is a primitive data type, and a Float is an object.

CERTIFICATION OBJECTIVE

Use Primitives, Enumerations, and Objects

This section will build on the fundamental concepts that were discussed in the previous sections. The OCA exam will not require that you write code from scratch. However, the exam creators have decided to present scenarios in which the candidate will need to determine the best-suited code from a list of segments. The exam will also present segments of code and ask varying questions about its elements. This section specifically covers literals and practical examples of primitives, enumerations, and objects.

Literals

A *literal* is a term used for a hard-coded value used within code. A literal is any value that is not a variable. The following example demonstrates the use of literals:

```
int daysInMay = 31;  
int daysInJune;  
daysInJune = 30;  
char y = 'Y';
```

As used here, 31, 30, and 'Y' are examples of literals. Valid literal value formats for all primitives except the boolean type include character, decimal, hexadecimal, octal, and Unicode. Literals values for booleans must be true or false.

on the job *Starting in Java 7, it is now possible to insert underscores in numeric literals. This allows the developer to create more readable code. The underscores are ignored at runtime and are used solely to make the code easier to read. The underscores can be placed in both whole and floating-point numbers. They cannot be placed before the first number or after the last number, or adjacent to a decimal point. They also cannot be placed prior to the f and l suffixes for a float or long. The following are some examples of underscores in use:*

```
long creditCardNumber = 5555_5555_5555_5555L;  
int largeNumber = 1_000_000;  
float pi = 3.14_159_265;
```

Examples of Primitives, Enumerations, and Objects

This section will provide a few examples of all the topics covered in this chapter so far. Each example will be accompanied by an explanation. These examples will mimic the types of scenarios likely found on the OCA exam.

Primitives in Action

Primitives are the most basic data types in Java. As stated, they can only be set or read. The following sample program uses each primitive that will be on the OCA. This program calculates a baseball pitcher's earned run average (ERA).

```

public class EraCalculator{
    public static void main(String[] args) {
        int earnedRuns = 3;
        int inningsPitched = 6;
        int inningsInAGame = 9;
        float leagueAverageEra = 4.25f;
        float era = ((float)earnedRuns / (float)inningsPitched) *
            inningsInAGame;
        boolean betterThanAverage;
        if (era < leagueAverageEra) {
            betterThanAverage = true;
        } else {
            betterThanAverage = false;
        }
        char yesNo;
        if (betterThanAverage) {
            yesNo = 'Y';
        } else {
            yesNo = 'N';
        }
        System.out.println("Earned Runs\t\t" + earnedRuns);
        System.out.println("Innings Pitched\t\t" + inningsPitched);
        System.out.println("ERA\t\t\t" + era);
        System.out.println("League Average ERA\t" + leagueAverageEra);
        System.out.println("Is player better than league average " +
            yesNo);
    }
}

```

Notice that on the line where the variable ERA is calculated, the two `int` variables, `earnedRuns` and `inningsPitched`, are cast to a `float`. The cast to a `float` is needed so the division will be performed on variables of type `float` instead of their original type of `int`. The variables `earnedRuns`, `inningsPitched`, `inningsInAGame`, and `leagueAverageEra` are all set by literals. The preceding program shows how primitives are used. When the code is executed, it will produce the following output:

| | |
|--------------------------------------|------|
| Earned Runs | 3 |
| Innings Pitched | 6 |
| ERA | 4.5 |
| League Average ERA | 4.25 |
| Is player better than league average | N |

Primitives and Their Wrapper Class

The code segment that follows shows four variables being declared. Of the four, two are primitives and two are objects. The objects are instances of primitive wrapper classes. On the exam, pay close attention to how the variables are declared. There will likely be a segment where the question will ask how many of the variables are primitives.

```

Integer numberOfCats;
Float averageWeightOfCats;
int numberOfDogs;
float averageWeightOfDogs;

```

In the preceding example, `int` and `float` are primitives and `Integer` and `Float` are objects. Note that the capital F in `Float` signifies an object.

Enumerations

Enumerations are used for variables that can have the value of only a few predefined constants. The following example demonstrates a small class. This class has an enumeration defined that contains three different shoe types. The `createRunningShoes()` method is used to set the `shoe` variable to the enumerated type of a running shoe:

```
public class EnumExample {  
    enum TypeOfShoe { RUNNING, BASKETBALL, CROSS_TRAINING }  
    TypeOfShoe shoe;  
    void createRunningShoes() {  
        shoe = TypeOfShoe.RUNNING;  
    }  
}
```

Objects

The OCA exam does not require that you develop your own objects. But it is important that you understand the content of an object and be able to recognize it in code. It is also important that you understand how to use methods that an object contains. In the following example, a class is defined. This class is the `Heater` class. As its name suggests, it would be used to represent a heater or furnace. This is a very basic class, because it has only two methods and one instance variable. A more useful class would have many more of both.

```
class Heater{  
    int temperatureTrigger;  
    int getTemperatureTrigger() {  
        return temperatureTrigger;  
    }  
    void setTemperatureTrigger(int temperatureTrigger) {  
        this.temperatureTrigger = temperatureTrigger;  
    }  
}
```

The preceding `Heater` class stores the temperature that will trigger the system to turn on in the variable named `temperatureTrigger`. The two methods are used to get the value and set the value. These are called *getters* and *setters* and will be discussed in [Chapter 7](#).

The following code segments will use the `Heater` class to create an object:

```
Heater houseHeater = new Heater();  
houseHeater.setTemperatureTrigger(68);  
System.out.println(houseHeater.getTemperatureTrigger());
```

This code segment declares a new variable as a `Heater` object. The object is then initialized with the `new` keyword. The next line uses the `setTemperatureTrigger` method to modify the state of the

object. The final line uses the `getTemperatureTrigger` method to read this value and display it to the user. For the OCA exam, it is important that you be familiar with this syntax and understand how the methods are used.

EXERCISE 4-2

Creating Getters and Setters

Creating a dozen getter and setter methods by hand could take a while to complete. Fortunately, most modern IDEs have an automated way of creating getter and setter methods. Using this automated feature produces the methods with just a few mouse clicks. Follow along with this exercise to get a feel for how this works in the Eclipse IDE.

1. Create a few instance variables.
2. If you are using Eclipse
 - a. Highlight the desired instance variable you want to produce getters and setters for, and then right-click the mouse.
 - b. A pop-up menu will be displayed. Select Source followed by Generate Getters and Setters....
 - c. Another dialog box will be displayed with additional options that are not needed for this example. Finally, click the OK button to generate the methods.
3. If you are using Netbeans
 - a. In the NetBeans text editor highlight the instance variable and right-click.
 - b. From the drop-down box, select Refactor | Encapsulate Fields....
 - c. A pop-up box will display the highlighted instance variable already selected. You can then click the Refactor button to add the getter and setter with the default settings.

See [Chapter 7](#) for more information on getter and setter methods.

CERTIFICATION SUMMARY

In this chapter, some of the most fundamental concepts of Java were discussed in relationship to basic classes and variables. Even though the OCA exam includes only two objectives covering these concepts, many advanced concepts and objectives are built upon the content of this chapter. Your understanding of this chapter will result in a better understanding of the next few chapters.

Java primitives were examined first. Primitives are the basic building blocks of a Java program. The eight primitive variable data types that will appear on the OCA are `int`, `double`, `boolean`, `char`, `byte`, `short`, `long`, and `float`. It is important that you remember these primitives and what type of data they are designed to store.

Objects were then discussed. Objects are a very important concept for you to understand for the OCA exam. Objects are an advanced Java data type that can be custom created or found in the many Java packages that are included with the Java Virtual Machine. Objects are the pieces that interact to make up an application. Java is an object-oriented language, which means that nearly every aspect of the program is represented as objects, and the interaction between the objects is what gives an application its functionality.

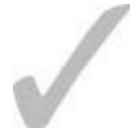
Arrays were then discussed. Arrays are good for keeping like data together. They use a zero-based index to access the individual elements of the array. Arrays can be of objects or primitives and must be initialized.

Enumerations were the last group of data types discussed. Enumerations are special objects that are used to denote a value among a preknown set of values. Although regular objects can be used to achieve the same results, enumerations provide a way to limit the data set without implementing a lot of custom code.

Next, the details of what makes Java a strongly typed language were examined. In general, the Java language allows for variables to change data type only by explicitly casting them to a new data type. If a variable is cast to a data type that it is not compatible with, an exception will be generated.

The final Java concept covered was Java naming conventions. Even though there are very few limitations on how variables and classes can be named, it is good coding practice to follow the conventions used by nearly every Java developer. Not following these conventions is a quick way to test the patience of your fellow developers.

The chapter concluded with a group of examples and explanations. These examples are important for you to understand. The OCA will not ask you to write large sections of code, but you must be able to understand code segments and determine what the output will be or if errors are present in the code.



TWO-MINUTE DRILL

Understand Primitives, Enumerations, and Objects

- Primitives are the fundamental data type in Java.
- int is a primitive data type that is used to store integer values. It is the default value for whole numbers.
- double is a primitive data type for large floating-point values. It is the default value for floating-point numbers.
- boolean is a primitive data type that is used to store true or false values.
- char is a primitive data type that is used to store a single Unicode character.
- byte is a primitive used to store small numbers which are a byte (8 bits) or smaller.
- short is a primitive used to store whole numbers up to 16 bits.
- long is a primitive used to store large whole numbers up to 64 bits.
- float is a primitive data type used to store floating-point values.
- Primitive data types all start with a lowercase letter, while classes start with a capital letter.
- Each primitive data type has a corresponding wrapper class: Integer, Double, Boolean, Character, Byte, Short, Long, and Float. Notice the capital letters.
- Objects are more advanced data types. They may be defined by a developer or found in a built-in Java package.

Use Primitives, Enumerations, and Objects

- Objects must be initialized by using the new keyword.
- Arrays allow you to store multiple variables together that can be accessed by an index.
- Enumerations allow a developer to create a predefined set of constants. A variable can then be set only to one of the predefined values.
- Java is a strongly typed language. Variables must be declared as a type, and any value that is

stored must be compatible with this type.

- It is possible to cast a variable to a different data type. If incompatible types are cast, an exception will be thrown.
- A literal is a value that is hard-coded in code as the value itself.
- Java naming conventions dictate that a class should be named with the first letter capitalized, along with each sequential word in the name.
- Java naming conventions dictate that a variable should be named with the first letter being lowercase, and with each sequential word in the name beginning with a capital letter.

SELF TEST

Understand Primitives, Enumerations, and Objects

- 1.** You need to create an application that is used to calculate the attendance at a baseball game. What data type would be most appropriate for storing the attendance?

 - A.** boolean
 - B.** char
 - C.** float
 - D.** int
- 2.** What is the best data type to use if you are required to perform many addition, subtraction, and multiplication calculations on a whole number?

 - A.** double
 - B.** Double
 - C.** int
 - D.** Integer
- 3.** You are writing a class that will store the status of an on/off switch. Which data type is most appropriate for storing this value?

 - A.** boolean
 - B.** char
 - C.** short
 - D.** int
- 4.** You have decided on the data type for a variable that will store the information about the on/off switch. Now you must determine a name for it. Which of the following names follows the Java naming conventions?

 - A.** LIGHTSWITCHENABLED
 - B.** lightswitchenabled
 - C.** lightswitchenabled
 - D.** x
- 5.** What is the best data type to use when storing a status code that may have one of the following values: success, failed, success with errors, or undetermined?

 - A.** Object
 - B.** Class
 - C.** boolean
 - D.** enum
 - E.** int
- 6.** A system has three sensors attached to it. You need a way to represent this system in your

program. What would be the best data type to use to model this system and sensors?

- A. Object
- B. boolean
- C. enum
- D. int

7. The new keyword is used to initialize which of the following data types? (Choose all that apply.)

- A. Object
- B. boolean
- C. Boolean
- D. Float
- E. float
- F. float[]

8. In the following line of code, what does the (int) represent?

```
number = (int)sensorReading;
```

- A. Rounding the sensorReading variable to the nearest int.
- B. Casting the sensorReading variable to the int data type.
- C. Nothing; it is there as a comment.

9. Given the following line of code, which of the lines of code listed are incorrect? (Choose all that apply.)

```
char c;
```

- A. c = new char();
- B. c = 'Y';
- C. c = '\u0056';
- D. c = "Yes";

Use Primitives, Enumerations, and Objects

10. Which of the following variables are being set with the use of a literal? (Choose all that apply.)

- A. int tvChannel = 4;
- B. char c = '5';
- C. char d = '\u0123';
- D. char e = c;
- E. int oldChannel = tvChannel;

11. Given the following line of code, what lines below are valid? (Choose all that apply.)

```
enum Sports { FOOTBALL, BASKETBALL, BASEBALL, TRACK }
```

- A. Sports sport = FOOTBALL;
- B. Sports sport = Sports.FOOTBALL;
- C. Sports sport = Sports.HOCKEY;
- D. Sports sport = 'TRACK'

12. What is wrong with the following method?

```
public double interestDue(double currentBalance, float interestRate){  
    double interestDue = currentBalance * interestRate;  
    return interestDue;  
}
```

- A. It should use all `float` primitives.
- B. It should use all `Float` objects.
- C. It should use all `double` primitives.
- D. It should use all `Double` objects.
- E. It should use `BigDecimal` objects.
- F. Nothing is wrong with this method.
- G. It does not compile because you cannot do math operations with primitives that are not the same type.

13. What is the correct way to create an array with five `int` data types? (Choose all that apply.)

- A. `int intArray = new int[5];`
- B. `int intArray = new int(5);`
- C. `int[] intArray = new int[5];`
- D. `int intArray[] = new int[5];`

14. What is the correct way to initialize a variable declared with the data type of `Book`, as a new `Book` object?

- A. `Book b;`
- B. `Book b = new Book();`
- C. `Book b = new Book[];`
- D. `Book b = Book();`

15. What is the difference between an `int` and an `Integer`?

- A. Nothing. They are both fully interchangeable.
- B. An `int` is an object and `Integer` is a primitive. An `int` is fastest when performing calculations.
- C. An `int` is a primitive and `Integer` is an object. An `int` is fastest when performing calculations.
- D. This is a trick question. There is no such thing as an `Integer`.
- E. This is a trick question. An `Integer` can be defined to be anything a developer wants it to be.

16. What is the result of using a method of an uninitialized object?

- A. Null is returned.
- B. The object is automatically initialized and an appropriate value is returned.
- C. A `NullPointerException` is thrown from the Java Virtual Machine.

17. What is wrong with the following code segment? (Choose all that apply.)

```
1:     float a = 1.2f;  
2:     int b = 5;  
3:     short c = 9;  
4:     long d = 6;  
5:     double e = b;  
6:     Double f = e;  
7:     Short g;  
8:     Float h = g.floatValue();
```

- A. Nothing is wrong.

- B.** There is an error on line 1.
- C.** There is an error on line 2.
- D.** There is an error on line 3.
- E.** There is an error on line 4.
- F.** There is an error on line 5.
- G.** There is an error on line 6.
- H.** There is an error on line 7.
- I.** There is an error on line 8.

18. What is the value of variable *x* in the following code segment?

```
float x = 0.0f;  
int y = 5;  
long z;  
x = y + 3.3f;  
x = x + z;
```

- A.** 0
- B.** 0.0f
- C.** 5.0f
- D.** 8.3f
- E.** 8.3
- F.** This code will not compile.

SELF TEST ANSWERS

Understand Primitives, Enumerations, and Objects

- 1.** You need to create an application that is used to calculate the attendance at a baseball game. What data type would be most appropriate for storing the attendance?

- A. boolean
- B. char
- C. float
- D. int

Answer:

- D.** The attendance of a baseball game is going to be a whole number within the range of an int.
 - A, B, and C** are incorrect. **A** is incorrect because boolean variables are used to store literals with values of true or false. **B** is incorrect because the char data type is used to store a single Unicode character. **C** is incorrect because float is used to store floating-point numbers.
-

- 2.** What is the best data type to use if you are required to perform many addition, subtraction, and multiplication calculations on a whole number?

- A. double
- B. Double
- C. int
- D. Integer

Answer:

- C.** An int is used to store whole numbers and is a primitive. Primitive variables perform calculations faster than their associated wrapper class.
 - A, B, and D** are incorrect. **A** is incorrect because a double is used for floating-point numbers. **B** is incorrect because Double is a primitive wrapper class used for floating-point numbers. **D** is incorrect because the Integer data type is the wrapper class for an int. You can tell that it is not a primitive because the first letter is capitalized like all class names. Performing calculations with an Integer would be much slower than doing so with the primitive int.
-

- 3.** You are writing a class that will store the status of an on/off switch. Which data type is most appropriate for storing this value?

- A. boolean
- B. char
- C. short
- D. int

Answer:

- A.** A boolean primitive is used to store true or false, which can be applied to a switch.
 - B, C, and D** are incorrect. They are all primitives used for different types of data.
-

- 4.** You have decided on the data type for a variable that will store the information about the on/off

switch. Now you must determine a name for it. Which of the following names follows the Java naming conventions?

- A. LIGHTSWITCHENABLED
 - B. LightSwitchEnabled
 - C. lightSwitchEnabled
 - D. x
-

Answer:

- C. A variable should begin with a lowercase letter, with each sequential word capitalized. The name should also be descriptive of what the variable is used for.
 - A, B, and D are incorrect.
-

5. What is the best data type to use when storing a status code that may have one of the following values: success, failed, success with errors, or undetermined?

- A. Object
 - B. Class
 - C. boolean
 - D. enum
 - E. int
-

Answer:

- D. An enum, or enumeration, is used to store data that has the possibility to be one of a few predefined data types.
 - A, B, C, and E are incorrect. A is incorrect because objects are used to store complex data structures. B is incorrect because classes are used to create objects. C and E are incorrect because both are primitives and not suitable for this specific application.
-

6. A system has three sensors attached to it. You need a way to represent this system in your program. What would be the best data type to use to model this system and sensors?

- A. Object
 - B. boolean
 - C. enum
 - D. int
-

Answer:

- A. An Object data type is one that the developer can define to represent the system and its state in the application's code.
 - B, C, and D are all incorrect because they are primitive types and cannot be defined to hold complex data structures.
-

7. The new keyword is used to initialize which of the following data types? (Choose all that apply.)

- A. Object
- B. boolean
- C. Boolean
- D. Float
- E. float

F. float[]

Answer:

- A, C, D, and F are correct. New is used to initialize any variable that is not a primitive.
 - B and E are incorrect because float and boolean are both primitive data types.
-

8. In the following line of code, what does the (int) represent?

```
number = (int)sensorReading;
```

- A. Rounding the sensorReading variable to the nearest int.
 - B. Casting the sensorReading variable to the int data type.
 - C. Nothing; it is there as a comment.
-

Answer:

- B. It is casting the variable sensorReading to an int.
 - A and C are incorrect.
-

9. Given the following line of code, which of the lines of code listed are incorrect? (Choose all that apply.)

```
char c;
```

- A. c = new char();
 - B. c = 'Y';
 - C. c = '\u0056';
 - D. c = "Yes";
-

Answer:

- A and D. A is a correct answer because the new keyword cannot be used with the primitive char. D is a correct answer because char cannot store a string.
 - B and C are incorrect answers because both lines are valid lines of code.
-

Use Primitives, Enumerations, and Objects

10. Which of the following variables are being set with the use of a literal? (Choose all that apply.)

- A. int tvChannel = 4;
 - B. char c = '5';
 - C. char d = '\u0123';
 - D. char e = c;
 - E. int oldChannel = tvChannel;
-

Answer:

- A, B, and C are correct. A literal is a value that is not a variable. A has the literal 4. B has the literal '5'. C has the literal '\u0123'.
- D and E are incorrect. D is incorrect because the variable c is used to set this char. E is incorrect because tvChannel is a variable.

11. Given the following line of code, what lines below are valid? (Choose all that apply.)

```
enum Sports { FOOTBALL, BASKETBALL, BASEBALL, TRACK }
```

- A. Sports sport = FOOTBALL;
- B. Sports sport = Sports.FOOTBALL;
- C. Sports sport = Sports.HOCKEY;
- D. Sports sport = 'TRACK'

Answer:

- B.** This is the only line that uses a sport that is in the enumeration and that uses the correct syntax.
- A, C, and D** are incorrect. **A** is incorrect because it uses incorrect syntax. **C** is incorrect because HOCKEY is not defined as a sport type. **D** is incorrect because the syntax is incorrect.

12. What is wrong with the following method?

```
public double interestDue(double currentBalance, float interestRate){  
    double interestDue = currentBalance * interestRate;  
    return interestDue;  
}
```

- A. It should use all float primitives.
- B. It should use all Float objects.
- C. It should use all double primitives.
- D. It should use all Double objects.
- E. It should use BigDecimal objects.
- F. Nothing is wrong with this method.
- G. It does not compile because you cannot do math operations with primitives that are not the same type.

Answer:

- E.** Primitives should not be used when working with currency on any value that cannot lose precision.
- A, B, C, D, F, and G** are incorrect.

13. What is the correct way to create an array with five int data types? (Choose all that apply.)

- A. int intArray = new int[5];
- B. int intArray = new int(5);
- C. int[] intArray = new int[5];
- D. int intArray[] = new int[5];

Answer:

- C and D.** **C** is the preferred way to declare an array. **D** is correct but does not follow standard conventions.
- A and B** are incorrect.

14. What is the correct way to initialize a variable declared with the data type of Book, as a new Book object?

- A. Book b;
- B. Book b = new Book();
- C. Book b = new Book[];
- D. Book b = Book();

Answer:

- B. The correct way to declare an object is to use new and then the object name followed by parentheses. The parentheses are used to pass arguments to the constructor if needed.
 - A, C, and D are incorrect. A is incorrect because it does not initialize a new Book object. C is incorrect because the square brackets are used instead of parentheses. D is incorrect because the new keyword is missing.
-

15. What is the difference between an int and an Integer?

- A. Nothing. They are both fully interchangeable.
- B. An int is an object and Integer is a primitive. An int is fastest when performing calculations.
- C. An int is a primitive and Integer is an object. An int is fastest when performing calculations.
- D. This is a trick question. There is no such thing as an Integer.
- E. This is a trick question. An Integer can be defined to be anything a developer wants it to be.

Answer:

- C. An int is a primitive, and primitives are faster when performing calculations. An Integer is an object. The capital letter I should help you distinguish objects from primitives.
 - A, B, D, and E are incorrect.
-

16. What is the result of using a method of an uninitialized object?

- A. Null is returned.
- B. The object is automatically initialized and an appropriate value is returned.
- C. A NullPointerException is thrown from the Java Virtual Machine.

Answer:

- C. A NullPointerException is thrown from the Java Virtual Machine.
 - A and B are incorrect. A is incorrect because a method cannot be called on an uninitialized object. However, an uninitialized object has a value of null and can be checked before a method is called. B is incorrect because an object cannot be automatically initialized, because it may need special parameters for its constructor or other custom initialization.
-

17. What is wrong with the following code segment? (Choose all that apply.)

```
1:     float a = 1.2f;
2:     int b = 5;
3:     short c = 9;
4:     long d = 6;
5:     double e = b;
6:     Double f = e;
7:     Short g;
8:     Float h = g.floatValue();
```

- A. Nothing is wrong.
- B. There is an error on line 1.
- C. There is an error on line 2.
- D. There is an error on line 3.
- E. There is an error on line 4.
- F. There is an error on line 5.
- G. There is an error on line 6.
- H. There is an error on line 7.
- I. There is an error on line 8.

Answer:

- I. The object g is never initialized. Therefore, when a method is called, it causes a compile time error.
- A, B, C, D, E, F, G, and H are incorrect. These are all valid lines of code.

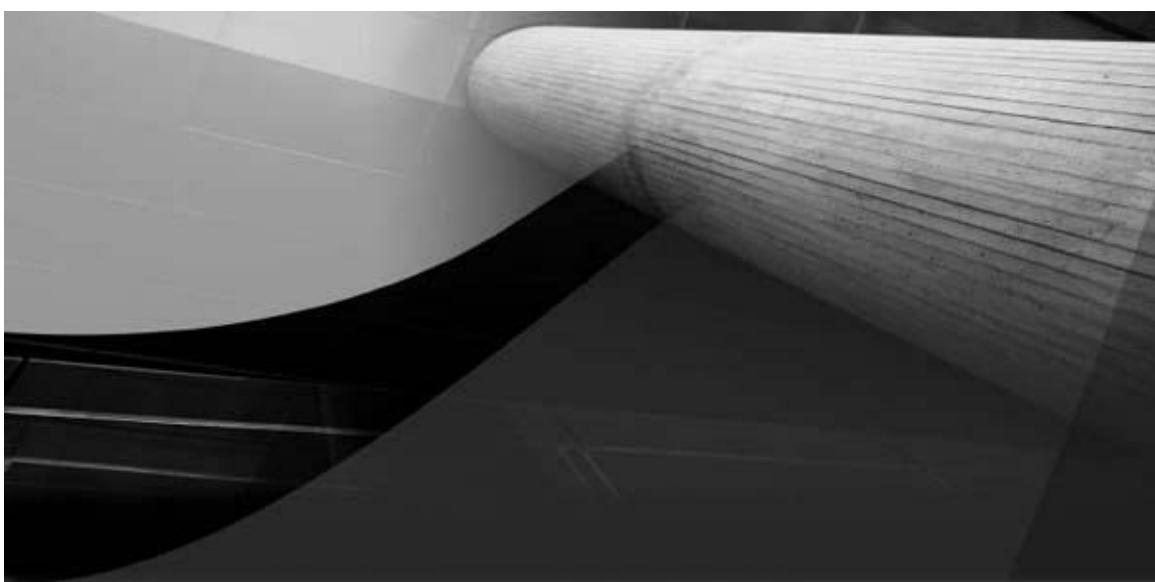
18. What is the value of variable x in the following code segment?

```
float x = 0.0f;
int y = 5;
long z;
x = y + 3.3f;
x = x + z;
```

- A. 0
- B. 0.0f
- C. 5.0f
- D. 8.3f
- E. 8.3
- F. This code will not compile.

Answer:

- F. The primitive z is never assigned a value. Therefore, this code will not compile.
- A, B, C, D, and E are incorrect. D would be correct if z was initialized to 0.



5

Understanding Methods and Variable Scope

CERTIFICATION OBJECTIVES

- Create and Use Methods
 - Pass Objects by Reference and Value
 - Understand Variable Scope
 - Create and Use Constructors
 - Use this and super Keywords
 - Create Static Methods and Instance Variables
- ✓ Two-Minute Drill

Q&A Self Test

This chapter will look at some of the details of methods and variable scoping. You are likely familiar with methods; this chapter will review the nuances of method and constructor creation. The effect of a static variable and method will be reviewed and demonstrated with code samples. The this and super keywords and their effect on the flow of execution will also be covered.

Create and Use Methods

Exam Objective 2.5 Call methods on objects

Exam Objective 6.1 Create methods with arguments and return values

Exam Objective 6.3 Create an overloaded method

This section will explore the construction and use of methods. Methods operate on data encapsulated within an object. They make up the backbone of your application. The way objects interact with methods defines the functionality of the software. The OCA exam will ask questions that expect you to understand how methods work. Code segments will be presented that use many different methods. You must understand the flow of the code as well as whether it is valid. This section will demonstrate how to create and use methods:

- Using method syntax
- Making and calling a method
- Overloading a method

Using Method Syntax

Methods are made up of five parts: the access modifier, the return type, the method name, the parameter list, and the body of the method. All methods must be defined inside of a class. Some of these parts can be empty. The following example shows the parts of a method and then a few valid examples:

```
//Parts of a method
<Access Modifier> <Return Type> <Method Name> (<Parameter List>)
{<Body>}
//Valid examples
public int getAttendance() /*Method Body*/
private double average(double firstNum, double secondNum)
/*Method Body*/
void saveToDisk() /*Method Body*/
```

Notice that although the first example does not include any parameters, it is valid. A method that has a no parameter list means that it does not need any outside input to perform its action. The last example has no access modifier. In this case, the method is using the default access modifier; this method is also valid.

Access Modifier

The Java language allows for methods to be given access modifiers that indicate to the compiler and Java virtual machine (JVM) what other objects can access this code. Java has four access modifiers. They are, in order of most restrictiveness, `private`, `default (package-private)`, `protected`, and `public`.

The `private`, `protected`, and `public` access modifiers are all indicated by placing them at the start of the method declaration. If no access modifier is declared, then the method will use the default modifier. Here is an example of all four being used:

```
private int getScore() /*Method Body*/
/*No access modifier is declared so
this will use default (package-private)*/
int getScore() /*Method Body*/
protected int getScore() /*Method Body*/
public int getScore() /*Method Body*/
```

Access modifiers, and what they do, will be discussed in more detail in [Chapter 7](#). For now, it is important that you understand that every method has an access modifier, even if it is the default one. For the rest of this chapter, all of the methods will use the least restrictive public access modifier for simplicity.

Return Type

Methods are able to return data to the code that has called them. A method can return either one variable or none at all. The returned variable may be an object or a primitive. Every method must declare a return type or use the void keyword to indicate that nothing will be returned to the caller from this method. Once a method declares that it will return a variable, a return statement must be included in the method body to pass the data back to the calling code. A return statement must include the keyword return followed by a variable or a literal of the declared return type. Once the return statement is executed, the method is finished. No code will execute after the return statement.

The following example shows a few methods; notice the return type in the method declaration:

```
public boolean isActive() {return true;}
public int getCurrentTotal() {return 5;}
public void processPendingData() /*Method Body*/
public ArrayList getAllAccounts() {return new ArrayList();}
```

In this example we see a boolean, an int, and an ArrayList all being returned by the different methods. The void indicates that the method does not return any data. In this simple example, the literal true, a 5, and a new ArrayList are returned.

Method Name

The method name indicates the method that will be used throughout the application to access its functionality. The name should be descriptive as to what the method does. [Chapter 7](#) will provide details about standard Java naming conventions.

The next example demonstrates a method with the name sampleMethod:

```
public void sampleMethod() /*Method Body*/
```

Parameter List

A method's parameter list follows the method name. It is enclosed in parentheses and is a comma-delimited list. A method may have no parameters, in which case the parentheses are empty. A method can contain up to 255 parameters, though it is best not to utilize this upward limit, since passing more than a handful of parameters is often a sign of bad design.

Parameters can be objects, enumeration types, or primitives. When creating the parameter list, the type is placed in front of the variable name. This is continued for each parameter and is separated by

commas. Following are a few examples of method declarations with various sized parameter lists:

```
public double areaOfRectangle(double height,  
    double width, double length) /*Method Body*/}  
public double areaOfCube(double sideLength) /*Method Body*/}  
public void drawCube() /*Method Body*/}
```

The first example has three parameters: (double height, double width, double length). The next example has only one parameter, (double sideLength), and the final example has no parameters.

Method Body

The method body is the main part of the method and contains all the code that makes up the functionality of the method. It may contain as few as one line of code or several hundred lines. If the method declares a return type, it must have a `return` statement that returns a literal or variable to match that type. If the return type is `void`, no `return` statement is required; however, the method can be used without a variable or literal to exit the method.

Following are two examples of a simple method with a complete body:

```
public int sum(int num1, int num2) {  
    int sum = num1 + num2;  
    return sum;  
}  
  
public void printString(String stringToPrint) {  
    System.out.println(stringToPrint);  
}
```

The first example demonstrates using the `return` statement to return a value. The second example does not return any data, which is indicated by the `void` keyword.

on the job *When you are creating methods, keep them focused and concise. If your method includes more than a few hundred lines of code, you may want to consider refactoring it into more than one method.*

Making and Calling a Method

A class comprises two things, variables and methods. The instance variables make up the state of the object, and the methods are responsible for all actions. When a class is created, the author must decide what its purpose is and implement that functionality as methods. Keep in mind the syntax rules discussed earlier in this chapter while we look at some scenarios.

The class for this example will be used to perform some simple math calculations. The first method that we'll create will determine the lower value of two `int` primitives. The public access modifier will be included since we do not want any restrictions on where this method can be used. This method should return to the caller the `int` that is smaller. Therefore, this method must have a return type of `int`. The method will be called `findLowerValue`. This method will accept two parameters; both are `int` primitives and will be called `number1` and `number2`. The method body will use conditional statements to determine what variable has a lower value. Here is this method:

```

public class MathTools {
    public int findLowerValue(int number1, int number2) {
        int result;
        if (number1 < number2)
            result = number1;
        else
            result = number2;
        return result;
    }
}

```

The parameters in this example are compared, and then the `int` with the lower value is stored in the `result` variable. This variable is then returned.

Methods are contained in objects. A method must be called from an instance of the object it is part of. The `findLowerValue` method in the preceding code is part of the `MathTools` class. To use this method, you must create an instance of `MathTools`. The syntax for calling a method is object name followed by a dot (.), and then the method name with its parameter list. The following code segment calls the `findLowerValue` method:

```

MathTools mTools = new MathTools();
int x = 8;
int y = 13;
int lowestInt = mTools.findLowerValue(x,y);
System.out.println("Result1 : " + lowestInt);
System.out.println("Result2 : " + mTools.findLowerValue(x,y));

```

First, this example must create a `MathTools` object and call it `mTools`. Two `int` primitives are then created. Next, the `findLowerValue` method is called. The `x` and `y` variables are passed to this method. Notice that the variable names in the parameter list do not have to match names; however, they must be a compatible type. The `findLowerValue` method returns the parameter that has the least value. This is stored in the `lowestInt` variable. Then the results are printed to standard out. The results are printed twice to highlight the fact that a method call with a return value can also be used in place of a variable of the same type. The output is as follows:

```

Result1: 8
Result2: 8

```

If a method has no parameters, the parameter list is left empty. Methods that do not return any data, return type `void`, cannot be used to set a variable as we did in the preceding example. However, the data from methods that do return a variable does not have to be used. In that case, the caller would be interested only in the action that the method performs.

The NetBeans IDE provides refactoring support allowing for automated actions such as safely renaming methods, moving methods to superclasses or subclasses, introducing a new method from an existing block of code (where the existing block of code is replaced with a call to the method), and generating accessor (getters) and mutator (setters) methods. For more information on using the refactoring features of the NetBeans IDE please reference NetBeans IDE Programmer Certified Expert Exam Guide (Exam 310-045), by Robert Liguori and Ryan Cuprak (McGraw-Hill,



Overloading a Method

Overloading methods is when more than one method shares the same name, but has a different parameter list. The parameter list may comprise more or less parameters than the other methods, or different types. Overloaded methods must not all have the same return type. This is a useful feature when a method may, for example, have five parameters; three of them are nearly always defaulted to a certain value, however. In this case, a method can be provided with all five parameters, plus another for convenience that has just two, and sets the other three to their defaults.

Overloading methods can also be useful when a method can produce a similar result from different data types as its parameters. In the previous example of the `findLowerValue` method, two `int` primitives were compared to find the one with the lower values. This method can be overloaded to work the same with `double` primitives. Here is an example of this method overloaded:

```
public double findLowerValue(double number1, double number2) {
    double result;
    if (number1 < number2)
        result = number1;
    else
        result = number2;
    return result;
}
```



Pay close attention to overloaded methods on the exam. It is easy to mistake which method is being used.

This method can be added to the `MathTools` class. The `findLowerValue` is now overloaded to work with both `int` and `double` primitives. The compiler determines what method is called by matching the parameter lists. If two `double` primitives are used, it executes the code in the second method; two `int` primitives would still use the original method.

The next example is a class called `LogManager` that can be used to perform some basic log management. It has two methods for printing log information to standard out. Here is this class:

```
public class LogManager {
    public void logInfo(String message, int errorNumber) {
        System.out.println("Error: " + errorNumber + " | " + message);
    }
    public void logInfo(String message) {
        logInfo(message, -1);
    }
}
```

The first method in this class has two parameters: the first is a `String` and is the message to be printed, and the second is a corresponding error number. This method will then print a line to standard out. The second method has only a `String` as a parameter. The `logInfo` method is overloaded since

there is more than one method with the same name. The second `logInfo` method will be used when a specific error number for the message is not included. In these cases, the error number will default to -1. Instead of implementing most of the same code as the first method, the second method will simply call the first method and pass in its default value.

The following code segment uses this case:

```
LogManager logManager = new LogManager();
logManager.logInfo("First log message", -299);
logManager.logInfo("Second log message");
```

Here's the output:

```
Error: -299 | First log message
Error: -1 | Second log message
```

A good design habit is to ensure that any overloaded methods behave in a similar manor. Confusing code can be created if two methods with the same name but different parameters have vastly different results.

CERTIFICATION OBJECTIVE

Pass Objects by Reference and Value

Exam Objective 6.8 Determine the effect upon object references and primitive values when they are passed into methods that change the values

This section will explore the way that Java passes data between methods. A variable will be passed by value if it's a primitive and by reference if it's an object. The differences between these two concepts are subtle, but can have a big effect on application design. The following is covered:

- Passing primitives by value to methods
- Passing objects by reference to methods

Passing Primitives by Value to Methods

When a primitive is used as an argument, a copy of the value is made and provided to the method. If the method sets the value of the parameter to a different value, it has no effect on the variable that was passed to the method. The following is an example of a method that adds 2 to the int primitive that is passed to it:

```
void addTwo(int value) {
    System.out.println("Parameter: value = " + value);
    value = value + 2;
    System.out.println("Leaving method: value = " + value);
}
```

Because primitives are passed by value, a copy of the variable value is passed to the method. Even

though the method modifies the parameter (since it is just a copy of the original argument used to invoke the method), the original argument remains unchanged from the perspective of the calling code. The following is a code segment that could be used to call this method:

```
int value = 1;
System.out.println("Argument: value = " + value);
addTwo(value);
System.out.println("After method call: value = " + value);
```

If this code segment were executed, it would produce the following results. Read the output that follows and walk through the code.

```
Argument: value = 1
Parameter: value = 1
Leaving method: value = 3
After method call: value = 1
```

Passing Objects by Reference to Methods

Objects are passed by reference to a method. This means that instead of making a copy of the object and passing it, a reference to the original object is passed to the method. A *reference* is basically an internal index that represents the object.



Note that this section can start a very technical conversation, which is beyond the scope of this book. The OCA is not going to drill into the details of how an object is passed internally. In short, for the OCA exam, it is important that you understand that any object is passed by reference. This means that the object passed to a method as an argument is the same as the object that the method receives as a parameter. Any changes to a parameter variable will be present when the code returns to the calling method.

The following example is similar to the example that demonstrated how primitives were passed by value. Instead of passing an int this time, it will pass a custom Number object. The following is the Number class:

```
public class Number {
    int number;
    public Number(int number) {
        this.number = number;
    }
    int getNumber() {
        return this.number;
    }
    void setNumber(int number) {
        this.number = number;
    }
}
```

Here is the method that will be called; this time, it will add 3 to the value passed to it:

```
void addThree(Number value) {  
    System.out.println("Parameter: value = " + value.getNumber());  
    value.setNumber(value.getNumber() + 3);  
    System.out.println("Leaving method: value = " + value.getNumber());  
}
```

Finally, here's the code segment used to call this method:

```
Number value = new Number(1);  
System.out.println("Argument: value = " + value.getNumber());  
addThree(value);  
System.out.println("After method call: value = " + value.getNumber());
```

This example is almost identical to the earlier one. The only difference is that now an object is passed by reference and the method adds 3 instead of 2. If this code segment were executed, the following would be the output. Notice that, this time, when the method returns to the calling code, the object has been modified.

```
Argument: value = 1  
Parameter: value = 1  
Leaving method: value = 4  
After method call: value = 4
```

e x a m watch

The OCA exam will include questions asking about the difference between passing variables by reference and value. The question will not directly ask you to identify the difference. It will present some code and ask for the output. In the group of answers to select from, you'll see an answer that will be correct if you assume the arguments are passed by value, and another answer that will be correct if the arguments are passed by reference. It is easy to get this type of question incorrect if you don't fully understand the concept of passing variables by reference and value.

CERTIFICATION OBJECTIVE

Understand Variable Scope

Exam Objective 1.1 Define the scope of variables

Exam Objective 2.4 Explain an object's lifecycle

This section will explore the way variables are organized in your code. As you can imagine, any nontrivial application will have countless variables. If variables could be accessed anywhere in the code, it would be difficult to find unique names that still conveyed a meaning. This scenario would

also promote bad coding practices. A programmer may try to access a variable that is in a completely different part of the program. To solve these problems, Java has variable scope. *Scope* refers to the section of code that has access to a declared variable. The scope may be as small as a few lines, or it may include the entire class.

In this section, we will cover the following topics:

- Local variables
- Method parameters
- Instance variables

Local Variables

The first variable scope that will be discussed is local variable scope. *Local variables* are the variables that are declared inside of methods. As the name implies, these variables are used locally in code. They are commonly declared at the start of a method and in loops, but they can be declared anywhere in code. A local variable may be a temporary variable that is used just once, or it can be used throughout a method.

The block of code in which a variable is declared determines the scope of the local variable. A block of code is determined by braces: { }. For example, if the variable is declared at the start of a method after the left brace ({), it would remain in scope until the method is closed with the right brace (}). Once a variable goes out of scope, it can no longer be used and its value is lost. The JVM may reallocate the memory that it occupies at any time.

A block of code can be created anywhere. A block can also be nested inside another block. A variable is in scope for the code block in which it is declared, and for all code blocks that exist inside it. The most common blocks are `if` statements and `for` or `while` loops.

The following example demonstrates the use of local variables in code blocks:

```
void sampleMethod() { // Start of code block A
    int totalCount = 0;
    for (int i = 0; i < 3; i++) { // Start of code block B
        int forCount = 0;
        totalCount++;
        forCount++;
        { // Start of code block C
            int block1Count = 0;
            totalCount++;
            forCount++;
            block1Count++;
        } // End of code block C
        { // Start of code block D
            int block2Count = 0;
            totalCount++;
            forCount++;
            block2Count++;
        } // End of code block D
        /* These two variables have no relation to the above ones of
         * the same name */
        int block1Count;
        int block2Count;
    } // End of code block B
} // End of code block A
```

Code block A is the method. Any variable that is declared in this block is in scope for the entire method. The variable `totalCount` is declared in block A; therefore, it can be accessed from anywhere else in the example method.

Code block B starts with the `for` loop. The variable `i` is declared in this block; even though it is not between the brackets, because it was declared in the `for` statement, it is considered to be in code block B. The variable `forCount` is also declared in block B. Because both of these variables are declared in block B, they are in scope only for block B and any blocks contained within B. They are out of scope for block A, and a compiler error would be generated if they were accessed from this block.

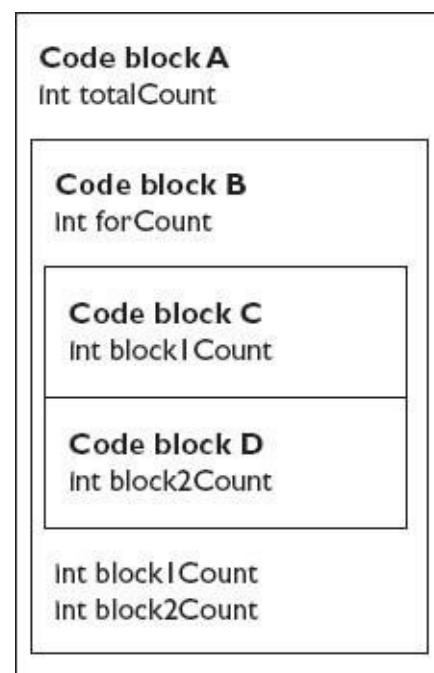
Contained inside block B is code block C. This block is started arbitrarily. In Java, it is valid to start a block of code at any time, although this is not done often in practice. The variable `block1Count` is declared in this block. It is in scope only for block C. However, any code inside block C also has access to the variables that have been declared in blocks A and B.

Block C is closed, and block D is created in block B. Block D contains the variable `block2Count`. This variable is in scope only for block D. Like block C, block D can also access variables that have been declared in blocks A and B. Understand that the variables in block C are not in scope for block D, and variables in D are not in scope for C.

In block B, the line after the closure of block D, two new variables are declared: `block1Count` and `block2Count`. These variables happen to have the same names as the variables that were declared in blocks C and D. Because the variables in blocks C and D are now out of scope, these two new variables have no relationship with the out scope variables. Giving variables the same names as others that are out of scope is valid in Java; however, it is not good coding practice because it can make the code difficult to maintain.

[Figure 5-1](#) represents another way to visualize this code example. Each code block represents scope. This figure shows each variable as it is declared in the code example. A variable is in scope from where it is declared until the end of its block; nested code blocks are included. These blocks correspond to the braces in the code example.

FIGURE 5-1 Code blocks visualized



The exam will ask scope-related questions, for which you will need to conceptualize the scope of various variables clearly. To help with scope conceptualization when working on a problem, you may want to draw out the code in blocks, as in [Figure 5-1](#). All the variables in the examples are considered local variables. Understanding variable scope is important. It is likely that at least a few questions on the test will center around a variable's scope.

A local variable is used when data needs to be accessed in only a certain part, or locally, in your code. For example, a variable used as a counter should be a local variable. Often times, a variable that you want to return will start as a local variable.

on the job When developing Java source code, variables should be declared with the most limited scope possible. This is a coding practice that helps reduce programming mistakes and improves code readability.

Method Parameters

Method parameters are the variables that are passed to the method from the calling segment of code. They are passed as arguments to the method. Method parameters may be primitives or objects. A method can have as many parameters as the developer defines up to 255. These variables are in scope for the entire method block. Method parameters are defined in the declaration for the method.

The following example contains two method parameters:

```
float findMilesPerHour(float milesTraveled, float hoursTraveled) {  
    return milesTraveled / hoursTraveled;  
}
```

In this example, `milesTraveled` and `hoursTraveled` are both method parameters. They are declared in the method's declaration. In this example, they are declared as `floats`. When this method is called, two `floats` must pass to the method as arguments. These two variables may be accessed anywhere in this method.

Instance Variables

Instance variables are declared in the class. They are called *instance variables* because they are created and remain in memory for as long as the instance of the class exists. Instance variables store the state of the object. They are not within the scope of any one particular method; instead, they are in scope for the entire class. They exist and retain their value from the time a class is initialized until that class is either reinitialized or no longer referenced.

The following example demonstrates two instance variables:

```
public class Television {  
    int channel = 0;  
    boolean on = false;
```

```

void setChannel(int channelValue) {
    this.channel = channelValue;
}
int getChannel() {
    return this.channel;
}
void setOn(boolean on) {
    this.on = on;
}
boolean isOn() {
    return this.on;
}
}

```

In this example, `channel` is declared as an `int`, and `on` is declared as a `boolean`. These are both instance variables. Remember that instance variables must be declared in the class, not in a method. The four methods in this class each access one of the instance variables. The `setChannel` method is used to set the instance variable `channel` to the value of the `int` that was passed to it as an argument. The `getChannel` and `isOn` methods return the values that are stored in the two instance variables, respectively. Notice that the `setOn` method has a parameter that is the same name as an instance variable. This is valid code. In a method that has these conditions, if the variable is referenced, it will be the method argument. To reference the instance variable, use the `this` keyword, which will be discussed in detail later in this chapter.

The following code segment demonstrates the `Television` class in use:

```

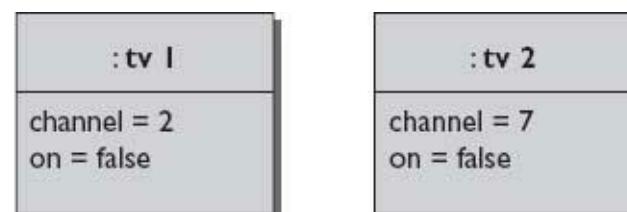
Television tv1 = new Television();
Television tv2 = new Television();
tv1.setChannel(2);
tv2.setChannel(7);
System.out.println("Television channel for tv1: " + tv1.getChannel());
System.out.println("Television channel for tv2: " + tv2.getChannel());

```

The first two lines of this example create two unique instances of the `Television` class. When the instance of the class is created, each object gets an instance variable that will store a channel. The next two lines use the `setChannel` method to set the channel to 2 and 7, respectively.

[Figure 5-2](#) represents the two objects that have been created and the value of both of their instance variables. The last two lines of code use the `getChannel` method to retrieve the value stored in the `channel` instance variable.

FIGURE 5-2 Two instances of the `tv` class



If the code were executed, the following would be the output. Remember that each `tv` object has a unique set of instance variables.

```
Television channel for tv1: 2  
Television channel for tv2: 7
```

The following Scenario & Solution covers types of variable scope and a likely use for each.

| SCENARIO & SOLUTION | |
|---|-------------------|
| What variable scope would be best suited for a counter in a loop? | Local variable |
| What variable scope must be used to store information about the state of an object? | Instance variable |
| What variable scope must be used to pass information to a method? | Method parameter |

An Object's Lifecycle

Objects are created and destroyed many times during the course of an application. This cycle is known as an object's *lifecycle*. Objects start out by being declared, when the data type and variable name are assigned. At this point, the code is telling the compiler that a variable of a certain name and type can be referenced in other places.

An object must then be *instantiated* and *initialized*. Instantiation of an object occurs when the new operator is used. At this point, the JVM allocates space for the object. Then the object is initialized with the object's *constructor*. The constructor sets all of the object's initial values and prepares the object to be used.

The object next enters the main part of its life. It is ready to have its methods called, which will perform actions and will store the data it was intended to store. As long as other active objects retain a reference to the object, it remains in this state. This is where the object will spend most of its time.

Once there are no other active references to the object, it is eligible to be removed from memory by the Java garbage collector. At this point, the object can no longer be called. The garbage collector will eventually reclaim the memory used for the discarded object, making it available for other uses.

CERTIFICATION OBJECTIVE

Create and Use Constructors

Exam Objective 6.4 Differentiate between default and user-defined constructors

Exam Objective 6.5 Create and overload constructors

A *constructor* is a special type of method in Java that, as its name implies, is used to construct an object. Every class in Java has a constructor. A class may have one or more user-defined constructors or it may use the compiler-added default constructor. This section will look at how to create and use constructors:

- Making a constructor
- Overloading a constructor
- Using the default constructor

Making a Constructor

The constructor is a special method that is used to initialize an object. The constructor is called after

the new operator. The next example shows the constructor for the `Integer` object being called, with one parameter:

```
Integer intObj = new Integer(7);
```

Every class is required to have at least one constructor, which may be user defined or the implicated default constructor that is added by the Java compiler. By defining a constructor, the developer is able to initialize the object any way he or she sees fit. This may mean initializing the instance variables to predetermined defaults and/or setting up resources such as a database connection.

A constructor always shares the name of the class, including the capitalized first letter. It is possible for a constructor to use any of the four access modifiers. However, only more common public access modifiers will be included on the OCA exam. Other modifiers tend to be used only in more advanced design patterns, and these patterns are not a part of the exam. Unlike a method, a constructor does not declare any return value, including void.

Here is an example of a constructor being defined in a class:

```
public class LoanDetails {  
    private int term;  
    private double rate;  
    private double principal;  
  
    public LoanDetails() {  
        term = 180;  
        rate = .0265; //Interest rate as a decimal  
        principal = 0;  
    }  
  
    public void setPrincipal(double p) {  
        principal = p;  
    }  
  
    public double monthlyPayment() {  
        return (rate * principal / 12)  
            / (1.0 - Math.pow(((rate / 12) + 1.0), (-term)));  
    }  
}
```

This class is used to calculate details about a basic loan. The constructor is called `LoanDetails`. It does not accept any parameters. Notice that it has no return type declared. When a `LoanDetails` object is created, the constructor sets the term, rate, and principal instance values to their default settings. This class assumes that the user will later set the principal amount based on the loan in question—that is why the setter, `setPrincipal`, for the principal instance variable is included with this class. This class includes another method for calculating the monthly payment of a loan. (You do not need to understand the details of this calculation, but it is accurate and based on industry standard formulas.)

Remember that constructors have no return type declared, not even the void keyword.

The following code segment demonstrates the use of this class:

```
LoanDetails ld = new LoanDetails();
ld.setPrincipal(150000);
System.out.println("Payment: " + ld.monthlyPayment());
```

This segment will produce the following output:

```
Payment: 1010.809999701624
```

Overloading a Constructor

Just as methods can be overloaded, so can constructors. To overload a constructor, the programmer must declare another constructor with the same name but a different parameter list. This is useful when a class wants to provide a simple constructor for most cases and a more advanced one for when it is needed. Continuing with the preceding example, the `LoanDetails` constructor can be overloaded to provide parameters for setting the term, rate, and principal. The next example shows the constructor that can be added to the `LoanDetails` class:

```
public LoanDetails(int t, double r, double p) {
    term = t;
    rate = r;
    principal = p;
}
```

This new constructor has three parameters, (`int t, double r, double p`), and allows the entire object to be set up and initialized directly from the constructor.

The next code segment demonstrates both constructors in use:

```
LoanDetails firstLD = new LoanDetails();
firstLD.setPrincipal(150000);
System.out.println("Payment 1 : " + firstLD.monthlyPayment());
LoanDetails secondLD = new LoanDetails(10, .025, 125000);
System.out.println("Payment 2 : " + secondLD.monthlyPayment());
```

This code is valid since the constructor is overloaded. The new constructor allows for the object to be initialized with the parameters passed to it. It is possible to overload the constructor as many times as needed as long as each constructor has a unique set of parameters. The parameters are considered unique if the data types do not match any of the other constructors' data types.

The output is as follows:

```
Payment 1 : 1010.809999701624
Payment 2 : 12643.676288957713
```

Using the Default Constructor

Every class must have a constructor. If no constructor is defined by the developer, the Java compiler adds one at compile time. This is called the *default constructor*. This constructor has no parameters. When called, it simply calls the superclass's constructor. This constructor is always named the same as the class. The default constructor is accessed in code the same way a user-defined constructor is called if it has no parameters. In fact, the calling code does not know whether the constructor is user-defined or the default.



If one user-defined constructor is defined, the compiler will not add a default one. This means methods are not guaranteed to have a constructor that takes no parameters.

CERTIFICATION OBJECTIVE

Use this and super Keywords

Exam Objective 7.5 Use super and this to access objects and constructors

The this and super keywords are used to access Java elements explicitly in relation to the current class. This section will look at how each work and when they should be used.

The this Keyword

The this keyword is used to refer explicitly to the current object. It is most commonly used when accessing instance variables. If a method or local variable has the same name as an instance variable, this can be used to refer explicitly to the instance variable. Following is an example of a class with one method and uses this to set the instance variable from the method parameter:

```
public class ScoreKeeper {  
    private int currentScore;  
    public void setCurrentScore(int currentScore) {  
        this.currentScore = currentScore;  
    }  
}
```

In this example the class contains an instance variable, currentScore. The setter for this variable has a parameter by the same name. To access the instance variable, this is used. If this were not used, any reference to currentScore in the setter would refer to the parameter.

The this keyword can also be used to call constructors from inside of the method. The next example will rewrite the LoanDetails class using the this keyword:

```

public class LoanDetails {
    private int term;
    private double rate;
    private double principal;

    public LoanDetails() {
        term = 180;
        rate = 0.0265;
        principal = 0;
    }

    public LoanDetails(int term, double rate, double principal) {
        this.term = term;
        this.rate = rate;
        this.principal = principal;
    }

    public void setPrincipal(double principal) {
        this.principal = principal;
    }

    public double monthlyPayment() {
        return (rate * principal / 12)
            / (1.0 - Math.pow((rate / 12) + 1.0, (-term)));
    }
}

```

In this example the parameter names of the constructor and setter have been changed to make the code more readable. However, they are now named the same as the instance variable. In both, `this` is used to access the instance variable. If `this` were not used, the parameter would be accessed instead.

on the job *It is always good practice to use the `this` keyword to access instance variables. It provides clarity when reading the code and lets the reader know immediately that an instance variable is being accessed.*

The `this` keyword can also be used to refer to methods and constructors in the same object. Often times, when a class has an overloaded constructor, the constructor that accepts the most parameters will do all of the initialized work and the other constructors will call it with sensible defaults. We can further rewrite the `LoanDetails` class to demonstrate this. The next code segment shows the constructor that does not accept any parameters rewritten:

```

public LoanDetails() {
    this(180, 0.025, 0);
}

```

This new constructor uses `this` to call the other constructor and passes it three parameters. It is valid to use `this` only in the first line of the constructor. A compiler error will be generated if `this` is used after that. This new constructor has the same effect as the original version of this method. However, this version is a better design. In some cases, the constructors may include many lines of code. It is easier to maintain one constructor and have the others call it than to maintain many different constructors.

The `this` keyword can also be used in front of methods in the same manner it is used with instances variables. However, this is not often used. An example is the following line of code, which would have to be in the `LoanDetails` class to be valid:

```
System.out.println("Payment " + this.monthlyPayment);
```

The super Keyword

The `super` keyword is used to refer to an object's superclass. It can be used to access methods that have been overridden in the current class or the constructor of a superclass. When the default constructor is used, it automatically calls its parent class's constructor by using `super`. A user-defined constructor can call its parent's constructor by using `super` on the first line. A compiler error will be generated if it is used past the first line. If a user-defined method does not use `super`, the compiler will automatically make a call with `super` to the parent class's constructor with no parameters. If this constructor does not exist in the parent class, a compile time error will be generated.

The following two classes show the `super` keyword in use:

```
public class ParentClass {  
    public ParentClass() {  
        System.out.println("ParentClass Constructor");  
    }  
    public ParentClass(String s) {  
        System.out.println("ParentClass Constructor " + s);  
    }  
}  
  
public class ChildClass extends ParentClass{  
    public ChildClass() {  
        System.out.println("ChildClass Constructor");  
    }  
    public ChildClass(String s) {  
        super(s);  
        System.out.println("ChildClass Constructor " + s);  
    }  
}
```

Both classes have two constructors. One accepts no parameters and the other accepts a string.

The `ChildClass` extends `ParentClass`. The following code segment shows these classes in use:

```
ChildClass childClass1 = new ChildClass();  
ChildClass childClass2 = new ChildClass("test");
```

The first line in this code segment calls the first constructor in the `ChildClass` class. Since there is no reference to `super`, the compiler automatically calls the constructor with no parameters of the parent class. The second line of code calls the `ChildClass` class's constructor that accepts a string as a parameter. This constructor then uses `super` to call its parent class's constructor that also accepts a string. Here is the output of this code segment:

```
ParentClass Constructor  
ChildClass Constructor  
ParentClass Constructor test  
ChildClass Constructor test
```

Trace through the code and ensure that you understand the flow of execution. The concepts presented here may be confusing at first but are important to grasp, because they will be seen in some form on the OCA exam.

The super keyword works with methods in a way similar to how it works with constructors. When a method has been overridden super can be used to gain access to the method from the parent class. Following is an example of this. This method would be added to the ParentClass:

```
public String className() {  
    return " ParentClass ";  
}
```

This method would be added to the ChildClass:

```
public String className(){  
    return "ChildClass -> " + super.className();  
}
```

Notice how this method is called super.methodName. That will call the method that is overridden in the parent class. Here is a code segment to demonstrate this:

```
ChildClass childClass = new ChildClass();  
System.out.println(childClass.className());
```

This segment creates a new object and then prints the results of the className method to standard out. The result of this code is as follows:

```
ParentClass Constructor  
ChildClass Constructor  
ChildClass -> ParentClass
```

The same text from the constructor is displayed as before. The className method appends its current name with that of its parent class by using super. Finally, it is printed to standard out.

exam watch

The this and super keywords can be used only on the first line of the constructor. Otherwise, a compile-time error will be generated.

CERTIFICATION OBJECTIVE

Create Static Methods and Instance Variables

Exam Objective 6.2 Apply the static keyword to methods and fields

A static method and field are also known as *class methods* and *class variables*. Both use the `static` keyword to indicate that they belong to the class and not an instance of a class. This section will look at how the `static` keyword is used to create class methods, class variables, and a common form for creating constants in Java.

Static Methods

The `static` keyword can be used to indicate that a method is a static, or class, method instead of a standard method. A static method belongs to the class, and not to an object or the instances of the class. Because these methods do not belong to the instance of an object, they cannot access any instance variables or standard methods. Static methods are also unable to use the `this` and `super` keywords. However, they can access other static methods or static variables. A static method is accessed by using the class name in place of the object instance name. Here is an example of a class with a static method:

```
public class Tools {  
    public static String formatDate(){  
        Date date = new Date();  
        Format formatter = new SimpleDateFormat("MMM-dd-yy");  
        return formatter.format(date);  
    }  
}
```

The `Tools` class has one method called `formatDate`. The `static` keyword is used to indicate that this is a static method. As shown in this example, the `static` keyword must be placed after the access modifier.

The next code segment is an example of this static method being used.

```
System.out.println(Tools.formatDate());
```

This single line of code prints the output of the method to standard out. Notice how the class name is used to call the method instead of an object. This is able to happen because the method is static and therefore belongs to the class, not to an object. Here is the output for this line of code:

Jun-30-12

Static methods are most commonly used as utility methods. For example, if you look though the Java API you will find that many of the methods in the `Math` class are static. This is because these methods are intended to perform a single task and have no reason to maintain the state of their own object. For example, the `sin` method in the `Math` class is used to find the trigonometric sine of an angle. It has no need for instance variables and is used more as a utility, or helper, method.

 **on the job** Next time you start a project, take a look at the main method. Notice that it is a static method.

Static Variables

The `static` keyword can also be used to create class variables. A class variable is similar to an instance variable, but instead of belonging to the instance of the object, it belongs to the instance of the class. The key difference between the two is that every instance of an object has access to the same class variable, but each instance has access to its own set of unique instance variables.

This example will demonstrate a class that is used to assign tracking numbers to packages. Each package must have a unique number. This class is called `ShippingPackage` and is shown next:

```
public class ShippingPackage {  
    public static int nextTrackingNumber = 100000;  
    private int packageTrackingNumber;  
  
    public ShippingPackage() {  
        this.packageTrackingNumber = nextTrackingNumber;  
        nextTrackingNumber++;  
    }  
    public int getPackageTrackingNumber() {  
        return packageTrackingNumber;  
    }  
}
```

This class has a static class variable that is called `nextTrackingNumber`. It is used to store the next available number for a package. This variable is set to 100000 initially, representing the first valid tracking number. The class also has an instance variable that is used to store the assigned tracking number. When the constructor is called it sets the instance variable, `packageTrackingNumber`, to the next available number stored in `nextTrackingNumber`. It then increments this variable. Since the `nextTrackingNumber` is static, every instance of the `ShippingPackage` class will access the same variable and data that it contains.

The next code segment will use the `ShippingPackage` class:

```
ShippingPackage packageOne = new ShippingPackage();  
ShippingPackage packageTwo = new ShippingPackage();  
ShippingPackage packageThree = new ShippingPackage();  
  
System.out.println("Package One Tracking Number: " +  
    packageOne.getPackageTrackingNumber());  
System.out.println("Package Two Tracking Number: " +  
    packageTwo.getPackageTrackingNumber());  
System.out.println("Package Three Tracking Number: " +  
    packageThree.getPackageTrackingNumber());
```

This code segment creates three instances of the `ShippingPackage` class. It then prints to standard out the tracking number that was assigned to each from the `ShippingPackage` constructor. The output of the code is as follows:

```
Package One Tracking Number: 100000  
Package Two Tracking Number: 100001  
Package Three Tracking Number: 100002
```

In this output, notice how each tracking number is incremented by 1. This happens because the static

variable, `nextTrackingNumber`, is a class variable and therefore common between all instances of the object.

Class variables with less restrictive access modifiers can be accessed in code similar to how class methods are accessed. The method name can be used. In the preceding example, the `nextTrackingNumber` variable is public. The following line of code can be used to get the variable and print it to standard out:

```
System.out.println("Next Tracking Number: " +  
    ShippingPackage.nextTrackingNumber);
```

Notice how the class name is used to access this public variable. The following will be output from this example:

```
Next Tracking Number: 100003
```

Constants

Static variables can also be used to create constants in code. A *constant* is simply a variable that has a set value at compile time that can never be changed. A Java constant is made by adding the `final` keyword to a class variable. The `final` keyword simply indicates that this variable can never have its value changed. Here is an example of a Java constant:

```
public static final double PI = 3.14;
```

This constant is for the value of `PI`. This can be accessed anywhere in your code by using its class name. This constant is already defined in the Java API, but to a much higher precision.

CERTIFICATION SUMMARY

Most of the objectives in this chapter cover a smaller section of the Java language. However, they are equally important to understand if you want to become a great developer.

This chapter began by discussing methods. Method syntax was reviewed and each part of a method declaration was examined. You must understand methods or you will find it difficult to understand deeper topics. Overloading methods is another critical topic covered in the first section. It is important that you remember that methods that are overloaded, which means they have the same name, must each have a unique parameter list.

