Je‹a nne 3cyai sky ‹a incl Scott Selikoff

Oracle Certified Associate Java SE 8 Programmer I

STUDY GUIDE

* 3 C L7 StO III |D Lñ C t iCc c X fiMI S
* r•tore th an 200 Elec t roni c Flu sh en r‹Is

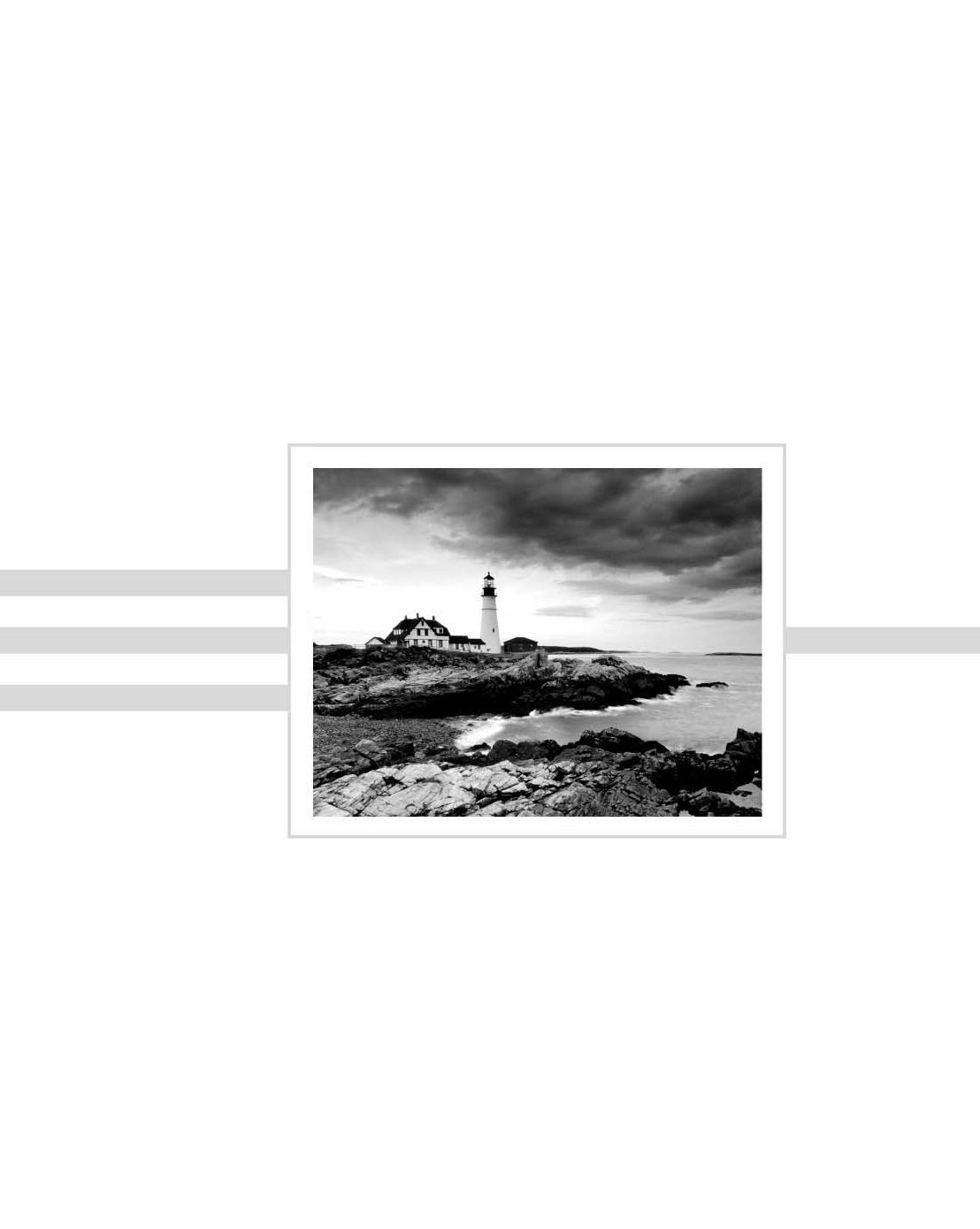
+ Sea re lv ‹I bIe key tc rf›1 \*,loss at y

# OCA: Oracle®

## Certified Associate Java® SE 8 Programmer I

**Study Guide Exam 1Z0-808**

# OCA: Oracle®



## Certified Associate Java® SE 8 Programmer I

**Study Guide Exam 1Z0-808**

Jeanne Boyarsky Scott Selikoff

Senior Acquisitions Editor: Kenyon Brown Development Editor: Alexa Murphy

Technical Editors: Ernest Friedman-Hill, Matt Dalen Production Editor: Rebecca Anderson

Copy Editor: Liz Welch

Editorial Manager: Pete Gaughan

Vice President and Executive Group Publisher: Richard Swadley Associate Publisher: Jim Minatel

Production Manager: Kathleen Wisor Media Supervising Producer: Rich Graves Book Designers: Judy Fung and Bill