The difference between passing objects to methods and passing primitives was also examined in this chapter. When a primitive is passed as an argument, a copy is made and the original will remain unchanged. However, when an object is passed as an argument, a reference of the object is passed to the method, and any changes made to the object in the method will be present in the original calling object.

Mastering the scope of variables is important as you design and read large Java applications. The next section covered these topics. First, local variables were discussed. These are the variables that should be used when the data needs to be stored for only a short amount of time and accessed from only one place in a class. The scope of these variables is the block of code in which they are declared.

Method parameters are the variables passed to the method. These variables will be used only when the method needs outside input from the calling code. They are declared in the method's signature and have a scope of the entire method.

The final scope was instance variables. These variables are declared inside of a class, but outside of a method. These variables are accessible throughout the entire class. They provide the state of an object and retain their value through the lifecycle of the object. Different methods may access them to read or modify the instance variables data.

Class constructors are a lot like normal methods, except that they are used only with the new operator to initialize an object. Unlike methods, constructors do not have any return type, including void. Every class must have a constructor. It may be user-defined, or a default constructor will be added if none has been defined. Like methods, constructors can also be overloaded.

The this and super keywords are used to reference objects that are in relation to the current object. The this keyword will reference the current object, and as the super keyword will reference the parent, or superclass, of the current object. The this keyword can be used on the first line of a constructor to call other constructors from the same class. The super keyword can similarly be used on the first line of a constructor, but it will call the parent class constructor.

Finally, this chapter reviewed the use of the static keyword. Java allows for variables and methods to be declared static, and the method or object belongs to the class, and not an object instance. Static methods, or class methods, can access only other static methods and variables. Static variables, or class variables, are shared between all instances of an object. Java constants are defined by using creating a static variable and marking it as final.

This chapter covers different material that may not always seem to be tightly related. However, your understanding all of these concepts will help you create rich applications that use sophisticated design patters. An OCA test taker is expected to understand these concepts. They will not all make up a huge part of this test, but a good understanding will help push your score higher.



TWO-MINUTE DRILL

Create and Use Methods

- Methods may return one variable or none at all.
- A method can be a primitive or an object.
- A method must declare the data type of any variable it returns.
- If a method does not return any data, it must use void as its return type.
- The syntax for a method is <Access Modifiers> <Return Type> <Method Name> (<Parameter List>) {<Body>}.
- Methods can be overloaded by creating another method with the same name but a different parameter list.
- Methods can be overloaded as much as needed.

Pass Objects by Reference and Value

- Primitives are passed by value.
- When passing a primitive by value, the value is copied into the method.
- Objects are passed by reference.
- When passing an object by reference, a reference to the object is copied into the method. Changes to the object will be present in all references to that object.

Understand Variable Scope

- A variable's scope defines what parts of the code have access to that variable.
- An instance variable is declared in the class, not inside of a method. It is in scope for the entire method and remains in memory for as long as the instance of the class it was declared in remains in memory.
- Method parameters are declared in the method declaration; they are in scope for the entire method.
- Local variables may be declared anywhere in code. They remain in scope as long as the execution of code does not leave the block in which they were declared.

Create and Use Constructors

- Constructors are used to initialize an object.
- If you do not define a constructor, the compiler will add a default one for you.
- A constructor can be overloaded with a unique parameter list as many times as needed.

Use this and super Keywords

- The `this` keyword is a reference to the current object.
- It is good practice to append `this` in front of instance variables.
- `this()` can be used to access other constructors in the current class.
- `this()` can be used only on the first line of a constructor.
- The `super` keyword is a reference to the parent object of the current object.
- `super` can be used to access the overridden methods of a parent.
- `super()` can be used to call the constructor of the parent class.
- `super()` can be used only on the first line of a constructor.

Create Static Methods and Instance Variables

- The `static` keyword can be used to create static methods and variables.
- A static method is also known as a class method.
- A static variable is also known as a class variable.
- Static methods belong to the class, not an object instance.
- Static variables belong to the class, not an object instance.
- Static methods can access only other static methods and variables.
- Every object of a class has access the same variable if it is static.
- The `static` keyword with the `final` keyword can be used to create Java constants.

SELF TEST

Create and Use Methods

1. A method needs to be created that accepts an array of `floats` as an argument and does not return any variables. The method should be called `setPoints`. Which of the following method declarations is correct?
 - A. `setPoints (float[] points) {...}`
 - B. `void setPoints (float points) {...}`
 - C. `void setPoints (float[] points) {...}`
 - D. `float setPoints (float[] points) {...}`
2. When the `void` keyword is used, which of the following statements are true? (Choose all that

apply.)

- A. A return statement with a value following it must be used.
- B. A return statement with a value following it can optionally be used.
- C. A return statement with a value following it should never be used.
- D. A return statement by itself must be used.
- E. A return statement by itself can optionally be used.
- F. A return statement by itself should never be used.
- G. A return statement must be omitted.
- H. A return statement can optionally be omitted.
- I. A return statement should never be omitted.

3. Given the SampleClass, what is the output of this code segment?

```
SampleClass s = new SampleClass();
s.sampleMethod(4.4, 4);

public class SampleClass {
    public void sampleMethod(int a, double b) {
        System.out.println("Method 1");
    }
    public void sampleMethod(double b, int a) {
        System.out.println("Method 2");
    }
}
```

- A. Method 1
- B. Method 2
- C. Method 1
Method 2
- D. Method 2
Method 1
- E. Compiler error

Pass Objects by Reference and Value

4. Given the class `FloatNumber` and method `addHalf`, what is the output if the following code segment is executed?

```

public class FloatNumber {
    float number;
    public FloatNumber(float number) {
        this.number = number;
    }
    float getNumber() {
        return number;
    }
    void setNumber(float number) {
        this.number = number;
    }
}

void addHalf(FloatNumber value) {
    value.setNumber(value.getNumber() + (value.getNumber()/2f));
}

/* CODE SEGMENT */
FloatNumber value = new FloatNumber(1f);
addHalf(value);
System.out.println("value = " + value.getNumber());

```

- A.** value = 1
B. value = 1.5
C. value = 2
D. value = 0
- 5.** Objects are passed by _____.
A. Value
B. Sum
C. Reference
D. Pointer
- 6.** Primitives are passed by _____.
A. Value
B. Sum
C. Reference
D. Pointer

Understand Variable Scope

- 7.** You need to create a class to store information about books contained in a library. What variable scope is best suited for the variable that will store the title of a book?
A. Local variable
B. Static variable
C. Global variable
D. Method parameter
E. Instance variable
- 8.** Given the SampleClass, when the following code segment is executed, what is the value of the instance variable size?

```
SampleClass sampleClass = new SampleClass(5);
public class SampleClass {
    private int size;
    public SampleClass(int size) {
        size = size;
    }
}
```

- A. 0
- B. 1
- C. 5
- D. Compiler error
- E. Runtime error

Create and Use Constructors

9. Given the SampleClass, what is the output of this code segment?

```
SampleClass sampleClass = new SampleClass();
public class SampleClass {
    private int size;
    private int priority;
public SampleClass(){
    super();
    System.out.println("Using default values");
}

public SampleClass(int size) {
    this.size = size;
    System.out.println("Setting size");
}

public SampleClass(int priority){
    this.priority = priority;
    System.out.println("Setting priority");
}
}
```

- A. Using default values
- B. Setting size
- C. Setting priority
- D. Compiler error

10. What constructor is equivalent to the one listed here?

```
public SampleConstructor() {  
    System.out.println("SampleConstructor");  
}
```

- A. public SampleConstructor() {
 this();
 System.out.println("SampleConstructor");
}
- B. public SampleConstructor() {
 super();
 System.out.println("SampleConstructor");
}
- C. public SampleConstructor() {
 this.SampleConstructor();
 System.out.println("SampleConstructor");
}
- D. public SampleConstructor() {
 super.SampleConstructor();
 System.out.println("SampleConstructor");
}
- E. None of the above

Use this and super Keywords

11. Given the SampleClass, what is the output of this code segment?

```
SampleClass sampleClass = new SampleClass();  
public class SampleClass {  
    private int size;  
  
    public SampleClass(){  
        this(1);  
        System.out.println("Using default values");  
    }  
  
    public SampleClass(int size) {  
        this.size = size;  
        System.out.println("Setting size");  
    }  
}
```

- A. Using default values
- B. Setting size
- C. Using default values
Setting size
- D. Setting size
Using default values

E. Compiler error

12. What is the effect of the following line of code?

`super()`

- A.** The method that is overridden by the current method is called.
- B.** The parent class's constructor is called.
- C.** The current class's constructor is called.
- D.** The child class's constructor is called
- E.** The current method is recursively called.

Create Static Methods and Instance Variables

13. Given the `SampleClass`, what is the output of this code segment?

```
SampleClass s = new SampleClass();
SampleClass.sampleMethodOne();
public class SampleClass {
    public static void sampleMethodOne() {
        sampleMethodTwo();
        System.out.println("sampleMethodOne");
    }
}

public void sampleMethodTwo() {
    System.out.println("sampleMethodTwo");
}
}
```

- A.** sampleMethodOne
- B.** sampleMethodTwo
- C.** sampleMethodOne
sampleMethodTwo
- D.** sampleMethodTwo
sampleMethodOne
- E.** Compiler error

14. Given the `SampleClass`, what is the value of `currentCount` for the instance of object `x` after the code segment had be executed?

```
SampleClass x = new SampleClass();
SampleClass y = new SampleClass();
x.increaseCount();
public class SampleClass {
    private static int currentCount=0;

    public SampleClass() {
        currentCount++;
    }

    public void increaseCount() {
        currentCount++;
    }
}
```

- A. 0
- B. 1
- C. 2
- D. 3
- E. Compiler error
- F. Runtime error

15. Static methods have access to which of the following? (Choose all that apply.)

- A. Static variables
- B. Instance variables
- C. Standard methods
- D. Static Methods
- E. None of the above

SELF TEST ANSWERS

Create and Use Methods

1. A method needs to be created that accepts an array of `floats` as an argument and does not return any variables. The method should be called `setPoints`. Which of the following method declarations is correct?

- A. `setPoints (float[] points) {...}`
 - B. `void setPoints (float points) {...}`
 - C. `void setPoints (float[] points) {...}`
 - D. `float setPoints (float[] points) {...}`
-

Answer:

- C. `void` must be used for methods that do not return any data.
 - A, B, and D are incorrect. A is incorrect because it is missing the return type. If the method is not going to return a variable, it still must use `void`. B is incorrect because it does not have an array of `floats` as a parameter. D is incorrect because it uses the incorrect return type.
-

2. When the `void` keyword is used, which of the following statements are true? (Choose all that apply.)

- A. A return statement with a value following it must be used.
 - B. A return statement with a value following it can optionally be used.
 - C. A return statement with a value following it should never be used.
 - D. A return statement by itself must be used.
 - E. A return statement by itself can optionally be used.
 - F. A return statement by itself can never be used.
 - G. A return statement must be omitted.
 - H. A return statement can optionally be omitted.
 - I. A return statement should never be omitted.
-

Answer:

- C, E, and H. When `void` is used, it indicates that there is no returned data from the method. C is correct because it is not valid to return any data. E is correct because it is valid to optionally use a `return` by itself. H is correct because the `return` statement can also be optionally omitted.
 - A, B, D, F, G, and I are incorrect. A and B are incorrect since the `void` keyword means you cannot return a value. D, F, G, and I are incorrect because the `return` statement is optional when `void` is used.
-

3. Given the `SampleClass`, what is the output of this code segment?

```
SampleClass s = new SampleClass();
s.sampleMethod(4.4, 4);

public class SampleClass {
    public void sampleMethod(int a, double b){
        System.out.println("Method 1");
    }
    public void sampleMethod(double b, int a){
        System.out.println("Method 2");
    }
}
```

- A. Method 1
- B. Method 2
- C. Method 1
Method 2
- D. Method 2
Method 1
- E. Compiler error

Answer:

- B. This is an example of a class with an overloaded method. Since it passes double and int primitives as parameters, it will call the second method.
 - A, C, D, and E are incorrect since this is valid code and only the first method is called.
-

Pass Objects by Reference and Value

4. Given the class `FloatNumber` and method `addHalf`, what is the output if the following code segment is executed?

```
public class FloatNumber {
    float number;
    public FloatNumber(float number) {
        this.number = number;
    }
    float getNumber() {
        return number;
    }
    void setNumber(float number) {
        this.number = number;
    }
}

void addHalf(FloatNumber value) {
    value.setNumber(value.getNumber() + (value.getNumber()/2f));
}

/* CODE SEGMENT */
FloatNumber value = new FloatNumber(1f);
addHalf(value);
System.out.println("value = " + value.getNumber());
```

- A. value = 1
 - B. value = 1.5
 - C. value = 2
 - D. value = 0
-

Answer:

- B. The `FloatNumber` object is passed by reference. Therefore, when the method changes its value, this change is still present when the code returns to the original calling code segment.
 - A, C, and D are incorrect. A is incorrect because the `FloatNumber` is passed by reference. If it were a primitive `float` this would be the correct answer. C and D are incorrect regardless how if the variable is passed by reference or value.
-

5. Objects are passed by _____.

- A. Value
 - B. Sum
 - C. Reference
 - D. Pointer
-

Answer:

- C. Objects are always passed by reference.
 - A, B, and D are incorrect. A is incorrect because primitives are passed only by value. B is incorrect because *Sum* is not a real term. D is incorrect because Java does not use the term *pointer*.
-

6. Primitives are passed by _____.

- A. Value
 - B. Sum
 - C. Reference
 - D. Pointer
-

Answer:

- A. Primitives are always passed by value.
 - B, C, and D are incorrect. B is incorrect because *Sum* is not a real term. C is incorrect because objects are passed only by reference. D is incorrect because Java does not use the term *pointer*.
-

Understand Variable Scope

7. You need to create a class to store information about books contained in a library. What variable scope is best suited for the variable that will store the title of a book?
- A. Local variable
 - B. Static variable
 - C. Global variable
 - D. Method parameter
 - E. Instance variable
-

Answer:

- E.** In a class that stores information about books, you would want to store the title of the book in a variable that will remain in scope for the life of the object.
- A, B, C, and D** are incorrect. **A** is incorrect because a local variable is best suited for data that will only be used for a short amount of time. **B** is incorrect because a static or class variable is best suited for data that all instances of the class would need to access. **C** is incorrect since global variables are discouraged in Java and would not make much sense in this situation. **D** is incorrect since it can only be used with methods.
-

- 8.** Given the `SampleClass`, when the following code segment is executed, what is the value of the instance variable `size`?

```
SampleClass sampleClass = new SampleClass(5);  
  
public class SampleClass {  
    private int size;  
    public SampleClass(int size) {  
        size = size;  
    }  
}
```

- A.** 0
B. 1
C. 5
D. Compiler error
E. Runtime error

Answer:

- A.** The instance variable is hidden with the parameter since they both have the same name. To set `size` to 5, `this.size = size;` would have to be used. The instance variable is 0 because that is the default value assigned to instance variables.
- B, C, D, and E** are incorrect. **B** is incorrect because the instant variable is set to 0, and not 1, by default. **D** and **E** are incorrect because this code is valid.
-

Create and Use Constructors

- 9.** Given the `SampleClass`, what is the output of this code segment?

```
SampleClass sampleClass = new SampleClass();
public class SampleClass {
    private int size;
    private int priority;

    public SampleClass() {
        super();
        System.out.println("Using default values");
    }

    public SampleClass(int size) {
        this.size = size;
        System.out.println("Setting size");
    }

    public SampleClass(int priority) {
        this.priority = priority;
        System.out.println("Setting priority");
    }
}
```

- A. Using default values
 - B. Setting size
 - C. Setting priority
 - D. Compiler error
-

Answer:

- D. This would generate a compiler error because you cannot overload a constructor or method and have the same data types for the parameters.
 - A, B, and C are incorrect. A is incorrect, however, if this were valid code it would be the correct answer. B and C are incorrect because even if these were valid code they do not represent the flow of execution of the code segment.
-

10. What constructor is equivalent to the one listed here?

```

public SampleConstructor() {
    System.out.println("SampleConstructor");
}

A. public SampleConstructor() {
    this();
    System.out.println("SampleConstructor");
}

B. public SampleConstructor() {
    super();
    System.out.println("SampleConstructor");
}

C. public SampleConstructor() {
    this.SampleConstructor();
    System.out.println("SampleConstructor");
}

D. public SampleConstructor() {
    super.SampleConstructor();
    System.out.println("SampleConstructor");
}

E. None of the above

```

Answer:

- B.** If super is not called on the first line of a constructor, the compiler will automatically add it for you.
- A, C, D, and E** are incorrect. **A** is incorrect because this() calls a constructor of the current class. **C** and **D** are not valid uses of super or this.
-

Use this and super Keywords

11. Given the SampleClass, what is the output of this code segment?

```

SampleClass sampleClass = new SampleClass();
public class SampleClass {
    private int size;
public SampleClass() {
    this(1);
    System.out.println("Using default values");
}

public SampleClass(int size) {
    this.size = size;
    System.out.println("Setting size");
}

```

- A.** Using default values
- B.** Setting size
- C.** Using default values
Setting size

- D. Setting size
Using default values
 - E. Compiler error
-

Answer:

- D. The first constructor is called. It uses this(1); to call the second constructor. The second constructor prints out its statement and then returns to the first constructor, where it prints out its statement.
 - A, B, C, and E are incorrect. A, B, and C are incorrect because they do not represent the correct execution of the code segment. E is incorrect because this is valid code.
-

12. What is the effect of the following line of code?

super()

- A. The method that is overridden by the current method is called.
 - B. The parent class's constructor is called.
 - C. The current class's constructor is called.
 - D. The child class's constructor is called.
 - E. The current method is recursively called.
-

Answer:

- B. The super keyword in this case is used to call the parent class's constructor. This must be done from the first line of a constructor in the current class.
 - A, C, D, and E are incorrect. A is incorrect because to refer to an overridden method super and the method name must be used. C is incorrect since this() would be used to call a correct class's constructor. D is incorrect since it is impossible to refer to a child's class's methods. E is incorrect because super has nothing to do with recursion.
-

Create Static Methods and Instance Variables

13. Given the SampleClass, what is the output of this code segment?

```
SampleClass s = new SampleClass();
SampleClass.sampleMethodOne();
public class SampleClass {
    public static void sampleMethodOne() {
        sampleMethodTwo();
        System.out.println("sampleMethodOne");
    }

    public void sampleMethodTwo() {
        System.out.println("sampleMethodTwo");
    }
}
```

- A. sampleMethodOne

- B. sampleMethodTwo
 - C. sampleMethodOne
sampleMethodTwo
 - D. sampleMethodTwo
sampleMethodOne
 - E. Compiler error
-

Answer:

- E. The method sampleMethodOne is a static method. This method is not able to call other standard methods; therefore, this will generate a compiler error when it is called.
 - A, B, C, and D are incorrect. A, B, and C are incorrect since the code has a compiler error. D is incorrect because of the compiler error, however, it would be correct if sampleMethodOne was not static and this was valid code.
-

- 14.** Given the SampleClass, what is the value of currentCount for the instance of object x after the code segment had been executed?

```
SampleClass x = new SampleClass();
SampleClass y = new SampleClass();
x.increaseCount();
public class SampleClass {
    private static int currentCount=0;

    public SampleClass() {
        currentCount++;
    }

    public void increaseCount() {
        currentCount++;
    }
}
```

- A. 0
 - B. 1
 - C. 2
 - D. 3
 - E. Compiler error
 - F. Runtime error
-

Answer:

- D. The variable currentCount is a static variable; therefore, every instance of this class has access to the same variable. The code segment creates two new instances of this class. The constructor increments this variable each time. Finally the increaseCount method is called, which also increments the variable.
 - A, B, C, E, and F are incorrect. A, B, and C are incorrect since currentCount is a static variable and the same value is incremented each time an object is created and in the increaseCount method. E and F are incorrect since this is valid code.
-

15. Static methods have access to which of the following? (Choose all that apply.)

- A. Static variables
 - B. Instance variables
 - C. Standard methods
 - D. Static methods
 - E. None of the above
-

Answer:

- A and D. Static methods can access only other static methods and static variables.
 - B, C, and E are incorrect. B and C are incorrect because static methods cannot access data that is associated with an instance of a class.
-



6

Programming with Arrays

CERTIFICATION OBJECTIVES

- Work with Java arrays
 - Work with `ArrayList` objects and their methods
- ✓ Two-Minute Drill

Q&A Self Test

A cornerstone of software development is working with data structures to store and retrieve data. Arrays are one of the most fundamental data structures; in fact, they can be found in nearly every programming language, and Java is no exception. Java inherited arrays from the C language, along with many of its other syntax rules. In addition to standard arrays, Java has an `ArrayList` class that is included in its software development kit. This is a modern approach to standard arrays and follows the principles of object-oriented programming. This chapter will discuss both type of arrays and prepare you for the questions you will encounter on the OCA exam.

CERTIFICATION OBJECTIVE

Work with Java Arrays

Exam Objective 4.1 Declare, instantiate, initialize, and use a one-dimensional array

Exam Objective 4.2 Declare, instantiate, initialize, and use multi-dimensional array

Java arrays are built into the language to handle multiple pieces of data that are of the same type. They allow the developer to use one variable with one or more indexes to access multiple independent pieces of data. Arrays can be used to store both primitives and objects.

Questions regarding arrays are included throughout the OCA exam. Questions will range from where to use an array, to what type of array to use, how to initialize an array, and how to work with arrays in code. The details of arrays are an important concept for you to understand to achieve success on the OCA exam. This section will review one-dimensional and multi-dimensional arrays.

One-Dimensional Arrays

A Java array is an object that acts as a container by storing a fixed number of the same type of values. The values are accessed by using an index. A one-dimensional array uses only a single index. Following is an example of an array of int primitives:

```
int[] arrayOfInts = new int[3];
arrayOfInts[0] = 5;
arrayOfInts[1] = 10;
arrayOfInts[2] = 15;
System.out.println("First: " + arrayOfInts[0]
    + " Second: " + arrayOfInts[1]
    + " Third: " + arrayOfInts[2]);
```

[Figure 6-1](#) is a visual representation of the above code segment.

FIGURE 6-1 int[] arrayOfInts

| | | |
|----------|-----------|-----------|
| 5 [0] | 10 [1] | 15 [2] |
|----------|-----------|-----------|

This code would produce the following output:

First: 5 Second: 10 Third: 15

This is a basic example of a one-dimensional array in action. Let's take a closer look at what happens here before we dig into the details of arrays.

First, the variable `arrayOfInts` is declared. It is declared with the type `int[]`. The square brackets indicate that this is an array of int primitives. Next, it is initialized with the `new` operator. The `new` operator must be used because all arrays are considered objects. It is initialized with its type, which is `int`, with the square brackets to indicate an array and finally a number inside the brackets. The number is used to assign a size to the array. In this case, it is an array containing three `int` primitives. The next three lines set a value for each index in the array. Notice that the first index has a value of 0. All arrays are zero-based and have a first number index of 0. Finally, the values are accessed and sent to standard out.

Declaring One-Dimensional Arrays

One-dimensional arrays are declared either by using square brackets after the type or by using the square brackets after the variable name. Both are valid. When the array is being declared, a number is never placed inside the brackets. An array is declared the same way for both objects and primitives. Here's an example:

```
/* Valid declaration of object and primitive array */
String[] clockTypes;
int[] alarms;
/*
 * Valid and equivalent to the above declarations
 * This is a less common syntax and is rarely used
 */
String clockTypes[];
int alarms[];
```

Initializing One-Dimensional Arrays

Once an array is declared, it must be initialized. An array can be initialized similarly to how an object is initialized. The new operator is used followed by the type with square brackets containing the length, or size, of the array. This initialization can be done on the same line as the declaration or on its own line. The following code segment demonstrates this:

```
/*
 * Each of these lines is a valid way to initialize
 * an array with the new operator
 */
String[] clockTypes = new String[3];
String clockTypes[] = new String[3];
clockTypes = new String[4];
/* Even arrays of primitives use the new operator */
int alarms[] = new int[2];
```

Once an array is assigned a size, it cannot be changed without initializing it again. If the declared array contains primitives, then every primitive inside the array is set to 0. An array of objects has each element initially set to null.

It is also possible to initialize an array with all of its values immediately after being declared. This is done by placing the values to be stored in the array in curly brackets. The curly brackets must follow the declaration on the same line. The values in the brackets are separated with commas. This will initialize the array and assign a value to each element in the array. The following code demonstrates this:

```
int[] alarms = {730,900};
String[] clockTypes = {"Wrist Watch","Desk Clock","Wall Clock"};
Clock[] clocks = {new Clock(1100), new Clock(2250)};
```

This example shows the alarms array, containing int primitives, being initialized with two int values. This array now has a length of two and is populated with 730 at index 0 and 900 at index 1. The next line is an array of String objects. This array is initialized with the three strings: "Wrist Watch", "Desk Clock", and "Wall Clock". The array has a length of three. The final example is an

object called Clock. This object has a constructor that forms a new object and requires the time to initialize the clock, by using an argument. The objects are created inline and both are stored in the array.

When an array is initialized with curly brackets, it must do so within the same statement as the declaration. Unlike the new operator, an array cannot be initialized with curly brackets on a different line of code.

on the job *This section has demonstrated how arrays can be declared with square brackets after the variable type or the variable name. In practice, most developers place the square brackets after the variable type, because the square brackets describe the type.*

Using One-Dimensional Arrays

One-dimensional arrays are very straightforward to use. Once declared and initialized, each element from the array can be accessed by using its corresponding index. Every object or primitive has an index number associated with it. It is important for you to remember that an array with a length of three has the indexes 0, 1, and 2. The first index is always 0.

The next example prints out the string that is stored in the array at index 1:

```
String[] clockTypes = {"Wrist Watch", "Desk Clock", "Wall Clock"};
System.out.println(clockTypes[1]);
```

The output would be as follows:

```
Desk Clock
```

This array is made up of String objects. When the array is used with an index number, it behaves just as a String object variable would. For example, the equalsIgnoreCase() method from the String class can be used like so:

```
if (clockTypes[0].equalsIgnoreCase("Grand Father Clock")) {
    System.out.println("It's a grandfather clock!");
}
```

It is also possible to get the length of an array. Arrays are objects in Java. You can access the public length field to obtain the length of the array. In the next example, notice that length is not a method with an open and close parentheses; it is a public field that can be accessed to get the length of the array:

```
System.out.println("length: " + clockTypes.length);
```

The example would produce the following output:

```
length: 3
```

Java has built-in methods for copying the data from one array to another. These methods copy the data and create two independent arrays upon completion. The arraycopy() method is a static method that is part of the System class. The method signature for arraycopy() is shown next:

```
public static void arraycopy(Object src, int srcPos,  
    Object dest, int destPos, int length)
```

This method has five parameters. The `src` parameter is the source array, the array from which you intend to copy data. The `srcPos` parameter is the starting position in the array and is where copying starts. The `dest` parameter is the array into which data will be copied. The parameter `destPos` is the starting position of where data will be placed in the array. Finally, the `length` parameter is the number of elements from the array to be copied. When using this method, you must ensure that you do not go outside the bounds of either array. The destination array must also be declared as the same type and initialized.

The following code listing demonstrates part of an array being copied into another array:

```
String[] clockTypes = {"Wrist Watch", "Desk Clock", "Wall Clock"};  
String[] newClockTypes = new String[2];  
System.arraycopy(clockTypes, 1, newClockTypes, 0, 2);  
  
for(String s : clockTypes){  
    System.out.println(s);  
}  
System.out.println("-----");  
for(String s : newClockTypes){  
    System.out.println(s);  
}
```

In this example, the last two elements from `clockTypes` are copied into `newClockTypes`. Then both are printed to standard out. The two arrays are independent. If values in one are modified, it will not affect the other array. Following is the output of this code segment:

```
Wrist Watch  
Desk Clock  
Wall Clock  
-----  
Desk Clock  
Wall Clock
```

on the job *The `Arrays` class of the Java utilities package provides sorting, searching, and comparing features for arrays. Static methods of the `Arrays` class include `asList`, `binarySearch`, `copyOf`, `copyOfRange`, `equals`, `fill`, and `sort`.*

Multi-Dimensional Arrays

Multi-dimensional arrays have more than one index. A multi-dimensional array with two dimensions, or indexes, is an array of arrays. An array can have three, four, or more dimensions. The Java language specification does not place a limit on the number of dimensions that an array can have. However, the Java virtual machine (JVM) specification does place a practical limit of 256 dimensions.

Here is a complete example of a two-dimensional array. We will go over all of the rules of the arrays in the next few sections.

```

char[][] ticTacToeBoard = new char[3][3];

for(int y=0;y<3;y++) {
    for(int x=0;x<3;x++) {
        ticTacToeBoard[x][y] = '-';
    }
}

ticTacToeBoard[0][0] = 'X';
ticTacToeBoard[1][1] = 'O';
ticTacToeBoard[0][2] = 'X';

for(int y=0;y<3;y++) {
    for(int x=0;x<3;x++) {
        System.out.print(ticTacToeBoard[x][y] + " ");
    }
    System.out.print("\n");
}

```

This example uses a two-dimensional array to represent a tic-tac-toe board. The array is first declared as `char` primitives and then initialized to have the size of 3-by-3. The first loop sets each value to a '-' character. Next, the board spaces at 0,0 and 0,2 are set to 'X', and 1,1 is set to 'O'. Finally the board is printed to standard out, resulting in the following output:

```

X - -
- O -
X - -

```

Declaring Multi-Dimensional Arrays

A multi-dimensional array is declared similarly to a one-dimensional array, with additional square brackets for each dimension. Where a one-dimensional array has one square bracket following the type or variable name, a two-dimensional array has two brackets, a three-dimensional array would have three, and so on. Following are some examples of multi-dimensional arrays being declared:

```

//An example of a two-dimensional array being declared both ways
String[][] chessBoard;
String chessBoard[][];
//An example of a three-dimensional array being declared both ways
int[][][] cube;
int cube[][][];

```

Initializing Multi-Dimensional Arrays

Multi-dimensional arrays can be initialized similarly to one-dimensional arrays. They are either initialized with the `new` operator or they use curly brackets with the values to be stored in the array:

```

String[][] square = {{"1","2"}, {"3","4"}};
String[][] square = new String[2][2];
int[][][] cube = new int[3][3][2];

```

In this example, the curly bracket groups are separated by commas. The values inside the curly brackets represent an array. You can think of multi-dimensional arrays as an array of arrays. This is because the first level is an array that contains an array at each element.

The arrays that are contained do not all have to be the same length. The next code segment demonstrates an array with different sized subarrays. [Figure 6-2](#) shows a visual representation of this array.

FIGURE 6-2 int[][] oddSizeArray

| | | |
|-------------|-------------|-------------|
| 1 [0][0] | 1 [1][0] | 1 [2][0] |
| 2 [0][1] | 2 [1][1] | 2 [2][1] |
| | 3 [1][2] | 3 [2][2] |
| 4 [1][3] | | |

```
int [] [] oddSizeArray = {{1,2},{1,2,3,4},{1,2,3}};
```

When using the new operator to initialize an array, the size of each dimension does not have to be defined. Because a multi-dimensional array is just an array of arrays, it is okay to define the length of only the first dimension. You can then either assign a preexisting array to that element or use the new operator again:

```
int [] [] [] array3D = new int [2] [] [] ;  
array3D[0]= new int [5] [] ;  
array3D[1]= new int [3] [] ;  
array3D[0] [0]= new int [7] ;  
array3D[0] [1]= new int [2] ;  
array3D[1] [0]= new int [4] ;
```

In this example, `array3D` initializes only the size of the first dimension when it is declared. Then on separate lines, both elements from the first dimension are initialized with another array. And finally, some elements from the second dimension are initialized with their arrays.

Using Multi-Dimensional Arrays

Multi-dimensional arrays are used similarly to one-dimensional arrays, except that additional dimensions need to be accounted for. When an element is accessed, an index must be provided in square brackets for each dimension:

```
int[][] grid = {{1,2},{3,4}};  
System.out.println(grid[0][0] + " " + grid[1][0]);  
System.out.println(grid[0][1] + " " + grid[1][1]);
```

It is also possible to assign one of the subarrays to another one-dimensional array. The following example continues with the last example to demonstrate this concept:

```
int[] subGrid = grid[1];
```

INSIDE THE EXAM

Searching for Clues on the Exam

When taking the OCA exam, pay close attention to questions containing multi-dimensional arrays. As this section has shown, these arrays can be used in many different ways. Think through the question and try to determine whether the array is valid. If you're in doubt, skip the question. While working on other questions, notice how the exam writers use arrays in these questions.

CERTIFICATION OBJECTIVE

Work with ArrayList Objects and Their Methods

Exam Objective 4.3 Declare and use an ArrayList

The Java `ArrayList` class is an object-orientated representation of the standard array discussed previously in this chapter. The `ArrayList` class is part of the `java.util` package.

Using the ArrayList Class

The `ArrayList` class can be used to create an object that can store other objects, including enumeration types (enums). An index is used to access the objects inside the `ArrayList`. The `ArrayList` provides additional flexibility over a standard array. It can be dynamically resized and allows for objects to be inserted in the middle of the `ArrayList` while automatically moving other elements to make room. Next is an example of an `ArrayList` in action:

```
Integer integer1 = new Integer(1300);  
Integer integer2 = new Integer(2000);  
ArrayList<Integer> basicArrayList = new ArrayList<Integer>();  
basicArrayList.add(integer1);  
basicArrayList.add(integer2);
```

In this example, two `Integer` objects are created: `integer1` and `integer2`. They will be later stored in an `ArrayList`. You should remember that `ArrayList` objects cannot store primitives as basic

arrays can. If a primitive is needed, it must be placed into a primitive wrapper object first. If a primitive is added to an `ArrayList`, autoboxing will occur to place it in its wrapper class automatically. Next, the `ArrayList` object `basicArrayList` is created. Notice the `<Integer>` within chevrons—this is a Java generic type indicator. This indicates that the `ArrayList` will be storing `Integer` objects. (Generics are not a component of the exam, so they will not be discussed in detail.) The default constructor is used to create an `ArrayList`. Finally, both objects are added. The default constructor creates an `ArrayList` with ten internal elements. The size of the `ArrayList` is still based on the number of objects in it, and an exception will be thrown if the index is not valid. If the internal size is surpassed, the `ArrayList` is automatically expanded.

The following line of code demonstrates how to access the elements in the `ArrayList`. `ArrayList` indexes are zero-based.

```
System.out.println(basicArrayList.get(0)
    + " - " + basicArrayList.get(1));
```

This would result in the following output:

```
1300 - 2000
```

To get the length of an `ArrayList`, the `size()` method is used. The `size()` method returns an `int` that represents the number of elements currently being stored.

```
System.out.println("Size: " + basicArrayList.size());
```

This example would produce the following output:

```
Size: 2
```

When the default constructor is used to create an `ArrayList`, it is given an initial capacity of ten. The capacity is different from the size. As you saw in the preceding example, the `basicArrayList` object used the default constructor and therefore had a capacity of ten. However, the `size()` returns only the number of elements that it is storing. The `ArrayList` will manage the capacity automatically. When the capacity is exceeded, the `ArrayList` will expand automatically. However, this does incur some overhead. When developing software, if you have a rough idea of the needed capacity, you can set this with a constructor. This reduces the overhead of the `ArrayList` expanding multiple times. Here is the method signature of this constructor:

```
public ArrayList(int initialCapacity)
```

The capacity of the `ArrayList` can be changed later with the `ensureCapacity()` method. This method increases the size of the `ArrayList` to the argument passed to the method. Here is the `ensureCapacity()` method signature:

```
public void ensureCapacity(int minCapacity)
```

As mentioned, it is possible to add an element into an `ArrayList` at a given index. If the index is in the middle of the `ArrayList`, the current element at that index and everything after it are shifted down by one. Continuing the `basicArrayList` example, the following code will place an element in the

middle of the ArrayList:

```
Integer interger3 = new Integer(900);
basicArrayList.add(1, interger3);
System.out.println(basicArrayList.get(0)
    + " - " + basicArrayList.get(1)
    + " - " + basicArrayList.get(2));
System.out.println("Size: " + basicArrayList.size());
```

A third Integer is created and the add() method is used, which accepts both the index to place the element and the element as arguments. When this code is executed, the following would be displayed:

```
1300 - 900 - 2000
Size: 3
```

Notice that the new Integer, 900, is placed at index one. The size has also been increased by one. The method signature for this add method is shown here:

```
public void add(int index, E element)
```

Objects can be just as easily removed from the ArrayList. When an object is removed, all of the elements after the removed element shift their indexes down by one. The size will also be decreased by one. An element can be removed by referencing its index number or by passing the object to the ArrayList. Here are the two method signatures to remove an object:

```
public E remove(int index)
public boolean remove(Object o)
```

This section covers only the most important methods of the ArrayList class. The bulk of the ArrayList exam questions will focus on these methods. However, it is still a good idea for you to review the Java API for the ArrayList class to get a feeling for all of the functionality that this class provides.

on the job *The ArrayList class belongs to a group of classes called the collection classes. These classes all implement the collection interface. As the name implies, the collection classes are used to store data. These classes are some of the most used in the Java language. Every developer should be familiar with most of the classes in this group. This group of classes contains stacks, sets, lists, queues, and arrays. Although ArrayList is the only such class on the OCA exam, it is still worthwhile for you to review the Java API for more information on these classes.*

ArrayList vs. Standard Arrays

This chapter has discussed two different valid ways to store data in your software. Which one is the best to use? As with any tool, your best bet is to use the one that fits the job. In most general cases, the ArrayList is likely to be easier to work with. It allows for growing and shrinking data sets, allows elements to be added or removed at any valid index, and offers many more methods that make working with ArrayList easy.

However, if you are concerned about overhead, the basic array may be better. If the size of the data

is known, then the automatic expanding and shrinking feature of `ArrayList` would not come into play. If primitives must be used, not in a wrapper class, then the standard array must be used.

The following table shows the pros and cons of using `ArrayList`:

| Pros | Cons |
|--|--|
| Automatically resizes when capacity is exceeded. | Cannot store primitives without being in wrapper classes. |
| Objects can be placed in the middle of the array and it will readjust all other elements. | Extra overhead is required when it is resized. |
| Objects can be easily removed from anywhere in the array and all other elements will readjust. | Must nest multiple <code>ArrayLists</code> in an <code>ArrayList</code> to work similar to a standard multi-dimensional array. |

The following table shows pros and cons of using standard one-dimensional and multi-dimensional arrays:

| Pros | Cons |
|--|---|
| Can store primitives. | Cannot be resized. |
| No overhead to access data. | When adding or removing data, the array must be managed manually. |
| Easy to create multi-dimensional arrays. | |

EXERCISE 6-1

Implement an `ArrayList` and Standard Array

This exercise will help you get more familiar with both the `ArrayList` class and standard arrays.

1. Create a Java project in the IDE of your choice.
2. Find the daily high temperature for the last seven days.
3. Create a standard one-dimensional array and enter each day's temperature into the array. It should contain seven elements when you are done.
4. Create an `ArrayList` and enter the same seven temperatures into it.
5. Use both the standard array and the `ArrayList` and find the average temperature over that time.
6. Print each value to standard out.
7. Ensure that you have calculated the same value from each array type.

CERTIFICATION SUMMARY

This chapter examined how to use standard Java arrays and `ArrayLists`. For the OCA exam, you need to know how to use each array type, how they work, and the semantic details of what is and is not valid. The exam will test you on all three areas in the same question.

Standard Java arrays and their syntax were inherited into the Java language from C. They are a very primitive way of storing multiple values of the same type. They incur low overhead, but changes must be managed manually. Once a size is declared, the array cannot grow. These arrays are able to store both primitive values and objects. It is also easy to create an array with many dimensions.

The `ArrayList` is a Java class that attempts to provide the same functionality as a standard array, but in an object-orientated manner. The `ArrayList` can store only objects, but primitives can be used when placed into their wrapper class first. The `ArrayList` will automatically adjust its size if its capacity is exceeded. It is also possible to add or remove elements from any location in the `ArrayList`. When objects are added or removed, the `ArrayList` will adjust the remaining objects.

The OCA exam will focus equally on using both array types as well as the proper syntax for both. It is not enough to know how to use a standard array or `ArrayList`. You must also know the details of what is and is not valid.



TWO-MINUTE DRILL

Work with Java Arrays

- Standard Java arrays are built into the Java language.
- Standard Java arrays can store both objects and primitives.
- Standard Java arrays can have one or many dimensions.
- Standard Java arrays are objects and must be initialized after they are declared.
- Standard Java arrays can be declared with square brackets after the type or variable name.
Example: `int[] myArray` or `int my Array[]`
- Standard Java arrays can be initialized with the new operator and their size in square brackets after the type. Example: `int[] myArray = new int[4]`
- Standard Java arrays can be initialized by placing the values that will populate the array in curly brackets. Example: `int[] myArray = {1, 2, 3, 4}`
- Standard Java arrays can be multi-dimensional; each set of square brackets represents a dimension.
- A three dimensional array would be declared as `int[][][] threeDArray`.
- Multi-dimensional arrays can be declared with the new operator by placing the size of each dimension in square brackets. Example: `int[][][] threeDArray = new int[3][2][2]`
- Multi-dimensional arrays can be initialized by placing the values that will populate the array in curly brackets. Each dimension is represented by a nested set of curly brackets. Example: `int[][][] threeDArray = {{{1, 2}, {3, 4}}, {{5, 6}, {7, 8}}, {{9, 10}, {11, 12}}}`
- Multi-dimensional arrays do not have to have the same size sub arrays. Example: `int[][][] oddSizes = {{1, 2, 3, 4}, {3}, {6, 7}}`
- Multi-dimensional arrays do not have to declare the size of every dimension when initialized.
Example: `int[][][] array5 = new int[3][][]`

Work with `ArrayList` Objects and Their Methods

- The `ArrayList` class is an object-orientated representation of a standard Java array.
- The `ArrayList` class is part of the `java.util` package.
- An `ArrayList` object will automatically resize if its capacity is exceeded.
- An `ArrayList` object can have its internal capacity adjusted manually to improve efficiency.
- An `ArrayList` object can have elements added at any index in the array and will automatically

move the other elements.

- A ArrayList object can have any element in it removed and will automatically move the other elements.
- In most cases, the ArrayList is the preferred method of storing data in an array.

SELF TEST

Work with Java Arrays

- 1.** I. What lines will compile without errors? (Choose all that apply.)

- A.** Object obj = new Object();
- B.** Object[] obj = new Object();
- C.** Object obj[] = new Object();
- D.** Object[] obj = new Object[];
- E.** Object[] obj = new Object [3]();
- F.** Object[] obj = new Object [7];
- G.** Object obj[] = new Object[];
- H.** Object obj [] = new Object [3]();
- I.** Object obj[] = new Object[7];
- J.** Object [8] obj = new Object [];
- K.** Object [3] obj = new Object [3]();
- L.** Object [7] obj = new Object [7];
- M.** Object obj [] = new {new Object (), new Object ()};
- N.** Object obj [] = {new Object (), new Object ()};
- O.** Object obj [] = {new Object [1], new Object [2]};

- 2.** What is the output of the following code segment?

```
String[] numbers = {"One", "Two", "Three"};  
System.out.println(numbers[3] + " " + numbers[2] + " " +  
numbers[1]);
```

- A.** One Two Three
- B.** Three Two One
- C.** A compile time error will be generated.
- D.** A runtime exception will be thrown.

- 3.** What is the output of the following code segment?

```
String[] numbers = {"One", "Two", "Three"};  
for(String s : numbers){  
    System.out.print(s + " ");  
}
```

- A.** One Two Three
- B.** Three Two One
- C.** A compile time error will be generated.
- D.** A runtime exception will be thrown.

4. What is the output of the following code segment?

```
int[] testScores = {80, 63, 99, 87, 100};  
System.out.println("Length: " + testScores.length);
```

- A. Length: 4
- B. Length: 5
- C. Length: 100
- D. A compile time error will be generated.
- E. A runtime exception will be thrown.

5. What is the output of the following code segment?

```
Integer[] integerArray1 = {new Integer(100), new Integer(1)  
    , new Integer(30), new Integer(50)};  
Integer[] integerArray2 = new Integer[2];  
integerArray2[0]=new Integer(100);  
System.arraycopy(integerArray1, 2, integerArray2, 1, 1);  
for(Integer i : integerArray2){  
    System.out.print(i + " ");  
}
```

- A. 100 1 30 50
- B. 100 1
- C. 100 30
- D. 100 1 30
- E. 600 1
- F. 600 30
- G. 600 1 30
- H. A compile time error will be generated.
- I. A runtime exception will be thrown.

6. What line *will* produce a compiler error?

- A. double[][] numbers;
- B. double[][][] numbers;
- C. double[][] numbers = {{1,2,3},{7,8,9},{4,5,6}};
- D. double[][] numbers = {{1,2,3},{7,8},{4,5,6,9}};
- E. double[][][] numbers = new double[7][][];
- F. double[][][] numbers = new double[][][];
- G. double[][][] numbers = new double[7][3][2];

7. What is the value of the variable sum at the end of this code segment?

```

int[] [] square = new int[3][3];
for(int i=0;i<3;i++) {
    square[i][i] = 5;
}
int sum=0;
for(int i=0;i<3;i++) {
    for(int j=0;j<3;j++) {
        sum+=square[i][j];
    }
}

```

- A.** 0
B. 5
C. 10
D. 15
E. 20
F. 25
G. 30
H. 35
I. 40
J. 45
K. 50
L. It will generate a compiler error.

8. The following code segment is valid. True or false?

```

int[] [] square = new int[2][];
square[0]= new int[5];
square[1]=new int[3];

```

- A.** True
B. False

Work with ArrayList Objects and Their Methods

9. What can be stored in an ArrayList?

- A.** Primitives
B. Objects
C. Standard arrays
D. Enums

10. What is the value of the size variable at the completion of this code segment?

```

ArrayList<Object> sampleArrayList = new ArrayList<Object>();
sampleArrayList.add(new Object());
sampleArrayList.ensureCapacity(15);
sampleArrayList.add(new Object());
int size = sampleArrayList.size();

```

- A. 0
 - B. 1
 - C. 2
 - D. 10
 - E. 15
- F. A compile time error will be generated.
G. A runtime exception will be thrown.

11. For standard situations what is typically the best type of array to use?

- A. ArrayList
- B. One-dimensional array
- C. Multi-dimensional array

12. What is the output of the following code segment?

```
ArrayList<String> sampleArrayList = new ArrayList<String>();  
sampleArrayList.add("One");  
sampleArrayList.add("Two");  
sampleArrayList.add(1, "Three");  
for(String s : sampleArrayList){  
    System.out.print(s + " ");  
}
```

- A. One Two Three
 - B. One Three Two
 - C. Three One Two
 - D. One Three
 - E. Three Two
- F. A compile time error will be generated.
G. A runtime exception will be thrown.

13. The ArrayList class is part of what Java package?

- A. java.lang
- B. java.util
- C. javax.tools
- D. javax.swing

14. What array type has the most overhead?

- A. One-dimensional array
- B. Multi-dimensional array
- C. ArrayList

15. What best describes the result of the following code segment? The ArrayList sampleArrayList has already been declared and initialized.

```
int i = 63;  
sampleArrayList.add(i);
```

- A. The int is successfully placed into the ArrayList.
- B. The int is converted to an Integer via autoboxing and then placed into the ArrayList.

- C. null is placed into the ArrayList.
- D. A compile time error will be generated.
- E. A runtime exception will be thrown.

SELF TEST ANSWERS

Work with Java Arrays

1. What lines will compile without errors? (Choose all that apply.)

- A. Object obj = new Object();
- B. Object [] obj = new Object();
- C. Object obj [] = new Object();
- D. Object [] obj = new Object [];
- E. Object [] obj = new Object [3]();
- F. Object [] obj = new Object [7];
- G. Object obj [] = new Object[] ;
- H. Object obj [] = new Object [3]();
- I. Object obj [] = new Object [7];
- J. Object [8] obj = new Object [];
- K. Object [3] obj = new Object [3]();
- L. Object [7] obj = new Object [7];
- M. Object obj [] = new {new Object(), new Object()};
- N. Object obj [] = {new Object(), new Object()};
- O. Object obj [] = {new Object [1], new Object [2]};

Answer:

- A, F, I, N, and O. Are all correct ways to declare and initialize an array.
 - B, C, D, E, G, H, J, K, L, and M are incorrect. B and C are incorrect because they have () instead of [] after the type. D and G are incorrect because they do not assign a size to the array. E and H are incorrect because the () after the [] is not the correct syntax. J and L are incorrect because the size is not assigned when it is declared. K is incorrect because the size is in the declaration and the extra () after the [] is not the correct syntax. M is incorrect because the first new operator is being used with the {} initialization method.
-

2. What is the output of the following code segment?

```
String[] numbers = {"One", "Two", "Three"};
System.out.println(numbers[3] + " " + numbers[2] + " " + numbers[1]);
```

- A. One Two Three
- B. Three Two One
- C. A compile time error will be generated.
- D. A runtime exception will be thrown.

Answer:

- D. Array indexes start at 0. The numbers[3] variable will throw a runtime ArrayIndexOutOfBoundsException.
 - A, B, and C are incorrect. If the indexes were each one number lower, B would be the correct answer.
-

3. What is the output of the following code segment?

```
String[] numbers = {"One", "Two", "Three"};  
for(String s : numbers){  
    System.out.print(s + " ");  
}
```

- A. One Two Three
- B. Three Two One
- C. A compile time error will be generated.
- D. A runtime exception will be thrown.

Answer:

- A. The enhanced for loop will print out each string in order.
 - B, C, and D are incorrect. B is incorrect since it is in the wrong order. C and D are incorrect because this code is valid.
-

4. What is the output of the following code segment?

```
int[] testScores = {80, 63, 99, 87, 100};  
System.out.println("Length: " + testScores.length);
```

- A. Length: 4
- B. Length: 5
- C. Length: 100
- D. A compile time error will be generated.
- E. A runtime exception will be thrown.

Answer:

- B. The length of an array equals the number of elements in the array.
 - A, C, D, and E are incorrect.
-

5. What is the output of the following code segment?

```
Integer[] integerArray1 = {new Integer(100), new Integer(1)  
    , new Integer(30), new Integer(50)};  
Integer[] integerArray2 = new Integer[2];  
integerArray2[0]=new Integer(100);  
System.arraycopy(integerArray1, 2, integerArray2, 1, 1);  
for(Integer i : integerArray2){  
    System.out.print(i + " ");  
}
```

- A. 100 1 30 50
- B. 100 1
- C. 100 30
- D. 100 1 30
- E. 600 1

- F. 600 30
G. 600 1 30
H. A compile time error will be generated.
I. A runtime exception will be thrown.
-

Answer:

- C. 100 is added to the array first, and then the 30 is copied into the array with the `arraycopy()` method.
 A, B, D, E, F, G, H, and I are incorrect. Each of these sets of numbers are not what will be outputted from this code segment.
-

6. What line *will* produce a compiler error?

- A. `double[][][] numbers;`
B. `double[][][] numbers;`
C. `double[][][] numbers = {{1,2,3},{7,8,9},{4,5,6}};`
D. `double[][][] numbers = {{1,2,3},{7,8},{4,5,6,9}};`
E. `double[][][] numbers = new double[7][][];`
F. `double[][][] numbers = new double[][][];`
G. `double[][][] numbers = new double[7][3][2];`
-

Answer:

- F. When the `new` operator is used, at least one array dimension in a multi-dimensional array must be given a size.
 A, B, C, D, E, and G are incorrect. These are all valid ways to work with arrays.
-

7. What is the value of the variable `sum` at the end of this code segment?

```
int [] [] square = new int [3] [3] ;
for(int i=0;i<3;i++) {
    square[i] [i] = 5;
}
int sum=0;
for(int i=0;i<3;i++) {
    for(int j=0;j<3;j++) {
        sum+=square[i] [j];
    }
}
```

- A. 0
B. 5
C. 10
D. 15
E. 20
F. 25
G. 30

- H. 35
- I. 40
- J. 45
- K. 50
- L. It will generate a compiler error.

Answer:

- D. All primitive array elements start with a value of 0. Then the first for loop sets square[0][0], square[1][1], and square[2][2] to 5. All the array elements are summed in the second for loop resulting in the sum variable having the value of 15 at the end.
 - A, B, C, E, F, G, H, I, J, K, and L are incorrect.
-

- 8.** The following code segment is valid. True or false?

```
int[][] square = new int[2][];  
square[0] = new int[5];  
square[1] = new int[3];
```

- A. True
- B. False

Answer:

- A. The code sample is valid, so the answer is true.
 - B is incorrect because the sample is valid.
-

Work with ArrayList Objects and Their Methods

- 9.** What can be stored in an ArrayList?

- A. Primitives
- B. Objects
- C. Standard arrays
- D. Enums

Answer:

- B, C, and D. Objects, standard arrays, and enums can be stored in an ArrayList.
 - A is incorrect. Primitives cannot be stored in an ArrayList unless they are placed in their primitive wrapper class.
-

- 10.** What is the value of the size variable at the completion of this code segment?

```
ArrayList<Object> sampleArrayList = new ArrayList<Object>();  
sampleArrayList.add(new Object());  
sampleArrayList.ensureCapacity(15);  
sampleArrayList.add(new Object());  
int size = sampleArrayList.size();
```

- A. 0

- B.** 1
 - C.** 2
 - D.** 10
 - E.** 15
 - F.** A compile time error will be generated.
 - G.** A runtime exception will be thrown.
-

Answer:

- C.** The size of an `ArrayList` is based on the number of objects in it, not its capacity.
 - A, B, D, E, F, and G** are incorrect because the size of an `ArrayList` is based on the number of objects in it, not its capacity.
-

11. For standard situations, what is typically the best type of array to use?

- A.** `ArrayList`
 - B.** One-dimensional array
 - C.** Multi-dimensional array
-

Answer:

- A.** An `ArrayList` is a modern implementation of an array. It provides more flexibility than standard arrays and its methods and automatic managing of internal resources will help reduce programming errors.
 - B** and **C** are incorrect because for both of these standard Java arrays are less flexible than an `ArrayList`.
-

12. What is the output of the following code segment?

```
ArrayList<String> sampleArrayList = new ArrayList<String>();
sampleArrayList.add("One");
sampleArrayList.add("Two");
sampleArrayList.add(1, "Three");
for(String s : sampleArrayList){
    System.out.print(s + " ");
}
```

- A.** One Two Three
 - B.** One Three Two
 - C.** Three One Two
 - D.** One Three
 - E.** Three Two
 - F.** A compile time error will be generated.
 - G.** A runtime exception will be thrown.
-

Answer:

- B.** The `Three` string is inserted at index 1. Since arrays have indexes starting at 0, an index of 1 places `Three` in the middle of the other two elements.
 - A, C, D, E, F, and G** are incorrect.
-

13. The `ArrayList` class is part of what Java package?

- A. `java.lang`
 - B. `java.util`
 - C. `javax.tools`
 - D. `javax.swing`
-

Answer:

- B.** The `java.util` package contains the collection classes and many other common Java classes.
 - A, C, and D** are incorrect. **A** is incorrect since `java.lang` contains many of the fundamental classes that make up Java. **C** is incorrect because `javax.tools` contains Java tools that can be invoked from programs. **D** is incorrect since `javax.swing` contains classes that are used to create graphical interfaces.
-

14. What array type has the most overhead?

- A. One-dimensional array
 - B. Multi-dimensional array
 - C. `ArrayList`
-

Answer:

- C.** `ArrayList` objects have more overhead involved than standard arrays. However, their flexibility often more than makes up for this.
 - A and B** are incorrect as these are examples of traditional Java arrays. They both have less overhead than an `ArrayList` and therefore are incorrect answers for this question.
-

15. What best describes the result of the following code segment? The `ArrayList sampleArrayList` has already been declared and initialized.

```
int i = 63;  
sampleArrayList.add(i);
```

- A.** The `int` is successfully placed into the `ArrayList`.
 - B.** The `int` is converted to an `Integer` via auto boxing and then placed into the `ArrayList`.
 - C.** `null` is placed into the `ArrayList`.
 - D.** A compile time error will be generated.
 - E.** A runtime exception will be thrown.
-

Answer:

- B.** Primitives cannot be stored in an `ArrayList`. However, if they are placed into their primitive wrapper class they can be stored. Java will automatically make that conversion via its autoboxing feature if a primitive is placed into an `ArrayList`.
 - A, C, D, and E** are incorrect. **A** is incorrect since an `int` cannot be directly placed into an `ArrayList`. **C** is incorrect because `null` is not placed into the `ArrayList`. **D** and **E** are incorrect because this is valid code.
-



7

Understanding Class Inheritance

CERTIFICATION OBJECTIVES

- Implement and Use Inheritance and Class Types
 - Understand Encapsulation Principles
 - Advanced Use of Classes with Inheritance and Encapsulation
- ✓ Two-Minute Drill

Q&A Self Test

Inheritance is a core feature of object-orientated programming languages. It allows for better architected software through code reuse, encapsulation, and data protection. It is critical that you understand this fundamental concept for the OCA exam. As a professional developer, you will be required to use inheritance daily in your own code and while interacting with existing code and frameworks. This chapter will cover the important theory of inheritance and demonstrate it with practical examples.

CERTIFICATION OBJECTIVE

Implement and Use Inheritance and Class Types

Exam Objective 7.1 Implement inheritance

Exam Objective 7.6 Use abstract classes and interfaces

Inheritance is a fundamental concept of the Java language that allows specific classes to inherit the methods and instance variables of more general classes. This creates code that is maintainable and emphasizes code reuse. You will need to have a thorough understanding of these topics for the exam.

This section also examines the differences between concrete classes and abstract classes. *Concrete* classes are the basic, normal class, whereas *abstract* classes are tied to inheritance. The OCA exam will surely include a few questions for which you will need to understand what type of class is being used.

Finally, interfaces will be discussed. In short, an interface allows the developer to specify an external public interface to a class. Any class that implements or uses this interface must abide by the specifications outlined in the interface.

This section is about inheritance and the details of how inheritance works. This concept will not only be a major part of the OCA exam but is also a very important concept to understand as a developer. The following topics will be covered over the next few pages:

- Inheritance
- Overriding methods
- Abstract classes
- Interfaces
- Advanced concepts of inheritance

Inheritance

Inheritance allows a developer to create general classes that can be used as the foundation for multiple specific classes. For example, a program may be required to have classes that represent animals. The animals that must be represented are dogs, cats, and horses. All of these animal classes share some common elements. In this simple example, each animal would have a `weight`, `age`, and `color` instance variable. Each animal class would also have methods that allow it to do such things as eat, rest, and move. These methods could be called `eat()`, `rest()`, and `move(int direction)`.

This can be implemented without inheritance by creating a class for each animal type and then defining each of the methods. This implementation approach will work, but it has a few drawbacks. Since each type of animal eats, rests, and moves very similarly, there will be a lot of duplicated code between each class. Duplicated code makes a program hard to maintain. If a bug is found in one class, the developer must remember to find it in every other class that has a copy of that code. The same problem exists for adding features to the duplicated code. It becomes very easy for code that *should* perform the same to slowly start performing differently as the code goes through the development and maintenance process. Another disadvantage of this approach is that polymorphism cannot be used.

Polymorphism is a technique that allows a specific object, such as a dog object, to be referred to in code as its more general parent animal. (Polymorphism will be covered in detail in [Chapter 8](#).)

Because this approach does not use inheritance, polymorphism is not possible. The following is an example of each animal class implemented in this approach. The details of the class are represented as comments to explain what functionality would be present if implemented.

```

public class Dog1 {
    int weight;
    int age;
    String hairColor;

    public void eat(){ /* Eat food by chewing */ }

    public void rest(){ /* Rest */ }

    public void move(int direction)
        { /* Walk in the direction given as a parameter */ }

    public void bark() { /* Bark */ }
}

public class Cat1 {
    int weight;
    int age;
    String hairColor;

    public void eat(){ /* Eat food by chewing */ }

    public void rest(){ /* Rest */ }

    public void move(int direction)
        { /* Walk in the direction given as a parameter */ }

    public void meow() { /* Meow */ }
}

public class Horse1 {
    int weight;
    int age;
    String hairColor;

    public void eat(){ /* Eat food by chewing */ }

    public void rest(){ /* rest */ }

    public void move(int direction)
        { /* Walk in the direction given as a parameter */ }

    public void neigh() { /* Neigh */ }
}

```

The first implementation of these animals is to create a unique class for each one. Each of the preceding classes has no relationship to the other. It is easy to see that the classes are all very similar and there is duplicated code among them. In fact, all the methods are the same except the `bark()`, `meow()`, and `neigh()` methods. Although there is no explicit relationship defined in the code, it is easy to infer that all three classes are related.

The same example can be better implemented by using inheritance. In the next simple example, three of the four methods that need to be implemented are common to each different animal. A dog, cat, and horse all eat, rest, and move in similar fashion. This common functionality can be placed in a general `Animal` class that defines all the general methods and instance variables that make up an animal. When the developer creates more specific types of animals such as dogs, cats, or horses, he or she can use the `Animal` class as a base, or superclass. The more specific classes will inherit all of the nonprivate methods and instance variables from the base `Animal` class. A class is inherited when it is extended. It is important to remember that a class can extend only one class. It is invalid to inherit multiple classes in one class. However, a class can inherit a class that then inherits another class, and so on. The `extends` keyword is used in the class signature line. The following is an example of the same animals being implemented using inheritance:

```
public class Animal {  
    int weight;  
    int age;  
    String hairColor;  
  
    public void eat() { /* Eat food by chewing */ }  
  
    public void rest() { /* Rest */ }  
  
    public void move(int direction)  
        { /* Walk in the direction given as a parameter */ }  
}  
  
public class Dog2 extends Animal{  
    public void bark() { /* Bark */ }  
}  
  
public class Cat2 extends Animal{  
    public void meow() { /* Meow */ }  
}  
  
public class Horse2 extends Animal{  
    public void neigh() { /* Neigh */ }  
}
```

This example creates `Dog2`, `Cat2`, and `Horse2` classes that are functionally the same as the first example. Each one of these classes extends, or inherits, the `Animal` class. The `Animal` class is used as their base, or superclass. The specific classes inherit all of the methods and instance variables from the `Animal` class, and are then permitted to add specific methods and variables that the particular class may need. In this example, each class added a method to make the noise of the animal. You may add as many instance variables or methods as needed to the class or use only those provided from the superclass.

Saying that class X extends class Y is the same as saying that class X inherits class Y.

When a class extends another class, any nonprivate methods that are contained in the superclass are accessible from the subclass. Later in this chapter, in the section “Understand Encapsulation Principles,” you’ll learn more about the four Java access modifiers; for now, assume that all of the examples use public methods. They can be invoked in the same manner as the methods implemented in the subclass. The following example demonstrates how the `Dog2` class can be used:

```
Dog2 dog = new Dog2();
dog.bark();
dog.eat();
```

In this example, a `Dog2` object named `dog` is created. Then, the `bark()` and `eat()` methods are called. Notice that both methods can be called in the same manner, even though only the `bark()` method is implemented in the `Dog2` class. This is because any `Dog2` object inherits all of the nonprivate methods in the `Animal` class.

Overriding Methods

Inheriting, or extending, a class is a very good approach for factoring out common functionality between classes. Specific classes extending more general classes allow code to be reused in a project. As stated previously, this helps keep the project more maintainable and less prone to bugs as the development cycle progresses.

The problem with this approach, however, is that the subclass that inherits the methods of the superclass is sometimes slightly different. For example, if a `Fish` class extends the `Animal` class, the `move()` method would not work since it is implemented by code that walks—and a fish needs to swim. A class that extends another class may override any inherited method. This is done by defining another method called `move()` with the same arguments and return type. When the `move()` method is invoked, the one that is implemented in the `Fish` class will be used. A class may override all, none, or just some of the methods it inherits from a parent class. The following is an example of the `Fish` class extending the `Animal` class and then overriding the `move()` method:

```
public class Fish extends Animal {
    public void move(int direction)
        { /* Swim in the direction given as a parameter */ }
}
```

Notice that the `move()` method signature is the same as in the `Animal` class. However, the `move()` method in the `Fish` class is overriding the `move()` method in the `Animal` class. When a `Fish` object is created and the `move()` method is called, the code that is located in the `Fish` class will be executed. To override a method, the method signatures, which is all of the parameters and return type, must be identical.

When a subclass overrides a method, it has the option of calling the method that is being overridden. This can be achieved by using the `super` keyword. The `super` keyword works just like the `this` keyword, but instead of referring to the current class, `super` refers to the superclass. When `super` is used, it must pass the correct arguments to the parent method. The following is an example of `super` being used in the `Horse3` class. Because horses normally rest standing, the `Horse2` class from earlier can be further modified to put the horse in a standing position before it performs the `rest()` method.

```
public class Horse3 extends Animal{
    public void rest(){
        /* Stand before rest */
        super.rest();
    }

    public void neigh() { /* Neigh */ }
}
```

When a `Horse3` object has its `rest()` method called, it will execute the code inside the `rest()` method of the `Horse3` class. This is because the `rest()` method overrides the `rest()` method in the `Animal` class. The `Horse3`'s `rest()` method makes the horse stand and then uses `super` to call the `rest()` method in the `Animal` class.

Abstract Classes

So far, all the examples presented use concrete classes. A *concrete class* is a regular class that can be instantiated. Java has another class type called an *abstract class*. An abstract class is different from a concrete class because it cannot be instantiated and must be extended. An abstract class may contain abstract methods. *Abstract methods* are methods that are not implemented. They have a valid method signature but must be overridden and implemented in the concrete class that extends the abstract class. However, an abstract class can extend another abstract class without implementing its methods. The following is an example of an abstract class:

```
public abstract class MusicPlayer {
    public abstract void play();
    public abstract void stop();

    public void changeVolume(int volumeLevel)
    { /* Set volume to volumeLevel */}
}
```

This example is an abstract class for a music player. This is intended to be the base class for different music-playing devices such as MP3 players or CD players. Notice how the class is defined; the keyword `abstract` is used to indicate that this is an abstract class. This class provides some functionality with the `changeVolume()` method. It also contains two abstract methods. An abstract method can exist only in an abstract class. The `abstract` keyword is used to mark a method as such. Every abstract method must be implemented in the concrete subclass that extends it.

An abstract class can extend other abstract classes. Abstract classes are not required to implement the methods of an abstract superclass. However, all of these methods must be implemented by the first concrete subclass. The purpose of an abstract method is to define the required functionality that any subclass must have. In this case, any music player must be able to play and stop. The functionality cannot be implemented in the `MusicPlayer` class because it is different from player to player. The following example is of two classes extending the `MusicPlayer` class:

```
public class MP3Player extends MusicPlayer{
    public void play() { /* Start decoding and playing MP3 */ }

    public void stop() { /* Stop decoding and playing MP3 */ }
}

public class CDPlayer extends MusicPlayer {
    public void play() { /* Start reading and playing disc */ }

    public void stop() { /* Stop reading disc and playing disc */ }
}
```

The `MP3Player` and `CDPlayer` classes are both types of music players. By extending the `MusicPlayer` class, they are required to implement the `play()` and `stop()` methods by overriding the abstract classes in the base class.

Interfaces

Interfaces are used in the Java language to define a required set of functionalities for the classes that implement the interface. Unlike extending base classes, a class is free to implement as many interfaces as needed. An interface can be thought of as an abstract class with all abstract methods.

When a concrete class implements an interface, it is required to implement all of the methods defined in the interface. An abstract class is not required to implement the interface methods, but the concrete class that extends it is still responsible to provide the functionality that the interface defines. Interfaces are used to create a standard public interface for similar items. This enables code to be more modular.

The `interface` keyword is used to create an interface in the next example:

```
public interface Phone {
    public void dialNumber(int number);

    public boolean isCallInProgress();
}
```

This example is a very basic interface for a phone.

The next example demonstrates this interface being implemented by a cell phone class and a landline phone. The keyword `implements` is used to implement an interface.

```

public class LandlinePhone implements Phone{
    private boolean callInProgress;

    public void dialNumber(int number)
        { /* Dial number via wired network */ }

    public boolean isCallInProgress() { return callInProgress; }
}

public class CellPhone implements Phone{
    private boolean callInProgress;

    public void dialNumber(int number)
        { /* Dial number via cell network */ }

    public boolean isCallInProgress() { return callInProgress; }
}

```

When an interface is implemented, all of its methods must then be implemented in that class. It is possible to implement multiple interfaces. When more than one interface is being used, they are separated in a comma-delimited list. When multiple interfaces are used, all of the methods defined in each one must be implemented.

Any unimplemented methods will cause the compiler to generate errors. The following is an example of a class implementing two interfaces:

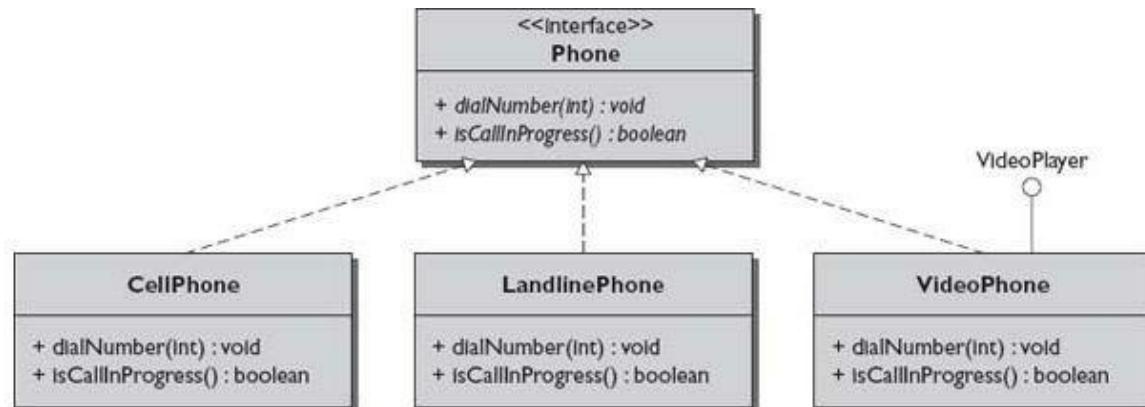
```

public class VideoPhone implements Phone, VideoPlayer{
    ...
}

```

The big advantage of using interfaces is that any class that uses the same interface will have the same public methods, as shown in [Figure 7-1](#). This means that the `CellPhone` class shown earlier could be used instead of `LandlinePhone`. Changing between these classes should not require any code change other than the type declared. This gets close to the idea of polymorphism and will be covered in detail in [Chapter 8](#).

FIGURE 7-1 Implementation of the Phone interface



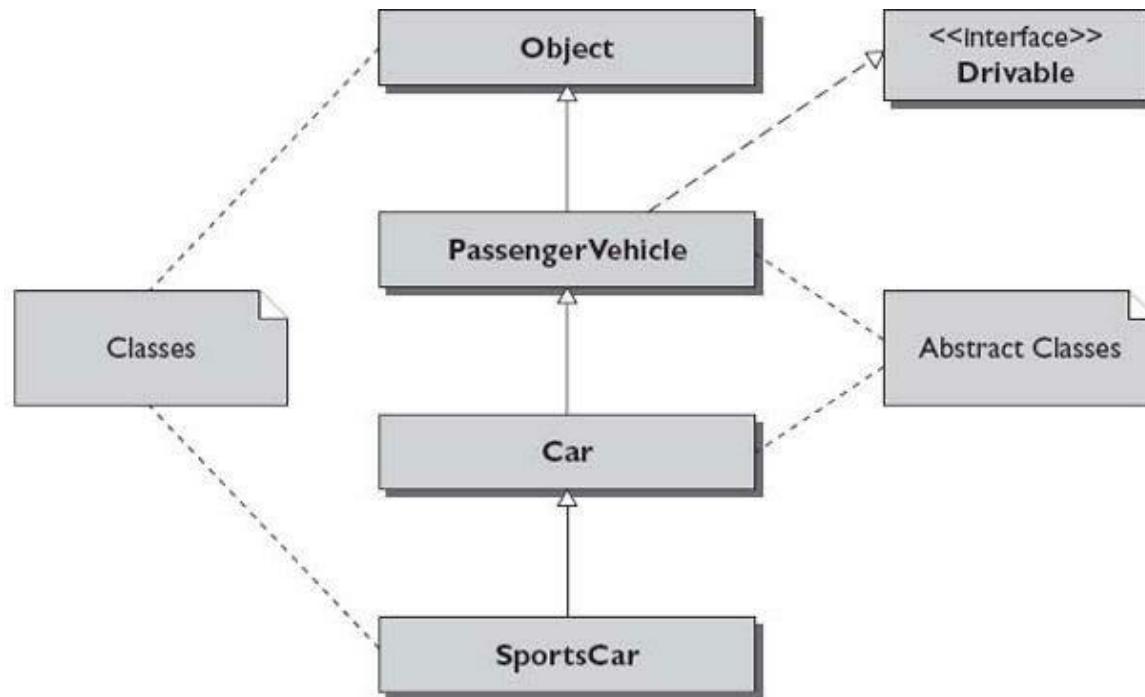
The last few sections discussed the basic cases of inheritance—one concrete class extending another concrete class, abstract class, or implementing interfaces. However, it is possible and common to have many levels of inheritance. In fact, every class in the Java language inherits from the base class `Object`. This includes the classes built by developers.

The use of `Object` as a base class is implied and does not have to be explicitly extended with the `extends` keyword. This means that in the preceding examples where a class was inherited, there were really two levels of inheritance. Looking back at the animal example, the `Dog2` class extended the `Animal1` class, and the `Animal1` class extended the `Object` class. This means that the `Dog2` class gained all of the functionality of both classes. If a class overrides the methods of another, the new method is then passed down the inheritance chain. The inheritance chain can continue as long as it is applicable to the application, meaning class A can extend class B that extends C that extends D and so forth. The classes can be a mixture of abstract and concrete. The classes are also able to implement any interfaces required.

Interfaces may also extend other interfaces. When an interface extends another interface, it gains all of the defined methods of the extended interface. Unlike a class, an interface may extend multiple interfaces. This is achieved by using the `extends` keyword, followed by a comma-delimited list of interfaces.

The diagram in [Figure 7-2](#) represents a possible inheritance tree. At the bottom of this tree is the concrete class `SportsCar`. This class extends the abstract class `Car`. The `Car` class extends the `PassengerVehicle` class, which extends the base `Object` class. The `PassengerVehicle` class also implements the `Drivable` interface.

FIGURE 7-2 An example of an inheritance tree



In an example like this, the `SportsCar` class has access to all the visible methods and instance variables in both the `PassengerVehicle` class and the `Car` class. The `SportsCar` class must also implement any methods that were abstract and unimplemented, including those required by the `Drivable` interface.

Understand Encapsulation Principles

Exam Objective 6.6 Apply access modifiers

Exam Objective 6.7 Apply encapsulation principles to a class

Encapsulation is the concept of storing data together with methods that operate on that data. Objects are used as the container for the data and code. This section discusses the principles of encapsulation and how it should be applied as a developer.

Encapsulation allows for data and method hiding. This concept is called *information hiding*. Information hiding makes it possible to expose a public interface while hiding the implementation details. Finally, this section will explore the JavaBean conventions for creating getter and setter methods. These are the methods used to read and modify properties of a Java object.

This section will expose the reader to some good basic design principles that should be used with the Java language. The OCAJ exam will require an understanding of these principles. These conventions will be used on the exam even when the question is not directly related to it. Your understanding them thoroughly will help you in understanding many exam questions.

Good Design with Encapsulation

The fundamental theory of an object-oriented language is that software is designed by creating discrete objects that interact to make up the functionality of the application. Encapsulation is the concept of storing similar data and methods together in discrete classes. In many non-object-oriented languages, there is no association between where the data is and where the code is. This can increase the complexity of maintaining the code, because often the variables that the code is using are spread apart over the code base. Bugs in the code can be hard to find and resolve due to different remote procedures using the same variables.

Encapsulation tries to solve these problems. It lets you create code that is easier to read and maintain by allowing you to group related variables and methods together in classes. Object-oriented software is very modular, and encapsulation is used for creating these modules.

A well-encapsulated class has a single clear purpose. This class should contain only the methods and variables that are needed to fulfill its purpose. For example, if a class was intended to represent a television, it should contain variables such as `currentChannel`, `volume`, and `isPoweredOn`. A `Television` class would also have methods such as `setChannel(int channel)` or `setVolume(int volume)`. These variables and methods are all related. They are specific to the properties and actions needed to create a `Television` class. The `Television` class would not contain methods such as `playDVD()`; this should be contained in a separate `DVD` class.

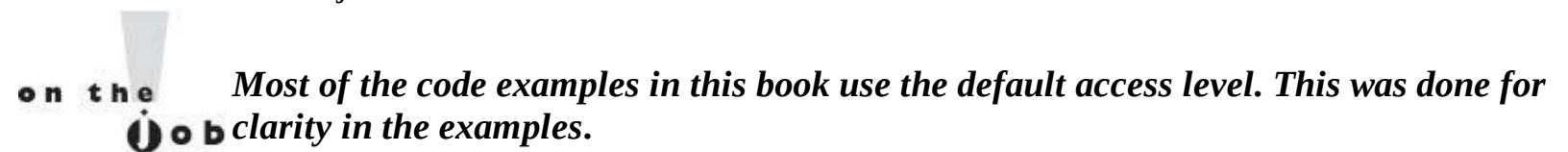
Encapsulation is about creating well-defined classes that have a clear purpose. These classes contain all the data and methods needed to perform their intended functions.

Encapsulation is defined slightly different depending on the source. Sometimes the definition indicates that encapsulation is solely about storing related data and methods together in a class. Other places will define encapsulation as also including information hiding of the implementation details. For this book, and the OCA exam, we will use the later definition.

Access Modifiers

Access modifiers are the keywords that define what can access methods and instance variables. The three access modifiers are `private`, `protected`, and `public`. These all change the default level of

access. The default access level does not use a keyword and is assigned to a method or instance variable when neither `private`, `protected`, nor `public` is used, and the area is left blank. Access modifiers are an important concept in object-oriented languages. They allow the implementation details to be hidden in a class. The developer can choose specifically what parts of a class are accessible to other objects.



Most of the code examples in this book use the default access level. This was done for clarity in the examples.

The OCA exam will focus on the different effects that each modifier has. The topics listed next will be covered in the following subsections:

- The access modifiers
- Information hiding
- Exposing object functionality

The Access Modifiers

Java uses three access modifiers: `private`, `protected`, and `public`. There is also the default access level, which is known as *package-private*. Each access level has different restrictions that allow or deny classes access to methods or instance variables. Access modifiers are also used when defining a class. This is beyond the scope of the OCA, so just assume all classes are `public`. The Java compiler will produce errors if a restricted method or instance variable is accessed by code that is unauthorized.

The `private` access modifier is the most restrictive and most commonly used access modifier. Any method or instance variable that is marked as `private` can be accessed only by other methods in the same class. Subclasses cannot access instance variables or methods that are `private`. The following is an example of the `private` keyword in use:

```
private int numberOfPoints;  
private int calculateAverage() { ... }
```

The default access level is the second most restrictive. It is often referred to as *package-private*. This access level allows access to its methods and instance variables from code that is in the same package. The default access level does not have a keyword to indicate it is in use. A method or instance variable is set to default when an access modifier is omitted. The following is an example of this access level in use:

```
int maxSpeed;  
float calculateAcceleration() { ... }
```

The `protected` access modifier is the third most restrictive. It is the same as the default access level but adds the ability of subclasses outside of the package to access its methods or instance variables. This means the methods that can access this data must either be in the same package (same as default) or be in a subclass of the class that contains the protected data. Remember that a subclass is a class that extends another class. The following is an example of the use of the `protected` access modifier:

```
protected boolean active;  
protected char getFirstChar() { ... }
```

The final access modifier is `public`. This is the least restrictive and second most common access modifier. The `public` access modifier provides no restriction to what can access its methods and instance variables. Any method can access a `public` method or instance variable regardless of which package it is contained or which superclass it extends. An item marked as `public` is accessible to the world. The following is an example of using the `public` access modifier:

```
public int streetAddress;  
public int findZipCode() { ... };
```

Information Hiding

Information hiding is the concept of hiding the implementation details of a class. Information hiding is achieved by using restrictive access modifiers. By hiding data, the developer can control how the data is accessed.

Instance variables are used to store the state of an object. If outside objects were able to access an object's entire set of instance variables, the risk of introducing bugs would be increased. A developer may create a new class that incorrectly tries to use the internal features of another class. Even if this approach works at first, it requires that the class's internal data structure not change. This concept also applies to methods. Not all methods need to be accessible by external classes. Often, a class will be composed of more methods used internally to perform tasks rather than methods designed for external objects.

| SCENARIO & SOLUTION | |
|--|--|
| You need to make an instance variable available only to the class in which it is declared. What access modifier would you use? | Use the <code>private</code> access modifier. |
| You need to make a method available only to other methods in the same package or a subclass of the class in which it is defined. What access modifier would you use? | Use the <code>protected</code> access modifier. |
| You need to make a method that is available to any other method in the application. What access modifier would you use? | Use the <code>public</code> access modifier. |
| You need to make an instance variable that is available only to other objects in the same package. What access modifier would you use? | Use the <code>package-private</code> modifier (default). |

A benefit of hiding data can be seen in this scenario: A class contains an instance variable that must be between a certain range. An outside object may set this variable and disregard the proper range. To prevent this, the variable can be marked `private` and a `public` method can be used to set it. This method would contain code that would change its value only if the new value were valid.

When working on a project, a general rule is that every method and instance variable should use the most restrictive access modifier possible. This promotes good encapsulated design by protecting data and helps reduce the areas in which bugs can

be inadvertently introduced.

Exposing Object Functionality

Once all of your internal implementation details are hidden, the class must have a set of public methods that expose its functionality to other objects. In most classes, all of the instance variables will use the private access modifier. The public methods should be the only required methods that other classes need to use this class. Any method used internally and not required by external classes should not be public.

Methods that are public can be compared to controls on the inside of a car. Only a few exist, but they allow the car to be driven. However, inside the car are many wires and mechanical controls that should not be altered and do not need to be altered to drive the car. These controls are like public methods, while the inside components are like private methods and instance variables.

Earlier in this chapter, interfaces were discussed. If a class implements an interface, it is required to implement each method in the interface as a public method. They are called interfaces because they represent the interface that other classes must use to work with this class. The public methods of any class can be thought of as an interface for the class. External objects have no knowledge of the underlying details of the class. They can see and use only the public interface that an object presents to them.

Setters and Getters

Setters and getters are the final concept of information hiding and encapsulation. As was discussed previously, it is good design to make all instance variables private. This means external classes have no way to access these variables. Sometimes an external object may need to read one of these variables to determine its state, or it may have to set it. To achieve this, a public method is created for the variable both to get and set the value. These are called *getters* and *setters*. They can be as simple as one line that only sets or returns a value.

The following example is a class that has one instance variable and a setter and getter:

```
public class ScoreBoard {  
    private int score;  
  
    public int getScore() {  
        return this.score;  
    }  
  
    public void setScore(int score) {  
        this.score = score;  
    }  
}
```

Notice in this example that a private instance variable named score. The two methods that are present are a getter and setter for the variable score. In this case, the class is giving read and write access to the variable via the methods. In some cases, a class may give only one or the other. The getter and setter in this example are simple and only set or return the value. However, if the class had to perform an action every time the score variable was changed, it could be done from the setter. For example, each time the score is changed, the class must record it to a log. This can be done in the setter. This is the benefit of keeping instance variables private. It gives control to the class as to how

its instance variables are accessed.

Getters and setters are the standard way of creating access to a class's instance variables. When developers are working with a class, they expect to find getters and setters. They also expect a JavaBeans naming convention to be followed. When creating a getter, the name should start with a lowercase get, followed by the variable name with no spaces and the first letter capitalized. The one exception to this is when a boolean value is returned. In this case, instead of using get, is is used with the same rules being applied to the variable name. When creating a setter, a similar convention should be followed. A setter should start with the word set, followed by the variable name with the first letter capitalized.

| Variable Type and Name | Getter and Setter Name |
|------------------------|---|
| int boatNumber | public int getBoatNumber() public void setBoatNumber(int boatNumber) |
| boolean boatRunning | public boolean isBoatRunning() public void setBoatRunning(boolean boatRunning) |
| Object position | public Object getPosition() public void setPosition(Object position) |

CERTIFICATION OBJECTIVE

Advanced Use of Classes with Inheritance and Encapsulation

Exam Objective 2.3 Read or write to object fields

This section will conclude the chapter by revisiting all of the concepts that have been discussed and demonstrating them with code examples. Each example will be followed by a detailed explanation of what is being highlighted and how it works. Pay close attention to the examples. They should help reinforce all of the concepts already covered.

Java Access Modifiers Example

The following example is a class implemented with its implementation details hidden. It uses public methods to expose an interface, as well as getters and setters to allow access to its instance variables.

```

public class PhoneBookEntry {
    private String name = "";
    private int phoneNumber = 0;
    private long lastUpdate = 0;

    public String getName() {
        return name;
    }

    public void setNameNumber(String name, int phoneNumber) {
        this.name = name;
        this.phoneNumber = phoneNumber;
        lastUpdate = System.currentTimeMillis();
    }

    public int getPhoneNumber() {
        return phoneNumber;
    }

    public void setPhoneNumber(int phoneNumber) {
        this.phoneNumber = phoneNumber;
        lastUpdate = System.currentTimeMillis();
    }
}

```

This example is a well-encapsulated class. It is a class that represents a basic phone book entry. It can store a name and a phone number. It also uses an instance variable to track the last time it was updated. All of the instance variables use the `private` access modifier. This means external classes are unable to read or modify them. It then uses getters and setters to modify the instance variables. In this example, there is no setter to set the `name` instance variable. To set the `name` instance variable, the object must also set the `phoneNumber` variable. This ensures that there is never a name without a phone number. If the instance variables were `public`, this class could not prevent another class from only setting a name without a number.

This example also uses its setters to update the `lastUpdate` variable. This variable is used to track the last time this class had its information updated. By using the getters and setters, the class can guarantee that any time an external object updates a field via a setter, the `lastUpdate` variable will also be updated. The details of how `lastUpdate` becomes updated are invisible to external objects.

Inheritance with Concrete Classes Examples

A concrete class is the standard Java class. All of its methods are implemented and it can be instantiated. The following example uses a `Bicycle` class. The base class represents a basic bicycle. Another class represents a ten-speed bicycle. It is called `TenSpeedBicycle` and extends the `Bicycle` class. The `TenSpeedBicycle` class is able to inherit some of its functionality while overriding the parts of the base class that need to behave differently. The `TenSpeedBicycle` class has the ability to change its gear ratio in addition to what the `Bicycle` class can do.

```
public class Bicycle {  
    private float wheelRPM;  
    private int degreeOfTurn;  
  
    public void pedalRPM(float pedalRPM) {  
        float gearRatio = 2f;  
        this.wheelRPM = pedalRPM * gearRatio;  
    }  
  
    public void setDegreeOfTurn(int degreeOfTurn) {  
  
        this.degreeOfTurn = degreeOfTurn;  
    }  
  
    public float getWheelRPM() {  
        return this.wheelRPM;  
    }  
  
    public int getDegreeOfTurn() {  
        return this.degreeOfTurn;  
    }  
}
```

The `Bicycle` class is a concrete class and, therefore, can be instantiated. It represents a basic bicycle. It has two instance variables: `wheelRPM`, which is used to store the RPM of the wheels, and `degreeOfTurn`, which is used to store the degree the handlebars are turned. Each variable has a getter, and `degreeOfTurn` has a setter. The `wheelRPM` variable is set with the method `pedalRPM(float pedalRPM)`. This accepts an argument that contains the RPM of the pedals and then multiplies that by a set gear ratio to find and set the `wheelRPM` variable.

The next example is the `TenSpeedBicycle` class. It extends the `Bicycle` class:

```
public class TenSpeedBicycle extends Bicycle {  
    private float gearRatio = 2f;  
    private float wheelRPM;  
  
    public void setGearRatio(float gearRatio) {  
        this.gearRatio = gearRatio;  
    }  
  
    public void pedalRPM(float pedalRPM) {  
        this.wheelRPM = pedalRPM * gearRatio;  
    }  
  
    public float getWheelRPM() {  
        return this.wheelRPM;  
    }  
}
```

The TenSpeedBicycle class represents a bicycle that has ten different possible gear ratios. The regular Bicycle class cannot be used, because it has a fixed gear ratio. The TenSpeedBicycle class adds a method and instance variable so a gear ratio can be set. It also overrides the wheelRPM variable. This must be done because the Bicycle class has no setter to set that variable directly. The TenSpeedBicycle class also overrides the pedalRPM(float pedalRPM) method. In the Bicycle class version of this method, the gear ratio was fixed. In the newer version, it uses the gear ratio that can be set. To retrieve the wheelRPM variable, the getter must also be overridden. This is because the original version of this method can return only the instance variable that is in its same class.

The next segment of code demonstrates both classes in use.

```
public class Main {  
    public static void main(String[] args) {  
        System.out.println("Starting...");  
        System.out.println("Creating a bicycle...");  
        Bicycle b = new Bicycle();  
        b.setDegreeOfTurn(0);  
        b.pedalRPM(50);  
        System.out.println("Turning: " + b.getDegreeOfTurn());  
        System.out.println("Wheel RPM: " + b.getWheelRPM());  
        System.out.println("Creating a 10 speed bicycle...");  
        TenSpeedBicycle tb = new TenSpeedBicycle();  
        tb.setDegreeOfTurn(10);  
        tb.setGearRatio(3f);  
        tb.pedalRPM(40);  
        System.out.println("Turning: " + tb.getDegreeOfTurn());  
        System.out.println("Wheel RPM: " + tb.getWheelRPM());  
    }  
}
```

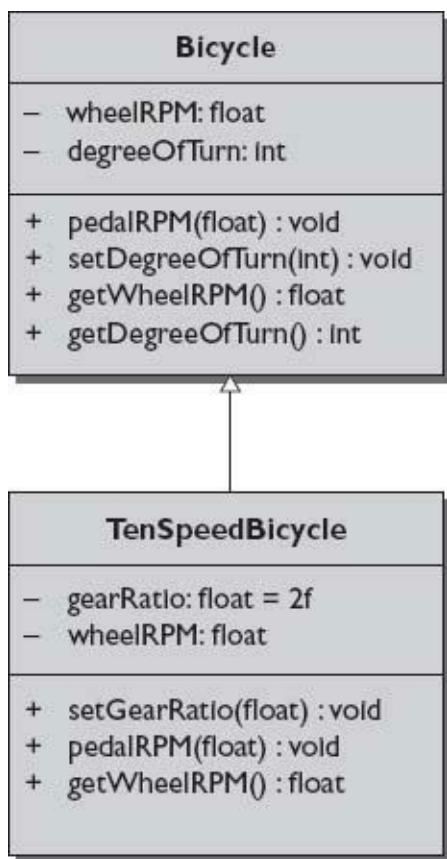
This code prints information to standard out for each step it takes. First, it creates a Bicycle object. It then sets the degree of turn to 0 and the pedal RPM to 50. The code then prints out the degree of turn, which will be 0, and the wheel RPM, which will be 100, since the gear ratio is 2 (2×50). Next, a TenSpeedBicycle object is created. This object has its degree of turn set to 10, its gear ratio set to 3, and its pedal RPM set to 40. Finally, this object prints out its degree of turn, which is 10, and its wheel RPM, which is 120 (3×40). Notice that the TenSpeedBicycle object's getDegreeOfTurn() and setDegreeOfTurn() were inherited from the base class Bicycle.

This is the output of the program after it was compiled and run:

```
Starting...  
Creating a bicycle...  
Turning: 0  
Wheel RPM: 100.0  
Creating a 10 speed bicycle...  
Turning: 10  
Wheel RPM: 120.0
```

This example shows most of the basic concepts of inheritance, as you can see in [Figure 7-3](#).

FIGURE 7-3 Basic inheritance



exam watch

As preparation for the OCA exam, you should review this code example until you fully understand how the preceding output was generated.

Inheritance with Abstract Classes Examples

This example will demonstrate an abstract class, a Java class that cannot be instantiated. Another concrete class must extend it. An abstract class may contain both concrete methods that have implementations and abstract methods that must be implemented by the subclass.

This example creates a plant simulator. It has a `Plant` abstract class that is extended by a `MapleTree` class and `Tulip` class. The `Plant` class is a good abstract class, because a plant is an abstract, or general, thing. Plants all share some characteristic that can be placed in this class. Each specific class can then contain the implementation details. The following code segment is the abstract `Plant` class:

```
public abstract class Plant {  
    private int age=0;  
    private int height=0;  
  
    public int getAge() {  
        return age;  
    }  
  
    public void addYearToAge() {  
        age++;  
    }  
  
    public int getHeight() {  
        return height;  
    }  
  
    public void setHeight(int height) {  
        this.height = height;  
    }  
  
    abstract public void doSpring();  
    abstract public void doSummer();  
    abstract public void doFall();  
    abstract public void doWinter();  
}
```

This abstract class offers a very simplistic view of what represents a plant. It contains two instance variables that every type of plant would use: `age` and `height`. There is both a getter and setter for `height` and a getter for `age`. The `age` instance variable has a method that is used to increment it each year.

The `Plant` class has four abstract methods. Each of these methods represents the actions that a plant must take during the specified season. These actions are specific to the type of plant and, therefore, cannot be generalized. Having them declared in the abstract `Plant` class guarantees that any class that extends the `Plant` class must implement them.

The next class is the `MapleTree` class:

```

public class MapleTree extends Plant {
    private static final int AMOUNT_TO_GROW_IN_ONE_GROWING_SEASON = 2;

    /*
     * A tree grows upwards a certain number of feet a year.
     * A tree does not die down to ground level during the winter.
     */
    private void grow() {
        int currentHeight = getHeight();
        setHeight(currentHeight + AMOUNT_TO_GROW_IN_ONE_GROWING_SEASON);
    }

    public void doSpring() {
        grow();
        addYearToAge();
        System.out.println("Spring: The maple tree is starting to grow " +
            "leaves and new branches");
        System.out.println("\tCurrent Age: " + getAge() + " " +
            "Current Height: " + getHeight());
    }

    public void doSummer() {
        grow();
        System.out.println("Summer: The maple tree is continuing to grow");
        System.out.println("\tCurrent Age: " + getAge() + " " +
            "Current Height: " + getHeight());
    }

    public void doFall() {
        System.out.println("Fall: The maple tree has stopped growing" +
            " and is losing its leaves");
        System.out.println("\tCurrent Age: " + getAge() + " " +
            "Current Height: " + getHeight());
    }

    public void doWinter() {
        System.out.println("Winter: The maple tree is dormant");
        System.out.println("\tCurrent Age: " + getAge() + " " +
            "Current Height: " + getHeight());
    }
}

```

The `MapleTree` class extends the `Plant` class and is used as a simple representation of a maple tree. Because the `Plant` class is abstract, the `MapleTree` class must implement all of its abstract methods. The `MapleTree` class contains one variable named `AMOUNT_TO_GROW_IN_ONE_GROWING_SEASON`. This variable is marked as `private static final int`. This is how Java declares a constant. These details are beyond the scope of the OCA exam. Just consider this a constant that is a primitive `int` and is `private`. This variable is used to set the amount of growth that a maple tree completes during a growing season.

The `MapleTree` class contains a method to grow, called `grow()`. This method is used to add the new height to the current height. The next four methods are all methods that are required to be implemented. These abstract methods are declared in the `Plant` class, with each one representing a different season. When they are invoked, they perform any required action that is needed for that season and then print two lines to standard out. The first line of text states what season it is and what the maple tree is doing. The next line displays the values of the age and height variables.

The next class is the `Tulip` class:

```

public class Tulip extends Plant {
    private static final int AMOUNT_TO_GROW_IN_ONE_GROWING_SEASON = 1;

    /*
     * A tulip grows each year to the same height. During
     * the winter they die down to ground level.
     */
    private void grow() {
        int currentHeight = getHeight();
        setHeight(currentHeight + AMOUNT_TO_GROW_IN_ONE_GROWING_SEASON);
    }

    private void dieDownForWinter() {
        setHeight(0);
    }

    public void doSpring() {
        grow();
        addYearToAge();
        System.out.println("Spring: The tulip is starting to grow " +
            "up from the ground");
        System.out.println("\tCurrent Age: " + getAge() + " " +
            "Current Height: " + getHeight());
    }

    public void doSummer() {
        System.out.println("Summer: The tulip has stopped growing " +
            "and is flowering");
        System.out.println("\tCurrent Age: " + getAge() + " " +
            "Current Height: " + getHeight());
    }

    public void doFall() {
        System.out.println("Fall: The tulip begins to wilt");
        System.out.println("\tCurrent Age: " + getAge() + " " +
            "Current Height: " + getHeight());
    }

    public void doWinter() {
        dieDownForWinter();
        System.out.println("Winter: The tulip is dormant underground");
        System.out.println("\tCurrent Age: " + getAge() + " " +
            "Current Height: " + getHeight());
    }
}

```

The `Tulip` class is intended to represent a tulip. It extends the `Plant` class and, therefore, must also implement all its abstract methods. Like the `MapleTree` class, the `Tulip` class also has a constant that is used to store the amount of growth per growing season.

The `Tulip` class has two private methods: a `grow()` method that is like the one present in the `MapleTree` class, and a `dieDownForWinter()` method that is used to reset the height to zero when the tulip loses all of its leaves during the winter.

The last four methods in the class are the abstract methods from the `Plant` class. Each season method performs the needed actions first, such as grow, die down, or age. It then prints to standard out a message about what it is doing, and what season it is. The second line of text contains the values of the `age` and `height` variables.

The final code segment is the `main()` method that uses both the `Tulip` and `MapleTree` classes:

```

public class Simulator{
    public static void main(String[] args) {
        System.out.println("Creating a maple tree and tulip...");
        MapleTree mapleTree = new MapleTree();
        Tulip tulip = new Tulip();
        System.out.println("Entering a loop to simulate 3 years");
        for (int i = 0; i < 3; i++) {
            mapleTree.doSpring();
            tulip.doSpring();
            mapleTree.doSummer();
            tulip.doSummer();
            mapleTree.doFall();
            tulip.doFall();
            mapleTree.doWinter();
            tulip.doWinter();
        }
    }
}

```

First, an object of each type is created. Then a `for` loop invokes the methods for all four seasons for each object. This loop represents a simple simulation program. Each time through the loop represents one year. Both objects age and grow from year to year.

When the preceding code is executed, it will produce the output shown next:

```

Creating a maple tree and tulip...
Entering a loop to simulate 3 years
Spring: The maple tree is starting to grow leaves and new branches
        Current Age: 1 Current Height: 2
Spring: The tulip is starting to grow up from the ground
        Current Age: 1 Current Height: 1
Summer: The maple tree is continuing to grow
        Current Age: 1 Current Height: 4
Summer: The tulip has stopped growing and is flowering
        Current Age: 1 Current Height: 1
Fall:   The maple tree has stopped growing and is losing its leaves
        Current Age: 1 Current Height: 4
Fall:   The tulip begins to wilt
        Current Age: 1 Current Height: 1
Winter: The maple tree is dormant
        Current Age: 1 Current Height: 4
Winter: The tulip is dormant underground
        Current Age: 1 Current Height: 0
Spring: The maple tree is starting to grow leaves and new branches
        Current Age: 2 Current Height: 6
Spring: The tulip is starting to grow up from the ground
        Current Age: 2 Current Height: 1
Summer: The maple tree is continuing to grow
        Current Age: 2 Current Height: 8
Summer: The tulip has stopped growing and is flowering
        Current Age: 2 Current Height: 1
Fall:   The maple tree has stopped growing and is losing its leaves
        Current Age: 2 Current Height: 8

```

```
Fall: The tulip begins to wilt
      Current Age: 2 Current Height: 1
Winter: The maple tree is dormant
      Current Age: 2 Current Height: 8
Winter: The tulip is dormant underground
      Current Age: 2 Current Height: 0
Spring: The maple tree is starting to grow leaves and new branches
      Current Age: 3 Current Height: 10
Spring: The tulip is starting to grow up from the ground
      Current Age: 3 Current Height: 1
Summer: The maple tree is continuing to grow
      Current Age: 3 Current Height: 12
Summer: The tulip has stopped growing and is flowering
      Current Age: 3 Current Height: 1
Fall: The maple tree has stopped growing and is losing its leaves
      Current Age: 3 Current Height: 12
Fall: The tulip begins to wilt
      Current Age: 3 Current Height: 1
Winter: The maple tree is dormant
      Current Age: 3 Current Height: 12
Winter: The tulip is dormant underground
      Current Age: 3 Current Height: 0
```

Notice how the maple tree continues to grow each year. The tulip, however, must regrow each year. Both the `Tulip` and the `MapleTree` objects have access to the `getAge()` and `getHeight()` methods that were implemented in the abstract `Plant` class. Review the code and the output thoroughly. Your understanding of the examples in this section will better prepare you for the OCA exam.

EXERCISE 7-1

Add Functionality to the Plant Simulator

This exercise will use the previous plant simulator and add new functionality to it.

1. Copy the plant simulator into the text editor or IDE of your choice.
 2. Compile and run the example to ensure that the code has been copied correctly.
 3. Add a new class called `Rose` that will represent a rose. Use the `Plant` base class and implement all of the required methods.
 4. Add your new class into the simulator and run the application.
-

Interface Example

This final example involves interfaces. An *interface* is a public set of methods that must be implemented by the class that uses the interface. By using an interface, a class is saying it implements the functionality defined by the interface. This example includes two interfaces. One is called `Printer` and provides a public interface that printers should implement. Any class that implements `Printer` can be said to have the ability to print. The other interface in this example is `Fax`. It provides the public interface for a faxing capability. Finally, this example has a class that implements both interfaces. This class represents an all-in-one printer/fax machine. The class is called `PrinterFaxCombo`.

This interface is for a printer. It provides a basic public interface that all printers should have. In this simple example, the printer can do two things: It can print a file with the `printFile(File f)` method, or it can check the ink levels with the `getInkLevel()` method.

```
public interface Printer {  
    public void printFile(File f);  
    public int getInkLevel();  
}
```

The next interface is for a fax machine. This simple fax machine can send a file with the `sendFax(File f, int number)` method or return a fax as an `Object` with the `getReceivedFaxes()` method.

```
public interface Fax {  
    public void sendFax(File f, int number);  
    public Object getReceivedFaxes();  
}
```

The following is the `PrinterFaxCombo` class. This class implements both interfaces.

```
public class PrinterFaxCombo implements Fax, Printer{  
    private Object incomingFax;  
    private int inkLevel;  
  
    public void sendFax(File f, int number) {  
        dialNumber(number);  
        faxFile(f);  
    }  
  
    public Object getReceivedFaxes() {
```

```

        return incomingFax;
    }

    public void printFile(File f) {
        sendFileToPrinter(f);
    }

    public int getInkLevel() {
        return inkLevel;
    }

    private boolean dialNumber(int number) {
        boolean success = true;
        /* Dial number set success to false if it is not successful */
        return success;
    }

    private void faxFile(File f){
        /* Send the File f as a fax */
    }

    private void sendFileToPrinter(File f){
        /* Print the File f */
    }

    /*
     * This class would contain many more methods to
     * implement all of this functionality.
     */
}

```

The `PrinterFaxCombo` class is a simplistic version of a printer that can also fax. The class is not fully implemented, but the comments in the empty methods should explain the purpose of each method. The important point of this example is that this class implements both the `Printer` and `Fax` interfaces. By implementing the interfaces, the `PrinterFaxCombo` class is obligated to implement each method they contain. Implementing interfaces allows an external object to know that this class provides the functionality of a printer and fax machine. Every class that implements the `Printer` interface provides printing functionality and has the same public interface. This creates modular code and allows easy swapping in and out of different classes based on the needs of the application. Interfaces also allow for polymorphism. This will be discussed in detail in [Chapter 8](#).

CERTIFICATION SUMMARY

This chapter has been about class inheritance and encapsulation. *Inheritance* is an important concept in Java. It is the term used to describe one class gaining the methods and instance variables of a parent class. This concept allows a developer to find commonality between classes and create a general parent class that each specific class can extend, or inherit, to then gain common functionality. This promotes code reuse.

Concrete classes and abstract classes are both able to be extended to create subclasses. The class that is extended is then considered the superclass, or base class. A class may extend only one class. Concrete classes are the standard class type with each method implemented. A class that extends a concrete class gains all of its visible methods. Abstract classes must be extended and cannot be instantiated in code. They contain a mixture of implemented and abstract, or unimplemented,

methods. When an abstract class is extended, all of its abstract methods must be implemented by the concrete subclass.

Interfaces are a set of unimplemented methods. When a class implements an interface, that class must then implement each method that is in the interface. Interfaces are used to define a predetermined set of exposed methods. Classes may implement as many interfaces as they need as long as all methods are then implemented in that class.

Next, Java access modifiers were discussed. The public, private, protected, and default access modifiers are used to prefix a method or instance variable. The public access modifier allows any code to access the method or instance variable that it prefixes. The private access modifier allows code only within its own class to access the method or instance variable. Protected restricts access only to classes in the same package. The default, or package-private, modifier allows any class in the same package, or in a subclass, to have access.

Another major concept covered in this chapter was encapsulation. Encapsulation is the design concept of allowing access to a class only through a public interface while hiding the rest of the implementation details. A public interface is created by the methods that have the public access modifier. Implementation details should be hidden by using the private, protected, or default access modifier. Getters and setters are normally used to access the hidden data. A getter is a simple method that returns an instance variable, and a setter is a method that sets an instance variable to the value passed to it as an argument.

This chapter concluded with code examples. These examples are important for you to understand since the OCAJ exam will have questions based on a given code segment.

TWO-MINUTE DRILL

Implement and Use Inheritance and Class Type

- Inheritance is used to place common code in a base class.
- Inheritance makes code more modular and easier to maintain.
- The extends keyword is used to extend or inherit a class.
- When a class inherits another class, it gains access to all of its public and protected methods and instance variables. If the two classes are in the same package, it also gains access to any methods or instance variables that have the default, or package-private, access modifier.
- The class that is being inherited is called the base class or superclass.
- The class that gains the functionality is called the subclass.
- A method in a superclass can be overridden by the subclass having a method with an identical signature.
- The super keyword can be used to access the overridden method.
- A class can extend only one other class.
- A concrete class is a class that can be instantiated; all of its methods have been implemented.
- An abstract class cannot be instantiated. It must be extended and may or may not contain abstract methods.
- When a concrete class extends an abstract class, all of the abstract methods must be implemented.
- An interface is used to define a public interface that a class must have.
- The keyword implements is used to implement an interface.
- A class may implement multiple interfaces by using a comma-delimited list.
- A class that implements an interface must implement all of the methods contained in the interface.

Understand Encapsulation Principles

- Encapsulation is the concept of storing related data and code together.
- Access modifiers can be used to restrict access to methods and instance variables.
- The public access modifier allows any class to access the public method or instance variable.
- The protected access modifier allows classes that are in the same package or are a subclass to have access to the method or instance variable.
- The default or package-private access modifier allows classes that are in the same package to access the method or instance variable.
- The private access modifier allows only methods in the same class to access the private method or instance variable.
- Information hiding is the concept of using restrictive access modifiers to hide the implementation details of a class.
- Both getters and setters should follow the JavaBeans naming convention. They should start with get, set, or is, followed by the variable name, starting with a capital letter.

Advanced Use of Classes with Inheritance and Encapsulation

- When creating methods or instance variables, the most restrictive access modifier possible should be used.
- A getter is used to access private instance variables.
- A setter is used to set private instance variables.

SELF TEST

Implement and Use Inheritance and Class Type

- 1.** What contains methods and instance variables and can be instantiated?
 - A. Concrete class
 - B. Abstract class
 - C. Java class
 - D. Interface
- 2.** What is used to define a public interface?
 - A. Concrete class
 - B. Abstract class
 - C. Java class
 - D. Interface
- 3.** What can contain unimplemented methods and instance variables and cannot be instantiated?
 - A. Concrete class
 - B. Abstract class
 - C. Java class
- 4.** Inheritance provides which of the following? (Choose all that apply.)
 - A. Allows faster execution times since methods can inherit processor time from superclasses
 - B. Allows developers to place general code in a class that more specific classes can gain through inheritance
 - C. Promotes code reuse
 - D. Is an automated process to transfer old code to the latest Java version

5. A class being inherited is referred to by what name? (Choose all that apply.)

- A. Subclass
- B. Superclass
- C. Base class
- D. Super duper class

Understand Encapsulation Principles

Refer to this class for the following two questions.

```
public class Account {  
    private int money;  
  
    public int getMoney() {  
        return this.money;  
    }  
  
    public void setMoney(int money) {  
        this.money = money;  
    }  
}
```

6. In the code segment, what is the method `getMoney()` considered?

- A. Get method
- B. Access method
- C. Getter method
- D. Instance variable method

7. In the code segment, what is the method `setMoney(int money)` considered?

- A. Set method
- B. Access method
- C. Setter method
- D. Instance variable method

8. Which of the following defines information hiding?

- A. Information hiding is hiding as much detail about your class as possible so others can't steal it.
- B. Information hiding is about hiding implementation details and protecting variables from being used the wrong way.
- C. Information hiding is used to obscure the interworking of your class so external classes must use the public interface.

9. What access modifier is used to make the instance variable or method available only to the class in which it is defined?

- A. `public`
- B. `private`
- C. `protected`
- D. `package-private` (default)

10. What access modifier is used for methods that were defined in an interface?

- A. `public`

- B. *private*
- C. *protected*
- D. *package-private* (default)

Advanced Use of Classes with Inheritance and Encapsulation

11. What is the proper signature for class X if it inherits class Z?

- A. `public class X inherits Z{ ... }`
- B. `public class X extends Z{ ... }`
- C. `public class X implements Z{ ... }`

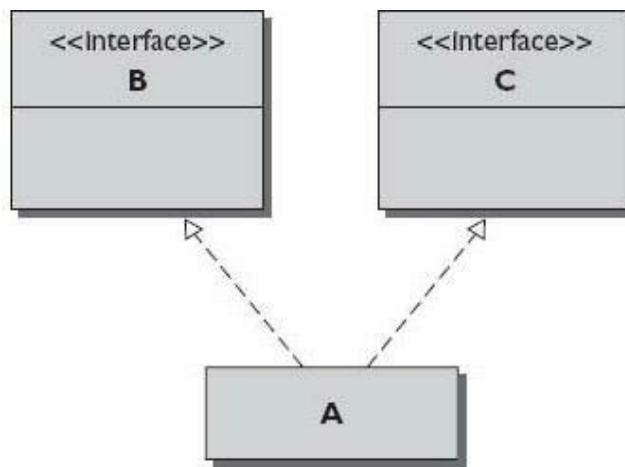
12. How many classes can a class extend directly?

- A. Zero
- B. One
- C. Two
- D. As many as it needs

13. How many interfaces can a class implement directly?

- A. Zero
- B. One
- C. Two
- D. As many as it needs

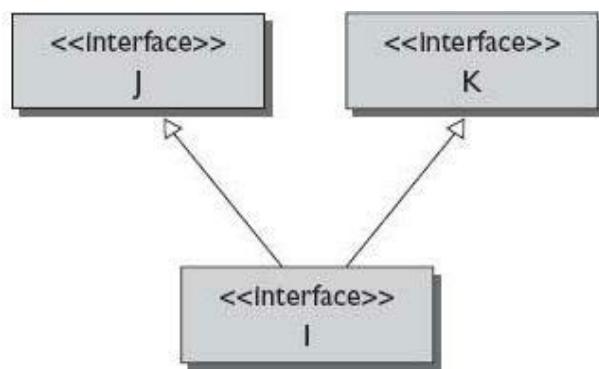
14. Consider the following UML illustration for assistance with this question:



What is the proper signature for class A if it implements interfaces B and C?

- A. `public class A implements B, implements C{ ... }`
- B. `public class A implements B, C{ ... }`
- C. `public class A interface B, interface C{ ... }`
- D. `public class A interface B, C{ ... }`
- E. `public class A extends B, C{ ... }`

15. Consider the following UML illustration for assistance with this question:



What is the proper signature for interface I to inherit interfaces J and K?

- A. public interface I extends J, K{ ... }
- B. public interface I implements J, K{ ... }
- C. public interface I implements J, implements K{ ... }
- D. public interface I interface J, K{ ... }

SELF TEST ANSWERS

Implement and Use Inheritance and Class Type

1. What contains methods and instance variables and can be instantiated?

- A. Concrete class
- B. Abstract class
- C. Java class
- D. Interface

Answer:

- A. A concrete class is the standard Java class that is used to create objects.
 - B, C, and D are incorrect. **B** is incorrect because an abstract class cannot be instantiated. **C** is incorrect because a Java class is a made-up term. **D** is incorrect because an interface does not contain methods and cannot be instantiated.
-

2. What is used to define a public interface?

- A. Concrete class
- B. Abstract class
- C. Java class
- D. Interface

Answer:

- D. An interface is used to define a public list of methods that must be implemented by the class. This represents a public interface.
 - A, B, and C are incorrect. **A** is incorrect because a concrete class is used to build objects. **B** is incorrect because abstract classes are used to define abstract methods for other classes to override. **C** is incorrect because a Java class is a made-up term.
-

3. What can contain unimplemented methods and instance variables and cannot be instantiated?

- A. Concrete class
- B. Abstract class
- C. Java class

Answer:

- B. An abstract class must always be extended; it cannot be instantiated to create an object. It can contain implemented and unimplemented methods.
 - A and C are incorrect. **A** is incorrect because a concrete class is not able to have any unimplemented methods. **C** is incorrect because a Java class is a made-up term.
-

4. Inheritance provides which of the following? (Choose all that apply.)

- A. Allows faster execution times since methods can inherit processor time from superclasses
- B. Allows developers to place general code in a class that more specific classes can gain through inheritance
- C. Promotes code reuse
- D. Is an automated process to transfer old code to the latest Java version

Answer:

- B** and **C**. Both statements are true about inheritance.
 A and **D** are incorrect. **A** is incorrect because inheritance has no effect on processor scheduling. **D** is incorrect because inheritance has no relationship to the Java version.
-

5. A class being inherited is referred to by what name? (Choose all that apply.)

- A. Subclass
B. Superclass
C. Base class
D. Super duper class

Answer:

- B** and **C**. The class that is inherited is the superclass or base class in reference to the class that extends it.
 A and **D** are incorrect. **A** is incorrect because the subclass is the class that inherits from another. **D** is incorrect because this is a made-up term.
-

Understand Encapsulation Principles

Refer to this class for the following two questions.

```
public class Account {  
    private int money;  
  
    public int getMoney() {  
        return this.money;  
    }  
  
    public void setMoney(int money) {  
        this.money = money;  
    }  
}
```

6. In the code segment, what is the method `getMoney()` considered?

- A. Get method
B. Access method
C. Getter method
D. Instance variable method

Answer:

- C**. This is a getter. Getters are used to retrieve a `private` instance variable. The name of a getter method is always `get` followed by the variable name with a capital letter. If the variable is a `boolean`, the `get` is replaced with `is`.
 A, B, and D are all incorrect and are not typical Java terms.
-

7. In the code segment, what is the method `setMoney(int money)` considered?

- A. Set method

- B. Access method
 - C. Setter method
 - D. Instance variable method
-

Answer:

- C. This is a setter. Setters are used to set a `private` instance variable. The name of a setter method is always `set`, followed by the variable name with a capital letter. They take one argument and use this to set the variable.
 - A, B, and D are all incorrect and are not typical Java terms.
-

8. Which of the following defines information hiding?

- A. Information hiding is hiding as much detail about your class as possible so others can't steal it.
 - B. Information hiding is about hiding implementation details and protecting variables from being used the wrong way.
 - C. Information hiding is used to obscure the interworking of your class so external classes must use the public interface.
-

Answer:

- B. Good class design hides as many methods and instance variables as possible. This is done by using the `private` access modifier. This is so external objects do not try to interact with the object in ways the developer has not intended. Hiding information makes code easier to maintain and more modular.
 - A and C are incorrect. A is incorrect because information hiding has nothing to do with protecting your code from others. C is incorrect because access modifiers should be used to force external classes to use the proper public interface.
-

9. What access modifier is used to make the instance variable or method available only to the class in which it is defined?

- A. `public`
 - B. `private`
 - C. `protected`
 - D. `package-private` (default)
-

Answer:

- B. The `private` access modifier is used to allow only the methods in the class to access the method or instance variable.
 - A, C, and D are incorrect. A is incorrect because `public` would make the instance variable available to every class. C is incorrect because `protected` would make the instance variable available to any subclass or class in the same package. D is incorrect because `package-private`, or the default access level, would make the instance variable available to any other class in the same package.
-

10. What access modifier is used for methods that were defined in an interface?

- A. `public`
- B. `private`

- C. protected
 - D. *package-private* (default)
-

Answer:

- A. The public access modifier must be used when implementing methods from an interface.
 - B, C, and D are incorrect. All three of these are incorrect because they limit the accessibility of the method and therefore would not be appropriate for an interface.
-

Advanced Use of Classes with Inheritance and Encapsulation

11. What is the proper signature for class X if it inherits class Z?

- A. `public class X inherits Z{ ... }`
 - B. `public class X extends Z{ ... }`
 - C. `public class X implements Z{ ... }`
-

Answer:

- B. The `extends` keyword is used to inherit a class.
 - A and C are incorrect. A is incorrect because `inherits` is not a valid Java keyword. C is incorrect because the `implements` keyword is used for interfaces, not classes.
-

12. How many classes can a class extend directly?

- A. Zero
 - B. One
 - C. Two
 - D. As many as it needs
-

Answer:

- B. A class can extend only one other class. However, it is possible to have one class extend a class that extends another class, and so on.
 - A, C, and D are incorrect.
-

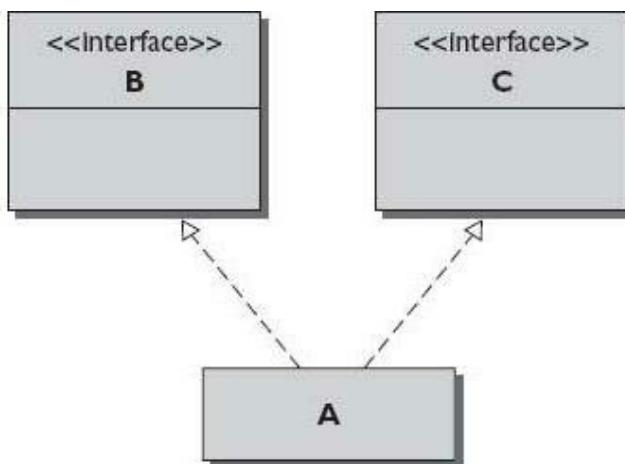
13. How many interfaces can a class implement directly?

- A. Zero
 - B. One
 - C. Two
 - D. As many as it needs
-

Answer:

- D. A class can implement as many interfaces as it needs.
 - A, B, and C are incorrect.
-

14. Consider the following UML illustration for assistance with this question:



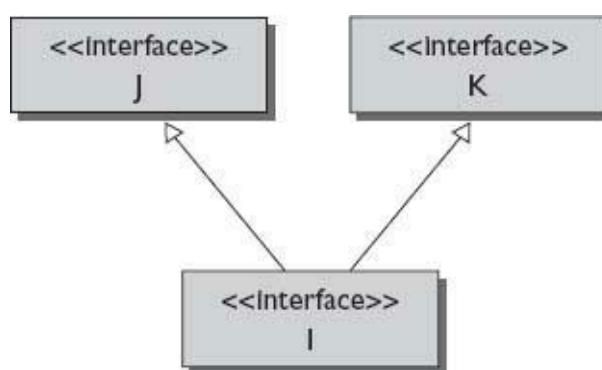
What is the proper signature for class A if it implements interfaces B and C?

- A. `public class A implements B, implements C{ ... }`
- B. `public class A implements B, C{ ... }`
- C. `public class A interface B, interface C{ ... }`
- D. `public class A interface B, C{ ... }`
- E. `public class A extends B, C{ ... }`

Answer:

- B.** A class uses the keyword `implements` to implement an interface. To implement multiple interfaces, the `implements` keyword should appear in a comma-delimited list after the keyword `implements`.
- A, C, D, and E** are incorrect. **A** is incorrect because the `implements` keyword should not be included more than once. **C** is incorrect because the `implements` keyword should be used instead of `interface`, and it should be used only once. **D** is incorrect because `implements` should be used instead of `interface`. **E** is incorrect because `extends` is used for classes, not interfaces; `implements` should be used instead.

15. Consider the following UML illustration for assistance with this question:



What is the proper signature for interface I to inherit interfaces J and K?

- A. `public interface I extends J, K{ ... }`
- B. `public interface I implements J, K{ ... }`
- C. `public interface I implements J, implements K{ ... }`
- D. `public interface I interface J, K{ ... }`

Answer:

- A.** An interface can also inherit other interfaces. Unlike classes, interfaces can inherit or extend as many other interfaces as needed. An interface uses the keyword `extends`, followed by a

comma-delimited list of all the other interfaces it wants to extend.

B, **C**, and **D** are incorrect. **B** is incorrect because only classes implement interfaces. An interface extends other interfaces. **C** is incorrect because extends should be used, and only once. **D** is incorrect because the `interface` keyword is not used correctly.



8

Understanding Polymorphism and Casts

CERTIFICATION OBJECTIVES

- Understand Polymorphism
- Understand Casting

✓ Two-Minute Drill

Q&A Self Test

The OCA exam will expect you to have a firm understanding of what polymorphism is and when it should be used. The exam will present scenarios using polymorphism in correct and incorrect ways.

Casting will also be covered on the OCA exam. On the surface, casting looks similar to polymorphism; however, it is very different. The OCA exam is likely to ask questions that require that you have a solid understand of when and how casting can be used.

At the conclusion of this chapter you should understand how to manipulate the type of an object by either polymorphism or casting. You should also know when each is needed or not needed. These are core concepts that will be asked directly on the OCA exam.

CERTIFICATION OBJECTIVE

Understand Polymorphism

Exam Objective 7.2 Develop code that demonstrates the use of polymorphism

Exam Objective 7.3 Differentiate between the type of a reference and the type of an object

Polymorphism is a fundamental aspect of object-oriented programming languages, including Java. Polymorphism allows the developer to write generic code that is flexible and that allows for easier code reuse, another fundamental object-oriented principle.

Concepts of Polymorphism

The word *polymorphism* comes from the Greeks and roughly means “many forms.” In Java, polymorphism means that one object can take the form or place of an object of a different type. Polymorphism can exist when one class inherits another. It can also exist when a class implements an interface. This section will describe how polymorphism can apply in both cases. Finally, this section will demonstrate what polymorphism looks like in Java code.

The following topics will be discussed:

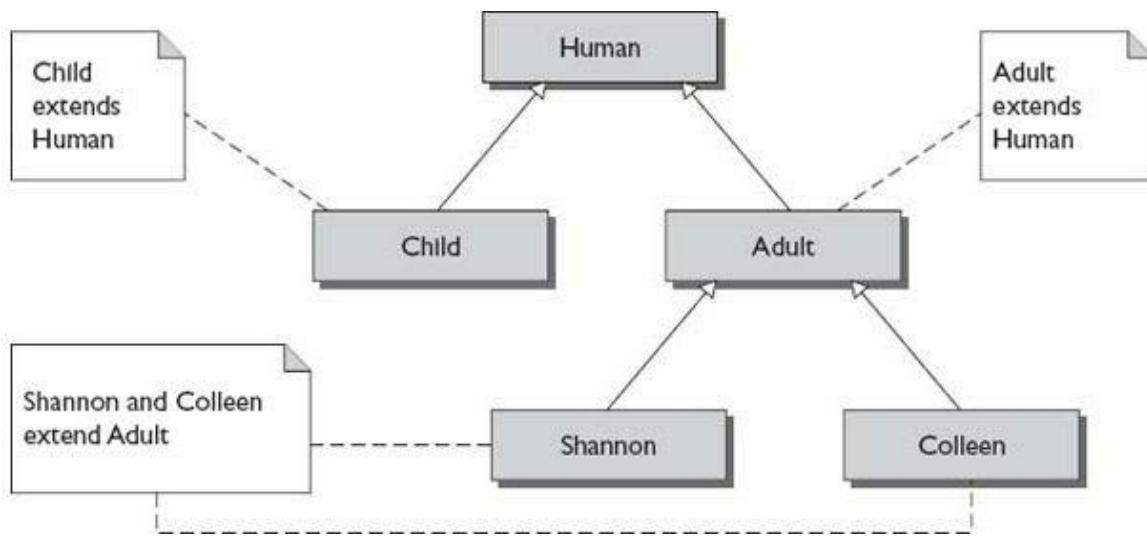
- Polymorphism via class inheritance
- Polymorphism via implementing interfaces
- Polymorphism in code

Polymorphism via Class Inheritance

A more specific object can polymorphically take the place of a more general object. You’ll recall that you can extend one object to create a more specific object that inherits the original object’s functionality in addition to the new functionality. For example, suppose a method requires a `Human` object. When the `Child` and `Adult` classes extend the `Human` class, they would each inherit all of the functionality of `Human`, plus all the more specific functionality of their specific classes. The `Child` and `Adult` objects are guaranteed to inherit all of the methods of the `Human` object. Therefore, both the `Child` and `Adult` objects would satisfy any operation that required a `Human` object. To continue the example further, suppose the `Shannon` and `Colleen` classes each extend the `Adult` class. Each of the objects created from the `Shannon` and `Colleen` classes would inherit the functionality of the more general `Adult` and `Human` classes and could therefore be used anywhere a `Human` or `Adult` object is required.

Polymorphism utilizes the “*is-a*” relationship. In [Figure 8-1](#), the `Child` object *is-a* `Human` object, and the `Adult` object *is-a* `Human` object. Both `Child` and `Adult` are specific types of a `Human` object. Furthermore, the `Shannon` object *is-an* `Adult` object and *is-a* `Human` object. This is also true for the `Colleen` object. The `Shannon` object is not only a more specific type of `Adult` object, but it’s also a more specific type of `Human` object. Any subclass object is a more specific type of its parent object. The *is-a* relationship is created when an object inherits, or extends, another. Any object that extends another object can be said to have an *is-a* relationship to the object that it extends. Any object that has an *is-a* relationship with another can polymorphically be used as that object.

FIGURE 8-1 Polymorphic objects



When an object is polymorphically acting as another object, the more specific object is restricted to using only the public interface of the more general object. In the preceding example, when the `Adult` object is used as a `Human` object, only the methods that are available in the `Human` class can be used. This is because the Java code that is using the `Adult` object as a `Human` object has no knowledge that this `Human` object is really an `Adult` object. This is the benefit of polymorphism. The Java code does not always have to be aware of the specifics of an object. If a general object meets the needs of a method, Java does not care whether the object is general or specific. The only requirement is that the object has an *is-a* relationship with the object that the method requires.

This relationship is unidirectional, so the more specific object can take the place of a general object, but not vice versa. For example, if an `Adult` object were needed, a more general `Human` object would not be able to provide all of the functionality of an `Adult` object.

exam watch

Abstract classes and concrete classes behave the same way with polymorphism. Because an abstract class cannot be instantiated, the only way to assign an object to an abstract data type is by using polymorphism. Pay close attention to how abstract classes are initialized.

Polymorphism via Implementing Interfaces

The application of polymorphism is not limited to class inheritance. Polymorphism can also be applied to the objects of classes that implement interfaces. When a class implements an interface, it is then required to implement all of the methods that the interface contains. By doing this, the class is guaranteed to have the functionality that the interface defines. This allows the objects created from these classes to be treated as instances of the interface.

An interface called `Display`, for example, can be used for classes that have the ability to display text on a screen. This interface contains two methods: One method is used to display text, and the second is used to get the text that is currently being displayed. Any class that implements this interface is declaring to other objects that it has the functionality of a `Display`. By implementing this interface, the class is then required to implement every method that the interface contains. Since the object was created from a class that implements the `Display` interface, it is guaranteed to have the functionality of a `display`. The object has an *is-a* relationship with `Display`. This object can now masquerade as an object of the `Display` type. An object can polymorphically act as any interface that

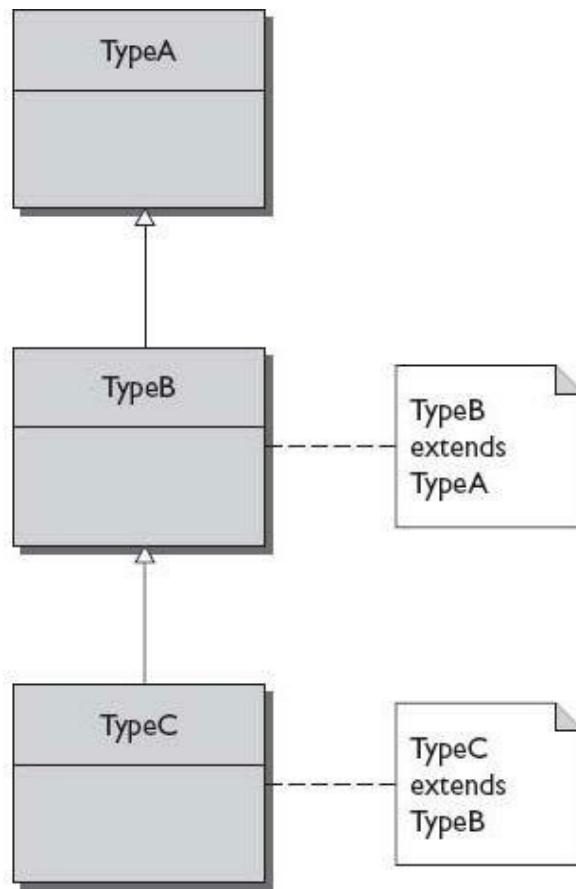
its class or any superclass implements.

on the job *Polymorphism and interfaces are very powerful tools. They are used extensively on large projects. As a professional developer, you'll find it helpful to study design patterns to look for common reusable software designs that make use of the concepts in this chapter. A good developer not only understands all of the basic concepts but also knows how best to use them.*

Polymorphism in Code

When one specific object can be used as another general object polymorphically, the specific object can be used in place of the more general one without being cast. For example (see [Figure 8-2](#)), if class TypeC extends TypeB, and TypeB extends TypeA, any time an object type of TypeA or TypeB is needed, TypeC can be used. The following code segment shows an example of this:

FIGURE 8-2 TypeA, TypeB, and TypeC



```
TypeA var1 = new TypeA();  
TypeA var2 = new TypeB();  
TypeA var3 = new TypeC();  
  
TypeB var4 = new TypeB();  
TypeB var5 = new TypeC();  
  
TypeC var6 = new TypeC();
```

In this example, any subclass can be used interchangeably with its superclass. The variable `var3` is declared as a `TypeA` object, but is initialized with a new `TypeC` object. Even though `var3` is really a `TypeC` object, it will be treated as a `TypeA` object anywhere `var3` is referenced. This is okay because the `TypeC` object has inherited all of the functionality of the `TypeA` and `TypeB` objects. However, since `var3` was declared as `TypeA`, it can now be treated only as an object of this type. If `TypeC` objects have additional methods that are not part of the `TypeA` class, these methods would be unavailable.

More commonly, polymorphism will be used for method arguments. This allows a method to be written more abstractly and therefore be more flexible. For instance, a method may be required to accept a type of animal object as its argument and use it to determine whether the animal is hungry. In this scenario, there is no benefit in creating a method that would accept a `Penguin` object and another that accepts a `PolarBear` object. Instead, it would be a better design to create one single method that accepts an `Animal` class. The state of hunger is general to the `Animal` class. The `Animal` class is a superclass for both the `Penguin` class and `PolarBear` class.

These basic examples are provided to help you understand the concepts of polymorphism. Keep in mind that they are described at a very high level here. This chapter will later offer more examples that show polymorphism in greater depth.

Programming to an Interface

“Programming to an interface” is the concept that code should interact based on a defined set of functionality instead of an explicitly defined object type. In other words, it is better for the public interfaces of objects to use data types that are defined as interfaces as opposed to a particular class when possible. When an object is implementing an interface, it is declaring that it has a certain set of functionalities. Many different classes can implement the same interface and provide its functionality. When a method uses an interface as its argument type, it allows any object, regardless of its type, to be used as long as it implements the interface. This allows the code to be more abstract and flexible and also promotes code reuse. Programming to an interface is also known as “design by contract.”



This chapter goes deeper into polymorphism than is required for the OCA exam, but the information provided here should help you better understand the test questions. Most of the questions on the test will either be a theory question regarding the definition of polymorphism or a simple scenario that will require the OCA candidate to choose a code segment that is correct.

Practical Examples of Polymorphism

The first section in this chapter approached polymorphism from a theoretical viewpoint; this section will look at coding examples. These examples are important for you to understand, and if you review them carefully, they should give you a clear understanding of the concepts presented in the earlier section.

The first example will demonstrate how polymorphism can be applied when a class extends another. There is no difference between the use of concrete and abstract classes. The next example demonstrates the use of polymorphism when interfaces are used. These examples will help reinforce the concepts covered in this chapter. The OCA exam will require that you know how to use polymorphism. Understanding these examples will better prepare you for the polymorphism questions

on the test.

Examples of Polymorphism via Class Inheritance

The following example is intended to demonstrate the use of polymorphism with class inheritance. This example has three classes. Two classes are used to represent phones. The Phone class is intended to be a simple representation of a standard phone. This class has a method to dial a number and return to the state of whether the phone is ringing or not. The second class represents a smart phone and is appropriately named SmartPhone . The SmartPhone class extends the Phone class. This class adds the functionality of being able to send and receive e-mails. The final class is named Tester and is used as a driver to test both phone classes and demonstrate polymorphism in action. The phone classes are simple representations, and most of their functionality is not implemented. Instead, it is noted as comments regarding its intended purposes. The following is the Phone class:

```
public class Phone {  
  
    public void callNumber(long number) {  
        System.out.println("Phone: Calling number " + number);  
        /* Logic to dial number and maintain connection. */  
    }  
  
    public boolean isRinging() {  
        System.out.println("Phone: Checking if phone is ringing");  
        boolean ringing = false;  
        /* Check if the phone is ringing and set the ringing variable */  
        return ringing;  
    }  
}
```

The Phone class is a simple class used for a normal phone with basic features. The class has a callNumber() method that is used to call the number that is passed as an argument. The isRinging() method is used to determine whether the phone is currently ringing. This class prints to standard out its class name and what action it is performing as it enters each method. The Phone class is the base class for the SmartPhone class. The SmartPhone class is listed next:

```
public class SmartPhone extends Phone {  
  
    public void sendEmail(String message, String address) {  
        System.out.println("SmartPhone: Sending Email");  
        /* logic to send email message */  
    }  
  
    public String retrieveEmail() {  
        System.out.println("SmartPhone: Retrieving Email");  
        String messages = new String();  
        /* Return a String containing all of the messages*/  
        return messages;  
    }  
}
```

```

public boolean isRingng() {
    System.out.println("SmartPhone: Checking if phone is ringing");
    boolean ringing = false;
    /* Check for email activity and only continue when there is none. */
    /* Check if the phone is ringing and set the ringing variable */
    return ringing;
}
}

```

The SmartPhone class represents a smart phone. This class extends the Phone class and therefore inherits its functionality. The SmartPhone class has a sendEmail() method that is used to send an e-mail message. It has a retrieveEmail() method that will return a String for any messages that have not yet been retrieved. This class also has an isRingng() method that overrides the isRingng() method from the superclass Phone. Similar to the Phone class, the SmartPhone class prints to standard out the class name and function that it will perform each time it enters a method.

The final class is named Tester. The class has the main() method for the demonstration program. This class exercises all of the methods in the Phone and SmartPhone classes.

```

public class Tester {
    public static void main(String[] args) {
        new Tester();
    }

    public Tester() {
        Phone landLinePhone = new Phone();
        SmartPhone smartPhone = new SmartPhone();
        System.out.println("About to test a land line phone " +
            "as a phone...");
        testPhone(landLinePhone);
        System.out.println("\nAbout to test a smart phone " +
            "as a phone...");
        testPhone(smartPhone);
        System.out.println("\nAbout to test a smart phone " +
            "as a smart phone...");
        testSmartPhone(smartPhone);
    }

    private void testPhone(Phone phone) {
        phone.callNumber(5559869447);
        phone.isRingng();
    }

    private void testSmartPhone(SmartPhone phone) {
        phone.sendEmail("Hi", "edward@ocajexam.com");
        phone.retrieveEmail();
    }
}

```

The main() method kicks off the program by creating a Tester object and therefore calling Tester() the constructor. The constructor is used to call each test method. In between each method call, it prints a line to standard out that indicates what the program is doing. The testPhone() method is used to test each method of the Phone class. It accepts a Phone object as an argument. The final

method is the `testSmartPhone()` method. This method tests each method of the `SmartPhone` class.

The `Tester()` constructor starts by creating two local variables. The first is called `landLinePhone` and is a `Phone` object. The second is called `smartPhone` and is a `SmartPhone` object. The constructor then displays a message and calls the `testPhone()` method with the `landLinePhone` variable as an argument.

Next, the constructor displays another message and again calls the `testPhone()` method. The `smartPhone` variable is used as the argument. The `testPhone()` method requires a `Phone` object as its argument, but the example has used a `SmartPhone` object instead. This is polymorphism. A smart phone is a more specific type of phone. A smart phone can do everything a landline phone can and more. This is represented in the `SmartPhone` class by it extending `Phone`. Notice that the `testPhone()` method is expecting a `Phone` object as an argument. It is perfectly acceptable if it gets a more specific type of phone. However, any additional method of the more specific class cannot be utilized. Since this method is designed for a `Phone` object as an argument, it can use only methods declared in the `Phone` class.

Finally, the constructor displays another status message and calls the `testSmartPhone()` method. This method exercises the methods declared in the `SmartPhone` object. Since polymorphism is unidirectional, the `testSmartPhone()` method cannot be called with a `Phone` object as its argument. The following is the output that would be generated by this program:

```
About to test a land line phone as a phone...
Phone: Calling number 5559869447
Phone: Checking if phone is ringing
```

```
About to test a smart phone as a phone...
Phone: Calling number 5559869447
SmartPhone: Checking if phone is ringing
```

```
About to test a smart phone as a smart phone...
SmartPhone: Sending Email
SmartPhone: Retrieving Email
```

When the `landLinePhone` variable is used with the `testPhone()` method, the output is simply generated from the `Phone` class since it is a `Phone` object. When the `smartPhone` variable is used with the `testPhone()` method, the flow of execution is more complex. Because the `SmartPhone` class extends the `Phone` class, the `SmartPhone` class inherits both the `callNumber()` and `isRinging()` methods. However, the `SmartPhone` class overrides the `isRinging()` method with its own. When the `callNumber()` method is invoked on a `SmartPhone` object, the method in the `Phone` class is used since it is not overridden. However, when the `isRinging()` method is called, the method in the `SmartPhone` class is used. This follows the basic rule of inheritance and overriding methods.

Examples of Polymorphism via Implementing Interfaces

This example will focus on an object's ability to behave polymorphically as an interface that its class implements. This allows objects that may be radically different, but share some common functionality, to be treated similarly. The common functionality is defined in an interface that each class must implement.

This example is composed of three classes and one interface. There is a `Tester` class to test the program. The other two classes are objects representing a goat and his home (that is, his shelter). Both in this program and conceptually, the objects are very different. A goat is a living animal and a goat

shelter is an inanimate item. However, they both share a common ability. Both the Goat class and the GoatShelter class can describe themselves. This functionality has been reflected in the fact that they both implement the Describable interface. Classes that implement this interface are then required to implement the getDescription() method. Here is the Describable interface:

INSIDE THE EXAM

Unidirectional Polymorphism

The OCA exam may try to present the test taker with a polymorphism question in which the more general object behaves as the more specific one. Remember that polymorphism works in only one direction. Only specific objects can behave as more general ones.

```
public interface Describable {  
    public String getDescription();  
}
```

This interface has only one method. The getDescription() method is used to return a description about the object. Any class that implements this interface is stating it has a method that can be used to get its description. The Goat class is shown next.

```
public class Goat implements Describable {  
  
    private String description;  
  
    public Goat(String name){  
        description = "A goat named " + name;  
    }  
  
    public String getDescription() {  
        return description;  
    }  
    /*  
     * Implement other methods for a goat  
     */  
}
```

The Goat class is a simple class that can be used to represent a goat. This class implements the Describable interface and therefore is required to implement the getDescription() method. The constructor of the Goat class has one parameter that it uses to place the name of the goat in the description string. The next class in this example is the GoatShelter class. It is listed next:

```

public class GoatShelter implements Describable {

    private String description;
    private int height;
    private int width;
    private int length;

    public GoatShelter (int height, int width, int length) {
        this.height = height;
        this.width = width;
        this.length = length;
        this.description = "A goat shelter that is " + height + " high, "
            + length + " long and " + width + " wide ";
    }

    public String getDescription() {
        return description;
    }
}

/*
 * Implement other methods for a goat shelter
 */
}

```

The GoatShelter class is designed generally to model a box. Its constructor requires that the dimensions of the box be used as arguments. The constructor also creates the description text that is returned in the getDescription() method. Similar to the Goat class, the GoatShelter class also implements the Describable interface. The final class is the Tester class. This class is used to demonstrate the concept of polymorphism with interfaces.

```

public class Tester {

    public static void main(String[] args) {
        new Tester();
    }

    public Tester() {
        Goat goat = new Goat("Bob");
        GoatShelter goatShelter = new GoatShelter (4, 4, 6);
        System.out.println(description(goat));
        System.out.println(description(goatShelter));
    }

    private String description(Describable d) {
        return d.getDescription();
    }
}

```

The Tester class contains the main() method that starts the execution of the program. This calls the Tester() constructor, where a Goat object and GoatShelter object are both created. The description() method is then used to print to standard out the description of each object. The

`description()` method requires a `Describable` object. It is impossible to have a true `Describable` object since it is an interface. However, classes that implement this interface are declaring that they have the functionality of `Describable`. These objects can then polymorphically act as if they were of type `Describable`. The following is the output of this program:

```
A goat named Bob  
A goat shelter that is 4 high, 4 long and 6 wide
```

EXERCISE 8-1

Add Functionality to the Describable Example

This exercise will use the preceding example. The goal of the exercise is to compile and run the example and add a class that implements the `Describable` interface.

1. Copy the example into the text editor or IDE of your choice.
 2. Compile and run the example to ensure the code has been copied correctly.
 3. Add a new class that implements the `Describable` interface.
 4. Compile and run the application.
-

Examples of Programming to an Interface

This example will demonstrate the concept of programming to an interface. This concept allows a developer to define the functionality that is required instead of defining an actual object type. This creates more flexible code that adheres to the object-oriented design principle of creating reusable code.

Suppose a developer creates a class that is used for creating log files. This class is responsible for creating and managing the log file on the file system, and then appending the log messages to it. This class is called `Logger`. The `Logger` class has a method called `appendToLog()` that accepts one object as an argument and then appends a message about it in the log.

The developer could overload this method with every possible data type that the program uses. Although this would work, it would be very inefficient. Suppose the program-to-an-interface concept is used instead, and the developer creates an interface that defines the required method for a logable class. This interface is called `Logable`. The `appendToLog()` method then uses the `Logable` interface as its argument. Any class that requires logging could implement this interface and then be used polymorphically with the `appendToLog()` method.

The following is the `Logable` interface:

```
public interface Logable {  
    public String getInitInfo();  
    public String getLogableEvent();  
}
```

The `Logable` interface is a basic interface that defines the methods required to work with the `appendToLog()` method in the `Logger` class. The `appendToLog()` method is not concerned with the

details of an object other than what pertains to logging. By using this interface, the developer has defined a functionality requirement as opposed to a strict object data type. This is what is meant by the phrase “programming to an interface.”

The `Logger` class is displayed next:

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class Logger {

    private BufferedWriter out;

    public Logger() throws IOException {
        out = new BufferedWriter(new FileWriter("logfile.txt"));
    }

    public void appendToLog(Logable logable) throws IOException {
        out.write("Object history: " + logable.getInitInfo());
        out.newLine();
        out.write("Object log event: " + logable.getLogableEvent());
        out.newLine();
    }

    public void close() throws IOException {
        out.flush();
        out.close();
    }
}
```

The `Logger` class creates a `BufferedWriter`, which is a means to write to a file. (This is beyond the scope of this chapter, and therefore will not be discussed.) The `appendToLog()` method is used to write to the log file. This class uses the `Logable` interface to remain flexible. This method will work with any other class that implements this interface and will follow the program-to-an-interface concept.

The next class is the `NetworkConnection` class. This class implements the `Logable` interface.

```
public class NetworkConnection implements Logable {  
  
    private long createdTimestamp;  
    private String currentLogMessage;  
  
    public NetworkConnection() {  
        createdTimestamp = System.currentTimeMillis();  
        currentLogMessage = "Initialized";  
    }  
  
    public void connect(){  
        /*  
         * Established connection  
         */  
        currentLogMessage = "Connected at " + System.currentTimeMillis();  
    }  
  
    public String getInitInfo() {  
        return "NetworkConnection object created " + createdTimestamp;  
    }  
  
    public String getLogableEvent() {  
        return currentLogMessage;  
    }  
}
```

This class implements the `Logable` interface and all of the methods required for it. When this class is polymorphically behaving as the `Logable` data type, the code being used does not care about the implementation details of the class. As long as the class implements the `Logable` interface, it is free to choose how the methods are implemented.

The `SystemStatus` class is the other class that uses the `Logable` interface.

```
public class SystemStatus implements Logable {  
  
    private long createdTimestamp;  
  
    public SystemStatus() {  
        createdTimestamp = System.currentTimeMillis();  
    }  
}
```

```

private int getStatus() {
    if(System.currentTimeMillis() - createdTimestamp > 1000) {
        return 1;
    }
    else{
        return -1;
    }
}

public String getInitInfo() {
    return "SystemStatus object created " + createdTimestamp;
}

public String getLogableEvent() {
    return String.valueOf("Status: "+getStatus());
}
}

```

The `SystemStatus` class's only similarity to the `NetworkConnection` class is that they both implement the `Logable` interface. This class chooses to implement the required `getInitInfo()` and `getLogableEvent()` methods in a different manner than the `NetworkConnection` class.

The final class is the `Tester` class. This is a simple class that demonstrates all of the preceding classes and interface in action.

```

public class Tester {

    public static void main(String[] args) throws Exception {
        new Tester();
    }

    public Tester() throws Exception {
        Logger logger = new Logger();
        SystemStatus systemStatus = new SystemStatus();
        NetworkConnection networkConnection = new NetworkConnection();
        logger.appendToFile(systemStatus);
        logger.appendToFile(networkConnection);
        networkConnection.connect();
        Thread.sleep(2000);
        logger.appendToFile(systemStatus);
        logger.appendToFile(networkConnection);
        logger.close();
    }
}

```

The `Tester` class does all of its work in its constructor. The class creates a new `Logger` object called `logger`. It then creates a `SystemStatus` object named `systemStatus` and a `NetworkConnection` object named `networkConnection`. It then uses the `appendToFile()` method from the `Logger` object. This method uses the `Logable` object as a parameter. Because both the `SystemStatus` and `NetworkConnection` classes implement this interface, their objects can be used polymorphically with this method. The following text is written to the log file:

```
Object history: SystemStatus object created 1238811437373
Object log event: Status: -1
Object history: NetworkConnection object created 1238811437374
Object log event: Initialized
Object history: SystemStatus object created 1238811437373
Object log event: Status: 1
Object history: NetworkConnection object created 1238811437374
Object log event: Connected at 1238811437374
```

CERTIFICATION OBJECTIVE

Understand Casting

Exam Objective 7.4 Determine when casting is necessary

So far, this chapter has discussed how to use a more specific object in place of a general one. Polymorphism allows for any subclass to fill in for its superclass. You saw that no special conversion is needed to do this. We are now going to look at the opposite situation. How can you take an object that is general and make it more specific? *Casting* will allow you to convert an object back to its original runtime type or any of its superclasses.

When Casting Is Needed

Polymorphism allows an object to be used as a more general object without the need to add new syntax. However, casting must be used when an object is to be used as a more detailed type or when converting a primitive to a type that will cause data to be lost. Polymorphism can occur without interaction because there is no possibility of incompatible data. When there is the chance of incompatible data types, the Java compiler requires the developer to declare formally their intention to use a variable as a different type.

Casting must occur when the primitive double is used as a float or a HashMap object is used as a LinkedHashMap object. This section will look at each case and explain when it is possible to convert the object or primitive and why you might need to do so.

The Java syntax to cast an object or primitive is relatively simple. To cast an object or primitive, place the type before it in parentheses. In the next example, detailedScore is declared as a double. It is then assigned to a float. For this code to be valid and compile, detailedScore must be cast to a float. The cast is performed by placing (float) in front of the double, detailedScore. This tells the compiler to treat detailedScore as a float and allow it to be assigned the variable score, which is declared as a float.

```
double detailedScore = 1.2;
float score = (float)detailedScore;
```

Casting Primitives

Primitives need to be explicitly cast when the conversion will potentially result in the loss of precision. If there is no potential for precision loss, the compiler will automatically cast the primitive. Precision is lost when a larger primitive is cast to a smaller primitive. Precision can also be lost when a primitive with a floating-point decimal is cast to a whole number primitive type. For example, an int is a 32-bit signed two's complement integer and has a minimum value of -2,147,483,648 and a

maximum value of 2,147,483,647 (inclusive). A byte is an 8-bit signed two's complement integer with a minimum value of -128 and a maximum value of 127 (inclusive). If an `int` that was storing the value 1236 were cast to a byte, there would be a loss of precision, because a byte cannot store a value as large as 1236.

So, if a byte cannot store a large `int`, but the compiler will allow you to cast the `int` into a byte, what happens at runtime? At runtime, the Java virtual machine (JVM) will truncate the bits from the `int`. In the cast of an `int` with the value of 1236, the JVM would truncate it to a byte with the value of -44. The flowing code will output Byte: -44:

```
int i = 1236;
byte b = (byte) i;
System.out.println("Byte: "+b);
```

To understand where the value -44 comes from, you need to look at the binary representation of the numbers. In the following example, the number 1236 is displayed as a 32-bit binary number. The number -44 is displayed as an 8-bit binary number. Remember that the most significant bit is used as a signed byte.

| | |
|--------|---|
| 1236 = | 0000 0000 0000 0000 0000 0100 1101 0100 |
| -44 = | 1101 0100 |

It is easy to see that the 24 most significant bits are being truncated from the `int`. The negative signed value is due to the most significant bit of the byte being a 1.

A cast is also required to convert a floating-point decimal number such as a `float` or `double` to a primitive whole number. The decimal value of the number is truncated. For example, the value of 5.7 would be truncated to 5. Rounding the number is not a consideration. Once the decimal value is removed, bits are truncated from the most significant bit onward until it is the proper new size.

TABLE 8-1 Primitive Casting

| Finished Primitive Type | | | | | | | |
|-------------------------|------------------------------|------------------------------|------------------------------|------------------------------|------------------------------|------------------------------|-----------------|
| Starting Primitive Type | byte | short | char | int | long | float | double |
| byte | | Safe Conversion | Safe Conversion |
| short | Explicit Cast Precision Lost | | Safe Conversion | Safe Conversion | Safe Conversion | Safe Conversion | Safe Conversion |
| char | Explicit Cast Precision Lost | Explicit Cast Precision Lost | | Safe Conversion | Safe Conversion | Safe Conversion | Safe Conversion |
| int | Explicit Cast Precision Lost | Explicit Cast Precision Lost | Explicit Cast Precision Lost | | Safe Conversion | Safe Conversion | Safe Conversion |
| long | Explicit Cast Precision Lost | | Explicit Cast Precision Lost | Safe Conversion |
| float | Explicit Cast Precision Lost | | Safe Conversion |
| double | Explicit Cast Precision Lost | |

Casting Between Primitives and Objects

Since the introduction of Java 5.0, it has become possible to cast primitives to and from their object wrapper classes. Not only has Java made it easy to make these basic conversions, but it will also automatically do the casting with a feature called “autoboxing” and “auto-unboxing”. Autoboxing and unboxing allows a primitive to be used interchangeably as its object wrapper class. The proper conversion is done automatically. The following code segment will demonstrate different ways of making an `Integer` object from an `int`:

```
int i = 8;
Integer obj1 = new Integer(i);
Integer obj2 = (Integer)i;
Integer obj3 = i;
Float obj4 = 5.7f;
//Below is invalid
Integer obj5 = obj4;
```

In this code segment, a primitive `int` named `i` is created and set to 8. This variable is then used to create three `Integer` objects. The object `obj1` is created using the method that was required before Java 5.0. It uses the `Integer` class’s constructor to instantiate a new object. This line of code demonstrates the basic way to instantiate an object and does not demonstrate any casting. Next, the `obj2` variable is set by casting a primitive `int` to an `Integer` object. Although this syntax is valid, it is rarely used. The `obj3` object is set by using only the primitive `i`. No cast is used in this case. However, the autoboxing feature of Java performs the cast for you. Most developers will use the autoboxing and unboxing method to move between primitives and their wrapper class. A `Float` object is also created. Autoboxing is used to convert the `5.7f` value to a `Float` object. Finally, an invalid example is shown.

Unlike a primitive where a `float` will be automatically truncated to fit into an `int`, a `Float` object will not automatically truncate to fit into an `Integer` object. This line of code would produce compiler warnings and throw a runtime exception. It is important to remember that you do not have to cast explicatively when writing the code, because a cast will be done for you automatically.

INSIDE THE EXAM

Hidden Casting

Be on the lookout for hidden casting. The Java compiler does not require you to cast explicitly between a primitive and its corresponding wrapper classes. However, a cast is still happening automatically as a convenience.

Casting Objects

An object can polymorphically become any object that is its superclass. Once an object is assigned a more general type, it can no longer access its more specific features. An object has to be cast back into its original runtime type to use these methods. It is important to ensure that the object to be cast was instantiated as that object or an object that inherited it. A runtime exception will be thrown if an object is incorrectly cast. Following are three example classes:

```
public class ClassA {  
    public String whoAmI() {  
        return "ClassA";  
    }  
    public String specialClassAMethod() {  
        return "ClassA only method";  
    }  
}  
public class ClassB extends ClassA {  
  
    public String whoAmI() {  
        return "ClassB";  
    }  
  
    public String specialClassBMethod() {  
        return "ClassB only method";  
    }  
}
```

`ClassA` in the example is a basic class with two methods: one method returns a string with its class name, and the other represents a unique method that only `ClassA` contains. `ClassB` extends `ClassA`. It overrides the `whoAmI` method with its own functionality. It also contains a method that is unique to itself, `specialClassBMethod`.

The following code segment creates a `ClassB` object named `obj1`. It then creates a `ClassA` object,

`obj2`. This object is initialized with a new `ClassB` object. This is valid because `ClassB` objects can polymorphically act as a `ClassA` object since `ClassB` extends `ClassA`. Finally a second `ClassA` object is created and initialized with a new `ClassA` object.

```
ClassB obj1 = new ClassB();
ClassA obj2 = new ClassB();;
ClassA obj3 = new ClassA();

System.out.println("obj1: " + obj1.whoAmI());
System.out.println("obj2: " + obj2.whoAmI());
System.out.println("obj3: " + obj3.whoAmI());
```

The following is the output when the preceding code is executed:

```
obj1: ClassB
obj2: ClassB
obj3: ClassA
```

The `obj2` object is assigned the type of `ClassA`, but is initialized with a `ClassB` object. This object retains all of the functionality of a `ClassB` object. This can be seen when the string from `obj2.whoAmI()` method is printed. However, since this object is assigned the type `ClassA`, it can be treated only as a `ClassA` object. For example, the following line of code would not compile:

```
System.out.println("obj2: " + obj2.specialClassBMethod());
```

Even though `obj2` was created as a `ClassB` object, it was assigned the `ClassA` type and therefore can be treated only as a `ClassA` object. To be able to access the functionality that `ClassB` provides, the object would have to be cast to the `ClassB` type. The next example demonstrates `obj2` being cast into a `ClassB` object:

```
ClassB obj4 = (ClassB) obj2;
System.out.println("obj4: " + obj4.specialClassBMethod());
```

It is also possible to cast the object inline. This shorthand syntax would be as follows:

```
System.out.println("obj2: " + ((ClassB) obj2).specialClassBMethod());
```

It is important that you understand that `obj3` cannot be cast to a `ClassB` object. You can cast to an object type successfully only if the object was instantiated as it or as one of its subclasses. If it were cast to `ClassB`, a runtime exception would be thrown.



When taking the OCA exam, you should remember that you have to cast objects only when you are going down the inheritance chain—that is, superclass to subclass. It is also important that you understand that the object at one point must have been the object that is being cast to, or a subclass of it.



When casting an object, you should always check whether the object can be cast without generating an exception. To check whether an object is the proper type, use the Java instanceof operator. This operator can be used in an if statement to determine whether object A is an instance of object B.

```
if(obj2 instanceof ClassB) {  
    /*Do cast*/  
}
```

CERTIFICATION SUMMARY

Polymorphism is a fundamental concept of any object-oriented programming language. This chapter has discussed the fundamental concepts of polymorphism and then demonstrated these concepts through examples.

The first part of this chapter defined polymorphism. Polymorphism is a tool that can be used to create more reliable code and produce it faster. Polymorphism allows you to treat a specific object as if it were a more general object. In other words, a class's object can masquerade as any object that the class uses to derive itself. The benefit is that applications can be written more abstractly. A common form of polymorphism is between classes that extend other classes. A class's object can be treated as any object that it extends, and this includes both concrete and abstract classes. Polymorphism also allows an object to be treated as any interface that it implements.

Polymorphism is most commonly used for method arguments. Often, a method will require only a general object. A more specific object can be used, however, since it will provide all of the functionality of the general object. You can think of the *is-a* relationship to help you understand polymorphism. For example, a specific object such as `Blue` *is-a* `Color`. So `Blue` is a specific object and extends the `Color` object.

This chapter then covered the benefits of programming to an interface. Programming to an interface allows the developer to specify the capabilities or behaviors that are expected, instead of strictly defining an expected object type. This allows the code to be more abstract and flexible. This is an example of polymorphism in use. Programming to an interface requires that an object polymorphically act as the interface it implements.

The chapter then used coding examples to clarify polymorphism and programming to an interface. These examples highlighted the important concepts discussed in theory in this chapter.

Finally, casting was examined. Casting is often used with polymorphism to return an object back to its original level of detail. It is important that you learn how to use casting correctly. Improper use can lead to unstable and difficult-to-maintain code that often crashes.

TWO-MINUTE DRILL

Understand Polymorphism

- Polymorphism is a fundamental concept of object-oriented languages, including Java.
- Polymorphism stimulates code reuse.
- Polymorphism allows one object to act as either one of its superclasses or as, an interface that it implements.
- In an *is-a* relationship, the subclass object (the more specific object) “is-a” superclass object (the

more general object).

- Polymorphism is unidirectional. More specific objects can act polymorphically only as more general objects.
- By implementing an interface, an object is declaring it has the functionality defined in the interface. This allows that object to act polymorphically as the interface.
- Programming to an interface is the concept in which the developer defines the required functionality instead of defining strict object data types. This allows other developers to interact with the code using any object they choose as long as it implements the required interfaces.
- An object can be used interchangeably with any of its superclasses without the need to be cast.
- An object can be used interchangeably with any interface that it implements without the need to be cast.
- When a more specific object is polymorphically used as a general object, the more specific functionality is not available.
- Polymorphism is commonly used for method arguments.

Understand Casting

- Casting is needed to convert an object back to a more detailed object in its inheritance chain.
- If you cast an object to an invalid type, a runtime exception will be thrown.
- To cast an object, you place the class name in front of it in parentheses.
- You must cast a primitive to another primitive type if there is the possibility that precision will be lost.
- Casting is not needed when going from a primitive to its wrapper class. Java's autoboxing/unboxing feature will do this automatically.
- Objects need to be cast when they move down the inheritance chain—that is, from a superclass to a subclass. Polymorphism allows objects to become more general.

SELF TEST

Understand Polymorphism

- 1.** Which statement is true about the term *polymorphism*?
 - A. It is a Latin word that roughly means “changeable.”
 - B. It is a Greek word that roughly means “many forms.”
 - C. It is an Old English word that roughly means “insectlike.”
 - D. It is a new technical term that means “Java object.”
- 2.** What type of object can polymorphically behave as another?
 - A. An object can act as any subclass of the class it was created from.
 - B. An object can act as any superclass of the class it was created from.
 - C. An object can act as any other abstract class.
- 3.** Polymorphism helps to facilitate which of the following? (Choose all that apply.)
 - A. Highly optimized code
 - B. Code reuse
 - C. Code obfuscation
 - D. Code that is generic and flexible
- 4.** What is a correct “is-a” relationship?
 - A. A specific object “is-a” more generic one.

- B. A generic object “is-a” more specific one.
 - C. A null object “is-an” object.
5. Which of the following statements explain why an object can polymorphically behave as an interface?
- A. By implementing the interface, the object is required to have all of the functionality that the interface represents.
 - B. By implementing the interface, the object inherits all the required methods it defines.
 - C. An object can behave as an interface because interfaces do not have a strict expected behavior and therefore any object can act as an interface.
6. What does it mean if a developer is programming to an interface?
- A. The developer is implementing an interface for the class he or she is working on.
 - B. The developer was given a set of interfaces he or she must implement.
 - C. The developer is defining the functionality instead of strict object types as much as possible.

The following code example will be referenced in questions 7 through 12. Afterward, see [Figure 8-3](#).

The Drivable interface:

```
public interface Drivable {  
    /*  
     * Drivable definitions  
     */  
}
```

The Tractor class:

```
public class Tractor implements Drivable{  
    /*  
     * Tractor functionality  
     */  
}
```

The Vehicle class:

```
public class Vehicle {  
    /*  
     * Vehicle functionality  
     */  
}
```

The Car class:

```
public class Car extends Vehicle implements Drivable{  
    /*  
     * Car functionality  
     */  
}
```

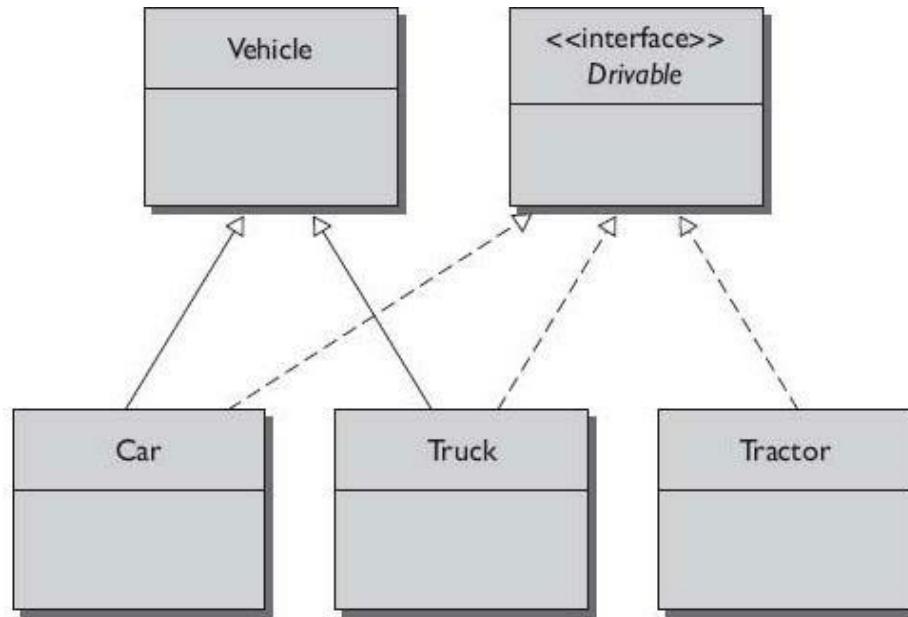
The Truck class:

```

public class Truck extends Vehicle implements Drivable{
/*
 * Truck functionality
 */
}

```

FIGURE 8-3 UML for questions 7–12



- 7.** Given the preceding classes and interface, would the following code segment produce errors when compiled?

```

Car car = new Car();
Vehicle vehicle = car;

```

- A. No errors would be produced.
- B. This code would result in compile errors.

- 8.** Given the preceding classes and interface, would the following code segment produce errors when compiled?

```

Truck truck = new Truck();
Drivable drivable = truck;

```

- A. No errors would be produced.
- B. This code would result in compile errors.

- 9.** Given the preceding classes and interface, would the following code segment produce errors when compiled?

```

Tractor tractor = new Tractor();
Vehicle vehicle = tractor;

```

- A. No errors would be produced.
- B. This code would result in compile errors.

10. Given the preceding classes and interface, would the following code segment produce errors when compiled?

```
Drivable drivable = new Drivable();  
Truck truck = drivable;
```

- A. No errors would be produced.
- B. This code would result in compile errors.

11. Given the preceding classes and interface, would the following code segment produce errors when compiled?

```
Vehicle vehicle = new Vehicle();  
Object o = vehicle;
```

- A. No errors would be produced.
- B. This code would result in compile errors.

12. Given the preceding classes and interface, would the following code segment produce errors when compiled?

```
Truck truck = new Truck();  
Object o = truck;
```

- A. No errors would be produced.
- B. This code would result in compile errors.

Understand Casting

13. In what cases is casting needed? (Choose all that apply.)

- A. Going from a superclass to subclass
- B. Going from a subclass to superclass
- C. Using an int as a double
- D. Using a float as a long

14. Why must the a variable be casted?

- A. So future developers understand the intended use of the variable
- B. To tell the compiler that the data conversion is safe to make
- C. Because it was used polymorphically before

15. What would cause an exception to be thrown from a cast?

- A. If precision is lost due to the cast
- B. If the object is cast to a superclass of its current type
- C. If the object was never instantiated as the object that it is being cast to or one of its subclasses
- D. If the object is being cast is null

SELF TEST ANSWERS

Understand Polymorphism

1. Which statement is true about the term *polymorphism*?

- A. It is a Latin word that roughly means “changeable.”
- B. It is a Greek word that roughly means “many forms.”
- C. It is an Old English word that roughly means “insectlike.”
- D. It is a new technical term that means “Java object.”

Answer:

- B. The word polymorphism comes from the Greeks and means “many forms.”
 - A, C, and D are incorrect.
-

2. What type of object can polymorphically behave as another?

- A. An object can act as any subclass of the class it was created from.
- B. An object can act as any superclass of the class it was created from.
- C. An object can act as any other abstract class.

Answer:

- B. An object inherits all of the functionality of its superclasses and can therefore polymorphically behave as they do.
 - A and C are incorrect. A is incorrect because an object cannot behave as its subclass since this class is more specific and contains functionality that is not present in the superclass. C is incorrect because there needs to be an “is-a” relationship between the classes. This answer does not mention what the relationship is.
-

3. Polymorphism helps to facilitate which of the following? (Choose all that apply.)

- A. Highly optimized code
- B. Code reuse
- C. Code obfuscation
- D. Code that is generic and flexible

Answer:

- B and D. Polymorphism aids in creating reusable code, because it allows the code to be written more abstractly, thus B is correct. Similar to B, polymorphism allows the code to be generic by using generic data types that any more specific object can fulfill. Thus, D is also correct.
 - A and C are incorrect. A is incorrect because polymorphism has no effect on the level of optimization of the code. C is incorrect because obfuscated code (code that is intentionally difficult to read) is not related to polymorphism.
-

4. What is a correct “is-a” relationship?

- A. A specific object “is-a” more generic one.
- B. A generic object “is-a” more specific one.
- C. A null object “is-an” object.

Answer:

- A.** A more specific object can be considered to be a more generic one. This is the fundamental principle of polymorphism.
 - B** and **C** are incorrect. **B** is incorrect because generic objects do not have all of the functionality of more specific ones and therefore do not possess an “is-a” relationship with the specific objects. **C** is incorrect because a null object has no effect on its relationship with other objects.
-

5. Which of the following statements explain why an object can polymorphically behave as an interface?

- A.** By implementing the interface, the object is required to have all of the functionality that the interface represents.
- B.** By implementing the interface, the object inherits all the required methods it defines.
- C.** An object can behave as an interface because interfaces do not have a strict expected behavior and therefore any object can act as an interface.

Answer:

- A.** When a class implements an interface, it is then required to implement all the methods the interface contains. This gives the class the functionality defined in the interface and therefore allows this class to behave as the interface.
 - B** and **C** are incorrect. **B** is incorrect because nothing is inherited when an interface is implemented. **C** is incorrect because each interface has a strict behavior expected of it. This is represented by the methods that must be implemented.
-

6. What does it mean if a developer is programming to an interface?

- A.** The developer is implementing an interface for the class he or she is working on.
- B.** The developer was given a set of interfaces he or she must implement.
- C.** The developer is defining the functionality instead of strict object types as much as possible.

Answer:

- C.** Programming to an interface means that a developer is defining functionality instead of object data types. Any object can then implement the required interface and be used. If an interface were not used, only objects of the specific defined data type would be usable.
 - A** and **B** are incorrect. **A** is incorrect because programming to an interface is a larger concept than just implementing one interface in one class. **B** is incorrect because in this situation the developer is just implementing a group of interfaces that have been predetermined.
-

The following code example will be referenced in questions 7 through 12.

The Drivable interface:

```
public interface Drivable {  
    /*  
     * Drivable definitions  
     */  
}
```

The Tractor class:

```
public class Tractor implements Drivable{
/*
 * Tractor functionality
 */
}
```

The Vehicle class:

```
public class Vehicle {
/*
 * Vehicle functionality
 */
}
```

The Car class:

```
public class Car extends Vehicle implements Drivable{
/*
 * Car functionality
 */
}
```

The Truck class:

```
public class Truck extends Vehicle implements Drivable{
/*
 * Truck functionality
 */
}
```

7. Given the preceding classes and interface, would the following code segment produce errors when compiled?

```
Car car = new Car();
Vehicle vehicle = car;
```

- A. No errors would be produced.
B. This code would result in compile errors.

Answer:

- A. No errors would be produced because the car class extends the Vehicle class and therefore can be used as a Vehicle object.
 B. is incorrect.

8. Given the preceding classes and interface, would the following code segment produce errors when compiled?

```
Truck truck = new Truck();  
Drivable drivable = truck;
```

- A. No errors would be produced.
- B. This code would result in compile errors.

Answer:

- A. No errors would be produced because the `Truck` class implements the `Drivable` interface and therefore can be used as a `Drivable` object.
 - B is incorrect.
-

9. Given the preceding classes and interface, would the following code segment produce errors when compiled?

```
Tractor tractor = new Tractor();  
Vehicle vehicle = tractor;
```

- A. No errors would be produced.
- B. This code would result in compile errors.

Answer:

- B. This code would result in compile errors because the `Vehicle` class is not a superclass for the `Tractor` class.
 - A is incorrect.
-

10. Given the preceding classes and interface, would the following code segment produce errors when compiled?

```
Drivable drivable = new Drivable();  
Truck truck = drivable;
```

- A. No errors would be produced.
- B. This code would result in compile errors.

Answer:

- B. This code would result in compile errors because the `Drivable` interface cannot be instantiated since it is an interface.
 - A is incorrect.
-

11. Given the preceding classes and interface, would the following code segment produce errors when compiled?

```
Vehicle vehicle = new Vehicle();  
Object o = vehicle;
```

- A. No errors would be produced.
 - B. This code would result in compile errors.
-

Answer:

- A. No errors would be produced because the Vehicle class is concrete, and the Object class is the superclass for every Java object.
 - B is incorrect.
-

12. Given the preceding classes and interface, would the following code segment produce errors when compiled?

```
Truck truck = new Truck();  
Object o = truck;
```

- A. No errors would be produced.
 - B. This code would result in compile errors.
-

Answer:

- A. No errors would be produced because the Object class is the superclass for all Java objects.
 - B is incorrect.
-

Understand Casting

13. In what cases is casting needed? (Choose all that apply.)

- A. Going from a superclass to subclass
 - B. Going from a subclass to superclass
 - C. Using an int as a double
 - D. Using a float as a long
-

- A and D. A is correct because a cast is always needed to go down the inheritance chain. D is correct because a long is not a floating-point number; precision will be lost and therefore it requires a cast.
 - B and C are incorrect. B is incorrect because polymorphism occurs when going from a subclass to a superclass. C is incorrect because no precision is lost when using an int as a double, so no cast is needed.
-

14. Why must the a variable be casted?

- A. So future developers understand the intended use of the variable
 - B. To tell the compiler that the data conversion is safe to make
 - C. Because it was used polymorphically before
-

- B. If there is a possibility of incompatible data, a cast must be used.
 - A and C are incorrect.
-

15. What would cause an exception to be thrown from a cast?

- A. If precision is lost due to the cast
 - B. If the object is cast to a superclass of its current type
 - C. If the object was never instantiated as the object that it is being cast to or one of its subclasses
 - D. If the object is being cast is null
-

C. If the object was not instantiated with the level of detail it is being cast to, it will cause a runtime exception.

A, B, and D are incorrect. A is incorrect because if precision is lost between primitive types, there is no exception. B is an example of polymorphism. A cast can be done but is not needed. D is incorrect because a null object can be cast. This will not cause an exception. However, if the cast is to an invalid type, the compiler will produce an error.



9

Handling Exceptions

CERTIFICATION OBJECTIVES

- Understand the Rationale and Types of Exceptions
- Understand the Nature of Exceptions
- Alter the Program Flow
- Recognize Common Exceptions
- ✓ Two-Minute Drill

Q&A Self Test

“**T**hrow and catch.” Once that phrase makes sense to you, you’ll know that you understand exception handling in Java. Throughout this chapter, we will explore the primary focus of exception handling in Java by throwing exceptions and handling them where appropriate. We’ll discuss the different types of exceptions, when and how to handle them as well as recognizing commonly seen checked exceptions, unchecked exceptions, and errors. By the end of the chapter, you should know enough about exception handling in Java to score well on the exceptions-related questions when you sit the exam. Good luck!

Understand the Rationale and Types of Exceptions

Exam Objective 8.3 Describe what exceptions are used for in Java

Exam Objective 8.1 Differentiate among checked exceptions, Runtime Exceptions, and Errors

The Java Language Specification provides the following definition for a software exception: “When a program violates the semantic constraints of the Java programming language, the Java virtual machine signals this error to the program as an exception.”

Exceptions are used in Java to handle events that affect the normal flow of the application’s execution. These events can occur from coding errors or issues with the resources in place. For portability and robustness, Java aims to manage exceptions in a predictable way. This is done by throwing and catching exceptions that have been or will be grouped in logical, sensible ways. We’ll explore the exception hierarchy in Java and the different types of exceptions.

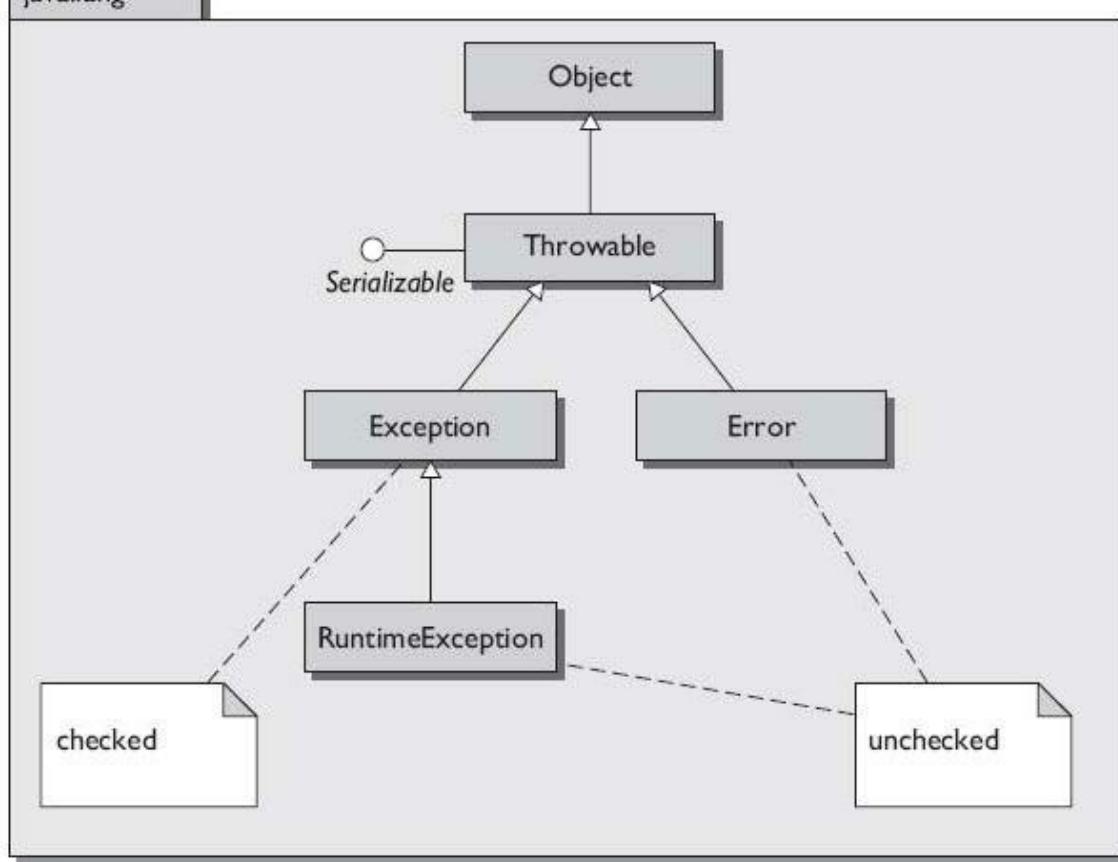
The following topics will be covered in the following pages:

- Exception hierarchy in Java
- Checked exceptions
- Unchecked exceptions
- Errors

Exception Hierarchy in Java

An exception in Java is defined by an instance of the class `Throwable`. Exception classes include the class `Throwable` and all of its subclasses. [Figure 9-1](#) depicts the exception class’s hierarchy.

FIGURE 9-1 Exception handling class hierarchy



In the hierarchy, the two direct subclasses to `Throwable` are `Exception` and `Error`. Another important subclass is `RuntimeException`, which is a direct subclass to the `Exception` class. `Runtime` exceptions and errors are not checked. All of the other exceptions are checked. Let's take a look at each category.

Checked Exceptions

Checked exceptions are checked by the compiler at compile time. Checked exceptions must be caught by a `catch` block or the thread will terminate, and so will the application if it is the only thread. The application will not terminate if you have started other threads. A handler can actually be registered to catch the exception in a multithreaded application. The following code demonstrates an application throwing an uncaught exception, but continuing to run because of the existence of other threads:

```

public class CEExample implements Runnable {
    public static void main(String args[]) throws IOException {
        Thread thrd = new Thread(new Main());
        thrd.start();
        try {
            Thread.sleep(5000);
        } catch (InterruptedException ie) {
            ie.printStackTrace();
        }
        throw new IOException("Oops");
    }
    public void run() {
        while (true) {
            try {
                Thread.sleep(5000);
            } catch (InterruptedException ie) {
                ie.printStackTrace();
            }
            System.out.println("Alive!");
        }
    }
}

```

Checked exceptions are all subclasses of the `Exception` class; however, `RuntimeException` and its subclasses are not subclasses of `Exception`.

Unchecked Exceptions

Unchecked exceptions are checked at runtime, rather than at compile time. Unchecked exceptions are all subclasses to the `RuntimeException` class, including `RuntimeException` itself. Unchecked exceptions and `Errors` do not need to be caught. More specifically, you should code your application with the assumption that unchecked exceptions would not be encountered; this is why you do not need to waste your effort trying to catch and manage them, as these exceptions should rarely occur. Most runtime exceptions occur due to programming mistakes.

 **Oracle provides excellent online coverage of exceptions through the Java Tutorials, “Lesson: Exceptions.” The section “Unchecked Exceptions - The Controversy” is of particular interest and is worth review.**

(Unchecked) Errors

Errors are unchecked exceptions that represent extreme conditions and will typically cause your application to fail. Most errors are unrecoverable external errors. Errors shouldn't be handled, but they can be.

 **Heatlamp (www.jmolly.com/heatlamp/) allows for the visualization of stack traces. Since every Java developer will be reading through stack traces and will need to be familiar with how they are laid out, why not use a visualization tool to aid in your analysis?**

EXERCISE 9-1

Determining When to Use Assertions in Place of Exceptions

This chapter does not cover assertions, because assertions are not covered on the exam. But you should know when to use them.

- Do a little research and determine when assertions should be used in place of traditional exception handling.
-

EXERCISE 9-2

Analyzing the Source Code of Java Exceptions

JDocs (www.jdocs.com) lets you take a look at the Java source code. This exercise will have you using JDocs to inspect the source code for one of each of the exception handling types of classes.

1. Use JDocs to look at the source for the `FileNotFoundException` checked exception. Write a short summary of your findings and observations.
 2. Use JDocs to look at the source for the `NumberFormatException` unchecked exception. Write a short summary of your findings and observations.
 3. Use JDocs to look at the source for the `AssertionError` unchecked error. Write a short summary of your findings and observations.
-

CERTIFICATION OBJECTIVE

Understand the Nature of Exceptions

Exam Objective 8.4 Invoke a method that throws an exception

Creating, throwing, and propagating exceptions is easier than it seems. In this section, we'll take a quick look at throwing exceptions; in the next section, we'll look further into catching them.

The following topics will be covered:

- Defining exceptions
- Throwing exceptions
- Propagating exceptions

Defining Exceptions

Exceptions in Java should have a no argument constructor and a constructor with a single `String` argument, as shown in the upcoming code sample. These guidelines are provided by convention and should be followed. To reinforce this point and provide a reference to the convention, *Murach's Java Programming, 4th Edition*, by Joel Murach (Mike Murach & Associates, 2011) states the following:

“By convention, all exception classes should have a default constructor that doesn’t accept any arguments and another constructor that accepts a string argument.” The following example demonstrates a custom exception called RecordException:

```
public class RecordException extends Exception {  
    public RecordException() {  
        super();  
    }  
    public RecordException(String s) {  
        super(s);  
    }  
}
```

This example demonstrates the creation of a checked exception as it is directly inherited from the Exception class. This Exception class could easily be made an unchecked exception by inheriting from the RuntimeException class.

All Exception subclass names should end with Exception—for example, SQLException and NumberFormatException. All Error subclass names should end with Error—for example, VirtualMachineError and OutOfMemoryError.

Throwing Exceptions

Methods use the throw statement to throw exceptions in Java. Only objects of the type or subtype of class Throwable can be thrown. Here is a valid throw statement:

```
throw new IllegalStateException();
```

An error will occur if the application attempts to throw an object that is not an instance of Throwable:

```
throw new String(); // Must be subtype of throwable
```

This statement will invoke a compilation error and will also result in the following error message if executed:

```
$ Exception in thread "main" java.lang.RuntimeException:  
  
Uncompilable source code - incompatible types  
required: java.lang.Throwable  
found:    java.lang.String  
at thrower.Thrower.main(Thrower.java:10)
```



Remember that any checked exception, unchecked exception, or error may be thrown.

Propagating Exceptions

Exceptions can be propagated all the way up to the `main` method. If the `main` method does not handle the exception, it will be thrown to the Java virtual machine (JVM) and the application will be terminated. When an application throws an exception, the method that contains the exception must catch the exception or send it to the calling method, or else the application will end. To send an exception to the calling method, the `throws` keyword must be included in the method declaration along with the exception name to be thrown. This is demonstrated in the following code:

```
import java.io.IOException;
public class Thrower {
    public static void main(String[] args) {
        Thrower t = new Thrower();
        try {
            t.throw1();
        } catch (IOException ex) {
            System.out.println("An IOException has occurred");
        }
    }
    public void throw1() throws IOException {
        throw2();
    }
    public void throw2() throws IOException {
        throw3();
    }
    public void throw3() throws IOException {
        throw4();
    }
    public void throw4() throws IOException {
        throw new IOException();
    }
}
```

A best practice is to use a logging API in conjunction with the items being caught in your catch statements. Take a look at your current projects. If the projects are littered with print statements (directed to standard out), especially in the catch clauses, consider doing some refactoring to leverage off of the logging APIs.

EXERCISE 9-3

Creating a Custom Exception Class

In this exercise, you will create a custom checked exception class.

1. Determine a name and purpose for your checked exception class.
2. Create a new class that inherits from the `Exception` class.
3. Create a no-argument constructor that calls `super()`.
4. Create a constructor with a single string argument that calls `super(s)`.
5. Develop code that will throw the exception.

CERTIFICATION OBJECTIVE

Alter the Program Flow

Exam Objective 8.2 Create a try-catch block and determine how exceptions alter normal program flow

Code within the scope of a `try` clause will allow any exceptions that are thrown within that clause to be evaluated by the associated catch clauses. Various statements work in conjunction with the `try` clause. We'll take a look at each one.

The following topics will be covered in these pages:

- The `try-catch` statement
- The `try-finally` statement
- The `try-catch-finally` statement
- The `try-with-resources` statement
- The `multi-catch` clause

The `try-catch` Statement

The `try-catch` statement contains code to “catch” thrown exceptions from within the `try` block, either explicitly or propagated up through method calls. In a `try` clause, the code within its block is attempting (trying) to complete without encountering any exceptions. If an exception is thrown, all statements after the exception in the `try` block will not be executed. This is demonstrated in the following code:

```
try {  
    System.out.print("What's up!");  
    throw new ArithmeticException();  
    System.out.print(", Hello!"); // code never reached  
    System.out.print(", Hi there! "); // code never reached  
} catch (ArithmetricException ae) {  
    System.out.print(", Howdy! ");  
    ae.printStackTrace();  
}  
$ What's up!, Howdy!
```

Always begin with the subclasses when ordering the catch clauses to catch exceptions. This necessary design (coding implementation) is illustrated in the next example:

```

public void demonstrateTryCatch() {
    try {
        throw new NumberFormatException();
    } catch (NumberFormatException nfe) { // Exception is caught here
        nfe.printStackTrace();
    } catch (IllegalArgumentException iae) {
        iae.printStackTrace();
    } catch (RuntimeException re) {
        re.printStackTrace();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

NumberFormatException is on the bottom of the hierarchy and is a subclass to IllegalArgumentException. IllegalArgumentException is a subclass to RuntimeException, and RuntimeException is a subclass to Exception. Therefore, the order of the catch clauses is relative to this order: NumberFormatException, IllegalArgumentException, RuntimeException, and Exception.

In the real world, it is not a best practice to catch RuntimeException or the Exception classes themselves. That is, you should catch their subclasses as appropriate to your code.

You have several options when you're printing data relative to an exception being caught, such as using the getMessage, toString, and printStackTrace methods. The getMessage method returns a detailed message about the exception. The toString method returns detailed messages about the exception and a class name. The printStackTrace method prints a detailed message, the class name, and a stack trace. These methods are demonstrated in the following code:

```

public void demonstrateTryCatch() {
    try {
        int result = (3 / 0); // throws ArithmeticException
    } catch (ArithmaticException ae) {
        System.out.println(ae.getMessage());
        System.out.println(ae.toString());
        ae.printStackTrace();
    }
}

```

They result in this output:

```

$ / by zero (divide by zero)
$ java.lang.ArithmaticException: / by zero
$ java.lang.ArithmaticException: / by zero
$     at com.ocajexam.exceptions_tester.TryStatements.demon-
strateTryCatch
(TryStatements.java:20)
     at com.ocajexam.exceptions_tester.Main.main(Main.java:26)

```

The catch clause should never be empty. This is considered *silencing* an exception and is not a good practice. A catch clause should either log a message, print the stack trace, or provide some sort of notification to the consumer (developer, user, and so on) of the application that an error has occurred. Here's an example:

```
public void demonstrateTryCatch() {  
    try {  
        throw new NumberFormatException();  
    } catch (NumberFormatException nfe) {  
        // This catch clause should not be empty  
    }  
}
```

When exceptions are being caught, if the exception class is not found, the system will look for the superclass of the exception being thrown. In the following example, `IllegalArgumentException` is the superclass of `NumberFormatException`, so it will catch the exception being thrown:

```
public void demonstrateTryCatch() {  
    try {  
        throw new NumberFormatException();  
    } catch (IllegalArgumentException iae) {  
        iae.printStackTrace();  
    }  
}
```

In the preceding examples, you'll notice a common naming convention for the exception parameters. It is customary to use the first letter of each word in the exception class for the parameter name. For example, for `ClassCastException`, the parameter name would be `cce`.

The try-finally Statement

In the `try-finally` statement, the `finally` clause is always executed after the `try` clause has completed, unless of course the application is terminated prior to the `try` clause finishing up.

The following example demonstrates the `try-finally` statement:

```
public void demonstrateTryFinally() {  
    try {  
        System.out.print("Jab");  
    } finally {  
        System.out.println(" and Roundhouse ");  
    }  
}
```

The resultant output of running the `demonstrateTryFinally` method is as follows:

```
$ Jab and Roundhouse
```

The following example demonstrates a `try-finally` statement in which the `try` clause exits prematurely with a call to `System.exit`:

```
public void demonstrateTryFinally() {  
    try {  
        System.out.print("Jab");  
        System.exit(0);  
    } finally {  
        System.out.println(" and Roundhouse ");  
    }  
}
```

The resultant output of running the `demonstrateTryFinally` method in this scenario is as follows:

```
$ Jab
```

Note that if an unchecked exception is thrown in a `try` block, the `finally` method will be invoked as shown next. Because the unchecked exception is not caught, the application terminates, but only *after* the `finally` statement completes.

```
public class Example {  
    public static void main(String[] args) {  
        System.out.print("Bread");  
        try {  
            throw new NumberFormatException(); //unchecked exception  
        } finally {  
            System.out.print(" and "); // this gets executed!  
        }  
        System.out.println(" butter"); // this statement is not  
reached  
    }  
}
```

Because of the termination of the application after completion of the `finally` statement, but before the end of the program, we do not get our butter:

```
$ Bread and
```

The try-catch-finally Statement

The `try-catch-finally` statement is exactly what it appears to be—a `try-catch` statement with a `finally` clause attached. The `finally` clause will execute after the `try-catch` portion of the statement, unless the application is terminated prior to the entry point of the `finally` clause.

The try-with-resources Statement

The `try-with-resources` statement provides the capability of declaring resources that must be closed when they are no longer needed. These resources must implement the `AutoCloseable` interface. Prior to Java SE 7, these resources were typically closed in the `finally` clause. With version 7, the resources are automatically closed at the end of the `try-with-resources` statement—that is, if an exception is thrown, the resource is closed, or if the code block reaches its end, the resource is closed.

The Javadoc documentation for `AutoCloseable`

(<http://docs.oracle.com/javase/7/docs/api/java/lang/AutoCloseable.html>) calls out all of the classes

that can be used with the `try-with-resources` statement. The following classes here implement `AutoCloseable`; they give you the big picture of the wide and possible usages of resources for the `try-with-resources` statement:

| | | |
|---|--|--|
| <code>AbstractInterruptibleChannel</code> | <code>AbstractSelectableChannel</code> | <code>AbstractSelector</code> |
| <code>AsynchronousFileChannel</code> | <code>AsynchronousServerSocketChannel</code> | <code>AsynchronousSocketChannel</code> |
| <code>AudioInputStream</code> | <code>BufferedInputStream</code> | <code>BufferedOutputStream</code> |
| <code>BufferedReader</code> | <code>BufferedWriter</code> | <code>ByteArrayInputStream</code> |
| <code>ByteArrayOutputStream</code> | <code>CharArrayReader</code> | <code>CharArrayWriter</code> |
| <code>CheckedInputStream</code> | <code>CheckedOutputStream</code> | <code>CipherInputStream</code> |
| <code>CipherOutputStream</code> | <code>DatagramChannel</code> | <code>DatagramSocket</code> |
| <code>DataInputStream</code> | <code>DataOutputStream</code> | <code>DeflaterInputStream</code> |
| <code>DeflaterOutputStream</code> | <code>DigestInputStream</code> | <code>DigestOutputStream</code> |
| <code>FileCacheImageInputStream</code> | <code>FileCacheImageOutputStream</code> | <code>FileChannel</code> |
| <code>FileImageInputStream</code> | <code>FileImageOutputStream</code> | <code>FileInputStream</code> |
| <code>FileLock</code> | <code>FileOutputStream</code> | <code>FileReader</code> |
| <code>FileSystem</code> | <code>FileWriter</code> | <code>FilterInputStream</code> |
| <code>FilterOutputStream</code> | <code>FilterReader</code> | <code>FilterWriter</code> |
| <code>Formatter</code> | <code>ForwardingJavaFileManager</code> | <code>GZIPInputStream</code> |
| <code>GZIPOutputStream</code> | <code>ImageInputStreamImpl</code> | <code>ImageOutputStreamImpl</code> |
| <code>InflaterInputStream</code> | <code>InflaterOutputStream</code> | <code>InputStream</code> |
| <code>InputStream</code> | <code>InputStream</code> | <code>InputStreamReader</code> |
| <code>JarFile</code> | <code>JarInputStream</code> | <code>JarOutputStream</code> |
| <code>LineNumberInputStream</code> | <code>LineNumberReader</code> | <code>LogStream</code> |
| <code>MemoryCacheImageInputStream</code> | <code>MemoryCacheImageOutputStream</code> | <code>MLet</code> |
| <code>MulticastSocket</code> | <code>ObjectInputStream</code> | <code> ObjectOutputStream</code> |
| <code>OutputStream</code> | <code>OutputStream</code> | <code>OutputStream</code> |
| <code>OutputStreamWriter</code> | <code>Pipe.SinkChannel</code> | <code>Pipe.SourceChannel</code> |
| <code>PipedInputStream</code> | <code>PipedOutputStream</code> | <code>PipedReader</code> |
| <code>PipedWriter</code> | <code>PrintStream</code> | <code>PrintWriter</code> |
| <code>PrivateMLet</code> | <code>ProgressMonitorInputStream</code> | <code>PushbackInputStream</code> |
| <code>PushbackReader</code> | <code>RandomAccessFile</code> | <code>Reader</code> |
| <code>RMIConnectionImpl</code> | <code>RMIConnectionImpl_Stub</code> | <code>RMICConnector</code> |
| <code>RMIIIOPServerImpl</code> | <code>RMIJRMPServerImpl</code> | <code>RMIServerImpl</code> |
| <code>Scanner</code> | <code>SelectableChannel</code> | <code>Selector</code> |
| <code>SequenceInputStream</code> | <code>ServerSocket</code> | <code>ServerSocketChannel</code> |
| <code>Socket</code> | <code>SocketChannel</code> | <code>SSLSocket</code> |
| <code>SSLSocket</code> | <code>StringBufferInputStream</code> | <code>StringReader</code> |
| <code>StringWriter</code> | <code>URLClassLoader</code> | <code>Writer</code> |
| <code>XMLDecoder</code> | <code>XMLEncoder</code> | <code>ZipFile</code> |
| <code>ZipInputStream</code> | <code>ZipOutputStream</code> | |

Now let's compare some code. Here is the legacy `try-catch-finally` code performing the `close` method explicitly in the `finally` statement:

```

public void demonstrateTryWithResources() {
    Scanner sc = new Scanner(System.in);
    try {
        System.out.print("Number of apples: ");
        int apples = sc.nextInt();
        System.out.print("Number of oranges: ");
        int oranges = sc.nextInt();
        System.out.println("Pieces of Fruit: " + (apples + oranges));
    } catch (InputMismatchException ime) {
        ime.printStackTrace();
    } finally {
        sc.close();
    }
}

```

Now let's refactor this code into a `try-with-resources` statement, where the `close` method will be called implicitly. Notice two changes: the first is the `Scanner` declaration as an argument in the `try` clause, and the second is the removal of the `finally` clause.

```

public void demonstrateTryWithResources() {
    try (Scanner sc = new Scanner(System.in)) {
        System.out.print("Number of apples: ");
        int apples = sc.nextInt();
        System.out.print("Number of oranges: ");
        int oranges = sc.nextInt();
        System.out.println("Pieces of Fruit: " + (apples + oranges));
    } catch (InputMismatchException ime) {
        ime.printStackTrace();
    }
}

```

The multi-catch Clause

The `multi-catch` clause allows for multiple exception arguments in one `catch` clause. Examine the following code segment:

```

...
} catch (ArrayIndexOutOfBoundsException aioobe) {
} catch (NullPointerException nbe) {}
...

```

These two catch clauses can be refactored into one `multi-catch` clause using the appropriate syntax:

```
catch (exceptionTypeA | exceptionTypeB [| ExceptionTypeX] ... e) {}
```

An example of `multi-catch` is shown in the following `demonstrateMultiCatch` method:

```

public void demonstrateMultiCatch2() {
    try {
        Random random = new Random();
        int i = random.nextInt(2);
        if (i == 0) {
            throw new ArrayIndexOutOfBoundsException();
        } else {
            throw new NullPointerException();
        }
    } catch (ArrayIndexOutOfBoundsException | NullPointerException e) {
        e.printStackTrace();
    }
}

```

on the job *The left margin of the NetBeans source editor will provide hints that encourage the automated refactoring of multiple related catch statements to a multi-catch statement. You can search for “Seven NetBeans Hints for Modernizing Java Code” on the Web to find more information.*

EXERCISE 9-4

Using NetBeans Code Templates for Exception Handling Elements

The NetBeans integrated development environment (IDE) includes a nice feature called *code templates*. Code templates are abbreviated strings that expand into fuller strings or blocks of code. There are currently six related code templates for exception handling. This exercise has you using them all:

1. Make use of the ca code template.
 2. Make use of the fy code template.
 3. Make use of the th code template.
 4. Make use of the tw code template.
 5. Make use of the twn code template.
 6. Make use of the trycatch code template.
-

CERTIFICATION OBJECTIVE

Recognize Common Exceptions

Exam Objective 8.5 Recognize common exception classes and categories

This objective is pretty much a review, because we've already discussed the differences between exception types. To help you grasp the specifics of the most common exceptions, let's take a look at the class hierarchies in some diagrams and produce some unchecked exceptions.

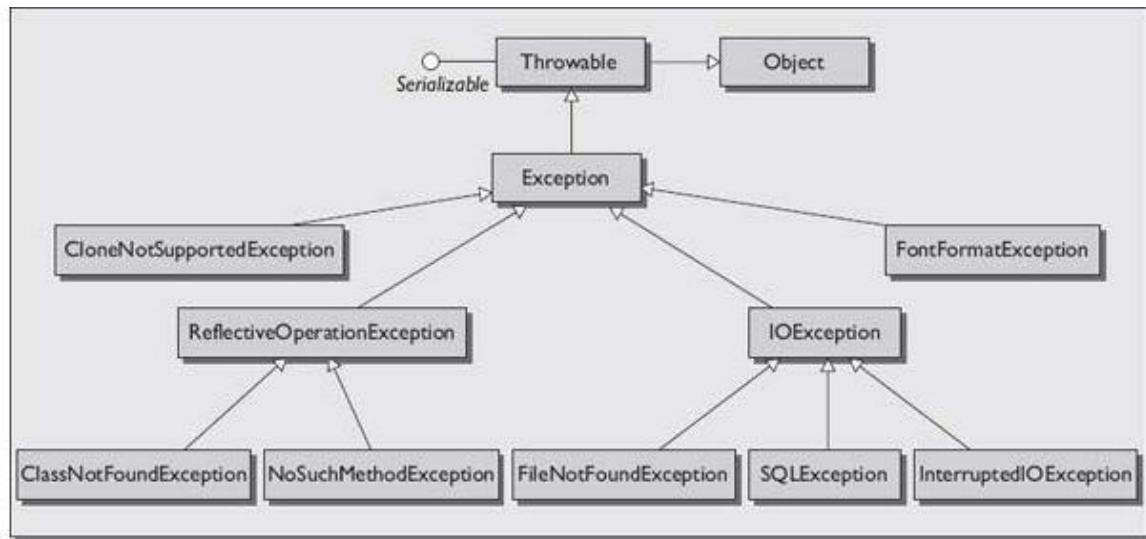
Common Checked Exceptions

Common checked exceptions include the following:

| | |
|----------------------------|------------------------|
| CloneNotSupportedException | ClassNotFoundException |
| NoSuchMethodException | IOException |
| FileNotFoundException | SQLException |
| InterruptedException | FontFormatException |

These exceptions are shown in [Figure 9-2](#) and are detailed in the following sections.

FIGURE 9-2 Common checked exceptions



CloneNotSupportedException

The `CloneNotSupportedException` is thrown when the `clone` method is called by an object that is not able to be cloned. Deep copy cannot be generated. Deep copying (for example, deep cloning) is when a cloned object makes an exact duplicate of an object's state. To learn more about deep copies (and shallow copies) consider reading the online article, "Deep Copy and Shallow Copy" from JusForTechies (http://www.jusfortechies.com/java/core-java/deepcopy_and_shallowcopy.php).

ClassNotFoundException

The `ClassNotFoundException` is thrown when a class cannot be loaded because of a failure to locate its definition.

NoSuchMethodException

The `NoSuchMethodException` is thrown when a called method is unable to be located. For example, a `NoSuchMethodException` would occur if you tried to use reflection and attempted to invoke a method that did not exist.

FileNotFoundException

The `FileNotFoundException` is thrown when a file is that cannot be found is attempted to be opened.

IOException

The `IOException` is thrown when a failed input/output operation occurs.

SQLException

The SQLException is thrown when a database error occurs.

InterruptedException

The InterruptedException is thrown when a thread is interrupted. This class has a bytesTransferred field that provides information on how many bytes were transferred successfully before the interruption occurred.

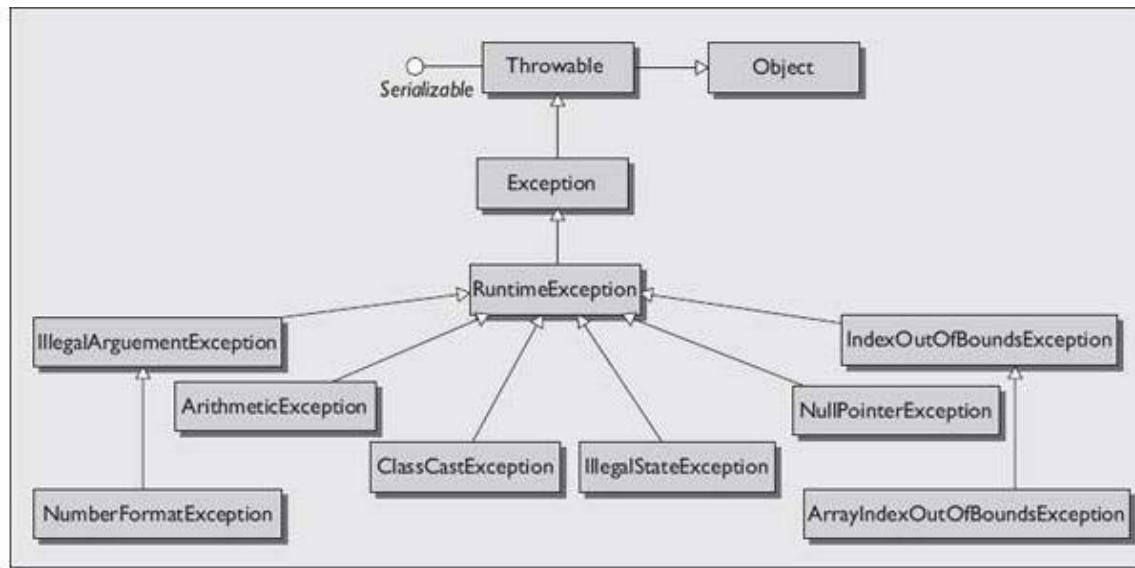
Common Unchecked Exceptions

Common unchecked exceptions include the following:

| | |
|--------------------------------|---------------------------|
| IllegalArgumentException | NumberFormatException |
| ArrayIndexOutOfBoundsException | IndexOutOfBoundsException |
| NullPointerException | IllegalStateException |
| IllegalStateException | ClassCastException |
| ArithmaticException | |

These exceptions are visualized in [Figure 9-3](#) and are detailed in the following sections with examples.

FIGURE 9-3 Common unchecked exceptions



The `IllegalArgumentException` Class

The `IllegalArgumentException` of the `java.nio.file` package is thrown when a method has been passed an illegal or inappropriate argument.

Here's an example:

```
public void forceIllegalArgumentException() {
    int s = 1;
    Path path = Paths.get(s); // IllegalArgumentException
}
```

The NumberFormatException Class

The `NumberFormatException` is thrown when the application has attempted to convert a string to one of the numeric types with the wrong format.

Here's an example:

```
public void forceNumberFormatException() {
    Double.parseDouble("2.1");
    Double.parseDouble("INVALID"); // NumberFormatException
}
```

The ArrayIndexOutOfBoundsException Class

The `ArrayIndexOutOfBoundsException` is thrown when an array has been accessed with an illegal index that is either less than, equal to, or greater than the size of the array.

Here's an example:

```
public void forceArrayIndexOutOfBoundsException() {
    Float[][] num = new Float[2][3];
    num[2][0] = (float)1.0;
    num[2][1] = (float)2.0;
    System.out.println(num[2][2]);
    System.out.println(num[2][3]); // ArrayIndexOutOfBoundsException
}
```

The IndexOutOfBoundsException Class

The `IndexOutOfBoundsException` is thrown when an index is out of range.

Here's an example:

```
public void forceIndexOutOfBoundsException() {
    List gorillaSpecies = new LinkedList();
    gorillaSpecies.add("Eastern");
    gorillaSpecies.add("Western");
    System.out.println(gorillaSpecies.get(1));
    System.out.println(gorillaSpecies.get(2)); // IndexOutOfBoundsException
}
```

The NullPointerException Class

The `NullPointerException` is thrown when an application is required to use an object but finds null instead.

Here's an example:

```
public void forceNullPointerException() {
    String iceCreamFlavor = "vanilla";
    iceCreamFlavor = null;
    System.out.println(iceCreamFlavor.length()); // NullPointerException
}
```

The IllegalStateException Class

The `IllegalStateException` is thrown when a method has been invoked at an illegal or inappropriate time due to being in an inappropriate state.

Here's an example:

```

public void forceIllegalStateException() {
    ArrayList chord = new ArrayList();
    chord.add("D");
    chord.add("G");
    chord.add("B");
    chord.add("G");
    Iterator it = chord.iterator();
    while (it.hasNext()) {
        it.next();
        it.remove();
        it.remove(); // IllegalStateException (remove depends on next)
    }
}

```

The ClassCastException Class

The `ClassCastException` is thrown when the code attempts to cast an object to a subclass of which it is not an instance.

Here's an example:

```

public void forceClassCastException() {
    Object x = new Float("1.0");
    System.out.println((Double) x);
    System.out.println((String) x); // ClassCastException
}

```

The ArithmeticException Class

The `ArithmaticException` is thrown when an exceptional arithmetic condition has occurred.

Here's an example:

```

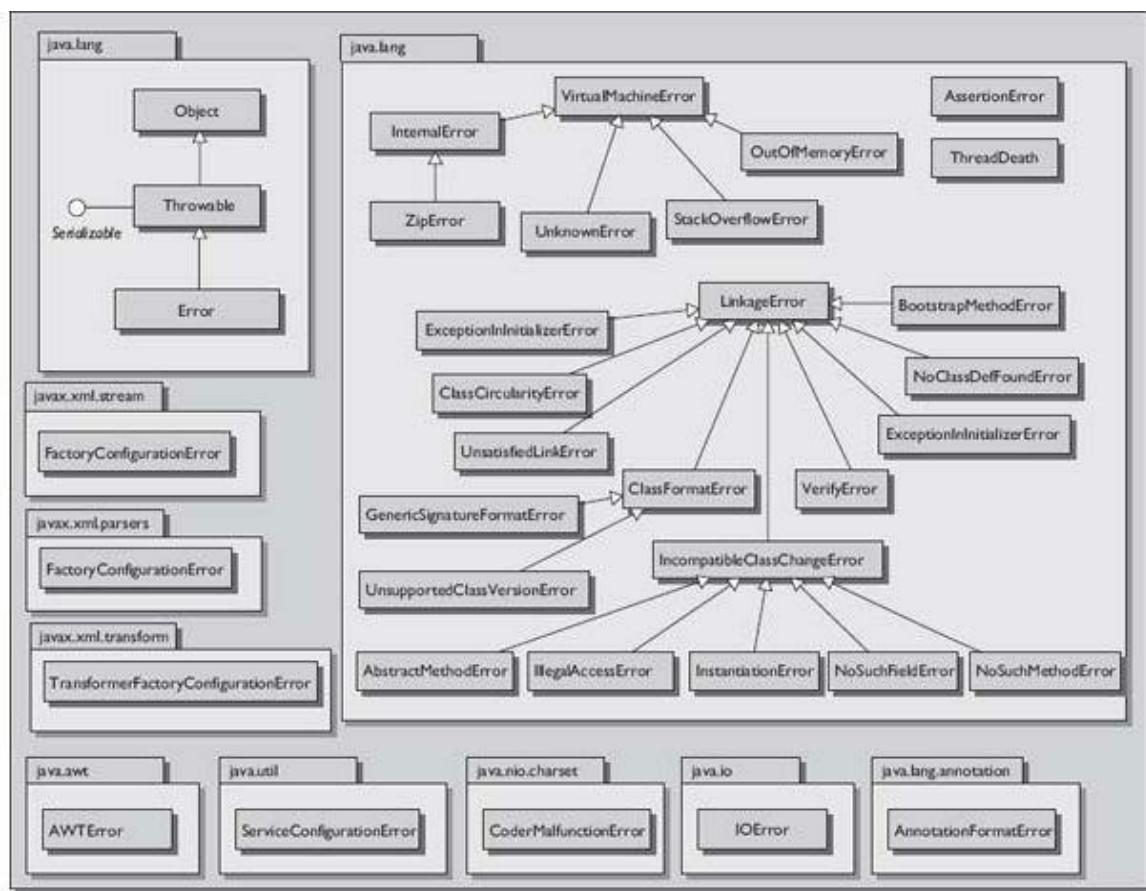
public void forceArithmaticException() {
    int apple;
    apple = (4 / 2);
    apple = (4 / 0); // ArithmaticException
}

```

Common Errors

Common (unchecked) errors include the following: `AssertionError`, `ExceptionInInitializeError`, `VirtualMachineError`, `OutOfMemoryError`, and `NoClassDefFoundError`. These errors as well as all of the error classes in JDK 7 (there are many) are visualized in [Figure 9-4](#). The five common errors are detailed in the following sections.

FIGURE 9-4 Common errors



The AssertionError Class

The `AssertionError` is thrown upon a failed assertion.

The ExceptionInInitializerError Class

The `ExceptionInInitializerError` is thrown when an unexpected exception occurs in a static initializer.

The VirtualMachineError Class

The `VirtualMachineError` is thrown when a JVM error occurs.

The OutOfMemoryError Class

The `OutOfMemoryError` is thrown when garbage collection is performed but is unable to free up any space.

The NoClassDefFoundError Class

The `NoClassDefFoundError` is thrown when the JVM cannot find a class definition that was found at compile time.

EXERCISE 9-5

Creating an Error Condition

Errors are not typically encountered, but when they are, your application is probably doomed. For demonstration on creating an error, the following code causes an `OutOfMemoryError` (“Exception in thread ‘main’ java.lang.OutOfMemoryError: Java heap space”):

```

public void forceStackOverFlowError() {
    Integer counter = 0;
    ArrayList<Integer> unstoppable = new ArrayList<>();
    while (true) {
        unstoppable.add(counter);
        counter++;
        if (counter % 10000 == 0) {
            System.out.println(counter
        }
    }
}

```

It's now your turn.

1. Select an error you'd like to force from [Figure 9-4](#).
 2. Study the error in the Javadoc documentation. You can find documentation for the subclasses of the `Error` class here: <http://docs.oracle.com/javase/7/docs/api/>.
 3. Develop and execute a simple application.
 4. Do whatever it takes to impress your selected `Error` on the application.
 5. Share what you did with the JavaRanch at this thread, “Various examples of having errors thrown relative to the `Error` class,” at www.coderanch.com/t/583633/java-SCJA/certification/Various-examples-having-errorsthrown#2656332.
-

CERTIFICATION SUMMARY

This chapter discussed the exception handling hierarchy in Java. We went over the different types/categories of exceptions in Java. We discussed the various `try` block statements and how to throw, catch, and handle exceptions. And finally, we discussed the various types of common exceptions and errors that you'll see when working with the Java programming language.

When you sit the OCA exam, you will see several questions on exception handling, most of them dealing with presented code samples. Your knowledge of working with exceptions must be strong, or you will feel defeated even before you finish the exam. If after reading this chapter, you still think your knowledge is weak on exceptions, read it through again, making sure you complete all the exercises. Also, for this chapter as well as the others, you should always feel comfortable with going straight into the Java Language Specification for clarity or information:

<http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf>.

Finally, mastering exception handling won't be important just for the exam; you'll be using it a lot in your own code for your projects at work, at school, and anywhere else.

TWO-MINUTE DRILL

Understand the Rationale and Types of Exceptions

- All exceptions and errors inherit from the `Throwable` class.
- Types include checked exceptions, unchecked exceptions, and errors.

- Checked exceptions are checked by the compiler at compile time.
- Checked exceptions must be caught by a catch block, or the application will terminate.
- Checked exceptions are all subclasses to the `Exception` class; however, `RuntimeException` and its subclasses are not part of the `Exception` class.
- Unchecked exceptions are checked at runtime, not compile time.
- Unchecked exceptions are all subclasses to the `RuntimeException` class, including `RuntimeException` itself.
- Unchecked exceptions and errors do not need to be caught.
- Errors represent extreme conditions and will typically cause your application to fail.

Understand the Nature of Exceptions

- Code cannot be placed between the `try` and `catch` blocks, `try` and `finally` blocks, or `catch` and `finally` blocks—that is, extra code cannot be immediately placed before or after the curly braces that separate the blocks in these statements.
- The common naming convention for `catch` clause arguments is the representation of a string containing the first letter of each word of the exception being passed.
- Exceptions are thrown with the `throw` keyword.
- The `throws` keyword is used in method definitions of methods that throw an exception.
- Methods of the `Throwable` class provide support of gathering information about a thrown exception. The methods `getMessage`, `toString`, and `printStackTrace` are commonly used.
- Thrown exceptions move up the call stack until they are caught. If they are not caught and reach the main method, the application will end.

Alter the Program Flow

- The `try` block must contain code that can throw an exception.
- The `try` block must have one `catch` or `finally` block.
- The `try-with-resources` statement declares resources that can be automatically closed. The objects must implement `AutoCloseable`.
- The `catch` block must be ordered by subtypes first.
- The multi-catch feature allows for multiple exception types to be caught in one `catch` block.
- The `try-catch` statement is a valid statement that does not include a `finally` clause.
- The `finally` block of the `try-catch-finally` and `try-finally` statements are always invoked, unless the JVM terminates first.
- The `finally` block is typically used for releasing resources.

Recognize Common Exceptions

- Common checked exceptions include `CloneNotSupportedException`, `ClassNotFoundException`, `NoSuchMethodException`, `IOException`, `FileNotFoundException`, `SQLException`, `InterruptedException`, and `FontFormatException`.
- Common unchecked exceptions include `IllegalArgumentException`, `NumberFormatException`, `ArrayIndexOutOfBoundsException`, `IndexOutOfBoundsException`, `NullPointerException`, `IllegalStateException`, `IllegalStateException`, `ClassCastException`, and `ArithmaticException`.
- Common (unchecked) errors include `AssertionError`, `ExceptionInInitializerError`, `VirtualMachineError`, `OutOfMemoryError`, and `NoClassDefFoundError`.

SELF TEST

Understand the Rationale and Types of Exceptions

1. Which class has the fewest subclasses: `Exception`, `RuntimeException`, or `Error`? Select the correct statement.
 - A. The `Exception` class has fewer subclasses than the `RuntimeException` and `Error` classes.
 - B. The `RuntimeException` class has fewer subclasses than the `Exception` and `Error` classes.
 - C. The `Error` class has fewer subclasses than the `Exception` and `RuntimeException` classes.
2. Of the following types of exceptions, which will an IDE help you in catching, if it is not handled in your code?
 - A. Checked exceptions.
 - B. Unchecked exceptions.
3. Which statement about the `Throwable` class is not correct?
 - A. The `Throwable` class extends the `Object` class.
 - B. The `Throwable` class implements the `Serializable` interface.
 - C. Direct subclasses to the `Throwable` class include the `RuntimeException` and `Error` classes.
 - D. The `Throwable` class is in the `java.lang` package.

Understand the Nature of Exceptions

4. Which of the following classes can be thrown? (Choose all that apply.)
 - A. `throw new Error();`
 - B. `throw new RuntimeException();`
 - C. `throw new Exception();`
 - D. `throw new Assertion();`
 - E. `throw new Throwable();`
5. Given:

```
public static void testMethod1() {  
    try {  
        testMethod2();  
    } catch (ArithmetricException ae) {  
        System.out.println("Dock");  
    }  
}  
  
public static void testMethod2() throws ArithmetricException {  
    try {  
        testMethod3();  
    } catch (ArithmetricException ae) {  
        System.out.println("Dickory");  
    }  
}  
  
public static void testMethod3() throws ArithmetricException {  
    throw new ArithmetricException();  
    System.out.println("Hickory");  
}
```

What will be printed when the `testMethod1` is invoked (after allowing the code to run with

compilation warnings)?

- A. “Hickory Dickory Dock” is printed
- B. “Dickory” is printed
- C. “Dock” is printed

6. Given:

```
public static void test() throws FileNotFoundException {  
    try {  
        throw FileNotFoundException();  
    } finally {  
    }  
}
```

Determine why it will not compile. Which statement is correct?

- A. The code will not compile without a catch clause.
- B. The code needs the new keyword after the throw keyword.
- C. The finally clause should be the final clause.
- D. There is no class called FileNotFoundException.

7. Can a method throw more than one exception?

- A. Yes, a method can throw more than one exception.
- B. No, a method cannot throw more than one exception.

Alter the Program Flow

8. What new features came with Java 7 to enhance exception handling capabilities? (Choose all that apply.)

- A. The multi-catch feature
- B. The boomerang feature
- C. The try-with-resources feature
- D. The try-with-riches feature

9. Given:

```
try {  
    throw new IIOException();  
} catch (IIOException iioe) {  
} catch (IOException ioe) {  
} catch (Exception e) {  
} finally {  
}
```

This code will not compile. Why?

- A. Although it's a best practice that exception classes have a no argument constructor, this isn't always followed, as in the case of the IIOException class that does not have a no argument constructor.
- B. The exceptions listed in the catch blocks should be in the reverse order. The catch blocks should be ordered like so: Exception, followed by IOException, followed by IIOException.
- C. The finally block must include statements.
- D. The throws keyword should be used in place of the throw keyword.

Recognize Common Exceptions

10. What do the `InternalError`, `OutOfMemoryError`, `StackOverflowError`, and `UnknownError` classes have in common? (Choose all that apply.)
- A. They are all subclasses to the `VirtualMachineError` class.
 - B. They all have a no argument constructor.
 - C. They all have a constructor that accepts a single `String` argument.
 - D. They are all subclasses to the `RuntimeException` class.
 - E. All of the above
11. The `bytesTransferred` field of which checked exception provides information on how many bytes were transferred successfully before a disruption occurred?
- A. `IOException`
 - B. `InterruptedIOException`
 - C. `IntrospectionException`
 - D. `TimeoutException`
12. Given:

```
String typeOfDog = "Mini Australian Shepherd";
typeOfDog = null;
System.out.println(typeOfDog.length);
```

Which of the following is true?

- A. A `NullPointerException` will be thrown.
- B. A `RuntimeException` will be thrown.
- C. An `IllegalArgumentException` will be thrown.
- D. A compilation error will occur.

SELF TEST ANSWERS

Understand the Rationale and Types of Exceptions

1. Which class has the fewest subclasses: Exception, RuntimeException, or Error? Select the correct statement.

- A. The Exception class has fewer subclasses than the RuntimeException and Error classes.
- B. The RuntimeException class has fewer subclasses than the Exception and Error classes.
- C. The Error class has fewer subclasses than the Exception and RuntimeException classes.

Answer:

- C. The Error class has fewer subclasses than the Exception and RuntimeException classes.
- A and B are incorrect. A is incorrect because the Exception class has the most subclasses compared to the RuntimeException class and the Error class. Remember that RuntimeException is a subclass of the Exception class. B is incorrect because the RuntimeException class has more subclasses than the Error classes.

2. Of the following types of exceptions, which will an IDE help you in catching, if it is not handled in your code?

- A. Checked exceptions.
- B. Unchecked exceptions.

Answer:

- A. An IDE will provide hints to help you catch checked exceptions that are not handled in your code.
- B is incorrect. An IDE will not provide hints to catch unchecked exceptions that are not handled in your code, as unchecked exceptions are not required to be caught.

3. Which statement about the Throwable class is not correct?

- A. The Throwable class extends the Object class.
- B. The Throwable class implements the Serializable interface.
- C. Direct subclasses to the Throwable class include the RuntimeException and Error classes.
- D. The Throwable class is in the java.lang package.

Answer:

- C. Direct subclasses to the Throwable class include the Exception and Error classes. The RuntimeException class is a subclass, but not a *direct* subclass.
- A, B, and D are incorrect. A is incorrect because the statement is correct; the Throwable class does extend the Object class. B is incorrect because the statement is correct; the Throwable class does implement the Serializable interface. D is incorrect because the statement is correct; the Throwable class is in the java.lang package.

Understand the Nature of Exceptions

4. Which of the following classes can be thrown? (Choose all that apply.)

- A. throw new Error();

- B. throw new RuntimeException();
 - C. throw new Exception();
 - D. throw new Assertion();
 - E. throw new Throwable();
-

Answer:

- A, B, C, and E are correct. The Error, RuntimeException, Exception, and Throwable classes can all be thrown.
 - D is incorrect. There is no Assertion class in Java. Though, there is an AssertionError class in Java that may be thrown.
-

5. Given:

```
public static void testMethod1() {  
    try {  
        testMethod2();  
    } catch (ArithmetricException ae) {  
        System.out.println("Dock");  
    }  
}  
  
public static void testMethod2() throws ArithmetricException {  
    try {  
        testMethod3();  
    } catch (ArithmetricException ae) {  
        System.out.println("Dickory");  
    }  
}  
  
public static void testMethod3() throws ArithmetricException {  
    throw new ArithmetricException();  
    System.out.println("Hickory");  
}
```

What will be printed when the testMethod1 is invoked (after allowing the code to run with compilation warnings)?

- A. “Hickory Dickory Dock” is printed
 - B. “Dickory” is printed
 - C. “Dock” is printed
-

Answer:

- B. “Dickory” is printed.
 - A and C are incorrect. A is incorrect because the “Hickory” statement is never reached and an exception is not thrown to the testMethod2 invocation. C is incorrect because an exception is not thrown to the testMethod2 invocation, which would have been necessary to print out “Dock”.
-

6. Given:

```
public static void test() throws FileNotFoundException {
    try {
        throw FileNotFoundException();
    } finally {
    }
}
```

Determine why it will not compile. Which statement is correct?

- A. The code will not compile without a catch clause.
- B. The code needs the new keyword after the throw keyword.
- C. The finally clause should be the final clause.
- D. There is no class called FileNotFoundException.

Answer:

- B. The code needs the new keyword after the throw keyword in this example.
- A, C, and D are incorrect. A is incorrect because a catch clause is not needed if a finally clause is provided. C is incorrect because using finally instead of final is correct. D is incorrect because there is an exception class called FileNotFoundException.

7. Can a method throw more than one exception?

- A. Yes, a method can throw more than one exception.
- B. No, a method cannot throw more than one exception.

Answer:

- A. There are no restrictions on how many different types of exceptions that a method can throw.
- B is incorrect because methods can throw one or more exceptions.

Alter the Program Flow

8. What new features came with Java 7 to enhance exception handling capabilities? (Choose all that apply.)

- A. The multi-catch feature
- B. The boomerang feature
- C. The try-with-resources feature
- D. The try-with-riches feature

Answer:

- A and C. Java 7 introduced the multi-catch and try-with-resources features.
- B and D are incorrect. B is incorrect because there is no boomerang feature. D is incorrect because there is no try-with-riches feature.

9. Given:

```
try {
    throw new IIOException();
} catch (IIOException iioe) {
} catch (IOException ioe) {
} catch (Exception e) {
} finally {
}
```

This code will not compile. Why?

- A. Although it's a best practice that exception classes have a no argument constructor, this isn't always followed, as in the case of the `IIOException` class that does not have a no argument constructor.
- B. The exceptions listed in the catch blocks should be in the reverse order. The catch blocks should be ordered like so: `Exception`, followed by `IOException`, followed by `IIOException`.
- C. The `finally` block must include statements.
- D. The `throws` keyword should be used in place of the `throw` keyword.

Answer:

- A. Although it's a best practice that exception classes have a no argument constructor, this isn't always followed. In this case, the `IIOException` class does not have a no argument constructor and will fail compilation.
 - B, C, and D are incorrect. B is incorrect because the catch blocks are ordered appropriately, with the subclasses listed first in the appropriate hierarchical manner. C is incorrect because the `finally` block does not have to include any statements—that is, it can remain empty. D is incorrect because the `throw` keyword is used appropriately.
-

Recognize Common Exceptions

- 10.** What do the `InternalError`, `OutOfMemoryError`, `StackOverflowError`, and `UnknownError` classes have in common? (Choose all that apply.)
- A. They are all subclasses to the `VirtualMachineError` class.
 - B. They all have a no argument constructor.
 - C. They all have a constructor that accepts a single `String` argument.
 - D. They are all subclasses to the `RuntimeException` class.
 - E. All of the above

Answer:

- A, B, and C. A is correct because `InternalError`, `OutOfMemoryError`, `StackOverflowError`, and `UnknownError` are all subclasses to the `VirtualMachineError` class. B is correct because `InternalError`, `OutOfMemoryError`, `StackOverflowError`, and `UnknownError` all have a no argument constructor. C is correct because `InternalError`, `OutOfMemoryError`, `StackOverflowError`, and `UnknownError` all have a constructor that accepts a single `String` argument.
 - D and E are incorrect. D is incorrect because `InternalError`, `OutOfMemoryError`, `StackOverflowError`, and `UnknownError` are not subclasses to the `RuntimeException` class. E is incorrect because all of the answers are not correct.
-

- 11.** The bytesTransferred field of which checked exception provides information on how many bytes were transferred successfully before a disruption occurred?
- A. IOException
 - B. InterruptedIOException
 - C. IntrospectionException
 - D. TimeoutException
-

Answer:

- B.** The InterruptedIOException class includes a bytesTransferred field that provides information on how many bytes were transferred successfully before the interruption occurred.
 - A, C, and D** are incorrect. **A** is incorrect because IOException does not have a bytesTransferred field. **C** is incorrect because IntrospectionException does not have a bytesTransferred field. **D** is incorrect because the TimeoutException does not have a bytesTransferred field.
-

- 12.** Given:

```
String typeOfDog = "Mini Australian Shepherd";
typeOfDog = null;
System.out.println(typeOfDog.length);
```

Which of the following is true?

- A. A NullPointerException will be thrown.
 - B. A RuntimeException will be thrown.
 - C. An IllegalArgumentException will be thrown.
 - D. A compilation error will occur.
-

Answer:

- D.** A compilation error will occur. To make use of the length method of the String class, parenthesis must be used. The statement should have read, System.out.println(typeOfDog.length()); and not System.out.println(typeOfDog.length);.
 - A, B, and C** are incorrect. **A** is incorrect because a NullPointerException will not be thrown. Note that the statement wouldn't have been incorrect if the length method were used properly. **B** is incorrect because a RuntimeException will not be thrown. **C** is incorrect because an IllegalArgumentException will not be thrown.
-



10

Working with Classes and Their Relationships

CHAPTER OBJECTIVES

- Understand Class Compositions and Associations
- Class Compositions and Associations in Practice
- ✓ Two-Minute Drill

Q&A Self Test

Object relationships are often described in terms of *composition* and *association*. The SCJA exam (predecessor to the OCA exam) included questions that required that you understand the relationship between objects and distinguish between a composition relationship and an association relationship. The OCA exam does not include specific objectives for this skill set, but it is implied that you understand class compositions and associations. As such, we provided this coverage as a bonus to aid in your exam preparation.

Understand Class Compositions and Associations

Composition and association are two general descriptions for object relationships. Composition and association encompass four specific types of relationship descriptors: direct association, composition association, aggregation association, and temporary association. These descriptors will be covered in

this chapter. By studying the four specific relationship types, you will have a greater understanding of the difference between composition and association. Multiplicities will also be discussed. Every object relationship has a multiplicity. The following topics will be covered in this chapter:

- Class compositions and associations
- Class relationships
- Multiplicities
- Association navigation

Class Compositions and Associations

Composition and *association* are the general terms used to describe a class relationship. An association or composition relationship is formed between two objects when one contains a reference to the other. The reference is often stored as an instance variable. The reference may be in one direction or bidirectional.

An association relationship is a relationship of two objects, where neither one directly depends on the other for its logical meaning. For example, suppose object A has an association relationship with object B. If this relationship was lost, both objects would still retain the same meaning they had before the relationship. These are considered weak relationships. Objects in an association relationship have no dependence on each other for the management of their life cycle. In other words, the existence of an object is not tied to the existence of the other object in the relationship. Another example would be a `CarFactory` object and a `CarFrame` object. The `CarFactory` object and `CarFrame` object have an association relationship. If this relationship no longer existed, each object could continue to make sense logically and would retain its original meaning on its own.

A composition relationship is stronger than an association relationship. Composition means that one object is composed of another. An object may be composed of one or multiple objects. If, for example, object A is composed of object B, object A depends on object B. This statement does not imply that object A is composed of only object B, however; in fact, object A may also be composed of other objects. If the relationship were lost between object A and object B, the logical meaning of both objects would be lost or significantly altered. In this example, object B—the inner object that object A is composed of—would depend on object A to manage its life cycle. The existence of object B is directly tied to the existence of object A, so when object A no longer exists, object B would also no longer exist. Object B would also become nonexistent if the relationship between the two objects were lost. Examples of objects that have a composition relationship tend to be more abstract than association relationships. Consider a `Car` object and `CarStatus` object, for example. The `Car` object is composed of the `CarStatus` object. Both objects depend on this relationship to define their meanings. The `CarStatus` object also depends on the `Car` object to maintain its life cycle. When the `Car` object no longer exists, the `CarStatus` object would also no longer exist.

SCENARIO & SOLUTION

| | |
|--|-------------|
| You have an object that controls the life cycle of another object. What term can be used to describe it? | Composition |
| You have an object that has a weak relationship with another object. What term can be used to describe it? | Association |
| You have an object that has a strong relationship with another object. What term can be used to describe it? | Composition |

In many cases, determining the type of relationship between two objects is not as clear as the

preceding examples. Some interpretation is needed to determine the relationship. A composition will always be responsible for an object's life cycle. Composition relationships also represent a stronger relationship compared to an association. Objects belonging to an association make more sense by themselves than objects of composition.

Class Relationships

This section will break down the four specific class relationships that are possible. Each represents a different way objects can have relationships. Association and composition relationships are not mutually exclusive. In fact, composition association is one of the detailed relationship types, and it is sometimes referred to as simply composition. The other three detailed relationship types, direct association, aggregation association, and temporary association, are often referred to as simply association. You'll need to have a strong understanding of these relationships.

The following topics will be covered in the next few sections:

- Direct association
- Composition association
- Aggregation association
- Temporary association

Direct Association

Direct association describes a “has a” relationship. This is a basic association that represents navigability. Direct association is a weak relationship and therefore can be generalized to an association. There is no life cycle responsibility and each object in the relationship can conceptually be independent. This tends to be the default association when nothing else can accurately describe the relationship.

If a truck was modeled as an object to create a `Truck` object, it may have a `Trailer` object that it has a direct association with. The `Truck` object and `Trailer` object are weakly associated, because each could be used without the other and still maintain its intended purpose. A `Truck` object does not need to have a `Trailer` object, and a `Trailer` object is not required for the construction of the `Truck` object. This direct association relationship is depicted in the Unified Modeling Language (UML) diagram in [Figure 10-1](#).

Composition Association

Composition associations are used to describe an object's relationship where one object is composed of one or more objects. A composition association is a strong relationship and will be generalized as a composition. The internal object makes conceptual sense only while stored in the containing object. This relationship represents ownership. In a composition association, object A is “composed of” object B. For example, a `Tire` object would be composed of a `RubberStrips` object. The `Tire` object requires a `RubberStrips` object; the `RubberStrips` object is not very useful by itself.

FIGURE 10-1 Direct association



FIGURE 10-2 Composition association



The containing object also has the responsibility of managing the life cycle of the internal object. It is possible for this object to pass the life cycle management to another object. Life cycle management means that the object composed of the second object, or the containing object, must maintain a reference to the inner object; otherwise the Java virtual machine will destroy it. If the containing object is destroyed, any objects that compose it will also be destroyed. This composition association relationship is depicted in the UML diagram in [Figure 10-2](#).

Aggregation Association

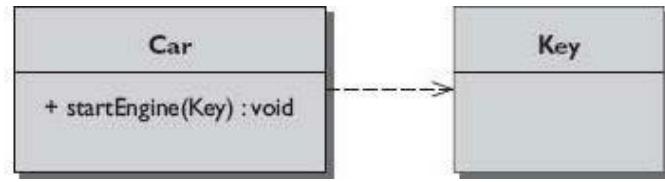
An aggregation association is a relationship that represents one object being part of another object. An aggregation association represents a “part of” the whole relationship. In this relationship, even though one object is a part of the other, each object can maintain its own meaning independently if the relationship is lost. Neither object depends on the other for its existence. The aggregation relationship does not require the object to perform life cycle management for the object that it references. Aggregation association is a weak relationship. It can be generalized as an association.

A `Motorcycle` object would have a `Windshield` object that it has an aggregation association with. The `Motorcycle` object and `Windshield` object are weakly associated, because each could be used without the other and still maintain its intended purpose. The `Windshield` object has a “part of” relationship with the `Motorcycle` object. This aggregation association relationship is depicted in the UML diagram in [Figure 10-3](#).

FIGURE 10-3 Aggregation association



FIGURE 10-4 Temporary association



Temporary Association

Temporary association is also known as a dependency. Typically, a temporary association will be an object used as a local variable, return value, or method parameter. It is considered a dependency because object A depends on object B as either an argument, a return value, or at some point a local variable. A temporary association is the weakest form of association. This relationship will not persist for the entire life cycle of the object.

For example, a `Car` object may have a method called `startEngine` that has a `Key` object as a parameter. The `Key` object as a parameter would represent a temporary association. This temporary association relationship is depicted in the UML diagram in [Figure 10-4](#).

Multiplicities

Every relationship has a multiplicity. *Multiplicity* refers to the number of objects that are part of a relationship. The three general classifications of multiplicity you should know are one-to-one, one-to-many, and many-to-many.

The following topics will be covered in the next few sections:

- One-to-one multiplicity
- One-to-many multiplicity
- Many-to-many multiplicity

on the job *Developers often just use the terms composition or association. When they say composition they are referring to composition association and a strong relationship. The term association is often used to refer to any of the other three weaker relationships: aggregation, direct, and temporary association.*

[Table 10-1](#) shows how each of the four specific relationship types relates to composition and association.

TABLE 10-1 Object Relationship Characteristics

| | Composition Association | Aggregation Association | Direct Association | Temporary Association |
|---|-------------------------|-------------------------|--------------------|-----------------------|
| General term is association | | ✓ | ✓ | ✓ |
| General term is composition | ✓ | | | |
| Strong relationship | ✓ | | | |
| Weak relationship | | ✓ | ✓ | ✓ |
| Has life cycle responsibility | ✓ | | | |
| Persists for most of the object's lifetime | ✓ | ✓ | ✓ | |
| Is used as a critical part of an object | ✓ | ✓ | | |
| Is often a local variable, return variable, or method parameter | | | | ✓ |

One-to-One Multiplicity

A one-to-one association is a basic relationship where one object contains a reference to another object. All four relationship types may have a one-to-one multiplicity. An example of a one-to-one relationship would be the `Motorcycle` object that has a relationship with a single `Engine` object.

One-to-Many Multiplicity

One-to-many relationships are created when one object contains a reference to a group of like objects. The multiple object references are normally stored in an array or a collection. All four relationship types may be one-to-many. The `Car` object can contain four `Tire` objects. This would be an aggregation association since the `Car` and `Tire` objects have a “part of” association between each other. The `Tire` objects can be stored in an array or collection. As the name implies, a one-to-many relationship may have many more than four objects. A many-to-one relationship is also possible.

Many-to-Many Multiplicity

Many-to-many relationships are possible only for aggregation associations, direct associations, and temporary associations. Composition association is a strong relationship that implies a life cycle responsibility for the object that composes it. If many objects have a relationship with an object, it is impossible for any individual object to control the life cycle of the other object in the relationship. If the car example were broadened to a traffic simulator application, it would include many other Car objects. Each of these Car objects contains references to many other TrafficLight objects. This represents a direct association since a single Car object does not maintain the life cycle of the TrafficLight objects. Each car object has a TrafficLight object. The relationship between a Car object and the TrafficLight object is weak. The TrafficLight objects are all shared between all of the car objects. A many-to-many association does not have to include an equal number of objects on each side of the relationship.

on the job Relationships (for example, aggregation and composition) and multiplicities can be easily depicted with UML. Drawing out class relationships can help you convey design concepts to your fellow employees. UML diagrams are covered in Appendix G.

Association Navigation

Association navigation is a term used to describe the direction in which a relationship can be traveled. An object that is contained within another object is said to be navigable if the containing object has methods for accessing the inner object. Most relationships are navigable in one direction, but if both objects contain references to the other, it is possible to have a bidirectional navigable relationship. Oftentimes, the methods for accessing inner objects are called *getters* and *setters*. A getter is a simple method that just returns an instance variable. A setter is a method that accepts an argument and uses it to set an instance variable.

Class Compositions and Associations in Practice

This section will review practical examples of association and composition relationships. The following examples and explanations should help provide a solid understanding of these concepts.

These topics are covered in the following sections:

- Examples of class association relationships
- Examples of class composition relationships
- Examples of association navigation

Examples of Class Association Relationships

This section will examine associations. The following three examples will demonstrate possible multiplicities of an aggregation association. An explanation will follow highlighting the important points from the example.

The topics covered include

- One-to-one class association
- One-to-many class association
- Many-to-many class association

One-to-One Class Association

The following example shows a `Truck` object and `Trailer` object. This is an example of a one-to-one direct association.

```
public class Truck {  
    /* This is an example of a one-to-one  
    direct association */  
    Trailer trailer;  
    void setTrailer(Trailer t){  
        trailer = t;  
    }  
    /*  
     * Remainder of Truck class would be here  
     */  
}
```

In this example of a one-to-one association, the `Truck` object contains a reference to the `Trailer` object. This is a one-to-one association because the variable `trailer` is a single variable. It is not part of an array or collection. This example is a direct association, because the `Truck` object is not responsible for the life cycle of the `trailer` variable. Another indication that it is a direct association is that, logically, the `Truck` object “has a” `Trailer` object.

In this example, and in most real-world situations, it is not always easy or even possible to determine if one object controls the life cycle of another. Oftentimes, you must make the best determination based on the information that is available. In this example, the `trailer` variable is being set by the method `setTrailer`. Since this method is used to set the variable, it can be assumed that other objects contain a reference to the `trailer` object and, therefore, no sole object is responsible for the life cycle of the object. Finally, since this was determined to be a direct association, the relationship can be generalized to just an association relationship.

One-to-Many Class Association

The next example will demonstrate an aggregation association. This example will have a relationship that is one-to-many. Here, `Wheel` objects are “part of” a `Car` object.

```
public class Car {  
    Wheel[] wheel = new Wheel[4];  
    void setWheels(Wheel w) {  
        wheel[0] = w;  
        wheel[1] = w;  
        wheel[2] = w;  
        wheel[3] = w;  
    }  
    // Remainder of Car class would be here  
}
```

This example has an array of four `Wheel` objects. Since there is one `Car` object that contains four `Wheel` objects, this relationship is one-to-many. With a one-to-many relationship, the multiple objects will normally be stored in an array or collection such as a `Vector`. This example is an aggregation association, because the `Wheel` object is “part of” the `Car` object. Because this is a weak relationship and there are no life cycle responsibilities, this can be generalized as an association.

Many-to-Many Class Association

The many-to-many relationship is more complex than the one-to-one and one-to-many relationships.

In this example, the relationship is between a group of car objects and a group of TrafficLight objects. The following is the code segment for the two objects:

```
// TrafficLight class
public class TrafficLight {
    int lightID;
    TrafficLight(int ID) {
        lightID = ID;
    }
}

// Car class
public class Car {
    TrafficLight[] allTrafficLights;
    Car(TrafficLight[] trafficLights) {
        allTrafficLights=trafficLights;
    }
}
```

This next segment is the code that creates both objects. This segment is important because it shows how the relationships are formed between the objects.

```
public class TrafficSimulator {
    Car[] cars = new Car[3];
    TrafficLight[] trafficLights = new TrafficLight[8];
    public static void main(String[] args) {
        new TrafficSimulator();
    }

    TrafficSimulator() {
        for (int i = 0; i < trafficLights.length; i++) {
            trafficLights[i] = new TrafficLight(i);
        }
        cars[0] = new Car(trafficLights);
        cars[1] = new Car(trafficLights);
        cars[2] = new Car(trafficLights);
    }
}
```

This segment contains a `main` method. The sole job of `main` is to create a new `TrafficSimulator` object. The `TrafficSimulator` object contains an array of `Car` objects and an array of `TrafficLight` objects. First, the `TrafficLight` objects are created. Each `TrafficLight` object stores a unique ID. Next, the `Car` objects are created. Each `Car` object contains an array of all the `TrafficLight` objects. This example is many-to-many, because each `Car` object contains the same group of multiple `TrafficLight` objects. This relationship can be classified as a direct association because the `Car` objects “has an” array of `TrafficLight` objects.

Examples of Class Composition Relationships

This section will be similar to the last section except that composition associations will be demonstrated. Composition associations have only two possible multiplicities. This section will show an example of each followed by an explanation.

One-to-One Class Composition

This example demonstrates a one-to-one composition relationship. This is a composition association

because that is the only type of association that can create a composition relationship.

```
public class Tire {  
    TireAirPressure tireAirPressure;  
    Tire(){  
        tireAirPressure = new TireAirPressure();  
    }  
}
```

In this example, the `Tire` object and the `TireAirPressure` object have a one-to-one relationship. The `Tire` object is “composed of” the `TireAirPressure` object. This represents a composition association. The relationship between the two objects is strong. The `Tire` object has life cycle management responsibilities to the `TireAirPressure` object. If the `Tire` object was destroyed, the `TireAirPressure` object would also be destroyed.

One-to-Many Class Composition

This final example demonstrates a composition relationship with a one-to-many multiplicity. The following code segment is of a `SensorStatus` class:

```
public class SensorStatus {  
    int status;  
    public SensorStatus(int newStatus) {  
        status = newStatus;  
    }  
}
```

The next segment demonstrates a `CarComputer` object that is “composed of” an array of five `SensorStatus` objects:

```
public class CarComputer {  
    SensorStatus[] sensorStatus = new SensorStatus[5];  
    public CarComputer() {  
        sensorStatus[0] = new SensorStatus(1);  
        sensorStatus[1] = new SensorStatus(1);  
        sensorStatus[2] = new SensorStatus(1);  
        sensorStatus[3] = new SensorStatus(1);  
        sensorStatus[4] = new SensorStatus(1);  
    }  
}
```

Because there is one `CarComputer` object and five `SensorStatus` objects, this represents a one-to-many relationship. The relationship is composition association. Again, notice how the relationship is strong, and that the `SensorStatus` array depends on the `CarComputer` object to manage its life cycle.

Examples of Association Navigation

Association navigation is the ability to navigate a relationship. The following example demonstrates a `PinStripe` object that is “composed of” a `Color` object:

```
public class PinStripe {  
    Color color = new Color(Color.blue);  
    Color getColor() {  
        return color;  
    }  
}
```

In this example, any object that had access to the `PinStripe` object could use its `getColor` method, which is considered a getter, to navigate to the `color` object. In this example, the navigation is only in a single direction.

CHAPTER SUMMARY

This chapter has discussed the different relationships that are possible among objects. Association and composition were the general description of relationships and are important for you to be familiar with.

Association is used to describe an object-to-object reference. This type of reference means that object A has a reference to object B and can access object A's public methods and member variables. Object B may or may not have a reference back to object A. A relationship of association means that both objects are independent and neither one relies on the other to maintain its existence. Direct association, aggregation association, and temporary association are all more detailed forms of association.

Composition relationships are a stronger form of association relationships. A composition relationship is a type of association where an object that is composed of another object is also responsible for the life cycle management of that object. A composition relationship may have one-to-one or one-to-many multiplicities. Composition association is an example of composition.

Next, this chapter covered each of the four possible relationships in detail. Direct association, aggregation association, and temporary association are three of the four relationship types. Each of these belongs in the general category of association. They imply no responsibility of life cycle management. Composition association belongs to the category of general composition. Composition association has a life cycle responsibility.

There may be three different multiplicities of relationships. A one-to-one relationship has one object that contains a reference to another object of a particular type. A one-to-many relationship has an object that contains an array of object references, or a collection such as an `ArrayList` or `Vector`. The final relationship is many-to-many. This relationship has many objects that contain a reference to the same collection or array of objects. The many-to-many relationship is unique for association and cannot exist for a composition relationship.

This chapter concludes with examples of each multiplicity for association and composition relationships. These examples are important for you to understand. In Appendix G of this book, these relationships will be revisited when UML modeling is discussed.



TWO-MINUTE DRILL

Understand Class Compositions and Associations

- Composition and association are both general descriptions for object-to-object relationships.
- A relationship is created when an object contains a reference to another object, often through an

instance variable.

- Direct association is a “has a” relationship.
- Direct association is a weak relationship.
- Direct association has no life cycle responsibilities.
- Direct association tends to be the default relationship if no other relationship seems to fit.
- Two objects that have a direct association will logically make sense if the relationship is lost.
- Composition association is a “composed of” relationship.
- Composition association is a strong association.
- Composition association has life cycle responsibilities.
- Composition association represents possession and ownership.
- Two objects that have a composition association will not logically make sense if the association is lost.
- When two objects have a composition association, the containing object often requires the inner object.
- Aggregation association is a “part of” relationship.
- Aggregation association is a weak relationship.
- Aggregation association has no life cycle responsibilities.
- Two objects that have an aggregation association will logically make sense if the relationship is lost.
- Temporary association is also known as a dependency.
- Temporary association is a weak relationship.
- Temporary association has no life cycle responsibilities.
- A temporary association relationship is created when a return value, method parameter, or local variable is used.
- One-to-one relationships are possible with both composition and association.
- One-to-one relationships have one object that contains a reference to another object.
- One-to-many relationships are possible with both composition and association.
- One-to-many relationships exist when one object contains a reference to an array or collection of similar objects.
- Many-to-many relationships are possible only with association.
- Many-to-many relationships have many similar objects that contain a reference to the same array or collection of objects.
- Association navigation is a term used to describe the ability to access an object that is contained in another object.
- Relationships may be able to navigate bidirectionally or unidirectionally.
- Getter methods are often used to navigate an inner object.

Class Compositions and Associations in Practice

- In an association, the inner object normally is not created in the containing object but is instead passed to the containing object as a method argument.
- In a composition, the inner object is normally created in the containing object.
- In a one-to-many relationship, the inner object is stored in an array or collection.
- A many-to-many relationship exists when objects in an array or collection each contain a reference to another array or collection.
- A many-to-many relationship can exist only for associations.

SELF TEST

Understand Class Compositions and Associations

- 1.** What associations are considered weak relationships? (Choose all that apply.)

 - A. Direct association
 - B. Temporary association
 - C. Composition association
 - D. Aggregation association
- 2.** What associations are considered strong relationships? (Choose all that apply.)

 - A. Direct association
 - B. Temporary association
 - C. Composition association
 - D. Aggregation association
- 3.** Which association can be described as “object A ‘has an’ object B?”

 - A. Direct association
 - B. Temporary association
 - C. Composition association
 - D. Aggregation association
- 4.** Which association can be described as “object A is ‘part of’ object B”?

 - A. Direct association
 - B. Temporary association
 - C. Composition association
 - D. Aggregation association
- 5.** Which association can be described as “object A is ‘composed of’ object B”?

 - A. Direct association
 - B. Temporary association
 - C. Composition association
 - D. Aggregation association
- 6.** Which association has a life cycle responsibility for the object it contains?

 - A. Direct association
 - B. Temporary association
 - C. Composition association
 - D. Aggregation association
- 7.** Association navigation is best described as which of the following?

 - A. The ability to navigate, or access, an object that is contained in another object
 - B. The ability to search for and find an object that is contained in another object
 - C. The possibility of passing an object to another object via a method
 - D. The ability to invoke methods of an object that will then change the path of code execution
 - E. The ability to invoke methods of an object to determine the current path of execution
- 8.** What would the multiplicity be in the following relationship? A `Lamp` object “has a” `LightBulb` object.

 - A. One-to-one
 - B. One-to-many
 - C. Many-to-many
- 9.** A composition association cannot exist in what multiplicity?

- A. One-to-one
- B. One-to-many
- C. Many-to-many

10. What would the multiplicity be in the following relationship? A `Bookshelf` object “has a” reference to an array made up of `Book` objects.

- A. One-to-one
- B. One-to-many
- C. Many-to-many

Class Compositions and Associations in Practice

Use the following code example for the next four questions:

```
public class Client {  
    Address address;  
    AccountNum[] accountNums;  
    void setAddress(Address newAddress) {  
        address = newAddress;  
    }  
    public Client() {  
        accountNums = new AccountNum[2];  
        accountNums[0] = new AccountNum();  
        accountNums[1] = new AccountNum();  
    }  
}
```

11. In the preceding code segment, what is the relationship of the `Client` object and the `address` variable?

- A. Direct association
- B. Temporary association
- C. Composition association
- D. Aggregation association

12. In the preceding code segment, what is the relationship of the `Client` object and the `accountNums` variable?

- A. Direct association
- B. Temporary association
- C. Composition association
- D. Aggregation association

13. In the preceding code segment, what is the multiplicity between the `Client` object and the `address` variable?

- A. One-to-one
- B. One-to-many
- C. Many-to-many

14. In the preceding code segment, what is the multiplicity between the `Client` object and the `accountNums` variable?

- A. One-to-one
- B. One-to-many
- C. Many-to-many

15. Which of the following statements are true? (Choose all that apply.)

- A. Association navigation can be quad-directional.
- B. Association navigation can be bidirectional.
- C. Association navigation can have no direction.
- D. Association navigation can be unidirectional.

SELF TEST ANSWERS

Understand Class Compositions and Associations

1. What associations are considered weak relationships? (Choose all that apply.)

- A. Direct association
 - B. Temporary association
 - C. Composition association
 - D. Aggregation association
-

Answer:

A, B, and D. Each of these associations is considered weak. This means that they do not have any life cycle responsibility and that if the relationship was lost, each object would still maintain its meaning.

C is incorrect. This is an example of composition and has a strong relationship.

2. What associations are considered strong relationships? (Choose all that apply.)

- A. Direct association
 - B. Temporary association
 - C. Composition association
 - D. Aggregation association
-

Answer:

C. This association is considered strong. It does have a life cycle responsibility and if the relationship was lost, each object would lose some or all of its meaning.

A, B, and D are incorrect. They are examples of associations that have weak relationships.

3. Which association can be described as “object A ‘has an’ object B”?

- A. Direct association
 - B. Temporary association
 - C. Composition association
 - D. Aggregation association
-

Answer:

A. Direct association is a “has a” relationship.

B, C, and D are incorrect. **B** is incorrect because temporary association is described as a dependency. **C** is incorrect because composition association is a “composed of” relationship. **D** is incorrect because aggregation association is a “part of” relationship.

4. Which association can be described as “object A is ‘part of’ object B”?

- A. Direct association
 - B. Temporary association
 - C. Composition association
 - D. Aggregation association
-

Answer:

D. Aggregation association is a “part of” relationship. One object will be used to make up

another object. However, neither object depends on the other for its existence and meaning.

A, B, and C are incorrect. **A** is incorrect because direct association is a “has a” relationship. **B** is incorrect because temporary association is described as a dependency. **C** is incorrect because composition association is a “composed of” relationship.

5. Which association can be described as “object A is ‘composed of’ object B”?

- A. Direct association
- B. Temporary association
- C. Composition association
- D. Aggregation association

Answer:

C. Composition association is a “composed of” relationship. One object will be used to make up another object. If this relationship was lost, the meaning of the objects would also change. This is a strong relationship and has a life cycle responsibility for the inner object.

A, B, and D are incorrect. **A** is incorrect because direct association is a “has a” relationship. **B** is incorrect because temporary association is described as a dependency. **D** is incorrect because aggregation association is a “part of” relationship.

6. Which association has a life cycle responsibility for the object it contains?

- A. Direct association
- B. Temporary association
- C. Composition association
- D. Aggregation association

Answer:

C. Composition association has the responsibility to maintain the life cycle of the object that it is composed of.

A, B, and D are incorrect. They are all weak relationships that have no life cycle responsibility.

7. Association navigation is best described as which of the following?

- A. The ability to navigate, or access, an object that is contained in another object
- B. The ability to search for and find an object that is contained in another object
- C. The possibility of passing an object to another object via a method
- D. The ability to invoke methods of an object that will then change the path of code execution
- E. The ability to invoke methods of an object to determine the current path of execution

Answer:

A. Association navigation is the ability to access an object that is contained in another.

B, C, D, and E are incorrect.

8. What would the multiplicity be in the following relationship? A `Lamp` object “has a” `LightBulb` object.

- A. One-to-one
- B. One-to-many

C. Many-to-many

Answer:

- A. There is one Lamp object and one LightBulb object; therefore, this is one-to-one.
 - B and C are incorrect. Neither side of this relationship has more than one object.
-

9. A composition association cannot exist in what multiplicity?

- A. One-to-one
 - B. One-to-many
 - C. Many-to-many
-

Answer:

- C. Composition association requires that it have responsibility for the life cycle of the objects it is composed of. It is impossible to have this responsibility in a many-to-many relationship because many objects would contain references to all of the objects.
 - A and B are incorrect. Composition can exist for cases since the one object can be composed of one or more objects.
-

10. What would the multiplicity be in the following relationship? A BookShelf object “has a” reference to an array made up of Book objects.

- A. One-to-one
 - B. One-to-many
 - C. Many-to-many
-

Answer:

- B. BookShelf is a single object and contains a reference to an array of Book objects. The key to this question is the fact that you are dealing with an array. This means there are many Book objects; therefore, it is a one-to-many relationship.
 - A and C are incorrect. A is incorrect because there are many Book objects. C is incorrect because there is only one BookShelf object.
-

Class Compositions and Associations in Practice

Use the following code example for the next four questions:

```
public class Client {  
    Address address;  
    AccountNum[] accountNums;  
    void setAddress(Address newAddress) {  
        address = newAddress;  
    }  
    public Client() {  
        accountNums = new AccountNum[2];  
        accountNums[0] = new AccountNum();  
        accountNums[1] = new AccountNum();  
    }  
}
```

11. In the preceding code segment, what is the relationship of the Client object and the address

variable?

- A. Direct association
 - B. Temporary association
 - C. Composition association
 - D. Aggregation association
-

Answer:

- A. Direct association is the best answer, because logically a `client` object “has an” `Address` object.
 - B, C, and D are incorrect.
-

12. In the preceding code segment, what is the relationship of the `client` object and the `accountNums` variable?

- A. Direct association
 - B. Temporary association
 - C. Composition association
 - D. Aggregation association
-

Answer:

- C. This is a composition association because the `client` object is “composed of” the `AccountNum` objects. This is a strong relationship since the `client` object maintains the life cycle of the `AccountNum` objects.
 - A, B, and D are incorrect.
-

13. In the preceding code segment, what is the multiplicity between the `client` object and the `address` variable?

- A. One-to-one
 - B. One-to-many
 - C. Many-to-many
-

Answer:

- A is the correct answer. Since there are no arrays or collections involved with either the `client` object or `Address` object, this must be one-to-one.
 - B and C are incorrect. **B** is incorrect because the `address` variable is a single variable and therefore does not qualify as many. **C** is incorrect because both the `client` object and `address` variable are not considered to be many since there is only one of each.
-

14. In the preceding code segment, what is the multiplicity between the `client` object and the `accountNums` variable?

- A. One-to-one
 - B. One-to-many
 - C. Many-to-many
-

Answer:

- B. The array of `AccountNum` objects should be a giveaway that this is one-to-many.
- A and C are incorrect. **A** is incorrect because the `accountNums` variable is an array and

therefore must be many. **C** is incorrect because there is only one client object.

15. Which of the following statements are true? (Choose all that apply.)

- A.** Association navigation can be quad-directional.
 - B.** Association navigation can be bidirectional.
 - C.** Association navigation can have no direction.
 - D.** Association navigation can be unidirectional.
-

Answer:

- B** and **D**. A relationship can only be bidirectional or unidirectional.
 - A** and **C** are incorrect. **A** is incorrect because quad-directional relationships do not exist. **B** is incorrect because it is impossible to have a relationship and not have a direction.
-



A

Java Platforms

Java-based technologies provide the components and means necessary for creating client, client-server, enterprise, and mobile applications.

The Java Standard Edition (SE) platform is designed primarily for client-side solutions. The Java Micro Edition (ME) platform is designed for embedded solutions. The Java Enterprise Edition (EE) platform is designed for enterprise solutions. To pass the OCA Java Associate SE 7 Programmer (OCA) exam, you'll need to have a general understanding of the Java SE platform only. Java ME and Java EE are presented here for your information only and you won't need to know anything about them for the exam. However, this appendix will help provide a better overall understanding of the Java ecosystem.

Let's start with a brief case study. Suppose you've spent several years working with teams of people on several projects. However, you've recently started a new job as the IT director for a small firm. You show up on the first day to find out that not only are you the IT director, but you will be playing the role of senior architect, designer, and developer, while managing a small group of junior programmers. When you meet the owner of the firm, he says, "There are a few improvements that I've wanted to add to our Java-based security system for a while now. There have been a lot of break-ins in the area in the last couple of months, so I need a system that is more informative. So here are the requirements. See if you guys can have the improved system up and running in 30 days." He hands you a small piece of paper with the following information:

- Requirements for alarm system improvements:
 - Must be able to record all security events into a database.

- Must be able to administer the system with a local application.
- Must automatically receive alarm notification via e-mail.
- Must be able to disable or enable the security system from my cell phone.
- Must be able to view the audit log from my home computer.
- Must be able to authenticate remote logins from a preexisting naming and directory service.

If you were unfamiliar with Java technologies, you would look at this list and think there is no way this can be done in such a small amount of time. However, if you are Java-technology savvy, you would quickly be able to associate a technology with each requirement. You would then be able to work up a quick architecture and delegate pieces of the assignment to different task members.

Let's take a look at how you might divide things up:

■ Alarm system improvements, forward plan:

- Task programmer A with building a SQL database and interface code with the Java Naming and Directory Interface (JNDI) API of the Java SE platform.
- Task programmer B with building a client application administration tool using JavaFX.
- Task programmer C with writing a notification module using the JavaMail API of the Java EE platform.
- Task programmer D with building a cell phone application to enable or disable the alarm system using the Java ME platform.
- Task programmer E with building a web-based application to log in and view the audit log using JavaServer Faces (JSF)/Facelets of the Java EE platform.
- Task programmer F with authenticating remote logins from a preexisting naming and directory service using the JNDI API of the Java SE platform.
- Task yourself with integrating all of the components together as the developers return them to you.

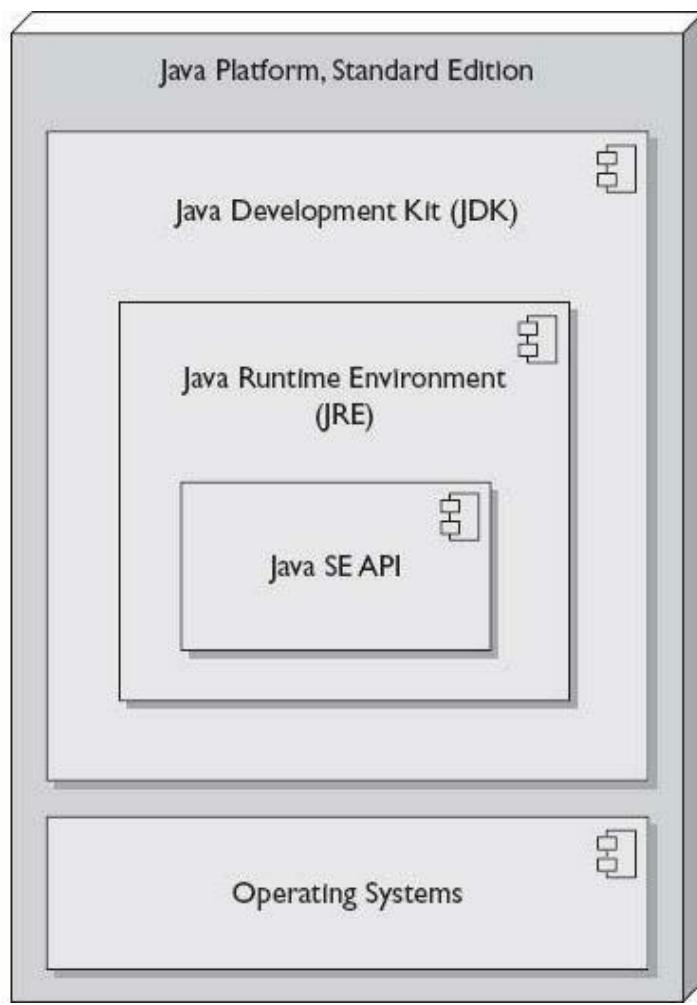
So now that you know the concept of selecting Java-based technologies for real-world solutions, you need to know exactly what the technologies are and when and where you would use them. This appendix details the platforms that house the technologies; the following sections will detail the technologies themselves:

- Java Platform, Standard Edition
- Java Platform, Micro Edition
- Java Platform, Enterprise Edition

Java Platform, Standard Edition

The Java SE is composed of the rich Java SE API, the Java Runtime Environment, the Java Development Kit, and the underlying operating system, as shown in [Figure A-1](#). Each of these elements serves its own specific purpose. Collectively, these components provide the means to develop, deploy, and run client and/or client-server-based systems.

FIGURE A-1 The Java SE platform



The Java SE API

The Java SE API is a collection of software packages. Each package houses a related set of classes and interfaces. You should have a general knowledge of the packages represented in [Table A-1](#). Many of these packages have subpackages that are not listed.

TABLE A-1 Commonly Used Java Packages

| Name | Description | Package Name |
|---|---|------------------------|
| Java Abstract Window Toolkit API | Provides native GUI functionality and an event-handling model | java.awt |
| Java Basic I/O API | Provides general input/output functionality | java.io |
| Java Database Connectivity API | Provides universal data access | java.sql, javax.sql |
| Java Core Language | Provides core Java language classes and interfaces | java.lang |
| Java Naming and Directory Interface API | Provides naming and directories services access | javax.naming |
| Java Networking API | Provides general networking functionality | java.net, javax.net |
| Java RMI API | Provides Remote Method Invocation functionality | java.rmi |
| Java Swing API | Provides GUI building functionality, largely superseding the AWT API | javax.swing |
| JavaFX API | Provides lightweight, hardware-accelerated UI functionality, superseding the AWT and Swing APIs | javafx.[packages] |
| Java Utilities API | Provides general utilities including the collections framework, event models, and time facilities | java.util |
| Extensible Markup Language (XML) API | Provides support for XML | java.xml |
| Web Services API | Provides support for JAX-WS | java.xml.ws |

The Java Runtime Environment

The Java Runtime Environment (JRE) is the set of software that allows Java applications to run. The JRE includes the following items:

- Java virtual machines (JVMs)

- Java HotSpot Client

- Java HotSpot Server

- Deployment technologies

- Java Plug-in

- Java Web Start Technology

- Java Control Panel

- Java Update Mechanism

Java is compiled into bytecode, and each operating system has its own JVM that will run that bytecode. This is what gives Java the “WORA” capability—WORA is the acronym for “Write once, run anywhere.”

The Java Development Kit

The Java Development Kit (JDK) is, in essence, a developer’s toolbox. Not only does it contain the JRE and the Java SE API, but it also contains all of the utilities you will need to compile, test, and debug your applications. [Table A-2](#) lists some of the common tools in the JDK that you will be using as a developer. For more information on the Java compiler and interpreter, see [Chapter 1](#).

TABLE A-2 Common Development Tools Used as Part of the JDK

| JDK Tool | Description |
|----------|--|
| jar | The Java archiving tool |
| java | The Java interpreter |
| javac | The Java compiler |
| javadoc | The API documentation generator |
| javap | The class file disassembler |
| javaw | Java interpreter without an associated console window |
| jconsole | The monitoring and management utility released with Java 5.0 |
| jdb | The Java debugger |
| keytool | Key and certificate management tool |
| rmiic | The Remote Method Invocation (RMI) stub and skeleton generator |
| xjc | Java Architecture for XML Binding compiler |

When you install a version of the JDK to your computer, you will want to take note of its location. This is important since you may need to specify its location in your system's path or point to it with your IDE.

Supported Operating Systems

Oracle directly supports the Solaris, Linux, and Microsoft Windows operating systems with fully compliant JVMs, JDKs, and JREs. You can get the latest JRE and JDK here:

www.oracle.com/technetwork/java/javase/downloads/. Legacy versions are kept in Oracle's archives here: www.oracle.com/technetwork/java/archive-139210.html. Third-party sources also make JREs and JDKs available for other operating systems. Many of these third-party solutions are listed and linked here: <http://java-virtual-machine.net/other.html>.

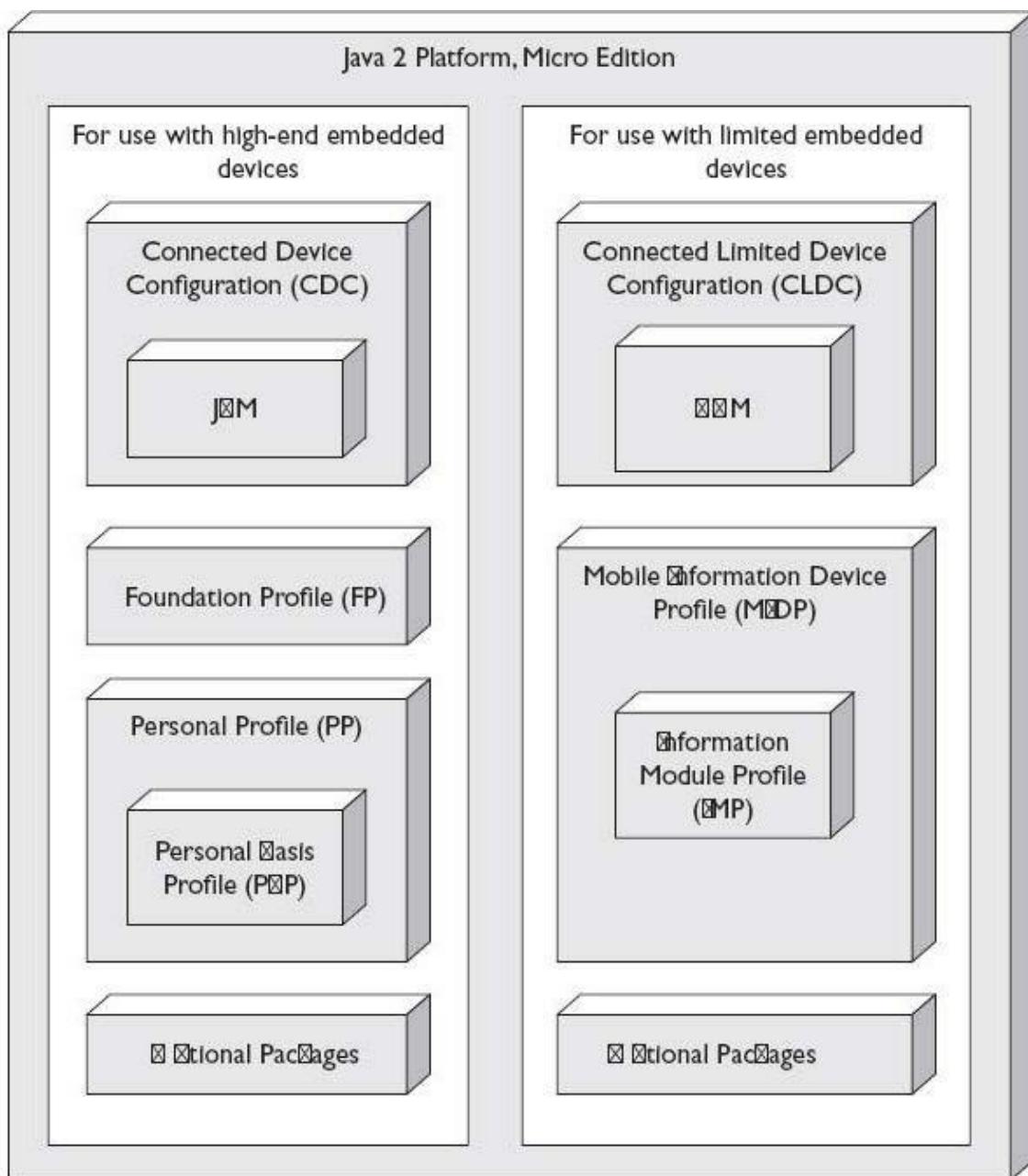
The OpenJDK–Mac OS X Port Project maintains a goal to produce an open source version of the JDK for the Mac; see <http://openjdk.java.net/projects/macosx-port/>.

Oracle provides support for the embedded market for the small footprint ARM, Power Architecture, and x86 architectures under a commercial license.

Java 2 Platform, Micro Edition

The Java ME is designed for embedded devices such as high-end PDAs and mobile phones. Java ME's architecture is based on configurations, profiles, and optional packages, as shown in [Figure A-2](#). Java ME is not on the exam however, Java Card (www.oracle.com/technetwork/java/javacard/overview/) and Java TV (www.oracle.com/technetwork/java/javame/javatv/overview) are current Java ME technologies worth investigating.

FIGURE A-2 The J2ME 1.4



Configurations

Java ME has two configurations: the Connected Device Configuration (CDC) and the Connected Limited Device Configuration (CLDC). These configurations contain platform a virtual machine containing a small but focused set of libraries making up the runtime environment. CDC is supplied with the standard JVM and is utilized for devices that do not have extreme constraints of resources. CLDC has a small compact virtual machine known as Oracle's K virtual machine (KVM) and a reduced set of class libraries.

Profiles

Profiles are required to work in conjunction with configurations as part of the necessary runtime environment. They are APIs that define the application's life-cycle model, user interface, and device properties access. CDC contains the Foundation Profile (FP), the Personal Profile (PP), and the Personal Basis Profile (PBP). PBP is a subset of PP. The CLDC contains the Mobile Information Device Profile (MIDP) and the Information Module Profile (IMP). IMP is a subset of MIDP.

Optional Packages

As discussed in detail in [Chapter 1](#), packages are collections of related classes and functionality. Additional packages can be included as needed to expand on Java ME functionality. The packages that can be optionally used are initially excluded (not included by default) to keep the Java ME footprint as small as possible.

Squawk

Squawk is a Java-compliant and CLDC-compatible virtual machine implementation, making it a piece of the Java ME architecture. Where most JVMs are written in C and C++, the majority of the Squawk JVM is pure Java. Squawk was designed to be as light as possible and is used with Oracle's wireless Small Programmable Object Technology kits (Sun SPOTs). These "hobby" kits, which include a 3-D accelerometer, temperature and light sensors, LCDs, and push buttons, are designed to encourage research and development of mobile technologies. The Squawk JVM is not on the exam, but you can find out more about the Sun SPOT project at www.sunspotworld.com.

Java Platform, Enterprise Edition

Java EE provides a means to create true enterprise systems that are flexible, scalable, and secure. A major benefit of enterprise systems is the separation of software components. Java EE follows the model-view-controller (MVC) architecture, where servlets work as the controller, JavaServer Pages (JSP) handle the view or presentation logic, and the business logic is represented as the model, typically the Enterprise JavaBeans (EJBs). Java EE requires a collection of optional packages that support each of these areas, as well as complementary technologies. The packages are actually implementations of specifications. [Table A-3](#) depicts the specifications of the Java EE 6 platform.

TABLE A-3 Java EE 6 Technology JSRs

| JSR | Specification Name | Abbreviation | Version |
|--------------------------------------|---|--------------|---------|
| Web Services Technologies | | | |
| JSR-67 | Java APIs for XML Messaging 1.3 | SAAJ | 1.3 |
| JSR-93 | Java API for XML Registries (JAXR) 1.0 | JAXR | 1.0 |
| JSR-101 | Java API for XML-Based RPC | JAX-RPC | 1.1 |
| JSR-109 | Implementing Enterprise Web Services 1.3 | N/A | 1.3 |
| JSR-181 | Web Service Metadata for the Java Platform | N/A | 2.0 |
| JSR-222 | Java Architecture for XML Binding (JAXB) 2.2 | JAXB | 2.2 |
| JSR-224 | Java API for XML-Based Web Services (JAX-WS) 2.2 | JAX-WS | 2.2 |
| JSR-311 | Java API for RESTful Web Services (JAX-RS) 1.1 | JAX-RS | 1.1 |
| Web Application Technologies | | | |
| JSR-45 | Debugging Support for Other Languages 1.0 | N/A | 1.0 |
| JSR-52 | A Standard Tag Library for JavaServer Pages | JSTL | 1.2 |
| JSR-315 | Java Servlet 3.0 | Servlets | 3.0 |
| JSR-314 | JavaServer Faces 2.0 | JSF | 2.0 |
| JSR-245 | JavaServer Pages 2.2/Expression Language 2.2 | JSP | 2.2 |
| Enterprise Application Technologies | | | |
| JSR-250 | Common Annotations for the Java Platform | N/A | 1.0 |
| JSR-299 | Context and Dependency Injection for Java (Web Beans 1.0) | Web Beans | 1.0 |
| JSR-303 | Bean Validation 1.0 | N/A | 1.0 |
| JSR-318 | Enterprise JavaBeans 3.1 | EJB | 3.1 |
| JSR-322 | Java EE Connector Architecture 1.6 | N/A | 1.6 |
| JSR-330 | Dependency Injection for Java 1.0 | N/A | 1.0 |
| JSR-907 | Java Transaction API (JTA) | JTA | 1.0 |
| JSR-914 | Java Message Service (JMS) API | JMS | 1.1 |
| JSR-919 | JavaMail 1.4 | N/A | 1.4.1 |
| JSR-925 | JavaBeans Activation Framework 1.1 | JAF | 1.1 |
| Management and Security Technologies | | | |
| JSR-77 | J2EE Management | N/A | 1.0 |
| JSR-88 | Java EE Application Deployment | N/A | 1.2 |
| JSR-115 | Java Authorization Contract for Containers | JACC | 1.1 |
| JSR-173 | Streaming API for XML (StAX) 1.0 | StAX | 1.0 |
| JSR-196 | Java Authentication Service Provider Interface for Containers | N/A | N/A |
| JSR-206 | Java API for XML Processing (JAXP) 1.3 | JAXP | 1.3 |
| JSR-221 | Java Database Connectivity 4.0 | JDBC | 4.0 |
| JSR-255 | Java Management Extensions (JMX) 2.0 | JMX | 2.0 |
| JSR-925 | JavaBeans Activation Framework (JAF) 1.1 | JAF | 1.1 |

A Java Specification Request (JSR) is the description of Java platform-related specifications—proposed and final. For more information on JSRs, visit the Java Community Process (JCP) home page: <http://jcp.org/en/home/index>. The JCP maintains the JSRs.

When developing Java EE systems, you will always need a JDK. Since the JDK is the main piece of the Java SE platform, you could essentially say that Java SE is part of Java EE. You will often have the option of using newer versions of the JDK with the Java EE implementation of your choice. Be aware that there is no direct correlation between the Java SE and Java EE version numbers. So you must check the documentation to see which versions of the JDK will work with your Java EE implementation. For a specific example of a case that would work, you could use Oracle's JDK 1.5 with Oracle's Application Server 10gR3, which is a J2EE 1.4 implementation.

Getting and staying involved in the Java community is an excellent way to keep up-to-date with the

latest Java Technologies. Consider joining a Java User Group (JUG) to share others' experiences and expertise. You can find out more about JUGs here: <http://home.java.net/jugs/java-user-groups>.

Online technology forums also offer a rich opportunity for acquiring Java knowledge. Consider frequenting Oracle Java forums: <https://forums.oracle.com/forums/category.jspa?categoryID=285>.



B

Java SE 7 Packages

Because Java is a programming language, the OCA exam focuses on many of the packages and classes within the core Java SE distributions. Java EE coverage was included in the prior SCJA exam but has been removed for the OCA. The tables provided in this appendix detail the full set of Java SE 7 packages.

You don't need to learn the low-level details of Java packages to perform well on the OCA exam. To a large extent, just knowing what the packages are designed for and what type of functionality they contain will help you achieve a high score.

Java SE 7 provides packages in the following areas:

- **Core packages** Language, utility, and base packages
- **Integration packages** Java Database Connectivity (JDBC), Java Naming and Directory Interface (JNDI), Remote Method Invocation (RMI)/RMI-Internet Inter-Orb Protocol (IIOP) and scripting packages
- **User interface packages** Swing API, Abstract Window Toolkit (AWT) API, sound, image I/O, printing packages
- **Security packages** Security and cryptography packages
- **XML-based packages** XML-based packages, web services

*The documentation for the OMG packages that are part of the Java SE 7 distribution are not listed in this appendix but can be viewed in the Java SE 7 API documentation:
<http://docs.oracle.com/javase/7/docs/api/>.*

Core Packages

The following tables detail the core Java packages (that is, language, utility, and base packages). The definitions provided in these tables as well as the other tables throughout this appendix are primarily those definitions used in the Java SE 7 API documentation.

TABLE B-1 Language Packages

| Language Packages | Description |
|---------------------------------------|---|
| <code>java.lang</code> | Provides classes that are fundamental to the design of the Java programming language. |
| <code>java.lang.annotation</code> | Provides library support for the Java programming language annotation facility. |
| <code>java.lang.instrument</code> | Provides services that allow Java programming language agents to instrument programs running on the JVM. |
| <code>java.lang.invoke</code> | Contains dynamic language support provided directly by the Java core class libraries and virtual machine. |
| <code>java.lang.management</code> | Provides the management interfaces for monitoring and management of the JVM and other components in the Java runtime. |
| <code>java.lang.ref</code> | Provides reference-object classes, which support a limited degree of interaction with the garbage collector. |
| <code>java.lang.reflect</code> | Provides classes and interfaces for obtaining reflective information about classes and objects. |
| <code>javax.lang.model</code> | Provides classes and hierarchies of packages used to model the Java programming language. |
| <code>javax.lang.model.element</code> | Provides interfaces used to model elements of the Java programming language. |
| <code>javax.lang.model.type</code> | Provides interfaces used to model Java programming language types. |
| <code>javax.lang.model.util</code> | Provides utilities to assist in the processing of program elements and types. |

TABLE B-2 Utility Packages

| Utility Packages | Description |
|--|---|
| <code>java.util</code> | Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array). |
| <code>java.util.concurrent</code> | Contains utility classes commonly useful in concurrent programming. |
| <code>java.util.concurrent.atomic</code> | Contains a small toolkit of classes that support lock-free thread-safe programming on single variables. |
| <code>java.util.concurrent.locks</code> | Provides interfaces and classes to use as a framework for locking and waiting for conditions that are distinct from built-in synchronization and monitors. |
| <code>java.util.jar</code> | Provides classes for reading and writing the Java Archive (JAR) file format, which is based on the standard ZIP file format with an optional manifest file. |
| <code>java.util.logging</code> | Provides classes and interfaces of the Java 2 platform's core logging facilities. |
| <code>java.util.prefs</code> | Allows applications to store and retrieve user and system preference and configuration data. |
| <code>java.util.regex</code> | Provides classes for matching character sequences against patterns specified by regular expressions. |
| <code>java.util.spi</code> | Provides service provider classes for the classes in the <code>java.util</code> package. |
| <code>java.util.zip</code> | Provides classes for reading and writing the standard ZIP and GZIP file formats. |

TABLE B-3 Base Packages

| Base Packages | Description |
|--------------------------------------|--|
| <code>java.beans</code> | Contains classes related to developing beans—components based on the JavaBeans architecture. |
| <code>java.beans.beancontext</code> | Provides classes and interfaces relating to bean context. |
| <code>java.applet</code> | Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context. |
| <code>java.io</code> | Provides for system input and output through data streams, serialization, and the file system. |
| <code>java.nio</code> | Defines buffers, which are containers for data, and provides an overview of the other NIO (New I/O) packages. |
| <code>java.nio.channels</code> | Defines channels, which represent connections to entities that are capable of performing I/O operations, such as files and sockets; defines selectors, for multiplexed, non-blocking I/O operations. |
| <code>java.nio.charset</code> | Defines charsets, decoders, and encoders for translating between bytes and Unicode characters. |
| <code>java.nio.channels.spi</code> | Provides service-provider classes for the <code>java.nio.channels</code> package. |
| <code>java.nio.file</code> | Defines interfaces and classes for the JVM to access files, file attributes, and file systems. |
| <code>java.nio.file.attribute</code> | Provides interfaces and classes that allow access to file and file system attributes. |

| | |
|--|--|
| <code>java.nio.file.spi</code> | Includes service-provider classes for the <code>java.nio.file</code> package. |
| <code>java.nio.charset.spi</code> | Includes service-provider classes for the <code>java.nio.charset</code> package. |
| <code>java.math</code> | Provides classes for performing arbitrary-precision integer arithmetic (<code>BigInteger</code>) and arbitrary-precision decimal arithmetic (<code>BigDecimal</code>). |
| <code>java.net</code> and <code>javax.net</code> | Provides the classes for implementing networking applications. |
| <code>javax.net</code> | Provides classes for networking applications. |
| <code>javax.net.ssl</code> | Provides classes for the secure socket package. |
| <code>java.text</code> | Provides classes and interfaces for handling text, dates, numbers, and messages in a manner independent of natural languages. |
| <code>java.text.spi</code> | Provides service provider classes for the classes in the <code>java.text</code> package. |
| <code>javax.management</code> | Provides the core classes for the Java Management Extensions. |
| <code>javax.management.loading</code> | Provides the classes that implement advanced dynamic loading. |
| <code>javax.management.modelmbean</code> | Provides the definition of the ModelMBean classes. |
| <code>javax.management.monitor</code> | Provides the definition of the monitor classes. |
| <code>javax.management.openmbean</code> | Provides the open data types and Open MBean descriptor classes. |
| <code>javax.management.relation</code> | Provides the definition of the Relation Service. |
| <code>javax.management.remote</code> | Provides interfaces for remote access to Java Management Extensions (JMX) MBean servers. |
| <code>javax.management.remote.rmi</code> | Provides the RMI connector is a connector for the JMX Remote API that uses Remote Method Invocation (RMI) to transmit client requests to a remote MBean server. |
| <code>javax.management.timer</code> | Provides the definition of the Timer MBean. |
| <code>javax.annotation</code> | Provides resource support for annotation types. |
| <code>javax.annotation.processing</code> | Provides facilities for declaring annotation processors and for allowing annotation processors to communicate with an annotation processing tool environment. |
| <code>javax.tools</code> | Provides interfaces for tools that can be invoked from a program—for example, compilers. |
| <code>javax.activation</code> | Provides interfaces and classes used by the JavaMail API to manage MIME data. |
| <code>javax.activity</code> | Contains service activity-related exceptions thrown by the Object Request Broker (ORB) machinery during unmarshalling. |

Integration Packages

The following tables detail the integration Java packages (JDBC, JNDI, RMI/RMI-IIOP, scripting and transactions packages).

TABLE B-4 Java Database Connectivity (JDBC) Packages

| Java Database Connectivity (JDBC) Packages | Description |
|--|---|
| java.sql | Provides the API for accessing and processing data stored in a data source (usually a relational database) using the Java programming language. |
| javax.sql | Provides the API for server-side data source access and processing from the Java programming language. |
| javax.sql.rowset | Provides standard interfaces and base classes for JDBC RowSet implementations. |
| javax.sql.rowset.serial | Provides utility classes to allow serializable mappings between SQL types and data types in the Java programming language. |
| javax.sql.rowset.spi | Provides standard classes and interfaces that a third-party vendor must use in its implementation of a synchronization provider. |

TABLE B-5 Java Naming and Directory Interface (JNDI) Packages

| Java Naming and Directory Interface (JNDI) Packages | Description |
|---|---|
| javax.naming | Provides the classes and interfaces for accessing naming services. |
| javax.naming.ldap | Provides support for Lightweight Directory Access Protocol (LDAPv3) extended operations and controls. |
| javax.naming.event | Provides support for event notification when accessing naming and directory services. |
| javax.naming.directory | Extends the javax.naming package to provide functionality for accessing directory services. |
| javax.naming.spi | Provides additional interfaces and classes for naming support. |

TABLE B-6 Remote Method Invocation (RMI) Packages

| Remote Method Invocation (RMI) Packages | Description |
|---|--|
| java.rmi | Provides the RMI package. |
| java.rmi.activation | Provides support for RMI Object Activation. |
| java.rmi.dgc | Provides classes and interface for RMI distributed garbage collection (DGC). |
| java.rmi.registry | Provides a class and two interfaces for the RMI registry. |
| java.rmi.server | Provides classes and interfaces for supporting the server side of RMI. |
| javax.rmi | Contains user APIs for RMI-IIOP. |
| javax.rmi.CORBA | Contains portability APIs for RMI-IIOP. |
| javax.rmi.ssl | Provides implementations of RMIClientSocketFactory and RMIServerSocketFactory over the Secure Sockets Layer (SSL) or Transport Layer Security (TLS) protocols. |

TABLE B-7 Scripting Packages

| Scripting Packages | Description |
|--------------------|---|
| javax.script | Provides the scripting API that includes interfaces and classes that define Java Scripting Engines and provides a framework for their use in Java applications. |

TABLE B-8 Transactions Packages

| Transactions Packages | Description |
|------------------------------------|---|
| <code>javax.transactions.xa</code> | Provides the API that defines the contract between the transaction manager and the resource manager, which allows the transaction manager to enlist and delist resource objects (supplied by the resource manager driver) in Java Transaction API (JTA) transactions. |
| <code>javax.transactions</code> | Contains three exceptions thrown by the ORB machinery during unmarshalling. |

User Interface Packages

The following tables detail the user-interface Java packages (Swing API, Abstract Window Toolkit [AWT] API, image I/O, sound, printing, and accessibility packages).

TABLE B-9 Swing API Packages

| Swing API Packages | Description |
|---|--|
| <code>javax.swing</code> | Provides a set of "lightweight" (all-Java language) components that, to the maximum degree possible, work the same on all platforms. |
| <code>javax.swing.border</code> | Provides classes and interface for drawing specialized borders around a Swing component. |
| <code>javax.swing.colorchooser</code> | Contains classes and interfaces used by the <code>JColorChooser</code> component. |
| <code>javax.swing.event</code> | Provides for events fired by Swing components. |
| <code>javax.swing.filechooser</code> | Contains classes and interfaces used by the <code>JFileChooser</code> component. |
| <code>javax.swing.plaf</code> | Provides one interface and many abstract classes that Swing uses to provide its pluggable look-and-feel capabilities. |
| <code>javax.swing.plaf.basic</code> | Provides user interface objects built according to the basic look and feel. |
| <code>javax.swing.plaf.metal</code> | Provides user interface objects built according to the Java look and feel (once code named Metal), which is the default look and feel. |
| <code>javax.swing.plaf.multi</code> | Provides user interface objects that combine two or more look and feels. |
| <code>javax.swing.plaf.nimbus</code> | Provides user interface objects built according to the cross-platform Nimbus look and feel. |
| <code>javax.swing.plaf.synth</code> | Provides a skinnable look and feel, in which all painting is delegated. |
| <code>javax.swing.table</code> | Provides classes and interfaces for dealing with <code>javax.swing.JTable</code> . |
| <code>javax.swing.text</code> | Provides classes and interfaces that deal with editable and noneditable text components. |
| <code>javax.swing.text.html</code> | Provides the class <code>HTMLEditorKit</code> and supporting classes for creating HTML text editors. |
| <code>javax.swing.text.html.parser</code> | Provides the default HTML parser, along with support classes. |
| <code>javax.swing.text.rtf</code> | Provides a class (<code>RTFEditorKit</code>) for creating Rich Text Format (RTF) text editors. |
| <code>javax.swing.tree</code> | Provides classes and interfaces for dealing with <code>javax.swing.JTree</code> . |
| <code>javax.swing.undo</code> | Allows developers to provide support for undo/redo in applications such as text editors. |

TABLE B-10 AWT API Packages

| AWT API Packages | Description |
|---------------------------|--|
| java.awt | Contains all of the classes for creating user interfaces and for painting graphics and images. |
| java.awt.color | Provides classes for color spaces. |
| java.awt.datatransfer | Provides interfaces and classes for transferring data between and within applications. |
| java.awt.dnd | Provides a mechanism to transfer information by dragging and dropping in the user interface. |
| java.awt.event | Provides interfaces and classes for dealing with different types of events fired by AWT components. |
| java.awt.font | Provides classes and interface relating to fonts. |
| java.awt.geom | Provides the Java 2D classes for defining and performing operations on objects related to two-dimensional geometry. |
| java.awt.im | Provides classes and interfaces for the input method framework. |
| java.awt.im.spi | Provides interfaces that enable the development of input methods that can be used with any Java Runtime Environment. |
| java.awt.image | Provides classes for creating and modifying images. |
| java.awt.image.renderable | Provides classes and interfaces for producing rendering-independent images. |
| java.awt.print | Provides classes and interfaces for a general printing API. |

TABLE B-11 Java Image I/O Packages

| Java Image I/O Packages | Description |
|--------------------------------|---|
| javax.imageio | Provides the main package of the Java Image I/O API. |
| javax.imageio.event | Deals with synchronous notification of events during the reading and writing of images. |
| javax.imageio.metadata | Deals with reading and writing metadata. |
| javax.imageio.plugins.bmp | Contains the public classes used by the built-in Bean-Managed Persistence (BMP) plug-in. |
| javax.imageio.plugins.jpeg | Provides classes supporting the built-in JPEG plug-in. |
| javax.imageio.spi | Contains the plug-in interfaces for readers, writers, transcoders, and streams, and a runtime registry. |
| javax.imageio.stream | Deals with low-level I/O from files and streams. |

TABLE B-12 Sound API Packages

| Sound API Packages | Description |
|---------------------------|--|
| javax.sound.midi | Provides interfaces and classes for I/O, sequencing, and synthesis of MIDI (Musical Instrument Digital Interface) data. |
| javax.sound.midi.spi | Supplies interfaces for service providers to implement when offering new MIDI devices, MIDI file readers and writers, or sound bank readers. |
| javax.sound.sampled | Provides interfaces and classes for capture, processing, and playback of sampled audio data. |
| javax.sound.sampled.spi | Supplies abstract classes for service providers to subclass when offering new audio devices, sound file readers and writers, or audio format converters. |

TABLE B-13 Java Print Service API Packages

| Java Print Service API Packages | Description |
|---------------------------------|---|
| javax.print | Provides the principal classes and interfaces for the Java Print Service API. |
| javax.print.attribute | Provides classes and interfaces that describe the types of Java Print Service attributes and how they can be collected into attribute sets. |
| javax.print.attribute.standard | Contains classes for specific printing attributes. |
| javax.print.event | Contains event classes and listener interfaces. |

TABLE B-14 Accessibility Package

| Accessibility Package | Description |
|-----------------------|--|
| javax.accessibility | Defines a contract between UI components and an assistive technology that provides access to those components. |

Security Packages

The following tables detail the security-related Java packages (such as cryptography packages).

TABLE B-15 Security Packages

| Security Packages | Description |
|---|--|
| <code>java.security</code> | Provides the classes and interfaces for the security framework. |
| <code>java.security.acl</code> | The classes and interfaces in this package have been superseded by classes in the <code>java.security</code> package. |
| <code>java.security.cert</code> | Provides classes and interfaces for parsing and managing certificates, certificate revocation lists (CRLs), and certification paths. |
| <code>java.security.interfaces</code> | Provides interfaces for generating RSA (Rivest, Shamir and Adleman AsymmetricCipher algorithm) keys as defined in the RSA Laboratory Technical Note PKCS#1, and DSA (Digital Signature Algorithm) keys as defined in NIST's FIPS-186. |
| <code>java.security.spec</code> | Provides classes and interfaces for key specifications and algorithm parameter specifications. |
| <code>javax.security.auth</code> | Provides a framework for authentication and authorization. |
| <code>javax.security.auth.callback</code> | Provides the classes necessary for services to interact with applications to retrieve information (authentication data including usernames or passwords, for example) or to display information (error and warning messages, for example). |
| <code>javax.security.auth.kerberos</code> | Contains utility classes related to the Kerberos network authentication protocol. |
| <code>javax.security.auth.login</code> | Provides a pluggable authentication framework. |
| <code>javax.security.auth.X500</code> | Contains the classes that should be used to store X500 Principal and X500 Private Credentials in a <code>Subject</code> . |
| <code>javax.security.auth.spi</code> | Provides the interface to be used for implementing pluggable authentication modules. |
| <code>javax.security.sasl</code> | Contains classes and interfaces for supporting Simple Authentication and Security Layer (SASL). |
| <code>javax.security.cert</code> | Provides classes for public key certificates. |
| <code>org.ietf.jgss</code> | Presents a framework to make use of security services such as authentication, data integrity, and data confidentiality from a variety of underlying security mechanisms such as Kerberos, using a unified API. |

TABLE B-16 Cryptography Packages

| Package | Description |
|--------------------------------------|--|
| <code>javax.crypto</code> | Provides the classes and interfaces for cryptographic operations. |
| <code>javax.crypto.interfaces</code> | Provides interfaces for Diffie-Hellman keys as defined in RSA Laboratories PKCS #3. |
| <code>javax.crypto.spec</code> | Provides classes and interfaces for key specifications and algorithm parameter specifications. |

XML-based Packages

The following tables detail the XML-related Java packages.

TABLE B-17 XML-based Packages

| XML-based Packages | Description |
|------------------------------------|--|
| javax.xml | Provides Extensible Markup Language (XML) support and constants. |
| javax.xml.bind | Provides a runtime binding framework for client applications including unmarshalling, marshalling, and validation capabilities. |
| javax.xml.bind.annotation | Defines annotations for customizing Java program elements to XML Schema mapping. |
| javax.xml.bind.annotation.adapters | Provides <code>XmlAdapter</code> and its spec-defined subclasses to allow arbitrary Java classes to be used with Java Architecture for XML Binding (JAXB). |
| javax.xml.bind.attachment | Enables the interpretation and creation of optimized binary data within an MIME-based package format, implemented by a MIME-based package processor. |
| javax.xml.bind.helpers | Provides partial default implementations for some of the <code>javax.xml.bind</code> interfaces (JAXB provider use only). |
| javax.xml.bind.util | Provides useful client utility classes. |
| javax.xml.crypto | Provides common classes for XML cryptography. |
| javax.xml.crypto.dom | Provides DOM-specific classes for the <code>javax.xml.crypto</code> package. |
| javax.xml.crypto.dsig | Provides classes for generating and validating XML digital signatures. |
| javax.xml.crypto.dsig.dom | Provides DOM-specific classes for the <code>javax.xml.crypto.dsig</code> package. |
| javax.xml.crypto.dsig.keyinfo | Provides classes for parsing and processing <code>KeyInfo</code> elements and structures. |
| javax.xml.crypto.dsig.spec | Provides parameter classes for XML digital signatures. |
| javax.xml.datatype | Provides XML/Java Type mappings. |
| javax.xml.namespace | Provides XML Namespace processing. |
| javax.xml.parsers | Provides classes allowing the processing of XML documents. |
| javax.xml.soap | Provides the API for creating and building SOAP messages. |
| javax.xml.stream | Provides interfaces and classes in support of XML streams. |
| javax.xml.stream.events | Provides interfaces in support of XML streams events. |
| javax.xml.stream.util | Provides interfaces and classes in support of stream events. |
| javax.xml.transform | Defines the generic APIs for processing transformation instructions and performing a transformation from source to result. |
| javax.xml.transform.dom | Implements Document Object Model (DOM)-specific transformation APIs. |
| javax.xml.transform.sax | Implements SAX2-specific transformation APIs. |
| javax.xml.transform.stax | Provides for Streaming API for XML (StAX)-specific transformation APIs. |
| javax.xml.transform.stream | Implements stream- and URI-specific transformation APIs. |
| javax.xml.validation | Provides an API for validation of XML documents. |
| javax.xml.ws | Contains the core Java API for XML Web Services (JAX-WS) APIs. |
| javax.xml.ws.handler | Defines APIs for message handlers. |
| javax.xml.ws.handler.soap | Defines APIs for SOAP message handlers. |
| javax.xml.ws.http | Defines APIs specific to the HTTP binding. |
| javax.xml.ws.soap | Defines APIs specific to the SOAP binding. |
| javax.xml.ws.spi | Defines SPIs for JAX-WS. |
| javax.xml.ws.spi.http | Provides HTTP SPI used for portable deployment of JAX-WS web services in containers. |
| javax.xml.ws.wsaddressing | Defines APIs related to WS-Addressing. |
| javax.xml.xpath | Provides an <i>object-model neutral</i> API for the evaluation of XPath expressions and access to the evaluation environment. |
| org.w3c.dom | Provides the interfaces for the DOM, which is a component API of the Java API for XML Processing. |

| | |
|------------------------------------|---|
| <code>org.w3c.dom.bootstrap</code> | Contains a factory class that enables applications to obtain instances of DOM Implementation. |
| <code>org.w3c.dom.events</code> | Provides interfaces and classes in support of DOM events. |
| <code>org.w3c.dom.ls</code> | Provides interfaces and exceptions support of DOM factory methods for creating Load and Save objects. |
| <code>org.xml.sax</code> | Provides the core SAX APIs. |
| <code>org.xml.sax.ext</code> | Contains interfaces to SAX2 facilities that conformant SAX drivers won't necessarily support. |
| <code>org.xml.sax.helpers</code> | Contains helper classes, including support for bootstrapping SAX-based applications. |
| <code>javax.jws.soap</code> | Provides support for SOAP bindings. |
| <code>javax.jws</code> | Provides annotation types in support of Java Web Services. |



C

Java Keywords

The following table represents all of the valid Java keywords.

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

Please note the following:

- Java keywords cannot be used as identifiers.
- The enum keyword was added in J2SE 5.0 (Tiger).
- The assert keyword was added in J2SE 1.4 (Merlin).
- Reserved literals named true, false, and null are not keywords.

Keywords const and goto are reserved Java keywords but are not functionally used. Note that since they are commonly used C language keywords, providing them as keywords allows the IDEs and compilers to provide better error messages when these keywords are encountered in a Java program.



D

Bracket Conventions

Java Bracket Conventions

The Java programming language, like many programming languages, makes strong use of brackets. The OCA exam requires that you be familiar with the different types of brackets. The following table contains each type of bracket that appears throughout this book and in the exam. This table details the bracket names as they are used in the Java Language Specification (JLS) as well as common alternative names.

| Brackets | JLS Nomenclature | Alternative Nomenclature | Usage |
|----------|------------------|---|--|
| () | Parentheses | Round brackets, curved brackets, oval brackets | Surrounds set of method arguments, encloses cast types, adjusts precedence in arithmetic expressions |
| { } | Braces | Curly brackets/braces, swirly brackets, squirrely brackets, fancy brackets, squiggly brackets | Surrounds blocks of code, initializes arrays |
| [] | Box brackets | Square brackets, closed brackets | Used with arrays, initializes arrays |
| < > | Angle brackets | Diamond brackets, chevrons | Encloses generics |

Miscellaneous Bracket Conventions

Guillemet characters, as represented in the next table, are used in Universal Modeling Language (UML). UML was covered in the SCJA exam, but does not appear in the OCA exam. This information is provided here because you may find yourself using UML at some point in your career.

| Bracket | Nomenclature | Alternative Nomenclature | Usage |
|---------|----------------------|--------------------------|---------------------------|
| « » | Guillemet characters | Angle quotes | Specifies UML stereotypes |



E

Unicode Standard

The Unicode Standard is a character coding system designed to form a universal character set. This standard is maintained by the Unicode Consortium standards organization. The characters in this set are technically known as Unicode scalar values (in other words, hex numbers). Commonly known as Unicode characters, the characters are primarily organized into symbol and punctuation characters, as well as by script characters (for example, spoken language characters).

Literal values in Java can be written using Unicode, as shown in the following examples:

```
int i = '\u0043' + '\u0021'; // 100 (67 + 33)
char[] cArray = {'\u004F', '\u0043', '\u0041'}; // OCA
```

Code charts for Unicode are maintained by the consortium for easy reference. The “Code Charts for Symbols and Punctuation” can be accessed from <http://unicode.org/charts/#symbols>. “The Unicode Character Code Charts by Script” can be accessed from <http://unicode.org/charts/index.html>.

Current Oracle release documentation states that Java SE 7 supports the Unicode Standard 6.0.0. Unicode 6.0.0 includes support for properties and data files as well as support for 2000 additional characters. The Java SE 6 and J2SE 5.0 API’s character information is based on the Unicode standard, version 4.0. The J2SE 1.4 API’s character information is based on the Unicode standard, version 3.0. This Unicode compliancy information is found in the documentation of the `Character` class.

Many Unicode standard groupings of characters exist, such as language characters, currency symbols, Braille patterns, arrows, and mathematical operators. The most commonly used characters are the ASCII punctuation characters.

ASCII Punctuation Characters

The first 128 characters are the same as those in the American Standard Code for Information Exchange (ASCII) character set. The Unicode Consortium references them as ASCII punctuation characters. [Table E-1](#) represents these characters. The values \u0000 to \u001F and 007F represent nonprintable ASCII characters. The values \u0020 to \u007E represent printable ASCII characters. The character \u0020 represents a blank space. As an example, the space could also be referenced by its decimal equivalent value (that is, 32), its octal equivalent value (040), its HTML equivalent value (), or directly by its printable character, as in `char c = ' ';`.

TABLE E-1 Printable and Nonprintable ASCII Characters

| | 000 | 001 | 002 | 003 | 004 | 005 | 006 | 007 |
|---|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 | NUL | DLE | SP | 0 | @ | P | ` | p |
| 1 | SOH | DC1 | ! | 1 | A | Q | a | q |
| 2 | STX | DC2 | " | 2 | B | R | b | r |
| 3 | ETX | DC3 | # | 3 | C | S | c | s |
| 4 | EOT | DC4 | \$ | 4 | D | T | d | t |
| 5 | ENQ | NAK | % | 5 | E | U | e | u |
| 6 | ACK | SYN | & | 6 | F | V | f | v |
| 7 | BEL | ETB | ' | 7 | G | W | g | w |
| 8 | BS | CAN | (| 8 | H | X | h | x |
| 9 | HT | EM |) | 9 | I | Y | i | y |
| A | LF | SUB | * | : | J | Z | j | z |
| B | VT | ESC | + | ; | K | [| k | { |
| C | FF | FS | , | < | L | \ | l | |
| D | CR | GS | - | = | M |] | m | } |
| E | SO | RS | . | > | N | ^ | n | - |
| F | SI | US | / | ? | O | _ | o | DEL |

The Java tutorials provide more information about the Unicode industry standard in their online trail: <http://docs.oracle.com/javase/tutorial/i18n/text/unicode.html>.



F

Pseudo-code Algorithms

Pseudo-code algorithms were covered in the SCJA exam. The coverage has been removed for the updated OCA exam. Therefore, we have reduced the pseudo-code-related material that appeared in the predecessor of this book, *SCJA; Sun Certified Java Associate Study Guide* (McGraw-Hill, 2009). The pseudo-code algorithms are covered here because the information is important for you to be familiar with in the real world.

Because you do not need to know about pseudo-code algorithms for the OCA exam, reviewing this appendix is optional.

Implementing Statement-Related Algorithms from Pseudo-code

Pseudo-code is a structured means to allow algorithm designers to express computer programming algorithms in a human-readable format. Pseudo-code is informally written and is compact and high-level in nature. Even though pseudo-code does not need to be tied to any specific software language, the designer will typically script the pseudo-code algorithms based on the structural conventions of the target software language.

You may be thinking, “Hey, pseudo-code sounds great! Where do I get started writing high-quality algorithms in pseudo-code?” Well, don’t get too excited. No standards exist for writing pseudo-code, since its main purpose is to help designers build algorithms in their own language. With so many different languages having varying structural differences and paradigms, creating a pseudo-code standard that applies to them all would be impossible. Essentially, writing pseudo-code allows for the quick and focused production of algorithms based on logic, not language syntax.

The following topics presented in the next sections will discuss working with basic pseudo-code and converting pseudo-code algorithms into Java code with an emphasis on statements:

Pseudo-code algorithms

Pseudo-code algorithms and Java

Pseudo-code Algorithms

The previous exam presented pseudo-code algorithms to the test candidate. In turn, the candidate had to decide which Java code segment correctly implemented the algorithms. This was tricky, since the pseudo-code algorithms did not need to represent Java syntax in any way, but the Java code segments had to be structurally and syntactically accurate to be correct.

Let's take a look at a pseudo-code algorithm:

```
value := 20
IF value >= 1
    print the value
ELSEIF value = 0
    print the value
ELSE
    print "less than zero"
ENDIF
```

While surfing the Internet, you'll come to the conclusion that there is no universally accepted convention for pseudo-code. For demonstration purposes, [Table F-1](#) gives you a general idea of how a typical representation of pseudo-code can be translated to Java.

TABLE F-1 Pseudo-code Conventions

| Pseudo-code Element | Pseudo-code Convention | Java Example |
|---------------------|--|---|
| Assignment | variable := value | wreckYear = 1511; |
| if statement | IF condition THEN //statement sequence ELSEIF //statement sequence ELSE //statement sequence ENDIF | if (wreckYear == 1502) wreck = "Santa Ana"; elseif (wreckYear == 1503) wreck = "Magdalena"; else wreck = "Unknown"; |
| switch statement | CASE expression OF Condition A: statement sequence Condition B: statement sequence Default: sequence of statements ENDCASE | switch (wreckYear) { case 1502: wreck = "Santa Ana"; break; case 1503: wreck = "Magdalena"; break; default: wreck = "Unknown" } |
| while statement | WHILE condition //statement sequence ENDWHILE | while (n < 4) { System.out.println(i); n++; } |
| for statement | FOR iteration bounds //statement sequence ENDFOR | for (int i=0; i<j; i++) { System.out.println(i); } |

Pseudo-code Algorithms and Java

Pseudo-code can be a fragment of a complete source file, and it is okay that some primitive declarations may be missing. However, conditional and iteration statements are always represented completely.

Let's take a look at some examples. Here's a pseudo-code algorithm:

```
fishngRods := 5
fishngReels := 4
IF fishngRods does not equal fishngReels THEN
    print "We are missing fishing equipment"
ELSE
    print "The fishing equipment is all here"
ENDIF
```

Here's a Java implementation:

```
int fishngRods = 5;
int fishngReels = 4;
if (fishngRods != fishngReels)
    System.out.print("We are missing fishing equipment");
else
    System.out.print("The fishing equipment is all here");
```

Note that this appendix leveraged off of the “PSEUDOCODE STANDARD” page of the Cal Poly State University web site, as the authors did a pretty good job proposing a standard:

http://users.csc.calpoly.edu/~jdalbey/SWE/pdl_std.html.



G

Unified Modeling Language

The Unified Modeling Language (UML) specification defines a modeling language for the specification, presentation, construction, and documentation of object-oriented system elements. UML was covered in the SCJA exam, but it has been removed in the updated OCA exam. It is covered here to offer you a quick understanding to complement the UML diagrams that appear in this book.

The UML standard is the culmination of the works from James Rumbaugh's object-modeling technique, Grady Booch's "Booch method," and Ivar Jacobson's object-oriented software engineering method. The collaborative effort of this trio has earned them the name, "The Three Amigos." The origins of their efforts leading to the UML standard are detailed in [Table G-1](#).

TABLE G-1 Object Methodologies Preceding UML

| Methodologists | Method | Emphasis | Circa |
|---|---|---|-------|
| James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen | Object Modeling Technique (OMT) | Object-oriented analysis (OOA) | 1991 |
| Ivar Jacobson | Objectory, object-oriented software engineering (OOSE) method | Object-oriented software engineering (OOSE) | 1992 |
| Grady Booch | Booch method | Object-oriented design (OOD) | 1993 |

The modern UML specification, maintained by the Object Management Group (OMG), has gone through several revisions, as represented in [Table G-2](#). The current UML 2.4.1 specification comprises four parts. The first two parts are the "OMG UML Infrastructure Specification version 2.4.1" and the

“OMG UML Superstructure Specification version 2.4.1.” The Infrastructure specification has a tighter focus concerning class-based structures and houses all of the basic information you should know. The Superstructure specification details user-level constructs and cross-references the Infrastructure specification so that the two parts may be integrated into one volume in the future. The remaining parts are the “Object Constraint Language (OCL)” for defining model element rules and the “UML Diagram Interchange” used for defining the exchange of UML 2 diagram layouts. The current versions of the specifications are obtainable from the OMG at www.omg.org/spec/UML/Current. In short, this appendix will teach you how to recognize the main diagram elements and relationships used by UML.

TABLE G-2 Evolving UML Specifications

| OMG Formal UML Specifications | Official Release Date | Significant Release Changes |
|-------------------------------|-----------------------|--|
| UML 2.4.1 | August 2011 | Added URI package attribute, updated actions and events, refined stereotypes, various revisions to 2.3 |
| UML 2.3 | May 2010 | Added final classifier, updated component diagrams, composite structures, clarified associations, various revisions to 2.2 |
| UML 2.2 | February 2009 | Adoption of the profile diagram, various revisions |
| UML 2.1.2 | November 2007 | Various minor revisions; bug fixes have been resolved |
| UML 2.1.1 | August 2007 | Minor updates including implementation of redefinition and bidirectional association |
| UML 2.0 | July 2005 | Several changes, enhancements, and additions, including enhanced support for structural and behavioral models |
| UML 1.3, UML 1.4.X, UML 1.5 | Various | Various minor revisions and bug fixes resolved |
| UML 1.1 | November 1997 | The OMG formally adopted UML |

The complete set of 14 UML diagrams from the UML 2.4 standard is shown in [Table G-3](#).

TABLE G-3 Types of UML Diagrams

| Structure Diagram | Behavior Diagram | Interaction Diagram |
|-----------------------------|-----------------------|------------------------------|
| Class diagram | Activity diagram | Communication diagram |
| Component diagram | State machine diagram | Interaction overview diagram |
| Composite structure diagram | Use case diagram | Sequence diagram |
| Deployment diagram | | Timing diagram |
| Object diagram | | |
| Package diagram | | |
| Profile diagram | | |

Two closely related UML focal areas are covered in this appendix. One focal area relates to the recognitions of simple class structure artifacts and basic OO principles. The other relates to depicting UML features related to class relationships.

To start with your first UML element—in this case, let’s call it an icon—we’ll take a look at the basic Java SE package icons represented in [Figure G-1](#). The package icons are typically represented by a folder with the package name located in the top-left compartment (also known as the tab). The package name may also be optionally placed into the larger compartment (as shown in [Figure G-1](#)), as

is commonly done when no other UML elements are enclosed in the package icon. Note that the package icon is not on the test, but we include it in many of the diagrams to show packages that enclose depicted classes.

FIGURE G-1 UML package icons



This is a good time to look at the core UML information that you should be familiar with. This appendix is filled with details on the representation of these UML elements. When you have read the appendix, you will be able to recognize all of the core UML elements, as well as relationships between elements.

Recognizing Representations of Significant UML Elements

Getting acquainted with the different UML elements can actually be quite fun, and the sense of accomplishment you'll have when mastering the art of reading and writing class relationship diagrams with UML is equally rewarding.

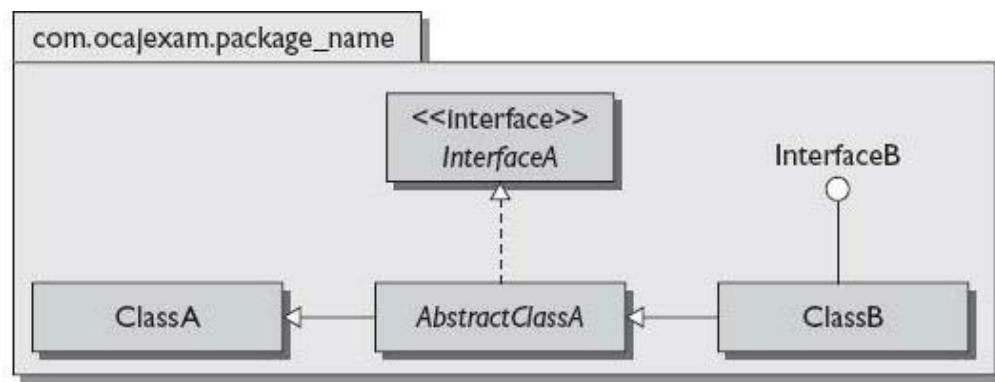
Our initial coverage is geared toward the class diagrams themselves. Attributes and operations compartments and visibility modifiers are also covered. Once you work though this section, you will know how to recognize the basic class elements of UML. These topics will be covered in the following subsections:

- Classes, abstract classes, and interface diagrams
- Attributes and operations
- Visibility modifiers

Classes, Abstract Classes, and Interface Diagrams

One of the simplest ways to represent classes and interfaces in UML is to show the class diagrams with only their name compartments. This holds true as well with representing interface implementations and class inheritances. [Figure G-2](#) depicts two interfaces, two classes, one abstract class, and their generalization and realization relationships. Abstract classes, concrete classes, and interfaces are all represented in a rectangle with their names in boldface type. Abstract classes are italicized. Interfaces are prefaced by the word “interface” between guillemot characters (like this: <<interface>>). An interface can be optionally depicted with its name beside the lollipop element (as with InterfaceB in the figure).

FIGURE G-2 Class diagram



The generalization and realization relationships between the classes in [Figure G-2](#) are further explained in the following sections.

Generalization

Generalization is expressed as an *is-a* relationship, where a class allows its more general attributes and operations to be inherited. In [Figure G-2](#), ClassB inherits from AbstractClassA and also from ClassA. AbstractClassA inherits from ClassA. We can also say ClassB *is-an* AbstractClassA, ClassB *is-a* ClassA, and AbstractClassA *is-a* ClassA. We could also say that ClassA and AbstractClassA are superclasses to ClassB, and, appropriately, ClassB would be their subclass. The generalization class relationship is depicted in the figure with a solid line and a closed arrowhead.

Realization

Realization is the general principle of implementing an interface. AbstractClassA implements the InterfaceA interface. ClassB implements the InterfaceB interface. The realization class relationship is depicted with a dotted line and a closed solid arrowhead or the lollipop element.

Code Engineering from UML Diagrams

UML provides many benefits; it is not limited to explaining existing code. When a system architect or system designer models the classes for a particular application, someone will need to develop code to those models. Many UML modeling tools can automatically generate the code structure for these models. However, most coders will use UML as a guide and choose to begin their coding from scratch.

Attributes and Operations

Attributes, also known as *member variables*, define the state of a class. *Operations*, sometimes called *member functions*, detail the methods of a class. Let's take a look at adding attributes and operations to a class UML diagram. The following is a code listing for an arbitrary `PrimeNumber` class. We will depict this class with UML.

```

import java.util.ArrayList;
import java.util.List;
public class PrimeNumber {
    private Boolean isPrime = true;
    private Double primeSquareRoot = null;
    private List<String> divisorList = new ArrayList<String>();
    public PrimeNumber(long candidate) {
        validatePrime(candidate);
    }
    public void validatePrime(Long c) {
        primeSquareRoot = Math.sqrt(c);
        isPrime = true;
        for (long j = 2; j <= primeSquareRoot.longValue(); j++) {
            if ((c % j) == 0) {
                divisorList.add(j + "x" + c / j);
                isPrime = false;
            }
        }
    }
    public List getDivisorList() {
        return divisorList;
    }
    public Double getPrimeSquareRoot() {
        return primeSquareRoot;
    }
    public Boolean getIsPrime() {
        return isPrime;
    }
    public void setIsPrime(Boolean b) {
        isPrime = b;
    }
}

```

Before we actually look at the associated UML diagram(s), let's examine the scope and required format for the information within the attributes and operations compartments.

Attributes Compartment

The attributes compartment houses the classes' attributes, also known as member variables. The attributes compartment is optionally present under the name compartment of the class diagram. The UML usage for each variable of the attributes compartment is detailed. For general knowledge it's good to be familiar with the following condensed attributes format:

[<visibility>] <variable_name> [: <type>] [= default_value]

Here, visibility defines the optionally displayed visibility modifier. The name would be the variable's name, and the type would be the type of the variable.

Operations Compartment

The operations compartment houses the classes' operations, also known as member functions or methods. The operations compartment is optionally present under the attributes compartment of the class diagram. If the attributes compartment is excluded, then the operations compartment may reside

under the name compartment of the class diagram. The UML usage for each method of the operations compartment is detailed. Again, it's good to be familiar with following condensed operations format:

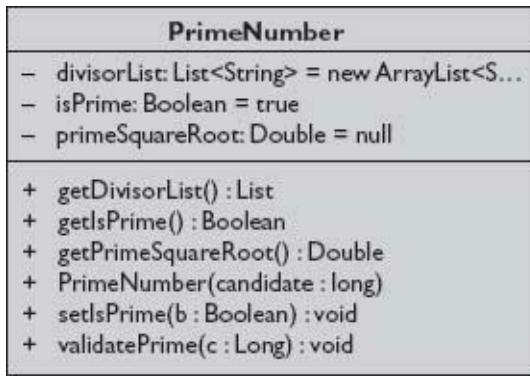
```
[<visibility>] <method_name> [<parameter-list>] [: <return-type>]
```

Here, visibility defines the optionally displayed visibility modifier. The name would be the method's name, the optionally displayed parameter-list is just as it says, and this is the same for the return-type.

Displaying the Attributes and Operations Compartments

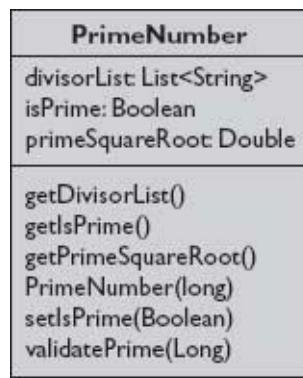
The display of level-of-detail information with regard to most UML elements is optional. This is true for the member variables and methods in the attributes and operations compartments as well. [Figure G-3](#) shows a more complete usage as defined in the compartment sections.

FIGURE G-3 Detailed attributes and operations compartments



In [Figure G-4](#), a more condensed representation of attributes and operations usages is shown.

FIGURE G-4 Abbreviated attributes and operations compartments



Both of the following representations are valid; by taking the time to understand this completely, you can avoid confusion when working with UML.

For attributes: `<variable_name> [: <type>]`

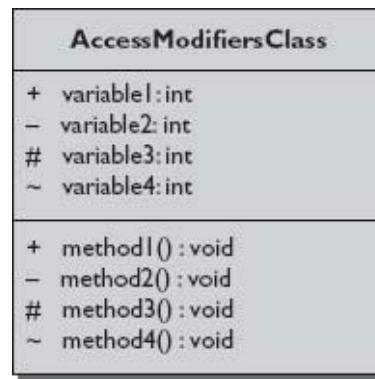
For operations: `<method_name> [<parameter-list>]`

Visibility Modifiers

As you are aware, there are four access modifiers: *public*, *private*, *protected*, and *package-private*.

These modifiers are depicted with symbols in UML and are used as shorthand in the attributes and operations compartments of a class diagram. These symbols are known as *visibility modifiers* or *visibility indicators*. The visibility indicator for the *public* access modifier is the plus sign (+). The visibility indicator for the *private* access modifier is the minus sign (-), and the visibility indicator for the *protected* access modifier is the pound sign (#), and the visibility indicator for the *package-private* modifier is the tilde (~) modifier. All four visibility modifiers are depicted in [Figure G-5](#) within the attributes and operations compartments. Visibility indicators are also optional and need not be displayed.

FIGURE G-5 Visibility modifiers



Several UML modeling tools are freely and commercially available in the marketplace. Being familiar with these tools will make you more productive in the workplace, will assist in collaboration, and will ultimately give you a more professional edge.

Recognizing Representations of UML Associations

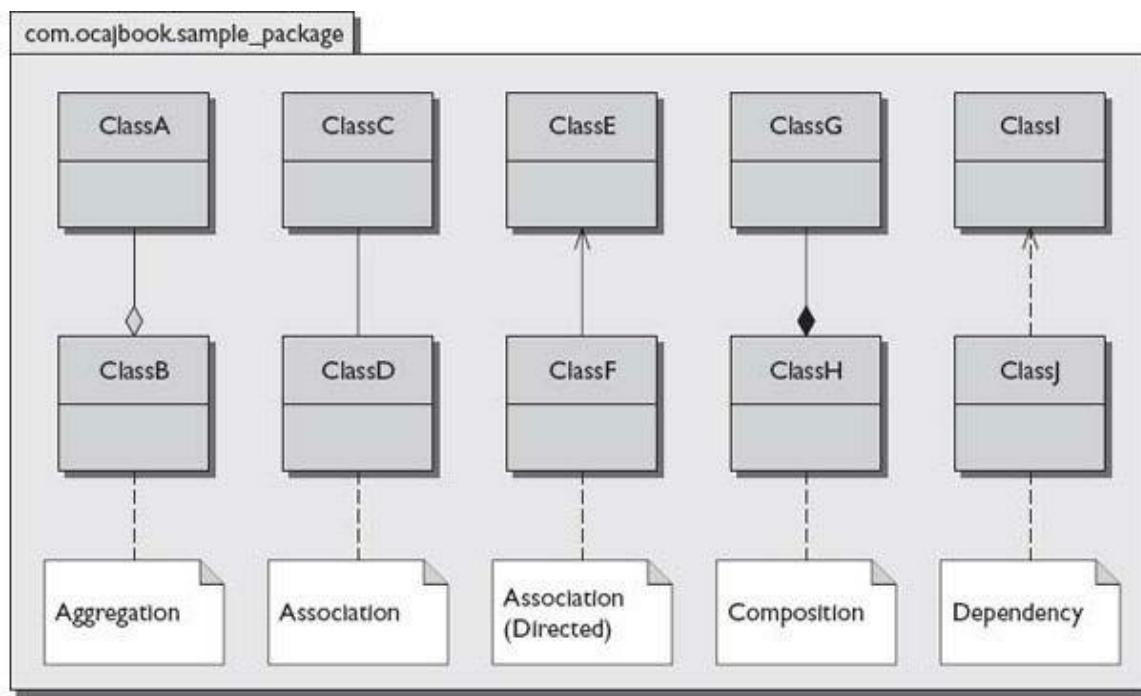
The preceding section solidified your knowledge of basic class diagrams and their main components. This section focuses on the relationships between classes in regard to their associations and compositions. Multiplicity indicators and role names are detailed as well to assist you in specifying the relationships between classes. When you have completed this section, you will know how to recognize connectors used between classes and how to interpret any specified multiplicity indicators and role names. The following topics will be covered:

- Graphic paths
- Relationship specifiers

Graphic Paths

The structure diagram graphic paths, also defined as class relationships, include notations for aggregation, association, composition, and dependency, as depicted in [Figure G-6](#). Generalization and realization graphic paths were covered in the preceding section.

FIGURE G-6 Graphic path notations



Aggregation Association

Aggregation association depicts one class as the owner of one or more classes. Aggregation is depicted with a solid line and an unfilled diamond. The diamond is on the side of the classifier. In [Figure G-6](#), you could say that a ClassA object is part of a ClassB object.

Association

An association that is not marked by navigability arrows is implied to be navigable in both directions; therefore, each association end is owned by the opposite classifier. Association is depicted with a solid line. In [Figure G-6](#), you could say there is an association between ClassC and ClassD objects.

Directed Association

An association has (directed) navigation when it is marked with a navigability arrow, also described as a stick arrow. This directed association's arrow denotes navigation in the direction of that end, the classifier has ownership of the marked association end, and the unmarked association's end is owned by the association. In addition to the navigability arrow, directed association is depicted with a solid line. In [Figure G-6](#), you could say that a ClassE object has a ClassF object.

Composition

Composition association depicts a class being composed of one or more classes. The component parts/classes live only as long as the composite class. Composition is depicted with a solid line and a filled diamond. The diamond is on the side of the classifier. In [Figure G-6](#), you could say that a ClassH object is composed of one or more ClassG objects.

Dependency

Dependency association depicts one class having a temporary association with another class. Dependency associations occur when a class needs another class to exist or when an object is used as a return value, local variable, or method argument. Dependency is depicted with a dotted line and a stick arrow. In [Figure G-6](#), you could say that a ClassJ object depends on a ClassI object.

As you probably noticed upon reading through the explanations of the relationships, class relationships can be written out using catchphrases between the objects. Common relationship catchphrases include “has-a,” “is-a,” “is composed of,” “is part of,” and “uses-a.”

Notes

Notes are represented in UML as a rectangle with a folded corner in the upper-right. Comments are placed into the notes element and a dotted line is drawn from the notes element to the artifact being commented on.

Relationship Specifiers

Sometimes depicting class relationships with the basic UML elements such as class diagrams and connectors is not enough to convey the true relationship between classes. A reader may clearly see that a relationship exists but may want to know more with regard to the constraints and high-level interaction. Multiplicity indicators and role names are specifiers used to define and clarify these relationships further.

Multiplicity Indicators

Multiplicity indicators are numerical representations used to depict the number of objects that may or must be used in an association. [Table G-4](#) defines the meanings of the different multiplicity indicators. If an association end does not show a multiplicity indicator, then the value is assumed to be 1. Multiplicity indicators can take the form of a single value or can be represented as a bounded relationship (<lowerbound>..<upperbound>).

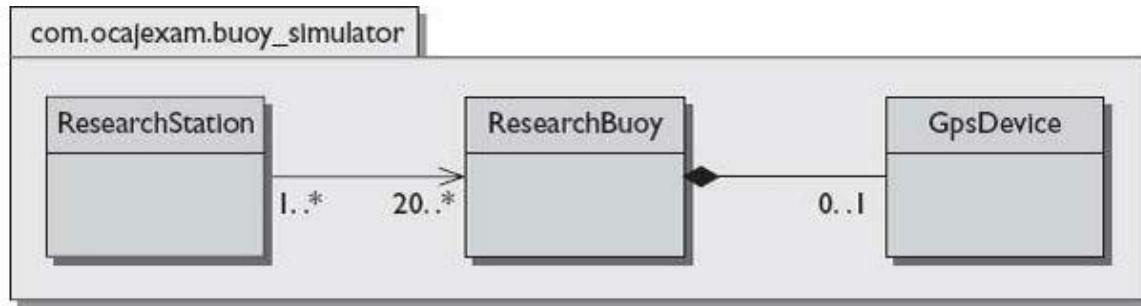
TABLE G-4 Multiplicity Indicators and Their Meanings

| Multiplicity Indicator | Example | Meaning of the Multiplicity Indicator |
|------------------------|---------|---|
| * | * | Object(s) of the source class may be aware of many objects of the destination class. |
| 0 | 0 | Object(s) of the source class are not aware of any objects of the destination class. This notation is not typically used. |
| 1 | 1 | Object(s) of the source class must be aware of exactly one object of the destination class. |
| [x] | 10 | Object(s) of the source class must be aware of the specified number of objects of the destination class. |
| 0..* | 0..* | Object(s) of the source class may be aware of zero or more objects of the destination class. |
| 0..1 | 0..1 | Object(s) of the source class may be aware of zero or one object of the destination class. |
| 0..[x] | 0..5 | Object(s) of the source class may be aware of zero or more objects of the destination class. |
| 1..* | 1..* | Object(s) of the source class must be aware of one or more objects of the destination class. |
| 1..[x] | 1..7 | Object(s) of the source class must be aware of one or up to the specified number of objects of the destination class. |
| [x]..[y] | 3..9 | Object(s) of the source class must be aware of the objects of the destination class within the specified range. |
| [x]..[y],[z] | 4..7,10 | Object(s) of the source class must be aware of the objects of the destination class within the specified range or the specified number. |

Multiplicity indicators in use are represented in [Figure G-7](#). Here you see the following:

ResearchStation objects must be aware of twenty or more ResearchBuoy objects. ResearchBuoy objects must be aware of at least one ResearchStation. Each ResearchBuoy must be composed of zero or more GpsDevice objects.

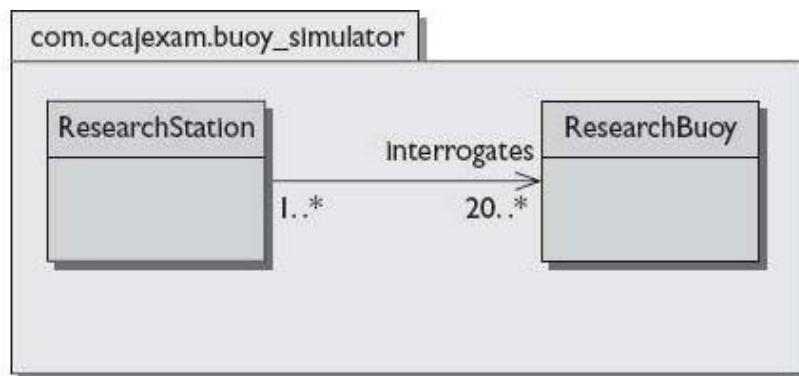
FIGURE G-7 Multiplicity indicators



Association Role Names

Role names are commonly used to clarify the usage of the associated objects and their multiplicities. In [Figure G-8](#), we see that the ResearchStation interrogates the ResearchBuoy. Without this descriptive role name, the relationship may have been unclear. We can also deduce that the ResearchStation object is aware of twenty ResearchBuoy objects, and each ResearchBuoy object is associated with one or more ResearchStation objects.

FIGURE G-8 Association role name



Appendix H

Practice Exam

Q1: Package statements

1. Which statement is not true about package statements?

- A. Package statements are optional.
- B. Package statements are limited to one per source file.
- C. Standard Java coding convention for package names reverses the domain name of the organization or group creating the package.
- D. The package names beginning with `javal.*` and `javaw.*` are reserved.

Hint: Consider the package names `java.util` and `javax.swing`.

Reference: [Chapter 1](#): Packaging, Compiling, and Interpreting Java Code

Objective: Understand Packages

Q2: Import statements

2. Which statement represents a valid statement that will allow for the inclusion of classes from the `java.util` package?

- A. `import java.util;`
- B. `import java.util.*;`
- C. `#include java.util;`
- D. `#include java.util.*;`

Hint: Consider valid Java keywords.

Reference: [Chapter 1](#): Packaging, Compiling, and Interpreting Java Code

Objective: Understand Packages

Q3: Collections API

3. List four interfaces of the Collections API.

- A. `ArrayList`, `Map`, `Set`, `Queue`

B. List, Map, Set, Queue

C. List, Map, HashSet, PriorityQueue

D. List, HashMap, HashSet, PriorityQueue

Hint: Consider the differences between interfaces and implementations (concrete classes).

Reference: [Chapter 1](#): Packaging, Compiling, and Interpreting Java Code

Objective: Understand Package-Derived Classes

Q4: Basic input/output classes

4. Which class in the `java.io` package allows for the reading and writing of files to specified locations within a file?

A. File

B. FileDescriptor

C. FilenameFilter

D. RandomAccessFile

Hint: This class implements `DataOutput`, `DataInput`, and `Closeable`.

Reference: [Chapter 1](#): Packaging, Compiling, and Interpreting Java Code

Objective: Understand Package-Derived Classes

Q5: Java interpreter

5. Which MS Windows-based command-line utility will allow you to run the Java interpreter without launching the console window?

A. javaw

B. interpw

C. java -wo

D. jconsole

Hint: The name is similar to that of the basic interpreter utility.

Reference: [Chapter 1](#): Packaging, Compiling, and Interpreting Java Code

Objective: Compile and Interpret Java Code

Q6: Import for ArrayList

6. What is the correct `import` package needed to use the `ArrayList` class?

- A. `import java.awt.*;`
- B. `import java.io.*;`
- C. `import java.net.*;`
- D. `import javax.swing.*;`
- E. `import java.util.*;`

F. This is a trick question, because it is part of the `java.lang` package that is imported automatically.

Hint: The package names give you some clue as to what it contains.

Reference: [Chapter 1](#): Packaging, Compiling, and Interpreting Java Code

Objective: Understand Package-Derived Classes

Q7: Packages for graphical toolkits

7. Of the following packages, which contain classes for building a graphical interface? (Choose all that apply.)

- A. `java.awt`
- B. `java.io`
- C. `java.net`
- D. `javax.swing`
- E. `java.util`

Hint: Rule out the packages with obvious names. Packages for graphical interfaces have names that make the least sense.

Reference: [Chapter 1](#): Packaging, Compiling, and Interpreting Java Code

Objective: Understand Package-Derived Classes

Q8: Extending classes

8. Which of the following statements is correct?

- A. A Java class can extend only one superclass.
- B. A Java class can extend multiple superclasses.
- C. A Java class cannot extend any superclasses.
- D. A Java class does not extend superclasses; it implements them.

Hint: Interfaces are implemented.

Reference: [Chapter 1](#): Packaging, Compiling, and Interpreting Java Code

Objective: Understand Class Structure

Q9: Packages

9. You have created a set of classes for your company and would like to include them in a package. Which one of the following would be a valid package name?

- A. your company name
- B. com.your company name
- C. java.your company name
- D. java.your_company_name
- E. com.your_company_name

Hint: Look for the answer that uses invalid names or characters.

Reference: [Chapter 1](#): Packaging, Compiling, and Interpreting Java Code

Objective: Understand Packages

Q10: The Java compiler

10. What is the result of running the following command:

```
javac Simulator.java
```

- A. The simulator program would be executed.
- B. A bytecode file `Simulator.class` would be created.

C. A bytecode file `simulator.java` would be created.

D. An error would be displayed because this is the wrong syntax.

Hint: `javac` is the Java compiler.

Reference: [Chapter 1](#): Packaging, Compiling, and Interpreting Java Code

Objective: Compile and Interpret Java Code

Q11: Statements

[11.](#) Of the following, which types of statements must be used to count the number of nickels in a `String` array of various coins?

A. Assignment

B. Assertion

C. Iteration

D. Conditional

Hint: An array will need to be stepped through, the string values of the array must be checked against the string (for example, "nickel"), and a counter must be incremented.

Reference: [Chapter 2](#): Programming with Java Statements

Objectives: Understand Assignment Statements, Create and Use Conditional Statements, Create and Use Iteration Statements

Q12: The `do-while` statement

[12.](#) The following short code segment contains two errors. What are they?

```
int a = 1;  
does  
a = 2;  
while (a == 1)
```

A. The declaration for the variable `a` must occur within the `do-while` statement.

B. Braces must be included for `do-while` statements even when only one statement is enclosed.

C. There is no `does` keyword; only the `do` keyword is allowed as part of a conditional statement.

D. The `do-while` statement must end with a semicolon.

Hint: Know your keywords and when to terminate a statement.

Reference: [Chapter 2](#): Programming with Java Statements

Objective: Create and Use Conditional Statements

Q13: Transfer of control statements

[13.](#) The `continue` and `break` statements are allowed within what types of statements?

- A. Loop statements
- B. All conditional statements
- C. The `switch` statement
- D. Expression statements

Hint: Iterations and cases

Reference: [Chapter 2](#): Programming with Java Statements

Objective: Create and Use Conditional Statements, Create and Use Iteration Statements

Q14: Javadoc comments

[14.](#) Select the commenting style that uses symbols designed to work with Javadoc.

- A. `// @author Robert`
- B. `/* @author Robert */`
- C. `/** @author Robert */`
- D. `/** @author Robert ;`

Hint: Javadoc comments must be properly terminated.

Reference: [Chapter 1](#): Packaging, Compiling, and Interpreting Java Code

Objective: Understand Class Structure

Q15: Assignments

[15.](#) Given the following code segments, which answer is NOT a valid Java implementation because it won't compile?

- A. `int variableA = 10;`

```
A. float variableA = 10.5;
   float variableB = 10.5f;
   int variableC = variableA + variableB;

B. int variableA = 10;
   float variableB = 10.5f;
   float variableC = variableA + variableB;

C. byte variableA = 10;
   float variableB = 10.5f;
   float variableC = variableA + variableB;

D. byte variableA = 10;
   double variableB = 10.5f;
   double variableC = variableA + variableB;
```

Hint: Consider when implicit casts occur.

Reference: [Chapter 2](#): Programming with Java Statements

Objective: Understand Assignment Statements

Q16: Statements

[16.](#) Which of the following code segments contain invalid Java statements? It can be assumed that all variables have been properly declared and initialized. (Choose all that apply.)

```
A. int x = getSize();

B. int a = 1;
   int b = 2;
   int c = a = b;

C. int mostPoints = 100;
   int score = 87;

   if ( mostPoints = score ) { setWinner(); }

D. boolean stillRunning = true;
   if ( stillRunning = true ) { run(); }
```

Hint: Remember that = is an assignment and == is a relational operator.

Reference: [Chapter 2](#): Programming with Java Statements

Objective: Create and Use Conditional Statements

Q17: Conditional statements

[17.](#) Which conditional statement would be best suited for determining the flow of code based on an `int` that may have a value between 1 and 10? The program must react differently for each value.

- A. `if-else` statement
- B. `if-else-if` statement with multiple `if-elses`
- C. multiple `if` statements
- D. `switch` statement

Hint: Most of these answers will satisfy the goal of the code; however, only one will produce the cleanest and most maintainable code.

Reference: [Chapter 2](#): Programming with Java Statements Objective: Create and Use Conditional Statements

Q18: Java statements

[18.](#) What Java code is used for declaring and initializing a null object?

- A. `Object object = new Object();`
- B. `Object object = new Object(null);`
- C. `object object = null;`
- D. `Object object = null();`

Hint: `null` is a keyword in Java and represents the absence of a value.

Reference: [Chapter 2](#): Programming with Java Statements

Objective: Understand Assignment Statements

Q19: Logical operators

[19.](#) Given the following code segment, what would be the output?

```
int value1 = 5;
int value2 = 7;
boolean bool1 = true;
boolean bool2 = false;
if (bool1 && (( value1 < value2 )|| bool2 ) && !(bool2)) {
    System.out.println("Result set one");
}
else {
    System.out.println("Result set two");
}
```

A. Result set one

B. Result set two

Hint: Start from the innermost parentheses and work outward, but also consider short-circuit operators.

Reference: [Chapter 3](#): Programming with Java Operators and Strings

Objective: Understand Fundamental Operators

Q20: Strings

20. What is the output for the following Java code segment? The text in parentheses is meant to describe the whitespace and is not part of the output.

```
String fiveSpaces = "      ";
String SCJA = "SCJA";
String lineToDisplay = "fiveSpaces + SCJA + fiveSpaces";
System.out.println(lineToDisplay.trim());
```

A. scJA (no trailing and leading spaces)

B. SCJA (five leading and trailing spaces)

C. SCJA (four leading and trailing spaces)

D. fiveSpaces + SCJA + fiveSpaces

Hint: Pay attention to where quotation marks are placed and how they affect the output.

Reference: [Chapter 3](#): Programming with Java Operators and Strings

Objective: Use String Objects and Their Methods

Q21: Strings

21. Given:

```
public class StringModifier {  
    public static void main (String[] args) {  
        String a = "Supercalifragilisticexpialidocious!";  
        String b = a.substring(x,y);  
        char [] c = {a.charAt(u), a.charAt(v)};  
        System.out.print(b + String.valueOf(c));  
    }  
}
```

What integer declarations are needed to print the string fragile!?

- A. int x = 8; int y = 14; int u = 2; int v = 33;
- B. int x = 10; int y = 16; int u = 4; int v = 35;
- C. int x = 8; int y = 15; int u = 5; int v = 32;
- D. int x = 9; int y = 15; int u = 3; int v = 34;

Hint: Indexing starts at zero.

Reference: [Chapter 3](#): Programming with Java Operators and Strings

Objective: Use String Objects and Their Methods

Q22: Whitespace omission

22. You want to remove all leading and trailing whitespace from a string. Which method invocation will allow this to occur?

- A. stringName.trim();
- B. stringName.trim(' ');
- C. stringName.trim(" ");
- D. stringName.trimWhiteSpace();

Hint: How many arguments does the trim method accept?

Reference: [Chapter 3](#): Programming with Java Operators and Strings

Objective: Use String Objects and Their Methods

Q23: Operator precedence

[23.](#) From highest precedence to lowest, which list of operators is ordered properly?

- A. *, +, &&, =
- B. *, &&, +, =
- C. *, =, &&, +
- D. +, *, &&, =

Hint: Multiplication is high on the list, and assignments are low.

Reference: [Chapter 3](#): Programming with Java Operators and Strings

Objective: Understand Operator Precedence

Q24: Modulus operator

[24.](#) Given the following code that uses the modulus operator, what will be printed?

```
System.out.print((24 % 8) + (10 % 7) + (100 % 99) + (38 % 6));
```

- A. 5
- B. 6
- C. 0312
- D. 0302

Hint: The Java modulus operator returns the remainder's value. The plus sign concatenates strings, though it can add integers under certain conditions.

Reference: [Chapter 3](#): Programming with Java Operators and Strings

Objective: Understand Fundamental Operators

Q25: Equality test

[25.](#) Given the following code segment testing the equality of two strings, what will be printed?

```
1. String string = "Dollar bill";
2. string.replace("Dollar bill", "Silver dollar");
3. if ("Dollar bill".equals(string)) {
4.     System.out.println ("I have a dollar bill.");
5. } else {
6.     System.out.println ("I have a silver dollar.");
7. }
```

- A. I have a dollar bill.
- B. I have a silver dollar.
- C. The code will not compile because of an error at line 2.
- D. The code will not compile because of an error at line 3.

Hint: Since strings are immutable, consider the effect from the absence of an assignment in line 2.

Reference: [Chapter 3](#): Programming with Java Operators and Strings

Objective: Use String Objects and Their Methods

Q26: `StringBuilder` append declarations

[26.](#) Which append declaration does not exist in Java 7?

- A. `public StringBuilder append (short s) {...}`
- B. `public StringBuilder append (int i) {...}`
- C. `public StringBuilder append (long l) {...}`
- D. `public StringBuilder append (float f) {...}`
- E. `public StringBuilder append (double d) {...}`

Hint: Consider which primitive is automatically casted to an integer when used as an integer literal.

Reference: [Chapter 3](#): Programming with Java Operators and Strings

Objective: Use `StringBuilder` Objects and Their Methods

Q27: Operator precedence

27. Given:

```
System.out.println("A" + 1 + (1 + 1) + 1);
```

What will print?

- A. A1111
- B. A121
- C. A13
- D. A31
- E. Compilation error

Hint: Association for the string concatenation operator is left to right.

Reference: [Chapter 3](#): Programming with Java Operators and Strings

Objective: Understand Operator Precedence

Q28: Using StringBuilder

28. Given:

```
StringBuilder s = new StringBuilder ("magic");
s.append("al").replace(1, 3, "us").matches("musical");
System.out.println(s);
```

What will be printed to standard out?

- A. magical
- B. musical
- C. true
- D. The code will not compile.

Hint: Consider the methods of the `StringBuilder` versus the string.

Reference: [Chapter 3](#): Programming with Java Operators and Strings

Objective: Use StringBuilder Objects and Their Methods

Q29: Conditional OR

[29.](#) Given:

```
boolean value = true;  
System.out.print( true || (value=false));  
System.out.println(", " + value);
```

What is printed to standard out?

- A. true, true
- B. true, false
- C. false, true
- D. false, true

Hint: Conditional OR is a short circuit operator.

Reference: [Chapter 3](#): Programming with Java Operators and Strings

Objective: Understand Fundamental Operators

Q30: Equality Testing

[30.](#) Given:

```
String value1 = "null";  
String value2 = "null";  
System.out.println(value1.equalsIgnoreCase(value2));
```

What result will be seen upon compilation and/or execution?

- A. A NullPointerException is thrown at runtime.
- B. equalsIgnoreCase is not a valid method name and causes a compilation error.
- C. "true" is printed to standard out.

D. "false" is printed to standard out.

Hint: When used properly, `equalsIgnoreCase` produces a Boolean value.

Reference: [Chapter 3](#): Programming with Java Operators and Strings

Objective: Test Equality Between Strings and other Objects

Q31: Increment operators

[31.](#) Given the following code segment, what would be the output?

```
int a = 0;
int b = 0;
int c = 0;
System.out.println(a++ + " " + --b + " " + c++ + " " + a + " " + b);
```

A. 1 0 1 0 0

B. 0 0 0 0 0

C. 1 0 1 1 -1

D. 0 -1 0 1 -1

Hint: When the operator is after the variable, the operation happens last; when the operator is before the variable, the operation happens first.

Reference: [Chapter 3](#): Programming with Java Operators and Strings

Objective: Understand Fundamental Operators

Q32: Mutable strings

[32.](#) Which of the following string classes create mutable strings? (Choose all that apply.)

A. String

B. StringBuffer

C. StringBuilder

Hint: Two of the answer choices are mutable.

Reference: [Chapter 3](#): Programming with Java Operators and Strings

Objective: Use String Objects and Their Methods

Q33: Variable type

33. Choose the simplest data structure to use to store multiple like variables that could then be accessed by an index.

A. Primitive

B. Object

C. Array

D. Enumeration

Hint: Try to remember the purpose of each item in the preceding list. The correct answer can be used in conjunction with the other answers. Find the one that is required to fulfill the question's requirements.

Reference: [Chapter 4](#): Working with Basic Classes and Variables

Objective: Understand Primitives, Enumerations, and Objects

Q34: Primitives and their values

34. Which of the statements are correct? (Choose all that apply.)

A. `3.0` is a valid literal for an `int`.

B. `3.0` is a valid literal for a `float`.

C. `3` is a valid literal for an `int`.

D. `3` is a valid literal for a `float`.

E. `3f` is a valid literal for an `int`.

F. `3f` is a valid literal for a `float`.

Hint: Consider each data type and the number. Recall the rules for what makes a number a particular type.

Reference: [Chapter 4](#): Working with Basic Classes and Variables

Objective: Understand Primitives, Enumerations, and Objects

Q35: The boolean primitive

[35.](#) What literal values are acceptable to use with the boolean primitive?

- A. true and false
- B. true, false, and null
- C. true, false, TRUE, and FALSE
- D. TRUE and FALSE

Hint: The null value is used with objects.

Reference: [Chapter 4](#): Working with Basic Classes and Variables

Objective: Understand Primitives, Enumerations, and Objects

Q36: Enumerations

[36.](#) Which code segments related to enumerations will result in a compiler error? (Choose all that apply.)

A. enum Coin {PENNY, NICKEL, DIME, QUARTER}

```
Coin coin = Coin.NICKEL;
```

B. enum Coin {PENNY, NICKEL, DIME, QUARTER}

```
Coin coin;
```

```
coin = Coin.NICKEL;
```

C. enum Coin {PENNY, NICKEL, DIME, QUARTER}

```
Coin coin = NICKEL;
```

D. enum Coin {penny, nickel, dime, quarter}

```
Coin coin = Coin.NICKEL;
```

Hint: The compiler error will look like this: “NICKEL cannot be resolved”.

Reference: [Chapter 4](#): Working with Basic Classes and Variables

Objective: Use Primitives, Enumerations, and Objects

Q37: Naming conventions

[37.](#) Consider the following declarations. Which declaration has an element name that does not conform to standard naming conventions?

- A. static final int varName; // variable
- B. Integer IntName; // variable
- C. void getName () {...} // method
- D. public class Name {...} // class

Hint: Only classes, interfaces, and enumerations should start with a capital letter, followed by lowercase letters.

Reference: [Chapter 4](#): Working with Basic Classes and Variables

Objective: Use Primitives, Enumerations, and Objects

Q38: Numeric primitives

[38.](#) Java 7 added the allowance of a special character to indentify places. Which declaration is correct?

- A. int investment = 1x000x000;
- B. int investment = 1_000_000;
- C. int investment = 1^000^000;
- D. int investment = 1-000-000;

Hint: Think about the packaging naming convention.

Reference: [Chapter 4](#): Working with Basic Classes and Variables

Objective: Understand Primitives, Enumerations, and Objects

Q39: Wrapper classes

[39.](#) What are the wrapper classes for the primitives boolean, char, short, int, and double?

- A. Boolean, Char, Short, Int, Double
- B. Boolean, Char, Short, Integer, Double
- C. Boolean, Character, Short, Int, Double
- D. Boolean, Character, Short, Integer, Double

Hint: Wrapper classes use the full spelling of the type's name.

Reference: [Chapter 4](#): Working with Basic Classes and Variables

Objective: Understand Primitives, Enumerations, and Objects

Q40: Literals

[40.](#) Which of the following statements contain literal values?

- A. `int maxHorsePower = 170;`
- B. `float currentHorsePower = (float) maxHorsePower;`
- C. `float idleHorsePower = ((float) currentHorsePower) / 10);`
- D. `System.out.println("Current HP: " + currentHorsePower);`

Hint: Remember that a literal value is not represented as a variable.

Reference: [Chapter 4](#): Working with Basic Classes and Variables

Objective: Use Primitives, Enumerations, and Objects

Q41: Initializing objects

[41.](#) What is the correct way to initialize a variable declared as a Penguin as a new Penguin object?

- A. `Penguin p;`
- B. `Penguin p = new Penguin();`
- C. `Penguin p = new Penguin[];`
- D. `Penguin p = Penguin();`

Hint: Initializing an object means a new one must be created.

Reference: [Chapter 4](#): Working with Basic Classes and Variables

Objective: Use Primitives, Enumerations, and Objects

Q42: Primitives

[42.](#) Which of the following are primitive data types? (Choose all that apply.)

- A. `int`

B. boolean

C. char

D. Float

E. string

Hint: If it is not a primitive, it is an object. Try to remember the naming conventions of Java primitives and objects.

Reference: [Chapter 4](#): Working with Basic Classes and Variables

Objective: Understand Primitives, Enumerations, and Objects

Q43: Arrays

[43.](#) Which code example makes use of arrays without producing a compiler or runtime error?

```
A. public class Actor {  
    String[] characterName = new String[3];  
    {  
        characterName[0] = "Captain Video";  
        characterName[1] = "Quizmaster";  
        characterName[2] = "J.C. Money";  
        characterName[3] = "Jersey Joe";  
    }  
}
```

```
B. public class Actor {  
    String[] characterName = new String[1..4]  
    {  
        characterName[0] = "Captain Video";  
        characterName[1] = "Quizmaster";  
        characterName[2] = "J.C. Money";  
        characterName[3] = "Jersey Joe";  
    }  
}
```

```
C. public class Actor {  
    String characterName = new String[4];  
    {  
        characterName[0] = "Captain Video";  
        characterName[1] = "Quizmaster";  
        characterName[2] = "J.C. Money";
```

```
    characterName[3] = "Jersey Joe";  
}  
}  
  
D. public class Actor {  
    String [] characterName = new String[4];  
    {  
        characterName[0] = "Captain Video";  
        characterName[1] = "Quizmaster";  
        characterName[2] = "J.C. Money";  
        characterName[3] = "Jersey Joe";  
    }  
}
```

Hint: Always start with 0 when working with array elements.

Reference: [Chapter 4](#): Working with Basic Classes and Variables

Objective: Understand Primitives, Enumerations, and Objects

Q44: Class variables

44. Class variables, also known as static fields, have only one instance in existence. Following standard naming conventions, which answer represents a class variable?

- A. variableName
- B. VariableName
- C. className:variableName
- D. className.variableName

Hint: The class name is used in conjunction with the static variable name to reference it.

Reference: [Chapter 5](#): Understanding Methods and Variable Scope

Objective: Understand Variable Scope

Q45: Local variables

[45.](#) What statement about local variables is true?

- A. Local variables are declared outside of methods and are initialized with a default value.
- B. Local variables are declared inside of methods and are initialized with a default value.
- C. Local variables are declared outside of methods and are not initialized with a default value.
- D. Local variables are declared inside of methods and are not initialized with a default value.

Hint: Instance variables are initialized with a default value.

Reference: [Chapter 5](#): Understanding Methods and Variable Scope

Objective: Understand Variable Scope

Q46: Constructors

[46.](#) Which declaration is a valid constructor for the class `CreateMiracle`?

- A. `public void createMiracle(){}`
- B. `public String createMiracle(String wish, String prayer) {}`
- C. `private String createMiracle(String wish) {}`
- D. `public CreateMiracle(String wish, String prayer){}`

Hint: Constructors do not return values.

Reference: [Chapter 5](#): Understanding Methods and Variable Scope

Objective: Create and Use Methods

Q47: Method declarations

[47.](#) You want to create a method that takes in three character parameters and returns a string. Which declaration is valid?

- A. `public void doMethod (char a, char b, char c) {String s = null; return s;}`
- B. `public String doMethod (Char a, Char b, Char c) {String s = null; return s;}`

- C. public String doMethod (char a, char b, char c) {String s = null; return s;}
- D. public string doMethod (char a, char b, char c) {String s = null; return s;}

Hint: Consider case and return values.

Reference: [Chapter 5](#): Understanding Methods and Variable Scope

Objective: Create and Use Methods

Q48: Modifiers

[48.](#) Which modifiers can be applied to constructors? (Choose all that apply.)

- A. abstract
- B. final
- C. native
- D. strictfp
- E. static
- F. synchronized
- G. transient
- H. volatile
- I. None of the above

Hint: Consider the purpose of each modifier.

Reference: [Chapter 5](#): Understanding Methods and Variable Scope

Objective: Create and Use Constructors

Q49: Default access modifiers for constructors

[49.](#) If a constructor does not include an access modifier, which modifier will it use by default?

- A. A constructor that does not include an access modifier will always be declared as public.
- B. A constructor that does not include an access modifier will make use of the same access modifier that is used for its class.

C. A compilation error will occur if a constructor does not include an access modifier.

Hint: Constructors may use any of the access modifiers (package-private, protected, private, or public).

Reference: [Chapter 5](#): Understanding Methods and Variable Scope

Objective: Create and Use Constructors

Q50: Scope

[50.](#) What is the scope of an instance variable?

- A. The block of code in which it is declared.
- B. The method for which it is a parameter.
- C. The class in which it is declared.

Hint: The three variable scopes are instance variables, method parameters, and local variables.

Reference: [Chapter 5](#): Understanding Methods and Variable Scope

Objective: Understand Variable Scope

Q51: Variable scope

[51.](#) What type of variable would be used to store the state of an object?

- A. Local variable
- B. Method parameter
- C. Instance variable
- D. Object variable

Hint: The name of the variable gives a clue as to where it should be used.

Reference: [Chapter 5](#): Understanding Methods and Variable Scope

Objective: Understand Variable Scope

Q52: Passing object arguments

[52.](#) Objects are passed by _____.

- A. Value
- B. Reference
- C. Copy
- D. Parameter

Hint: When an object is changed in a method, it is modified in the code that called the method.

Reference: [Chapter 5](#): Understanding Methods and Variable Scope

Objective: Create and Use Methods

Q53: Constructing methods

- [53.](#) What is the proper way to construct a method named `displayImage` that accepts two variables as arguments? The first variable is an array of `ints` called `points`, and the second is an `Image` object called `image`. This method does not return any variables.

- A. `displayImage(int[], Image) {...}`
- B. `displayImage(int[] points, Image image) {...}`
- C. `void displayImage(int[], Image) {...}`
- D. `void displayImage(int[] points, Image image) {...}`

Hint: Make sure no data is omitted from the method signature.

Reference: [Chapter 5](#): Understanding Methods and Variable Scope

Objective: Create and Use Methods

Q54: Declaring a return type

- [54.](#) Which of the following statements is/are true about declaring a return type? (Choose all that apply.)

- A. The keyword `void` is used to signify that a method does not return any value.
- B. The use of the keyword `void` is optional if the method isn't returning anything.
- C. If a method is returning an `int`, it must declare this by having `int` precede the method name in the method signature.

D. If the return type is omitted, a method may or may not return data, depending on the condition inside of the method.

Hint: A method must always clearly state what it intends to do.

Reference: [Chapter 5](#): Understanding Methods and Variable Scope

Objective: Create and Use Methods

Q55: Association

[55.](#) Which association can be stated as “object A ‘uses-an’ object B”?

- A. Direct association
- B. Temporary association
- C. Composition association
- D. Aggregation association.

Hint: “part-of,” “composed-of,” and “has-a” apply to the other answers.

Reference: [Chapter 10](#): Working with Classes and Their Relationships

Objective: Understand Class Compositions and Associations

Q56: Composition association

[56.](#) Which statement is false about composition association?

- A. Composition association models a whole-part relationship, where the whole is responsible for the lifecycle of its parts.
- B. Composition association is a weaker association than aggregation association.
- C. Composition association maintains a “composed-of” relationship.
- D. Composition association is considered a class relationship.

Hint: Understand the differences between composition and aggregation association.

Reference: [Chapter 10](#): Working with Classes and Their Relationships

Objective: Understand Class Composition and Associations

Q57: Association

- [57.](#) Given the following simple code segment, what association is being used in regards to the `Color.BLACK` object relating to the `cat` object?

```
public Class Cat {  
  
    public Color getColor () {  
        return Color.BLACK;  
    } }
```

- A. Containment
- B. Realization
- C. Dependency
- D. Navigability

Hint: Think about alternative names to the common associations.

Reference: [Chapter 10](#): Working with Classes and Their Relationships

Objective: Class Compositions and Associations in Practice

Q58: Class associations

- [58.](#) Of the following associations, which has/have a weak relationship? (Choose all that apply.)

- A. Direct association
- B. Temporary association
- C. Composition association
- D. Aggregation association
- E. Complete association

Hint: A relationship can be either weak or strong.

Reference: [Chapter 10](#): Working with Classes and Their Relationships

Objective: Understand Class Composition and Associations

Q59: Composition association

[59.](#) What is true about composition association? (Choose all that apply.)

- A. It has a weak relationship.
- B. It has a strong relationship.
- C. It has lifecycle responsibility for the objects in the relationship.
- D. It can have a one-to-one multiplicity.
- E. It can have a one-to-many multiplicity.
- F. It can have a many-to-many multiplicity.

Hint: A composition association can be envisioned as one object being composed of, or made up of, another object.

Reference: [Chapter 10](#): Working with Classes and Their Relationships

Objective: Understand Class Composition and Associations

Q60: Association

[60.](#) Which association does the following statement represent? “Object Z ‘has-an’ object Y.”

- A. Direct association
- B. Temporary association
- C. Composition association
- D. Aggregation association

Hint: The three types of association are “has-a,” “part-of,” and “composed-of.”

Reference: [Chapter 10](#): Working with Classes and Their Relationships

Objective: Understand Class Composition and Associations

Q61: Relationships in Java code

[61.](#) In the following code segment, what is the relationship between the `Store` class and the `MerchandiseItem` class?

```
public class Store {  
    MerchandiseItem item;  
  
    public Store() {  
        item = new MerchandiseItem();  
    }  
}
```

- A. Direct association
- B. Temporary association
- C. Composition association
- D. Aggregation association

Hint: Pay attention to where the variable for this class gets initialized.

Reference: [Chapter 10](#): Working with Classes and Their Relationships

Objective: Class Compositions and Associations in Practice

Q62: Multiplicity in Java code

- [62.](#) In the following code segment, what is the multiplicity between the `Album` class and the `Picture` class?

```
public class Album {  
    Picture[] pictures;  
  
    public Store() {  
        pictures = new Picture[3];  
        pictures[0] = new Picture();  
        pictures[1] = new Picture();  
        pictures[2] = new Picture();  
    }  
}
```

- A. One-to-one
- B. One-to-many

C. Many-to-many

D. Many-to-most

Hint: Count the classes at one end of the relationship and then the other end.

Reference: [Chapter 10](#): Working with Classes and Their Relationships

Objective: Class Compositions and Associations in Practice

Q63: Interfaces

[63.](#) Select the statement about interfaces that is correct.

A. An interface can extend only one interface.

B. An interface can extend more than one interface.

Hint: The behavior of extensions with interfaces is different from the behavior of how a class extends a class.

Reference: [Chapter 7](#) Understanding Class Inheritance

Objective: Implement and Use Inheritance and Class Types

Q64: Inheritance

[64.](#) Which two statements related to inheritance are true?

A. A concrete class can extend only one class.

B. A concrete class can extend multiple classes.

C. An interface can extend only one interface.

D. An interface can extend multiple interfaces.

Hint: You must be ultra-selective when deciding on which class to extend to a concrete class. Selectiveness doesn't matter when extending one or more interfaces from an interface.

Reference: [Chapter 7](#): Understanding Class Inheritance

Objective: Implement and Use Inheritance and Class Types

Q65: Abstract methods

[65.](#) Which Java elements can have abstract methods?

- A. Concrete classes
- B. Abstract classes
- C. Interfaces
- D. Packages

Hint: Concrete classes have all of their methods defined. Also, consider which answer makes no sense here.

Reference: [Chapter 7](#): Understanding Class Inheritance

Objective: Implement and Use Inheritance and Class Type

Q66: Encapsulation

[66.](#) Which access modifier used in conjunction with getter and setter methods provides tight encapsulation?

- A. package-private
- B. private
- C. protected
- D. public

Hint: Consider the minus sign visibility indicator in UML.

Reference: [Chapter 7](#): Understanding Class Inheritance

Objective: Understand Encapsulation Principles

Q67: Abstract classes

[67.](#) Which statement is NOT true about abstract classes?

- A. An abstract class must include the keyword `abstract` in its declaration.
- B. Abstract classes can be instantiated.
- C. Abstract classes contain a mixture of implemented and abstract methods.
- D. When an abstract class is extended, all of its abstract methods must be implemented by a non-abstract subclass.

Hint: Only concrete classes can be instantiated.

Reference: [Chapter 7](#): Understanding Class Inheritance

Objective: Advanced Use of Classes with Inheritance and Encapsulation

Q68: Access modifiers

[68.](#) The `private` and `protected` access modifiers can be used with which entities? (Choose all that apply.)

- A. Classes
- B. Interfaces
- C. Constructors
- D. Methods
- E. Data members

Hint: Think ‘inside.’

Reference: [Chapter 7](#): Understanding Class Inheritance

Objective: Understand Encapsulation Principles

Q69: Inheritance

[69.](#) What can contain a mixture of implemented and unimplemented methods and must be extended to use?

- A. Concrete class
- B. Abstract class
- C. Java class
- D. Interface

Hint: Try to recall what an unimplemented method can be called.

Reference: [Chapter 7](#): Understanding Class Inheritance

Objective: Implement and Use Inheritance and Class Types

Q70: Terms for class inheritance

[70.](#) If class A extends class B, what terms can be used to describe class A? (Choose all that apply.)

- A. Subclass
- B. Superclass
- C. Base class
- D. Parent class
- E. Child class

Hint: Think about the relationship between the two classes.

Reference: [Chapter 7](#): Understanding Class Inheritance

Objective: Implement and Use Inheritance and Class Types

Q71: Getters

[71.](#) Which of the following methods use the proper naming convention for a getter accessing the variable `totalPoints`?

- A. `TotalPoints()`
- B. `GetTotalPoints()`
- C. `getTotalpoints()`
- D. `getTotalPoints()`

Hint: The naming convention is often called camel case.

Reference: [Chapter 7](#): Understanding Class Inheritance

Objective: Understand Encapsulation Principles

Q72: Access modifiers

[72.](#) What access modifier, if any, should be used on all instance variables unless there is a particular reason not to use it?

- A. `public`
- B. `private`
- C. `protected`
- D. package-private (default)

Hint: Instance variables should always use the most restrictive modifier possible.

Reference: [Chapter 7](#): Understanding Class Inheritance

Objective: Understand Encapsulation Principles

Q73: Using interfaces

[73.](#) What is the correct method signature for class Z if it uses interface A?

- A. public class Z inherits A{ ... }
- B. public class Z extends A{ ... }
- C. public class Z implements A{ ... }
- D. public class Z uses A{ ... }

Hint: Interfaces cannot be inherited from.

Reference: [Chapter 7](#): Understanding Class Inheritance

Objective: Advanced Use of Classes with Inheritance and Encapsulation

Q74: Object-oriented principles

[74.](#) Which object-oriented principle is represented in the following declaration?

```
List<String> = new ArrayList<String>();
```

- A. Encapsulation
- B. Data abstraction
- C. Information hiding
- D. Polymorphism

Hint: Consider how ArrayList and List relate.

Reference: [Chapter 8](#): Understanding Polymorphism and Casts

Objective: Understand Polymorphism

Q75: Polymorphism

[75.](#) Which Java API allows for extensive use of the polymorphism OO principle?

A. Logging API

B. Collections API

C. Concurrency API

D. Networking API

Hint: Think `ArrayList`, `HashMap`, and `PriorityQueue`.

Reference: [Chapter 8](#): Understanding Polymorphism and Casts

Objective: Understand Polymorphism

Q76: Casting

[76.](#) Given:

```
Short s1 = (Short) 300;
short s2 = (Short) 300;
short s3 = (short) 300;
short s4 = 300;
```

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

A. The declaration for `s1` will compile.

B. The declaration for `s2` will compile.

C. The declaration for `s3` will compile.

D. The declaration for `s4` will compile.

Hint: Consider the rules behind explicit casting.

Reference: [Chapter 8](#): Understanding Polymorphism and Casts

Objective: Understand Casting

Q77: Boolean casting

[77.](#) Is it possible to cast a `boolean` type to another primitive type?

A. Yes, it is possible to cast a `boolean` type to another primitive type.

B. No, it is not possible to cast a `boolean` type to another primitive type.

Hint: You cannot cast a non-boolean primitive type to a boolean type.

Reference: [Chapter 8](#): Understanding Polymorphism and Casts

Objective: Understand Casting

Q78: Widening and narrowing

[78.](#) Given:

```
long longValue = 100;  
byte b = (byte) (short) (int) longValue; // narrowing  
  
byte byteValue = 100;  
long l = (long) (int) (short) byteValue; // widening  
  
System.out.println (b + l);
```

What will happen?

- A. "200" will be printed to standard out.
- B. The code will not compile due to issues with the declarations.
- C. The code will not compile due to an issue with the expression in the print statement.

Hint: Consider implicit casts in the print statement.

Reference: [Chapter 8](#): Understanding Polymorphism and Casts

Objective: Understand Casting

Q79: Program to an interface

[79.](#) What does it mean to “program to an interface”?

- A. Objects should be referenced by the interface they implement.
- B. Every object should implement at least one interface.
- C. Extending classes should be avoided in favor of using interfaces where possible.
- D. “Program to an interface” is another term for well-designed code.

Hint: Another way to explain “programming to an interface” is that interfaces are used wherever possible.

Reference: [Chapter 8](#): Understanding Polymorphism and Casts

Objective: Understand Polymorphism

Q80: Benefits of using polymorphism

[80.](#) What are the design benefits of using polymorphism? (Choose all that apply.)

- A. Objects can be generalized as return types and method parameters.
- B. Less code needs to be written.
- C. Superclasses and subclasses can be used interchangeably.
- D. Polymorphism allows code to work with many different types of classes that all share some commonality, such as implementing the same interface.

Hint: Polymorphism allows classes to be substituted by general types.

Reference: [Chapter 8](#): Understanding Polymorphism and Casts

Objective: Understand Polymorphism

Q81: Polymorphism

[81.](#) Polymorphism allows an object to be referred to as _____.
(Choose all that apply.)

- A. Any base class it extends
- B. Any interface that is implemented
- C. A primitive data type
- D. An instance variable
- E. Any subclass that extends it

Hint: It is easy to use a subset of methods from an object.

Reference: [Chapter 8](#): Understanding Polymorphism and Casts

Objective: Understand Polymorphism

Q82: Examples of polymorphism

[82.](#) Given that the `FoodStore` class extends the `Store` class, which of the following statements is/are valid? (Choose all that apply.)

- A. Store store = new FoodStore();
- B. FoodStore foodStore = new Store();
- C. Object obj = new FoodStore();
- D. Object obj = new Store();

Hint: Remember that the `Object` class is the base class of every class.

Reference: [Chapter 8](#): Understanding Polymorphism and Casts

Objective: Understand Polymorphism

Q83: Polymorphism in use

- [83.](#) What can be said about the following code?

```
public interface Audible {...}  
public class Stereo implements Audible {...}  
public class Computer implements Audible {...}  
  
public class AudioSource {  
    public void startPlaying(Audible sound) {...}  
}
```

- A. `Audible` is the base class for the `Stereo` and `Computer` classes.
- B. The `startPlaying` code should be changed so it uses polymorphism.
- C. `Object` is not a base class for the `AudioSource` class.
- D. The `startPlaying` method is using the design principle of “programming to an interface.”

Hint: Look carefully at the parameters of the `startPlaying` method.

Reference: [Chapter 8](#): Understanding Polymorphism and Casts

Objective: Understand Polymorphism

Q84: Exception categories

[84.](#) Exceptions fall into three categories. Which is not an exceptions category?

- A. Check Exceptions
- B. Unchecked Exceptions
- C. Assertions
- D. Errors

Hint: This answer contains a Boolean expression that is assumed to be true when the condition executes.

Reference: [Chapter 9](#): Handling Exceptions

Objective: Understand the Rationale and Types of Exceptions

Q85: Catching exceptions and errors

[85.](#) Is it possible to catch any checked exception, unchecked exception, or error programmatically? Choose the statement that is true.

- A. It is possible to catch any checked exception, unchecked exception, or error.
- B. It is not possible to catch any checked exception, unchecked exception, or error, because you cannot catch errors.

Hint: All exceptions and errors inherit from the `Throwable` class.

Reference: [Chapter 9](#): Handling Exceptions

Objective: Understand the Rationale and Types of Exceptions

Q86: Runtime exceptions

[86.](#) Identify the statements that correctly orient the exceptions to the superclasses. (Choose all that apply.)

- A. `NumberFormatException` is a subclass of `IllegalArgumentException`.
- B. `IllegalArgumentException` is a subclass of `NumberFormatException`.
- C. `IndexOutOfBoundsException` is a subclass of `ArrayIndexOutOfBoundsException`.
- D. `ArrayIndexOutOfBoundsException` is a subclass of `IndexOutOfBoundsException`.

Hint: `IllegalArgumentException` and `IndexOutOfBoundsException` are direct subclasses

to `RuntimeException`.

Reference: [Chapter 9](#): Handling Exceptions

Objective: Recognize Common Exceptions

Q87: Try-with-resources

[87.](#) What is the name of the interface that methods must implement to utilize the new try-with-resources feature?

- A. `AutoClose`
- B. `AutoCloseable`
- C. There is no such feature.

Hint: This feature was introduced in JDK 1.7.

Reference: [Chapter 9](#): Handling Exceptions

Objective: Alter the Program Flow

Q88: `IOException` classes

[88.](#) Which three classes are subclasses of the `IOException` class?

- A. `FileNotFoundException`
- B. `SQLException`
- C. `ClassNotFoundException`
- D. `InterruptedException`

Hint: Which one of the answers is a subclass of

`ReflectiveOperationException`?

Reference: [Chapter 9](#): Handling Exceptions

Objective: Recognize Common Exceptions

Q89: Exception reporting

[89.](#) Both the `getMessage` and `toString` methods return information about exceptions that are caught. Which method returns the basic information as well as the class name of the exception?

- A. getMessage()
- B. toString()
- C. getData()
- D. toOutput()

Hint: You may be most familiar with the `printStackTrace()` method, which returns the basic information, the class name, and a stack trace.

Reference: [Chapter 9](#): Handling Exceptions

Objective: Recognize Common Exceptions.

Q90: Unchecked exceptions

[90.](#) Which of the following code fragments will throw a `NumberFormatException`?

- A. `Integer.parseInt("INVALID");`
- B. `int e = (2 / 0);`
- C. `Object x = new Float("1.0"); Double d = (Double) x;`
- D. `String s = null; int i = s.length();`

Hint: The incorrect answers include a `NullPointerException`, an `ArithmaticException`, and a `ClassCastException`.

Reference: [Chapter 9](#): Handling Exceptions

Objective: Recognize Common Exceptions

ANSWERS

[1.](#) D. D is correct because the statement is false. Package names beginning with `javas.*` and `javaw.*` are NOT reserved and can be freely used.

However, the package names beginning with `java.*` and `javax.*` ARE reserved.

A, B, and C are incorrect answers because they are all true statements and the question was looking for the false statement.

[2.](#) B. To include all of the classes in the `java.util` package, the `import` statement is used, followed by the package name, including an asterisk.

A, C, and D are incorrect. A is incorrect because the `import` statement must specify a class directly, or in this case all of the classes with the asterisk wildcard. C is incorrect because of

the exclusion of the asterisk wildcard, along with the use of an invalid #include keyword. D is incorrect because of the invalid #include keyword, which is unique to the C and C++ programming languages.

3. B. Interfaces of the Collections API include the List, Map, Set, and Queue APIs. A, C, and D are incorrect. ArrayList, HashMap, HashSet, and PriorityQueue are all concrete classes and are considered to be implementations of their respective interfaces.
4. D. Class RandomAccessFile allows for the reading and writing of files to specified locations. A, B, and C are incorrect. A is incorrect because the File class provides a representation of file and directory pathnames. B is incorrect because the FileDescriptor class provides a means to function as a handle for opening files and sockets. C is incorrect because the FilenameFilter interface defines the functionality to filter filenames.
5. A. The javaw utility used on MS Windows machines will run the Java interpreter without a console window. B, C, and D are incorrect. B is incorrect because there is no interpw command. C is incorrect because there is no -wo switch. D is incorrect because jconsole is a management and monitoring application, not an interpreter.
6. E. ArrayList is located in the java.util package. A, B, C, D, and F are incorrect. The import statements in A, B, C, and D import packages that do not contain the ArrayList class. F is incorrect because ArrayList is not part of the java.lang package.
7. A and D. Swing and AWT are the two Java graphical toolkits and are included in the java.awt and javax.swing packages. B, C, and E are incorrect. B is incorrect because java.io is used for input and output type jobs. C is incorrect because java.net is used for network connections. E is incorrect because the java.util package contains classes such as Vector, ArrayList, Random, and many more, but no classes for building a GUI.
8. A. A Java class can extend only one superclass. B, C, and D are incorrect. B is incorrect because a Java class cannot extend multiple superclasses. C is incorrect because Java can extend a superclass. D is incorrect because a class will extend a class but implements an interface.
9. E. A package name should be your reverse domain name. A, B, C, and D are incorrect. A, B, and C are incorrect because they contain spaces that are not valid characters in package names. C and D are incorrect because java* is reserved for official Java packages.
10. B. javac is the Java compiler. It is used to compile a Java source file into a bytecode file. A bytecode filename has the .class extension. A, C, and D are incorrect. A is incorrect because the Java compiler will not interpret/execute the application. C is incorrect because the Java compiler does not generate Java source code. D is incorrect because there is nothing incorrect with the given syntax.

11. A, C, and D. The code will need to iterate through an array, test the condition to see if the array value equals a "nickel" string, and perform an assignment to increase the value of the counter. B is incorrect. Assertion statements are used to evaluate whether code is functioning as expected.

12. C and D. C contains an error because it needs a do-while statement, not an invalid does-while statement. D contains an error because the do-while statement must end with a semicolon.

A and B are incorrect. A is incorrect because declarations must occur outside of the do-while statement. As such, the declaration was represented correctly in the code segment. B is incorrect because when there is only one statement enclosed in the do-while statement, it does not need to be enclosed in braces.

13. A and C. Iteration statements and switch statements can include continue and break statements. Iteration statements, known as loops, include the for loop, enhanced for loop, while loop, and do-while loop.

B and D are incorrect. The only conditional statement that can include the break and continue statement is the switch statement. There is no logical reason to continue or break out of an expression statement.

14. C. Javadoc comments are included between /** and */.

A, B, and D are incorrect. A is incorrect because it shows a normal line comment. B is incorrect because it shows a normal block comment. D is incorrect because this Javadoc comment isn't terminated properly; it must terminate with */.

15. A. This implementation will not compile, because adding an int and a float together does not yield an integer literal as is expected for the assignment.

B, C, and D are incorrect. These answers all have code that will compile because the code has necessary implicit casts that occur within their expression and assignment statements.

16. C. This statement is not valid. The variable mostPoints is an int. It cannot be placed in an if statement without a relational operator.

Remember that the assignment of primitive literals evaluate to primitive values.

A, B, and D are incorrect. A is incorrect because the statement is valid—it is a simple assignment. B is incorrect because the code segment is valid. It is legal to have more than one variable assigned on a line as seen in the last line of B. The last line of this segment sets a equal to b, and then c equal to a. D is incorrect because the segment is valid. Because stillRunning is a boolean value, a relational operator is not needed. Note that while this answer is legal and compiles, it may not be what was intended, as relational operators are typically used in if statements.

17. D. A switch statement is best suited for situations where multiple possible conditions exist that must be accounted for.

A, B, and C are incorrect. A is incorrect because the if-else statement only allows two conditions and this question calls for ten conditions. B and C are incorrect because a

combination of `if` statements should be used only when limited conditions are being checked.

18. C. To create or set an object to `null`, the `null` keyword can be used. Objects are also set to `null` when they are declared but not yet initialized.
A, B, and D are incorrect. A is incorrect because when the `new` keyword is used, it initializes the object to a new non-null object. B is incorrect for the same reason as A, plus the `Object` constructor does not accept any arguments. D is incorrect because `null()` is a made up method and is invalid in Java.
19. A. The expression in the `if` statement evaluates to true. This expression is actually easier than it looks considering that `bool1` is on the left side of a short-circuit operator.
B is incorrect.
20. D. This is a trick question. Notice the quotation marks used to initialize the `lineToDisplay` variable. The quotation marks indicate that everything between them is passed as a string. This is not concatenating the variables, as it might seem at first. The `trim` method removes whitespace from the beginning and end of a string. The string is everything between the quotation marks. Since the string neither starts nor ends with spaces, nothing is trimmed.
A, B, and C are incorrect. If the quotation marks were not present, A would have been correct.
21. D. The integer declarations supplied in answer D causes the printing of `fragile!`
A, B, and C are incorrect. A is incorrect because the integer declarations supplied causes the printing of `ifragips`. B is incorrect because the integer declaration causes a `StringIndexOutOfBoundsException` exception to be thrown since there are only 34 characters to be accessed, not 35. C is incorrect because the integer declaration causes the printing of `ifragilcu`.
22. A. The method is called `trim` and it receives no arguments.
B, C, and D are incorrect. B and C are incorrect because these methods inappropriately try to receive arguments. D is incorrect because there is no `trimWhiteSpace` method.
23. A. In order of precedence, the following is correct: multiplication (*), addition (+), conditional AND (`&&`), assignment (=).
B, C, and D are incorrect because they are improperly ordered.
24. B. $24 \% 8 = 0$, $10 \% 7 = 3$, $100 \% 99 = 1$, $38 \% 6 = 2$. Added together, $0 + 3 + 1 + 2 = 6$.
A, C, and D are incorrect because they do not represent the logical answer.
25. A. Strings are immutable, so the statement at line 2 does not affect the string; it's as if nothing ever happened. Explicitly overwriting the value of the string object would have changed the result:
`string = string.replace("Dollar bill", "Silver dollar");`
B, C, and D are incorrect. B is incorrect because the condition results in true. C and D are incorrect because the code compiles fine.
26. A. The append declaration that includes a `short` is not included in the `StringBuilder` class. This is because the `short` value is automatically casted to an integer when passed for integer

literal. Therefore, the following declaration will also handle a short argument: public
StringBuilder append (int i) {...}
B, C, D, and E are incorrect. B is incorrect because the append declaration that includes an int is included in the StringBuilder class. C is incorrect because the append declaration that includes a long is included in the stringBuilder class. D is incorrect because the append declaration that includes a float is included in the StringBuilder class. E is incorrect because the append declaration that includes a double is included in the StringBuilder class.

27. B. Precedence occurs first with the enclosed $1+1$ equation. The evaluation then continues at the beginning of the expression with the string, and each value after the string is appended as a string, resulting in A121.

A, C, and D are incorrect because the resultant output is A121. E is incorrect because there is no compilation error.

28. D. The code will not compile because the matches method is not a method of the StringBuilder class. The matches method is a method of the String class, but that does not apply here.

A, B, and C are incorrect, because the code does not compile. If the matches method was removed from the chain, the word musical would be printed.

29. A. Since conditional OR (`||`) is a short-circuit operator, as soon as the left operand evaluates to true, the right operand is not evaluated.

Therefore, `true, true` is printed to standard out.

B, C, and D are incorrect because `true, true` is printed to standard out.

30. C. Both values of `null` are the same, so the evaluation prints "true". In this scenario, the method `equalsIgnoreCase` provided more functionality than was needed, and could have been replaced with just the `equals` method to return "true".

A, B, and D are incorrect. A is incorrect because there are no `null` values. The values are actually strings assigned the character sequence of "null". B is incorrect because `equalsignoreCase` is a valid method name so no compilation error will occur. D is incorrect because "true" is printed.

31. D. D is correct relative to the expected behavior of the decrement and increment operators
A, B, and C are incorrect.

32. B and C. Both the `StringBuffer` class and `StringBuilder` class create strings that are mutable.

A is incorrect. The `String` class creates immutable strings.

33. C. An array is used to store multiple variables that are the same data type variables.

A, B, and D are incorrect. A is incorrect because primitives can make up the contents of an array but an array is required to hold them. B is incorrect because an object may be used to simulate an array but it is much more complex than using just an array. D is incorrect because enumerations are used to limit the possible values that can be stored in a variable to a predefined list.

34. C, D, and F. C is correct because an `int` is used to store an integer number. D is correct because the compiler will automatically convert 3 to a floating-point number. F is also correct because when an `f` is appended to a number, it implies that that number is a floating-point number, even if it does not have a decimal.
- A, B, and E are incorrect. A is incorrect because an `int` cannot store a decimal number. B is incorrect because `3.0f` would be a valid literal for a float, but `3.0` would not (the compiler treats `3.0` as a double). E is incorrect because an `int` cannot have `f` appended to it.
35. A. Valid literal values for the `boolean` primitive include `true` and `false`.
- B, C, and D are incorrect. `TRUE`, `FALSE`, and `null` are all invalid literals for the `boolean` primitive. The `null` value is a valid literal for the `Boolean` wrapper class.
36. C and D. C and D are correct because the associated code segment will not compile. The code in C does not compile because `NICKEL` by itself cannot be resolved; the code must read `Coin.NICKEL`. The code in D does not compile due to the case mismatch between `nickel` and `NICKEL`.
- A and B are incorrect because they are all valid, compilable code segments.
37. B. B is correct because the proper naming convention is not used. All variables should start with a lowercase letter—thus, `IntName` should read `intName`.
- A, C, and D are incorrect because they make use of the proper naming conventions. Variable and method names should appear in all lowercase characters, using camel case where appropriate. Class, interface, and enumeration names should start with an uppercase character, followed by lowercase characters and camel case where appropriate. Note that “camel case” refers to the capitalization of the first letter of appropriate words within an identifier.
38. B. The underscore is used to identify places in numeric values.
- A, C, and D are incorrect because `x`, `^`, and `-` are incorrect; they are not used to identify places in numeric values.
39. D. The wrapper class reference types use the full spelling of the type names beginning with a capital letter. The full set of wrapper classes is `Boolean`, `Character`, `Byte`, `Short`, `Integer`, `Long`, `Float`, and `Double`.
- A, B, and C are incorrect. Classes named `Char` and `Int` do not exist.
- The appropriate wrapper classes are named `Character` and `Integer`.
40. A, C, and D. The `170` in A, the `10` in C, and the "current `HP`: " in D are all literals.
- B is incorrect because this statement does not contain any values that are literals.
41. B. It uses the correct syntax to declare and create a new object.
- A, C, and D are incorrect. A is incorrect because it declares a `Penguin` object but does not initialize it. C is incorrect because `[]` is used when it should be `()`. D is incorrect because it omits the `new` keyword.
42. A, B, and C. Primitive types include `int`, `boolean`, and `char`. Notice that they all start with a lowercase letter.

D and E are incorrect. D is incorrect because `Float` is the wrapper class for a `float`. E is incorrect because `String` is a reference to the `String` class. The capital F in `Float`, and in any other data type, should signal that this is an object.

43. D. The array declaration and element assignments are used appropriately and compile without error.

A, B, and C are incorrect. A compiles with an `ArrayIndexOutOfBoundsException` exception thrown at runtime. This is because the array index `characterName[3]` is out of bounds. B fails to compile because only an integer is expected in the box brackets. C fails to compile because the box brackets that are needed to create the array are missing.

44. D. Class (static) variables are referenced by the class name and variable name, delineated by a period.

A, B, and C are incorrect. A and B are incorrect because variables are not qualified by a class name. C is incorrect because when referencing static variables, colons are not used for delineation.

45. D. Local variables are declared inside of methods and are not initialized with a default value.

A, B, and C are incorrect because they do not represent true statements about the properties of local variables.

46. D. Constructors do not declare a return type, and therefore cannot return a value.

A, B, and C are incorrect because they are all method declarations.

47. C. This is a perfectly formed method declaration.

A, B, and D are incorrect. A is incorrect because the declaration states that no value (`void`) is to be returned, but a string is. B is incorrect because the wrapper class for character is `Character` and not `char`. D is incorrect because the return type of `String` does not start with a capital letter as it should. That is, "public string doMethod" should read "public String doMethod".

48. I. The following modifiers cannot be used with constructors: `abstract`, `final`, `native`, `strictfp`, `static`, `synchronized`, `transient`, and `volatile`.

A, B, C, D, E, F, G, and H are incorrect, relative to the correct answer.

49. B. A constructor that does not include an access modifier will make use of the same access modifier that is used for its class.

A and C are incorrect. A is incorrect because constructors that do not include an access modifier are not always declared as `public`; they are declared the same as their class. C is incorrect because a compilation error will not occur if a constructor does not have an access modifier.

50. C. An instance variable is declared in the class, and outside of any method. It is in scope for the entire class, and its lifecycle is tied to that class.

A and B are incorrect. A is incorrect because local variables are in scope for the block of code in which they are declared. B is incorrect because method parameters are in scope for the entire method.

51. C. Instance variables retain their value for the life of the object.
A, B, and D are incorrect. A is incorrect because a local variable is used for temporary items and stays in scope only until the block of code it is declared in is exited. B is incorrect because method parameters are the variables passed to a method as arguments. They are in scope only for that method. D is incorrect because the object variable does not exist.
52. B. Objects are always passed by reference. This means that a reference to the memory location of the object is passed to the method as opposed to a copy of the object.
A, C, and D are incorrect. A is incorrect because primitives are passed by value; this means that a copy of the variable is passed to methods. C and D are incorrect because copy and parameter are not valid ways to pass a variable.
53. D. This method signature uses the `void` keyword to signify that there isn't any data to return, and it contains the proper variable names.
A, B, and C are incorrect. A is incorrect because it is missing both the `void` keyword and the variable names. B is incorrect because it is missing the `void` keyword, and C is incorrect because it is missing the variable names.
54. A and C. A is correct because the `void` keyword means the method will not return any data. C is correct because the data type of the variable to be returned must precede the method name.
B and D are incorrect. B is incorrect because `void` is not optional if the method isn't returning anything. The `void` keyword must be included in this condition. D is incorrect because a method must always declare a return type or `void`. If it is a constructor, it can omit a return type and it will be implied as `void`.
55. B. In Temporary association indicates that object A “uses-an” object B.
A, C, and D are incorrect. A is incorrect because in direct association, one object “has-a” object. C is incorrect because in composition association, an object is “composed-of” another object. D is incorrect because in aggregation association, an object is “part-of” another object.
56. B. B is correct because it represents a false statement. Composition association is a stronger association than aggregation association.
A, C, and D are incorrect because they represent true statements. Composition association is a class relationship that models a whole-part relationship—also referred to as a “composed-of” relationship.
57. C. Temporary association, also known as dependency, exists when an object is temporarily used in a return statement.
A, B, and D are incorrect. Composition (containment), realization, and direct association (navigability) do not apply here.
58. A, B, and D. Each of these associations is considered weak. This means they do not have any lifecycle responsibility, and if the relationship was lost, each object would still maintain its meaning.
C and E are incorrect. C is incorrect because composition association has a strong relationship.

E is incorrect because complete association is a made-up term.

59. B, C, D, and E. B is correct because composition relationships are a strong relationship. C is correct because the object that is composed of the other objects has a lifecycle responsibility for these objects. D and E are correct because one object can be composed of only another single object, or an object can be composed of many objects, like an array.

A and F are incorrect. A is incorrect because composition association is a strong relationship, not weak. F is incorrect because it is impossible to have a many-to-many relationship that is also a strong relationship.

60. A. Direct association is a “has-a” relationship.

B, C, and D are incorrect. B is incorrect because temporary association is known as a dependency. C is incorrect because composition association is a “composed-of” relationship. D is incorrect because aggregation association is a “part-of” relationship.

61. C. The `Store` object is “composed-of” the `MerchandiseItem` object. There is a strong relationship between the two classes, and the `Store` object has a lifecycle responsibility for the `MerchandiseItem` object.

A, B, and D are incorrect.

62. B. One `Album` object contains three `Picture` objects: one-to-many.

A, C and D are incorrect. A is incorrect because the array being contained in `Album` means this is a many relationship on at least one side. C is incorrect because this answer is impossible since the relationship is a composition. A composition association cannot be many-to-many. D is incorrect as there is no such thing as a many-to-most multiplicity.

63. B. An interface can extend more than one interface.

A is incorrect, because an interface can extend more than one interface.

64. A and D. A concrete class can extend only one class. An interface can extend multiple interfaces.

B and C are incorrect.

65. B and C. Abstract classes and interfaces can have abstract methods.

A and D are incorrect. Concrete classes cannot have abstract methods. Packages in this context is an illogical answer.

66. B. Using the `private` access modifier in conjunction with getter and setter methods provides excellent encapsulation.

A, C, and D are incorrect. The package-private, `protected`, and `public` access modifiers do not provide for tight encapsulation.

67. B. B is correct because it's a false statement. Abstract classes cannot be instantiated.

A, C, and D are incorrect because they are true statements. An abstract class must include the keyword `abstract` in its declaration. Abstract classes contain a mixture of implemented and abstract methods. When an abstract class is extended, all of its abstract methods must be

implemented by the subclass.

68. C, D, and E. The `private` and `protected` access modifiers can be used with constructors, methods, and data members.

A and B are incorrect. The `private` and `protected` access modifiers cannot be used with classes and interfaces.

69. B is correct. An abstract class may contain implemented or unimplemented methods. An unimplemented method is also called an abstract method.

A, C, and D are incorrect. A is incorrect because a concrete class is a normal Java class that has all of its methods implemented. C is incorrect because the term “Java class” is a general term and does not apply to a specific type of class. D is incorrect because an interface is used to create a public interface for a class. It may not contain any implemented methods.

70. A and E. A class that gains the functionality of other classes is called either a subclass or child class.

B, C, and D are incorrect. Superclass, base class, and parent class are all terms to describe the class that is being inherited.

71. D. A getter should be start with `get`, with a lowercase letter g, followed by the name of the variable starting with a capital letter.

A, B, and C are incorrect. A is incorrect because it is missing `get`. B is incorrect because the G in `Get` is capitalized, and it should be lowercase. C is incorrect because the p in `points` is lowercase and should be capitalized.

72. B. Instance variables should have the most restrictive modifier, which is `private`.

A, C, and D are incorrect. All of these access modifiers, including the default access level, are less restrictive than `private`.

73. C. An interface is implemented.

A, B, and D are incorrect. A and D are incorrect because `inherits` and `uses` are made-up terms. B is incorrect because `extends` is used for classes, not interfaces.

74. D. The code example is polymorphic.

A, B, and C are incorrect. Encapsulation, data abstraction, and information hiding are not represented in the code example.

75. B. Java’s Collections API allows for extensive use of polymorphism.

A, C, and D are incorrect. Although polymorphism may be used in some areas of other Java APIs, the Collections API uses the feature quite commonly.

76. C and D. C is correct as an explicit `short` is applied. D is correct as an implicit `short` is applied.

A and B are incorrect because you cannot use wrapper classes for explicit casts in this manner.

77. B. It is not possible to cast a primitive `boolean` type to another primitive type.

A is incorrect, because it is not possible to cast a boolean type to another primitive type.

78. A. The widening and narrowing that occurs in the declaration is all acceptable. Also, implicit casts occur in the `print` statement.
B and C are incorrect as the code compiles fine without any compilation issues.
79. A. The statement “program to an interface” means that objects should be referenced by the interfaces that they implement. This creates more modular code.
B, C, and D are incorrect. None of these statements describe the concept.
80. A and D. A is correct because polymorphism allows objects to act as other objects. The other objects are more general forms of the original object. D is correct because polymorphism also allows different classes that share some common functionality to be treated the same. They must all implement the same interface.
B and C are incorrect. B is incorrect because less code may be required, but this is not always the case. C is incorrect because objects can be referenced only in their more general form. An object cannot be used as a more specific object that extends it later.
81. A and B. Polymorphism allows an object to be referenced by a more general data type. A more general data type is either a base class it has extended or an interface it has implemented.
C, D, and E are incorrect. C and D are incorrect because an instance variable is a type of variable and has nothing to do with polymorphism. E is not correct because polymorphism cannot work when moving to more specific objects.
82. A, C, and D. A is correct because `Store` is a base class for `FoodStore`. C is correct because `Object` is a base class for `FoodStore`. The `Object` class is always the base class for every object. `Store` implicitly extends `Object`, `Store` is extended by `FoodStore`, and so `FoodStore` extends `Object`. D is correct because `Store` implicitly extends `Object`.
B is incorrect because a `FoodStore` object cannot be initialized with a more general class.
83. D. This is an example of programming to an interface. The `startPlaying` method will expect as an argument any object that implements that `Audible` interface.
A, B, and C are incorrect. A is incorrect because `Audible` is an interface, not a class. B is incorrect because this example is already using polymorphism. C is incorrect because `Object` is the implied base class for all objects.
84. C. `Assertions` is not an exception category.
A, B and D are incorrect as `Check Exceptions`, `Unchecked Exceptions`, and `Errors` are all exception categories in Java.
85. A. It is possible to catch any checked exception, unchecked exception, or error.
B is incorrect. All exceptions and errors can be caught, as they all inherit from the `Throwable` class. However, errors typically are not recoverable.
86. A and D. `NumberFormatException` is a subclass of `IllegalArgumentException`.
`ArrayIndexOutOfBoundsException` is a subclass of `IndexOutOfBoundsException`.

B and C are incorrect. B is incorrect because `IllegalArgumentException` is not a subclass of `NumberFormatException`, because `IllegalArgumentException` is the a superclass of `NumberFormatException`. C is incorrect because `IndexOutOfBoundsException` is not a subclass of `ArrayIndexOutOfBoundsException`, because `IndexOutOfBoundsException` is a superclass of `ArrayIndexOutOfBoundsException`.

87. B. The `AutoCloseable` interface establishes a method definition `close()` that is used to close resources. This feature is used in conjunction with the new try-with-resources feature of Java SE 7.

A and C are incorrect. A is incorrect because there is no interface named `Autoclose`. C is incorrect because the try-with-resources feature does exist with JDK 1.7.

88. A, B, and D. `FileNotFoundException`, `SQLException`, and `InterruptedException` are all subclasses of `IOException`.

C is incorrect. `ClassNotFoundException` is not a subclass of `IOException`; it is a subclass of `ReflectiveOperationException`.

89. B. The `toString` method returns the basic information of an exception being caught as well as the class name—for example, “`java.io.FileNotFoundException: ... (The filename, directory name, or volume label syntax is incorrect)`”.

A, C and D are incorrect. A is incorrect because the `getMessage` method returns only the basic information of an exception being caught, not the class name—for example, “`... (The filename, directory name, or volume label syntax is incorrect)`”. C is incorrect because there is no `getData` method. D is incorrect because there is no `toOutput` method.

90. A. Evaluation of the statement causes a `NumberFormatException` to be thrown.

B, C, and D are incorrect. B is incorrect because evaluation of the statement causes an `ArithmaticException` to be thrown. C is incorrect because evaluation of the statement causes a `ClassCastException` to be thrown. D is incorrect because evaluation of the statement causes a `NullPointerException` to be thrown.

Appendix I

About the Download

This e-book includes a practice exam in standard text format, plus a free downloadable MasterExam practice exam and Enterprise Architect Project File.

MasterExam requires free online registration. The MasterExam software is easy to install on any Windows 2000/XP/Vista/7 computer and must be installed to access MasterExam. To download MasterExam, simply use the link below and follow the directions for the free online registration.

www.osborne.onlineexpert.com

To access the other available downloads, visit McGraw-Hill Professional's Media Center by clicking the link below and entering this e-book's ISBN and your e-mail address. You will then receive an e-mail message with a download link for the additional content.

<http://mhprofessional.com/mediacenter/>

This e-book's ISBN is 978-0-07-178944-8.

Because you purchased a McGraw-Hill e-book, you are also entitled to incredible savings on LearnKey online training courses. When you purchase the complete course you'll get 12 months access to all of the appropriate training materials including media-rich courseware, test prep software, study plans, reference material, and more. This special promotion is offered for a limited time and is available only to customers in the 50 United States. Click the Online Expert link above to explore this offer from LearnKey.

Steps to Download MasterExam

First, complete the free registration. Then, take the guided tour if you're new to Online Expert. Next, click the Student Login link on the home page and log in as a student. When you've logged in, click on the course that matches the title of your e-book, click the MasterExam tab in the upper right, and then follow the instructions to download your MasterExam.

System Requirements

Software requires Windows 2000 or higher and Internet Explorer 6.0 or above and 20 MB of hard disk space for full installation.

MasterExam

MasterExam provides you with a simulation of the actual exam. The number of questions, the type 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 lower right-hand corner of your screen. The clock will continue to count down to zero unless you choose to end the exam before the time expires.

Enterprise Architect Project File

Enterprise Architect (EA) is a CASE tool that supports UML modeling and reverse source code engineering. EA was used to create the draft UML diagrams for this book. Since knowing UML is a requirement for the exam, the authors have included the project file for these diagrams in the download to assist you in your learning.

To open up the project file you must have a version of EA. You can download a 30-day trial version of the application from the Spark Systems website at <http://www.sparxsystems.com/products/ea/trial.html>. Once installed, you will be able to view and modify each of the UML diagrams. You will find the diagrams to be organized in the EA project as they are presented in each chapter. You can reference [Appendix G](#) for detailed information on UML.

Help

A help feature for the MasterExam software is available through MasterExam.

Removing Installation

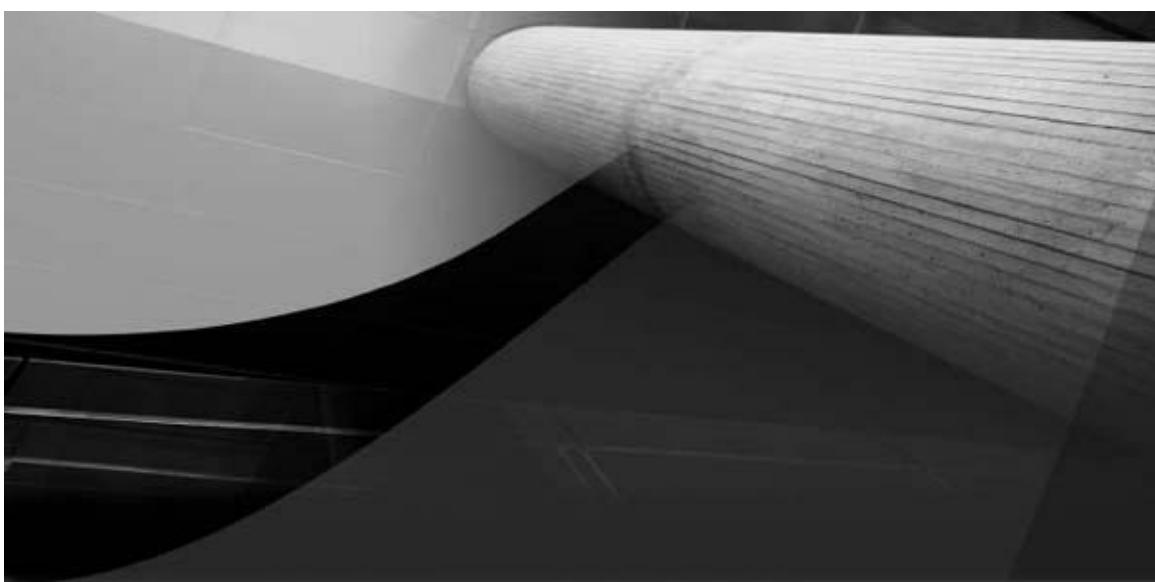
MasterExam is installed to your hard drive. For best results removing LearnKey programs, use the Start | All Programs | LearnKey Uninstall option.

Technical Support

For questions regarding the content of the download or MasterExam, please visit www.mhprofessional.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), please visit www.learnkey.com, e-mail techsupport@learnkey.com, or call toll free (800) 482-8244.



Glossary

absolute path The full path to a directory or file from the root directory. For example, C:\NetBeansProject\SampleProject is an absolute path. The paths ..\ SampleProject or Projects\ SampleProject are relative paths.

abstract A modifier that indicates that either a class or a method has some behavior that must be implemented by its subclasses.

access modifiers Modifiers that define the access privileges of interfaces, classes, methods, constructors, and data members. Access modifiers include *package-private*, **private**, **protected**, and **public**.

accessor method A method used to return the value of a private field. See also *getter*.

application server A (Java EE) server that hosts various applications and their environments.

array A fixed-length group of the same type variables or references that are accessed with an index.

ArrayList A resizable array implementation of the List interface; an object-oriented representation of an array.

arithmetic operator Java programming language operator that performs addition (+), subtraction (-), multiplication (*), division (/), or remainder production (%) operations.

assertions Boolean expressions that are used to validate whether code functions as expected while debugging.

assignment statement A statement that allows for the definition or redefinition of a variable by assigning it a value. It is represented by the equal operator (=) in Java code.

attributes The state of a class's instance and static variables (fields).

autoboxing A convenience feature in Java that allows a primitive to be used as its wrapper object class without any special conversion by the developer.

AutoCloseable A Java interface that represents a resource that must be closed when it is no longer needed. Objects that inherit from AutoCloseable may be used with the try-with-resources statement.

base class See *superclass*.

bean A reusable software component that conforms to design and naming conventions.

bitwise operator A Java programming language operator that may be used to compare two operands of numeric type or two operands of type boolean. Bitwise operators include bitwise AND (&), bitwise exclusive OR (^), and bitwise inclusive OR (|).

block Code placed between matching braces—for example, { int x; }.

boolean A Java keyword (boolean) that is used to define a primitive variable as having a Boolean type with a value of either true or false. The corresponding wrapper class is Boolean.

byte A Java keyword (byte) used to define a primitive variable as an integer with 1 byte of storage. The corresponding wrapper class is Byte.

casting Converting one type to another, such as converting/casting a primitive int to a primitive long.

char A Java keyword (char) used to define a variable as a specific Unicode character with 2 bytes of storage. The corresponding wrapper class is Character.

Checkstyle A development tool that assists programmers write Java code that adheres to coding standards. See <http://checkstyle.sourceforge.net/> for information.

child class See *subclass*.

class A Java type that defines the implementation of an object. Classes include instance variables, class variables, and methods. Classes also specify the superclass and interfaces in which they inherit and implement. All classes inherit from the Object class.

class variable See *static variable*.

classpath An environment variable that includes an argument set telling the Java virtual machine where to look for user-defined classes/packages.

comment Text within source files that provides explanations of associated code. In Java, comments are delimited with // (single line), /*...*/ (multiple line), or /**...*/ (multiple line) Javadoc comments.

composition association A whole–part relationship between classes whereby the whole is responsible for the lifetime of its parts. Composition is also known as *containment* and is a strong relationship.

compound assignment operator An operator with an abbreviated syntax that is used in place of the assignment of an arithmetic or bitwise operator. The compound assignment operator evaluates the two operands first, and then assigns the results to the first operand.

concatenation operator An operator (+) that is used to concatenate (that is, join) two strings.

concrete class A class that has all of its methods implemented. Concrete classes can be instantiated.

conditional statement A decision-making control flow used to execute statements and blocks of statements conditionally. Examples are `if`, `if else`, `if else if`, and `switch`.

constructor A method that initializes a new object.

declaration A statement that establishes an identifier with associated attributes. A class declaration lists its instance variables and methods. Declarations within methods define the type of local variables.

double A Java keyword (`double`) that is used to define a primitive variable as a floating-point number with 8 bytes of storage. The corresponding wrapper class is `Double`.

encapsulation The principle of defining (designing) a class that exposes a concise public interface while hiding its implementation details from other classes.

enumeration type A type with a fixed set of constants as fields.

Error A subclass of the class `Throwable`. The `Error` class indicates issues that an application should not try to catch.

Exception A subclass of the class `Throwable`. The `Exception` class indicates issues that an application may want to catch.

Executable JAR A JAR file containing a `manifest.mf` in `META-INF` directory with the `Main-Class` attribute set to a class with a `public static void main(String args[])` method.

expression statement A statement that changes part of the application's state. Expression statements include method calls, assignments, object creation, pre/post-increments and pre/post-decrements. An expression statement can be evaluated to a single value.

FindBugs A program that uses static analysis to look for bugs in Java code; see findbugs.sourceforge.net for more information.

float A Java keyword (`float`) that is used to define a primitive variable as a floating-point number with 4 bytes of storage. The corresponding wrapper class is `Float`.

getter A simple public method used to return a private instance variable.

Git An open-source distributed version control system.

heap A memory area where objects are stored.

IDE Acronym for *integrated development environment*. A development suite that allows developers to edit, compile, debug, connect to version control systems, collaborate, and do much more depending on the specific tool. Most modern IDEs have add-in capabilities for various software modules to enhance capabilities. Popular IDEs include the NetBeans IDE, JDeveloper, Eclipse, and IntelliJ IDEA.

import statement A statement used in the beginning of a class that allows for external packages to be made available within the class.

inheritance The ability of one Java class to extend another and gain its functionality.

instance variable A variable that is declared in the class instead of in a particular method. This variable has a life cycle that lasts for the duration of the object's existence. This variable is in scope for all methods.

int A Java keyword (`int`) that is used to define a primitive variable as an integer with 4 bytes of storage. The corresponding wrapper class is `Integer`.

interface A definition of public methods that must be implemented by a class.

iteration statement A control flow through which a statement or block of statements is iterated, based on a maintained state of a variable or expression. The `for` loop, enhanced `for` loop, and the `while` and `do-while` statements are used for iterating.

J2EE Acronym for *Java 2 Platform, Enterprise Edition*. The legacy term for Java EE. See Java EE.

J2ME Acronym for *Java 2 Platform, Micro Edition*. The legacy term for Java ME. See Java ME.

J2SE Acronym for Java 2 Platform, Standard Edition. The legacy term for Java SE. See Java SE.

JAR Acronym for *Java archive*. A JAR file is used to store a collection of Java class files. It is represented by one file with the `.jar` extension in the file system. It may be executable. The JAR file is based on the ZIP file format.

Java EE Acronym for *Java Platform, Enterprise Edition*. A software development platform that includes a collection of enterprise API specifications for Enterprise JavaBeans, servlets, and JavaServer Pages. Java EE compliance is reached when an application server (full compliance) or web container (partial compliance) implements the necessary Java EE specifications.

Java ME Acronym for *Java Platform, Micro Edition*. A software development platform that includes a collection of APIs designed for embedded devices.

Java SE Acronym for *Java Platform, Standard Edition*. A software development platform that includes a collection of APIs designed for client application development.

JavaBean A reusable Java component based on a platform-independent reusable component model

in which a standardized means is used to access and modify the object state of the bean.

Javadoc A tool that produces HTML documentation from extracted comments of Java source code. The comment symbols and annotations in the comments must comply with the Javadoc specification.

JavaFX A rich client platform that provides a lightweight, hardware-accelerated Java UI platform for enterprise business applications.

jConsole A monitoring and management tool that is a part of the JDK. It enables the local or remote monitoring of an application to track CPU as well as memory usage. It connects and uses information provided via the Java Management Extension (JMX).

JDK Acronym for *Java Development Kit*. A bundled set of development utilities for compiling, debugging, and interpreting Java applications. The Java Runtime Environment (JRE) is included in the JDK.

JRE Acronym for *Java Runtime Environment*, an environment used to run Java applications. It contains basic client and server JVMs, core classes, and supporting files.

JSR Acronym for *Java Specification Request*. Formalized documents that are used to request the inclusion of new technologies and specifications into the Java platform.

JVM Acronym for *Java virtual machine*, the platform-independent environment where the Java interpreter executes.

keyword A word in the Java programming language that cannot be used as an identifier (in other words, a variable or method name). Java SE 6 maintains 50 keywords, each designed to be used for a specific purpose.

library Set of compiled classes that add functionality to a Java application.

literal A value represented as an integer, floating point, or character value that can be stored in a variable. For example, 1115 is an integer literal, 12.5 is a floating-point literal, and A is a character literal.

local variable A variable that is in scope only for a single method, constructor, or block.

logical operator Java programming language operators that perform logical operations, such as the Boolean NOT (!), conditional AND (&&), and conditional OR (||).

long A Java keyword (`long`) used to define a primitive variable as an integer with a storage of 8 bytes. The corresponding wrapper class is `Long`.

method A procedure that contains the code for performing operations in a class.

method argument A variable that is passed to a method. A method may have multiple arguments, or none.

method parameter A variable that is in scope for the entire method. It is declared in the method

signature and is initialized from the method arguments.

modulus The remainder production operator (%).

multi-catch A clause that allows for multiple exception arguments in one catch clause.

mutator method A method used to set the value of a private field. See *setter*.

null A Java keyword (*null*) representing a reserved constant that points to nothing. Specifically, a null type has a null (void) reference represented by the literal *null*.

object An instance of a class created at runtime from a class file.

object-oriented The design principle that uses objects and their interactions to design applications.

operator A Java element that performs operations on up to three operands and returns a result.

operator precedence The order of which operators will be evaluated when more than one operator are included in an expression.

overloading The process of implementing more than one method with the same return type and name, while using various numbers and/or types of parameters to distinguish between them.

overriding The process of overriding a superclass's method by using the same method signature.

package A Java keyword (package) that begins a statement at the beginning of a class. This statement indicates the package name with which it is associated. The fully qualified name for a class includes this package name.

package-private modifier The default modifier that allows package-only access to the associated class, interface, constructor, method, or data member.

parent class See *superclass*.

pass-by-reference The action of passing an argument to a method in which the Java virtual machine gives the method a reference to the same object that was passed to it. This is how objects are passed.

pass-by-value The action of passing an argument to a method in which the Java virtual machine copies the value for the method. This is how primitives are passed.

polymorphism A concept that allows data of one type to be handled and referred to by a type that is more general. Generalities can be created by using inheritance and extending classes, or by implementing interfaces.

POSIX Acronym for *Portable Operating System Interface*. It is a group of standards defined by IEEE (Institute of Electrical and Electronics Engineers) that cover APIs and command line utilities.

postfix increment/decrement operator These operators provide a shorthand way of incrementing and decrementing the value of a variable by 1, after the expression has been evaluated.

prefix increment/decrement operators Operators that provide a shorthand way of incrementing and decrementing the value of a variable by 1, before the expression has been evaluated.

primitive A fundamental data type that is not an object. Examples include, but are not limited to `int`, `float`, and `boolean`.

primitive cast A technique in Java of changing the primitive data type of a variable to another primitive type.

private access modifier A Java keyword that allows class-only access to the associated constructor, method, or data member.

profile A term used in Java ME to describe more specific features that a Java virtual machine target implements.

protected access modifier A Java keyword that allows package-external subclass access and package-only access to the associated constructor, method, or data member.

pseudo-code A structured means to allow algorithm designers to express computer programming algorithms in a human-readable format.

public access modifier A Java keyword that allows unrestricted access to the associated class, interface, constructor, method, or data member.

relational operator A Java programming language operator that performs relational operations such as less than (`<`), less than or equal to (`<=`), greater than (`>`), greater than or equal to (`>=`), value equality (`==`), and value inequality (`!=`).

relative path A path that is not anchored to the root of a drive. Its resolution depends upon the current path. For example, `../usr/bin` is a relative path. If the current path was `/home/user/Documents`, this path would resolve to `/home/usr/bin`.

RuntimeException A subclass of the class `Exception`. The `RuntimeException` class is the superclass of those exceptions that can be thrown during normal runtime of the Java virtual machine.

scope The block of code in which a variable is in existence and may be used.

setter A simple public method that accepts one argument and is used to set the value of an instance variable.

short A Java keyword (`short`) that is used to define a primitive variable as an integer with a storage of 2 bytes. The corresponding wrapper class is `Short`.

statement A command that performs an activity when executed by the Java interpreter. Common Java statement types include expression, conditional, iteration, and transfer-of-control statements.

static variable A variable that is declared in the class, such as an instance variable. This variable, however, is common to all objects of the same type. Only one instance of this variable exists for all objects of a particular type. Each instance of the class shares the same variable.

String class A class representing an immutable character string.

StringBuilder The `StringBuilder` class represents a mutable sequence of characters.

subclass A class that is derived from another class through inheritance. Also called a *child class*.

super A Java keyword (`super`) used to invoke overridden methods.

superclass A class used to derive other classes through inheritance. Also called a *parent class* or *base class*.

SVN An open source version control system.

Swing API A rich GUI application programming interface complete with an event model that is used for creating and managing user interfaces.

this A Java keyword that is used to assist in referring to any member of the current object. The `this` keyword must be used from within an instance method or constructor.

transfer-of-control statement A statement used to change the controlled flow in an application. Transfer-of-control statements include `break`, `continue`, and `return` statements.

try-catch A statement that contains code to “catch” thrown exceptions from within the `try` block, either explicitly or propagated up through method calls.

try-catch-finally A `try-catch` statement that includes a `finally` clause.

try-finally A statement that contains a `finally` clause that is always executed after successful execution of the `try` block.

try-with-resources A statement that declares resources that can be automatically closed. The objects/resources that are declared must implement `AutoCloseable`.

unboxing A convenience feature in Java that allows a primitive wrapper object to be used as its native primitive without any special conversion by the developer.

Unicode character A 16-bit set of characters. The character set makes up the Unicode standard.

Unicode Standard A character coding system designed to form a universal character set. The Unicode Standard is maintained by the Unicode Consortium standards organization.

variable A term for a symbolic reference to data in Java code.

XML Acronym for *Extensible Markup Language*. A general-purpose specification used for creating markup languages. This specification allows for the creation of custom tags in structured text files. Web-based solutions make common use of XML files as configuration, deployment descriptor, and tag library files.

SYMBOLS

! (logical negation operators)
!= (not equal to operator)
" " (double quotes)
% (modulus operator)
&& (logical AND operator)
' ' (single quotes)
() (parentheses)
 conventions using
 listing method parameters in
 overriding operator precedence with
 return string lengths using
* (multiplication operator)
+ (plus sign)
 as addition operator
 as concatenation operator
++x operator
+= (assignment by addition)
- (subtraction operator)
--x operator
-= (assignment by subtraction)
, (comma)
. (dot)
[] (box brackets)
 conventions using
 indicating arrays with
 using in multi-dimensional array declarations
/ (forward slash)
 using as division operator
/* */
/** */
: (colon)
:= (pseudo-code assignment)
; (semicolon)
< (less than operator)
<= (less than or equal to operator)
< > (angle brackets), conventions using
<< >> (Guillemet characters)
= (assignment operator)
== (equals to operator)
> (greater than operator)
>= (greater than or equal to operator)
@override
_ (underscore), in numeric literals
{ } (braces)

conventions using
defining local variables within
enclosing multiple statements in
|| (logical OR operator)

A

abstract classes
defined
demonstrating inheritance with
depicting in UML
implementing interfaces
study materials on
using polymorphism with
abstract keyword
abstract methods
access modifiers
about
depicting in UML
design principles using
encapsulation using
implementing classes with hidden details
selecting
study materials for
using for methods
accessibility package
addition operator (+)
aggregation association
characteristics of
defined
diagramming
many-to-many relationships with
study materials for associations
AND operators
angle brackets (< >)
Apache Ant
append method
applications
detecting bugs with FindBugs
interpreting bytecode
invoking Java interpreter
arguments
exception classes without
passing primitives as
polymorphism used with method
using primitive as
arithmetic operators

ArithmeticException class
ArrayIndexOutOfBoundsException class

ArrayList class

- about
- iterating through standard arrays vs.
- study materials for using

arrays

- about
- ArrayList vs.
- multi-dimensional
- one-dimensional
- return string length as variable in
- study materials for

Arrays class

ASCII punctuation characters

AssertionError class

assertions

assignment expression statements

assignment operators

assignment statements

- combining with expression statements
- defined
- study materials for

association

- about
- aggregation association
- composition association
- composition relationships vs.
- defined
- diagramming in UML
- direct association
- many-to-many class
- not included in exam objectives
- one-to-many class
- one-to-one class
- role names for
- study materials for
- temporary

association navigation

- about
- example of
- study materials for

attributes

- package
- package statement

in UML diagrams
attributes compartment
autoboxing

about
using primitive interchangeable as object wrapper class
AWT (Abstract Window Toolkit) API
elements of
scenarios for using
types of

B

base packages for Java SE 7

blocks

defined
using local variables in code
visualizing code

boolean primitive

about
example using
when to use

boolean values

Boolean wrapper class objects

box brackets ([])

conventions using
indicating arrays with
using in multi-dimensional array declarations

braces({ })

conventions using
defining local variables within
enclosing multiple statements in

break statements

exiting iteration statements with
terminating switch statements with

byte primitive

bytecode

C

case keyword

case sensitivity

camel case naming conventions
primitives and wrapper class

casting

defined
hidden
objects
polymorphism vs.

primitives

between primitives and object wrapper classes

study materials for

variables

when to use

CDC (Connected Device Configuration)

certification objectives

casting

conditional statements

defining Java class structure

differentiating between reference and primitive variables

encapsulation principles

exception handling

fundamental operators

implementing inheritance and class types

invoking method that throws exception

iteration statements

package design

packages

passing objects between methods

polymorphism

precedence of operators

reading/writing to object fields

recognizing common exceptions

reference and primitive variables

static methods and instance variables

string objects and methods

StringBuilder class and methods

testing equality between objects and strings

this and super keywords

transfer of control statements

try-catch blocks altering normal program flow

understanding assignment statements

understanding package-derived classes

variable scope

working with Java arrays

chaining methods

char primitive

about

when to use

charAt method

CharSequence interface

checked exceptions

about

ClassNotFoundException

CloneNotSupportedException

common

`FileNotFoundException`
`InterruptedException`
`IOException`
`NoSuchMethodException`
`SQLException`
study materials on
throwing

class relationships

aggregation association
association navigation in
composition association
direct association
multiplicities in
representing in UML
temporary association

`ClassCastException` class

classes. *See also* abstract classes; association; composition; concrete classes; wrapper classes; and specific classes

abstract

Abstract Window Toolkit API

applying polymorphism via inheritance

`ArithmeticException`

`ArrayIndexOutOfBoundsException`

`ArrayList`

Arrays

`AssertionError`

Basic Input/Output API

`ClassCastException`

collection

concrete

creating custom exception

creating static variables for

depicting in UML

`ExceptionInInitializerError`

explicit importing of

extending vs. inheriting

hierarchy for Reader and Writer class

`IllegalArgumentException`

`IllegalStateException`

implementing interfaces for

implementing with hidden details

`IndexOutOfBoundsException`

inheritance tree for

instance variables for

Java Collections Framework

making and calling methods in

naming

Networking API

NoClassDefFoundError

NullPointerException

NumberFormatException

Object base class

objects vs.

OutOfMemoryError

overriding inherited methods for subclasses

package

package-derived

primitive wrapper classes

String, StringBuilder, and String-Buffer

structure for

subpackage

Swing API

Throwable

types of class relationships

Utilities API

VirtualMachineError

ClassNotFoundException exception

-classpath option for compiling

CloneNotSupportedException exception

collection classes

Collection interface

colon (:)

comma (,)

command-line tools

-D option for

-version option for

IDEs vs.

knowing Apache Ant and Maven

using command-line parameters

command switches

comments

compiler errors

caused by incorrect expression statements

introducing

compiling

-classpath option for

-d option for

detecting bugs in strings

operands of different types

and running objects

software from IDEs

source code

study materials for

composition

- about
- association relationships vs.
- one-to-many class
- one-to-one class
- study materials for
- composition association
 - about
 - diagramming in UML
 - study materials for associations
- compound assignment operators
- concat method
- concatenation operator (+)
- concrete classes
 - behavior with polymorphism
 - defined
 - example of inheritance for implementing interfaces
 - study materials on
- conditional statements
 - defined
 - evaluating strings in switch statements
 - if statements
 - if-then
 - if-then-else
 - overflow errors using conditional expressions in
 - study materials for
 - switch
 - types found on exam
 - unboxing Boolean wrapper class objects
- Connected Device Configuration (CDC)
- Connected Limited Device Configuration (CLDC)
- constants
 - creating with static variables
 - predefined for enumerations
- constructors
 - certification objectives for
 - default
 - defined
 - initializing objects with
 - making
 - overloading
 - return types not declared for
 - StringBuilder class
 - study materials for
 - using this in
- continue statements
- control statements. See transfer of control statements cryptography packages

curly brackets(brackets)

defining local variables within
enclosing multiple statements in

D

-d option for compiling
data. *See also* objects; primitives

boolean primitive

byte primitive

char primitive

compiling and interpreting Java code

constructors

creating and using conditional statements

creating and using methods

declaring and using ArrayList

effect of strongly typed variables on
enumerations

hiding

int primitive

knowing primitive used for specific

short primitive

types of primitive

declaring

arrays

and initializing primitives

multi-dimensional arrays

one-dimensional arrays

variables with limited scope

default constructor

default statements

delete method

deleteCharAt method

dependency association. *See* temporary association deployment technologies for JRE

direct association

characteristics of

defined

diagramming in UML

many-to-many relationships with

study materials for associations

division operator (/)

do-while iteration statements

double primitive

double quotes (" ")

E

encapsulation

about
access modifiers in
designing code with encapsulation
implementing classes with hidden details
information hiding
selecting most restrictive access modifier
setters and getters for
study materials on

`endsWith` method

enhanced for loops

enumerations (enum)

characteristics of
example of
study materials for
used for variables

epsilon

equality operators

`equals` method

`equals` to operator (==)

`equalsIgnoreCase` method

errors. *See also* compiler errors

`AssertionError`

common

compiling constructors using `this` and `super` keywords

conditional expressions leading to overflow

detecting bugs in strings with FindBugs application

`ExceptionInInitializerError` class

forcing error conditions

logging

`NoClassDefFoundError`

`OutOfMemoryError`

study materials on

throwing

as unchecked exceptions

`VirtualMachineError`

exam tips. *See also* certification objectives

casting

chaining methods

conditional statements found on exam

default constructors without parameters

exception handling

extending vs. inheriting classes

features of collection classes

following Java naming conventions for object types

`if` conditional statements

J ME not on exam

Java EE not on exam

- methods and variable scope
- overloading methods
- passing objects to method by reference
- polymorphism
- preparing for scope-related questions
- pseudo-code algorithms not on exam
- questions on multi-dimensional arrays
- return types not declared for constructors
- testing compiler reaction to errors
- theory of inheritance
- understanding inheritance examples
- using IDEs on
- watching for hidden casting
- working with arrays

Exception class

`ExceptionInInitializerError` class

exceptions. *See also* checked exceptions; unchecked exceptions

- about

- analyzing source code for

- `ArithmeticException` class

- `ArrayIndexOutOfBoundsException`

- checked

- class hierarchy for

- `ClassCastException`

- common unchecked

- creating multi-catch clauses for

- custom classes for

- defining

- errors as unchecked

- `IllegalArgumentException`

- `IllegalStateException`

- `IndexOutOfBoundsException`

- NetBeans templates for handling

- `NullPointerException`

- `NumberFormatException`

- propagating

- study materials on

- throwing

- try-catch-finally statements with

- try-catch statements to catch

- try-finally statements to catch

- try-with-resources statements with

- unchecked

- when to use assertions vs.

exercises

- adding functionality to interface

- adding new function to abstract class

- analyzing source code for exceptions
- compareTo method of String class
- compiling and running objects
- compound assignment operators
- constructors of StringBuilder class
- creating custom exception classes
- creating getter and setter methods
- evaluating String class in switch statement
- forcing error conditions
- implementing and using arrays and ArrayList
- iterating through ArrayList
- NetBeans code templates for exception handling
- performing code refactoring
- statement-related keywords
- uncovering bugs with FindBugs application
- using assertions vs. exceptions

expression statements

- about
- assigning variables with
- combining with assignment statements

extending

- concrete and abstract classes
- vs. inheriting classes

extends keyword

F

FileNotFoundException exception

final keyword

FindBugs application

float primitive

- about

- casting as int

- when to use

floating-point math

- choosing primitive to store values in
- comparing equality of numbers in

for loops

- enhanced

- iteration statements with

formatting

- code

- using IDEs for consistent

forward slash(/)

G

garbage collector

generalization

getters

accessing inner objects with

creating

study materials for

graphic paths

greater than operator (>)

greater than or equal to operator (>=)

Guillemet characters (<< >>)

H

Heatlamp

hidden casting

hiding

data

methods with encapsulation

I

IDEs (integrated development environments). *See also* NetBeans IDE

benefits and disadvantages of

compiling and interpreting software from

creating getter and setter methods with

ensuring formatting with

`if` conditional statements

exam tip for

identifying

overflow errors using conditional expressions

study materials for

unboxing Boolean wrapper class objects in

writing block of statements in

`if-then` conditional statements

identifying

overflow errors using conditional expressions

study materials for

when to use

`if-then-else` conditional statements

identifying

overflow errors using conditional expressions

study materials for

when to use

`IllegalArgumentException` class

`IllegalStateException` class

immutable

`implements` keyword

implicit import statements

import statements

importing `HashMap` class
importing source files using
including additional classes in source code with
replacing implicit with explicit

importing
 classes implicitly
 explicit classes
 using static import statements

`indexOf` method
`IndexOutOfBoundsException` class

information hiding
inheritance

 abstract classes and
 applying polymorphism via class
 concrete class with
 defined
 extending vs. inheriting classes
 implementing
 implementing interfaces
 inheritance tree
 overriding methods for classes with
 study materials on

initializing
 defined
 multi-dimensional arrays
 one-dimensional arrays
 primitives

initializing objects
 about
 with `new` operator
 role of constructors in

`insert` method

instance variables
 about
 accessing with `this`
 declaring
 selecting most restrictive access modifier for
 study materials for
 using as variable scope

instantiation

`int` primitive
 about
 examples using
 study materials on
 when to use

integrated development environments. See IDEs

integration packages for Java SE 7

interfaces

applying polymorphism by implementing
defined
depicting in UML
example using
illustrated
implementation of
programming to
study materials on
usefulness of

interpreting

-classpath option
-D option
-version option
bytecode

InterruptedException exception

invoking Java interpreter

IOException exception

iteration statements

about
continue statements terminating
defined
do-while
enhanced for loops in
exiting with break statements
for loops with
selecting best
study materials for
while

J

J ME (Java 2 platform, Micro Edition)

jalopy

Java 2 platform, Micro Edition (J ME)

Java Abstract Window Toolkit API

Java Basic Input/Output API

Java Collections Framework

Java Community Process (JCP)

Java Database Connectivity (JDBC) packages

Java Development Kit. *See* JDK

Java EE (Enterprise Edition)

Java Image I/O packages

Java interpreter

Java Naming and Directory Interface (JNDI) packages

Java Networking API

Java platforms. *See also* Java SE 7

J2ME

Java EE

Java SE features

selecting

Java Print Service API package

Java SE 7. *See also* JDK

about

class structure in

compiling and interpreting code for

core packages of

illustrated

integration packages for

Java Runtime Environment

Java SE API

naming conventions in

operating systems supported

pseudo-code algorithms with

representing packages in UML

strongly typed language

support for Unicode standard

symbols and separators for

user interface packages

Java SE API

about

contained in JDK

packages in

Java Specification Requests (JSRs)

Java User Groups (JUGs)

Java Utilities API

classes of Java Collections Framework

extended functionality of

scenarios for using

Java Virtual Machine. *See* JVM java.awt package

javafx prefix

java.io package

java.net package

java.util package

java.util.regex

javax.swing package

JConsole

JCP (Java Community Process)

JDBC (Java Database Connectivity) packages

JDK (Java Development Kit)

about

illustrated

Java SE API contained in

using with Java EE

utilities in

JDocs

JNDI (Java Naming and Directory Interface) packages

JSRs (Java Specification Requests)

JUGs (Java User Groups)

JVM (Java Virtual Machine)

automatic data conversions by

classes and

included in JRE

initializing objects with new operator

K

K virtual machine (KVM)

Kava Runtime Environment (JRE)

keywords. *See also* access modifiers; *and specific keywords*

abstract

exercise listing

extends

list of valid

making variables immutable with final

static

table of statement-related

this and super

void

KVM (K virtual machine)

L

labeled statements

language packages

length method

less than operator (<)

less than or equal to operator (<=)

lifecycle of objects

literals

defined

study materials on

underscores in numeric

local variables

logging errors

logical operators

logical negation operators

using logical AND and OR operators

long primitive

loops

continue statements terminating

do-while

iteration statements with for loops
using arrays in
while

M

main method

many-to-many class associations

many-to-many multiplicity

Matcher class

Maven

method parameters

listing in parentheses

passing

variable scope and

methods

abstract

access modifiers for

append

body of

chaining

charAt

concat

creating static

delete

deleteCharAt

depicting in UML diagrams

endsWith

equalsIgnoreCase

hiding with encapsulation

indexOf

insert

length

making and calling

name of

objects passed by reference between

overloading

passing primitive values to

providing public

replace

return types for

reverse

selecting most restrictive access modifier for

startsWith

String class

StringBuilder class

study materials for

substring
syntax for
toLowerCase method
toUpperCase method
trim method
writing code for
multi-catch clauses
multi-dimensional arrays
about
declaring
initializing
multiplication operator (*)
multiplicities
class composition
example of class association
many-to-many multiplicity
one-to-many multiplicity
one-to-one multiplicity
study materials for
types of
UML indicators for

N

naming
camel case conventions for
classes
methods
variables

NetBeans IDE
code templates for exception handling
refactoring features of
refactoring multiple catch statements in
viewing string declarations in

new keyword
new operator
calling constructors after
initializing objects with
instantiating objects with

NoClassDefFoundError class
NoSuchMethodException exception
not equal to operator (!=)
notes in UML
NullPointerException class
NumberFormatException class

O

Object base class
objects. *See also* association; composition
about
accessing with this
association and composition relationship for
casting
classes vs.
compiling and running
exposing functionality of
initializing
Java naming conventions for
knowledge required on exam about
lifecycle of
made up of primitives
methods within instance of
modeling methodologies for
passed by reference between methods
passing
polymorphic
polymorphism between general and specific
removing from memory
`StringBuilder` class
strings as immutable
study materials for
unboxing Boolean wrapper class
uninitialized
OCA (Oracle Certified Associate) process
one-dimensional arrays
 declaring
 defined
 initializing
 square brackets indicating
one-to-many class association
one-to-many class composition
one-to-many multiplicity
one-to-one class association
one-to-one class composition
one-to-one multiplicity
operands
operating systems
 illustrated
 supported by Java SE
operations compartment
operations in UML diagrams
operators
 arithmetic
 assignment

- compound assignment
- equality
- logical
- new
- precedence of
- prefix and postfix incrementing and decrementing
- relational
- string concatenation
- study materials on

OR operators

Oracle Certified Associate (OCA) process

`OutOfMemoryError` class

overflow errors

overloading

- constructors

- methods

overriding

- annotation for

- class methods with inheritance

- operator precedence

P

package-derived classes

- Collections Framework

- Java Abstract Window Toolkit API

- Java Basic Input/Output API

- Java Networking API

- Java Swing API

- Java Utilities API

- study materials for

package-private access modifier

package statements

- attributes of

- importing source files with

- valid

packages

- accessibility

- AWT API

- base

- commonly used Java SE API

- compiling and interpreting from IDE

- cryptography

- designing classes and attributes of

- extended functionality of Java Utilities API

- integration

- java and javax prefixes for

Java Image I/O
Java Print Service API
Java SE 7
base
language
objectives for
scripting
security
Sound API
study materials for
Swing API
transactions
user interface
utility
XML-based

parameters
default constructors without
method
using command-line
parentheses ()
conventions using
listing method parameters in
overriding operator precedence with
return string lengths using

passing objects
Pattern class
period (.)
platforms. *See Java platforms*
plus sign (+)
polymorphism
about
applying by implementing interfaces
casting vs.
defined
examples using
study materials for
unidirectional
via class inheritance

postfix decrement operator
postfix increment operator
precedence of operators
about
overriding
study materials on
table listing
precision of cast primitives
prefix decrement operator

prefix increment operator

primitives

about

boolean

byte

casting

char

characteristics of different

choosing type for floating-point math

declaring and initializing

double

examples using

float

int

knowing type used for specific data

long

passing values to methods

placing in ArrayList

short

storing simple values with

study materials for

types of data

using in assignment statements

wrapper classes for

private access modifier

about

using

using setters and getters with instance variables

profiles in J2ME

programming

defining exceptions

designing code with encapsulation

engineering code using UML diagrams

writing methods

programming to interface

about

examples of

propagating exceptions

protected access modifier

about

using

pseudo-code algorithms

conventions for

function of

using with Java

public access modifier

about

providing public methods
using

R

Reader class
realization
reference
 defined
 passing objects by
relational operators
 basic
 equality operators
relationships. *See* class relationships; multiplicities
Remote Method Invocation (RMI) packages
replace method
return statements
 optional nature of
 returning method variables with
 uses for
return types for methods
reverse method
RMI (Remote Method Invocation) packages
role names for associations
RuntimeException class

S

scope. *See* variable scope
scripting packages
security packages
semicolon (;)
setters
 accessing inner objects with
 creating methods for
 study materials for
short primitive
single quotes (' ')
Sound API packages
source code
 analyzing for exceptions
 compiling
 -d option for compiling
 using variables of limited scope in
source files
 package statements for
 viewing Java SE
SQLException exception
Squawk

src.jar
stack traces
startsWith method
statements. *See also specific statements*
about
assignment
break
conditional
continue
if conditional
if-then
import
iteration
labeled
listing of Java
package
return
terminating switch
transfer of control
try-catch-finally
try-finally
try-with-resources
types of
static import statements
static keyword
about
creating static variables
indicating static methods with
using
static methods
creating
study materials for
static variables
creating
study materials for
using as constants
String class
charAt method
compareTo method of
concat method
endsWith method
equals method of
equalsIgnoreCase method
evaluating in switch statements
indexOf method
length method
methods of

- replace method
- startsWith method
- study materials on
- substring method
- toLowerCase method
- toUpperCase method
- trim method
- using parentheses to return lengths of

StringBuffer class

StringBuilder class

- append method for
- constructors of
- delete method
- deleteCharAt method
- insert method
- methods of
- objects of
- reverse method
- scenarios using
- study materials on methods of

strings. *See also* String class; StringBuilder class

- choosing classes for
- concatenating
- creating
- immutable nature of
- methods of String class
- StringBuilder class and methods
- study materials on
- testing equality between objects and
- using toString method with classes

strongly typed variables

subclasses

- defined
- filling in for superclass
- overriding inherited methods for

substring method

subtraction operator (-)

super keyword

- accessing overridden methods with
- placement in constructors
- study materials for

superclasses

- defined
- objects polymorphically becoming
- study materials on

Swing API packages

switch conditional statements

compiler errors using same case values in
evaluating `String` class in
scenarios for using
using
switches for commands
symbols and separators for Java
syntax

casting objects or primitives
method

system properties

setting new values for
subset of

T

temporary association

characteristics of
defined
diagramming in UML
many-to-many relationships with
study materials for associations

terminating switch statements

this keyword

accessing instance variables with
placement in constructors
study materials for

`Throwable` class

throwing exceptions

about
catching with try-catch statements
creating multi-catch clauses
study materials on
try-catch-finally statements when
using try-finally statements when

`toLowerCase` method

`toString` method

`toUpperCase` method

transactions packages

transfer of control statements

break statements as
continue statements
defined
labeled
return
study materials for

`trim` method

try-catch-finally statements

try-catch statements
try-finally statements
try-with-resources statements

U

UML (Unified Modeling Language)

- about
- aggregation association
- attributes and operations
- classes, abstract classes, and interface diagrams with
- composition association
- depicting multiplicities in
- depicting visibility modifiers in
- direct association
- drawing class relationships in
- indicators for multiplicity
- methodologies preceding
- notes in
- releases of
- representing Java SE packages in
- showing class relationships in
- temporary association
- types of diagrams

unboxing

unchecked exceptions

- `ArithmeticException`
- `ArrayIndexOutOfBoundsException` class
- `ClassCastException`
- common
- defined
- `IllegalArgumentException` class
- `IllegalStateException`
- `IndexOutOfBoundsException`
- `NullPointerException`
- `NumberFormatException`
- study materials on
- throwing

underscores in numeric literals

Unicode Standards

unidirectional polymorphism

Unified Modeling Language. *See* UML

uninitialized objects

user interface packages

utilities. *See* Java Utilities API; JDK

utility packages

V

value, passing primitives by
variable scope

- choosing type of
- instance variables
- local variables
- mastering variable
- method parameters
- preparing for questions on
- study materials for

variables. *See also* instance variables; primitives; static variables; variable scope

- about Java primitive

- arrays

- case of

- casting to different types

- creating constants from static

- creating static

- declaring with limited scope

- depicting member variables in UML diagrams

- enumerations used for

- instance

- local

- method parameters

- naming conventions for

- passing to methods by reference and value

- strongly typed

versions of UML

viewing Java SE source files

visibility modifiers in UML. *See also* access modifiers

void keyword

- return types omitted for constructors with

- study materials for

- void return type for methods

W

while loop statements

“WORA” capability

wrapper classes

- casting between primitives and object

- examples using

- unboxing Boolean wrapper class objects

- used for primitives

X

x-- operator

x++ operator

XML-based packages

Are You Oracle Certified?

Professional development and industry recognition are not the only benefits you gain from Oracle certifications. They also facilitate career growth, improve productivity, and enhance credibility. Hiring managers who want to distinguish among candidates for critical IT positions know that the Oracle Certification Program is one of the most highly valued benchmarks in the marketplace. Hundreds of thousands of Oracle certified technologists testify to the importance of this industry-recognized credential as the best way to get ahead—and stay there.

For details about the Oracle Certification Program, go to oracle.com/education/certification.

Oracle University —
Learn technology from the source

ORACLE®
UNIVERSITY

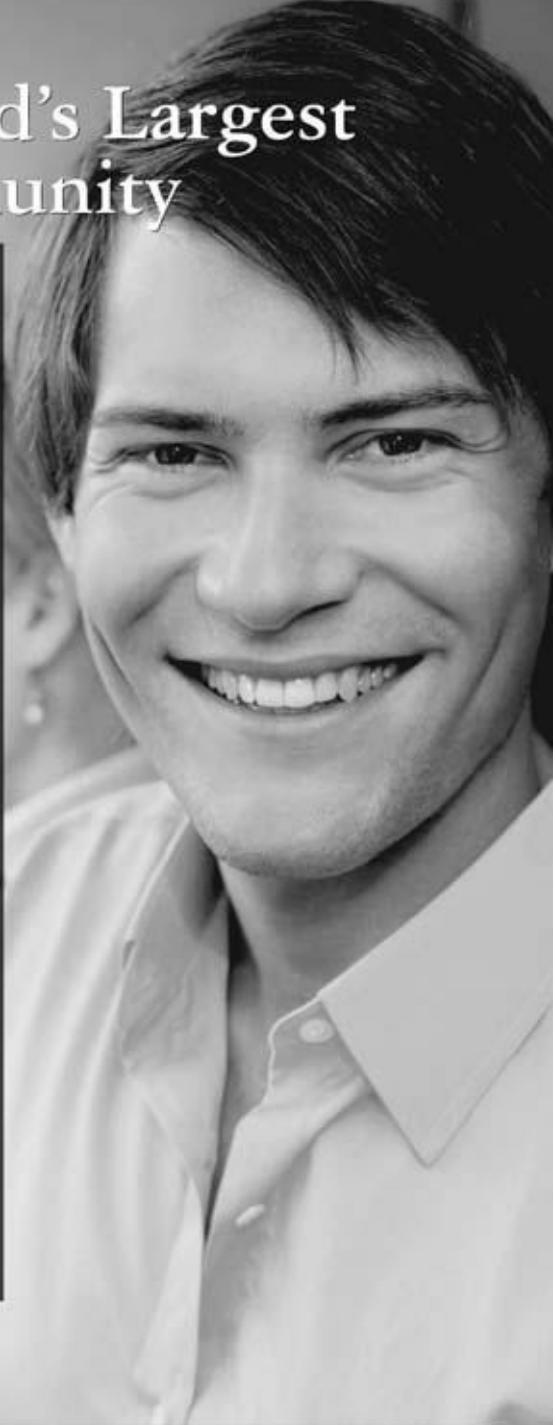


Join the World's Largest Oracle Community

With more than 5 million members, Oracle Technology Network (otn.oracle.com) is the best place online for developers, DBAs, and architects to interact, exchange advice, and get software downloads, documentation, and technical tips — all for free!

Registration is easy;
join Oracle Technology
Network today:
otn.oracle.com/join

ORACLE®
TECHNOLOGY NETWORK



GET YOUR **FREE SUBSCRIPTION** **TO ORACLE MAGAZINE**

Oracle Magazine is essential gear for today's information technology professionals.

Stay informed and increase your productivity with every issue of *Oracle Magazine*.

Inside each free bimonthly issue you'll get:



- Up-to-date information on Oracle Database, Oracle Application Server, Web development, enterprise grid computing, database technology, and business trends
- Third-party news and announcements
- Technical articles on Oracle and partner products, technologies, and operating environments
- Development and administration tips
- Real-world customer stories

If there are other Oracle users at your location who would like to receive their own subscription to *Oracle Magazine*, please photocopy this form and pass it along.

ORACLE
MAGAZINE

Three easy ways to subscribe:

① Web

Visit our Web site at oracle.com/oraclemagazine
You'll find a subscription form there, plus much more

② Fax

Complete the questionnaire on the back of this card
and fax the questionnaire side only to **+1.847.763.9638**

③ Mail

Complete the questionnaire on the back of this card
and mail it to **P.O. Box 1263, Skokie, IL 60076-8263**

ORACLE®

Copyright © 2000, Oracle and/or its affiliates. All rights reserved. Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

Want your own FREE subscription?

To receive a free subscription to *Oracle Magazine*, you must fill out the entire card, sign it, and date it (incomplete cards cannot be processed or acknowledged). You can also fax your application to +1.847.763.9638. Or subscribe at our Web site at oracle.com/oraclemagazine

Yes, please send me a FREE subscription *Oracle Magazine*. No.

- From time to time, Oracle Publishing allows our partners exclusive access to our e-mail addresses for special promotions and announcements. To be included in this program, please check this circle. If you do not wish to be included, you will only receive notices about your subscription via e-mail.
- Oracle Publishing allows sharing of our postal mailing list with selected third parties. If you prefer your mailing address not to be included in this program, please check this circle.

If at any time you would like to be removed from either mailing list, please contact Customer Service at +1.847.763.9638 or send an e-mail to oracle@listserve.com. If you no longer wish to receive promotional e-mail related to Oracle products, services, and events, if you want to completely unsubscribe from any e-mail communication from Oracle, please send an e-mail to unsubscribe-oracle@mail.com with the following in the subject line: REMOVE [your e-mail address]. For complete information on Oracle Publishing's privacy practices, please visit oracle.com/info/privacy.html.

Would you like to receive your free subscription in digital format instead of print if it becomes available? Yes No

YOU MUST ANSWER ALL 10 QUESTIONS BELOW.

① WHAT IS THE PRIMARY BUSINESS ACTIVITY OF YOUR FIRM AT THIS LOCATION? (check one only)

- 01 Aerospace and Defense Manufacturing
- 02 Automotive Services Provider
- 03 Automotive Manufacturing
- 04 Chemicals
- 05 Media and Entertainment
- 06 Construction/Engineering
- 07 Consumer Sector/Consumer Packaged Goods
- 08 Education
- 09 Financial Services/Insurance
- 10 Health Care
- 11 High Technology Manufacturing, OEM
- 12 Industrial Manufacturing
- 13 Independent Software Vendor
- 14 Life Sciences (biotech, pharmaceuticals)
- 15 Natural Resources
- 16 Oil and Gas
- 17 Professional Services
- 18 Public Sector (government)
- 19 Research
- 20 Retail/Warehouse/Distribution
- 21 Systems Integration, VAR/ISV
- 22 Telecommunications
- 23 Travel and Transportation
- 24 Utilities (electric, gas, sanitation, water)
- 99 Other Business and Services

② WHICH OF THE FOLLOWING BEST DESCRIBES YOUR PRIMARY JOB FUNCTION? (check one only)

- 01 Executive Management (President, Chair, CEO, CFO, Owner, Partner, Principal)
- 02 Finance/Administrative Management (VP/Director/Manager/Controller, Purchasing, Administration)
- 03 Sales/Marketing Management (VP/Director/Manager)
- 04 Computer Systems/Operations Management (CIO/VP/Director/Manager MIS/IT, Ops)
- 05 IS/IT Staff
- 06 Application Development/Programming Management
- 07 Application Development/Programming Staff
- 08 Consulting
- 09 Education/Training
- 10 Technical Support Director/Manager
- 11 Other Technical Management/Staff
- 98 Other

③ WHAT IS YOUR CURRENT PRIMARY OPERATING PLATFORM? (check all that apply)

- 01 Digital Equipment Corp UNIX/VAX/VMS
- 02 HP UNIX
- 03 IBM AIX
- 04 IBM UNIX
- 05 Linux (Red Hat)
- 06 Linux (SUSE)
- 07 Linux (Oracle Enterprise)
- 08 Linux (other)
- 09 Macintosh
- 10 MVS
- 11 Network
- 12 Network Computing
- 13 SCO UNIX
- 14 Sun Solaris/SentOS
- 15 Windows
- 16 Other UNIX
- 99 None of the Above

④ DO YOU EVALUATE, SPECIFY, RECOMMEND, OR AUTHORIZIZE THE PURCHASE OF ANY OF THE FOLLOWING? (check all that apply)

- 01 Hardware
- 02 Business Applications (ERP, CRM, etc.)
- 03 Application Development Tools
- 04 Database Products
- 05 Internet or Intranet Products
- 06 Other Software
- 07 Middleware Products
- 99 None of the Above

⑤ IN YOUR JOB, DO YOU USE OR PLAN TO PURCHASE ANY OF THE FOLLOWING PRODUCTS? (check all that apply)

- 01 CAD/CAM
- 02 Collaboration Software
- 03 Communications
- 04 Database Management
- 05 File Management
- 06 Finance
- 07 Java
- 08 Multimedia Authoring
- 09 Networking
- 10 Programming
- 11 Project Management
- 12 Scientific and Engineering
- 13 Systems Management
- 14 Workflow
- 15 Mainframe
- 16 Macintosh
- 17 Massively Parallel Processing

- 18 Microcomputer
- 19 Intel x86/32
- 20 Intel x86/64
- 21 Network Computer
- 22 Symmetric Multiprocessing
- 23 Workstation Servers

- SERVICES
- 24 Consulting
- 25 Education/Training
- 26 Maintenance
- 27 Online Database
- 28 Support
- 29 Technology-Based Training
- 30 Other
- 99 None of the Above

⑥ WHAT IS YOUR COMPANY'S SIZE? (check one only)

- 01 More than 25,000 Employees
- 02 10,001 to 25,000 Employees
- 03 5,001 to 10,000 Employees
- 04 1,001 to 5,000 Employees
- 05 101 to 1,000 Employees
- 06 Fewer than 100 Employees

⑦ DURING THE NEXT 12 MONTHS, HOW MUCH DO YOU ANTICIPATE YOUR ORGANIZATION WILL SPEND ON COMPUTER HARDWARE, SOFTWARE, PERIPHERALS, AND SERVICES FOR YOUR LOCATION? (check one only)

- 01 Less than \$10,000
- 02 \$10,000 to \$49,999
- 03 \$50,000 to \$99,999
- 04 \$100,000 to \$499,999
- 05 \$500,000 to \$999,999
- 06 \$1,000,000 and Over

⑧ WHAT IS YOUR COMPANY'S YEARLY SALES REVENUE? (check one only)

- 01 \$50,000,000 and above
- 02 \$100,000,000 to \$500,000,000
- 03 \$50,000,000 to \$100,000,000
- 04 \$5,000,000 to \$50,000,000
- 05 \$1,000,000 to \$5,000,000

⑨ WHAT LANGUAGES AND FRAMEWORKS DO YOU USE? (check all that apply)

- 01 Ajax
- 02 C
- 03 C++
- 04 C#
- 13 Python
- 14 Ruby/Rails
- 15 Spring
- 16 Struts

- 05 Macintosh
- 06 Java
- 07 Java
- 08 ASP
- 09 AIX
- 10 Perl
- 11 PHP
- 12 PL/SQL
- 13 SQL
- 14 J++/J#
- 15 Visual Basic
- 99 Other

⑩ WHAT ORACLE PRODUCTS ARE IN USE AT YOUR SITE? (check all that apply)

- ORACLE DATABASE
- 01 Oracle Database 11g
- 02 Oracle Database 10g
- 03 Oracle Database
- 04 Oracle Embedded Database (Oracle Lite, Times Ten, Berkeley DB)
- 05 Other Oracle Database Releases

- ORACLE FUSION Middleware
- 06 Oracle Application Server
- 07 Oracle Portal
- 08 Oracle Enterprise Manager
- 09 Oracle BPM, Process Manager
- 10 Oracle Identity Management
- 11 Oracle SOA Suite
- 12 Oracle Data Hub

- ORACLE DEVELOPMENT TOOLS
- 13 Oracle JDeveloper
- 14 Oracle Forms
- 15 Oracle Reports
- 16 Oracle Designer
- 17 Oracle Discoverer
- 18 Oracle BI Server
- 19 Oracle Warehouse Builder
- 20 Oracle WebCenter
- 21 Oracle Application Express

- ORACLE APPLICATIONS
- 22 Oracle E-Business Suite
- 23 PeopleSoft Enterprise
- 24 JD Edwards EnterpriseOne
- 25 JD Edwards World
- 26 Oracle Fusion
- 27 Hyperion
- 28 Siebel CRM

- ORACLE SERVICES
- 29 Oracle E-Business Suite On Demand
- 30 Oracle Technology On Demand
- 31 Siebel CRM On Demand
- 32 Oracle Consulting
- 33 Oracle Education
- 34 Oracle Support
- 99 None of the Above

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 for 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.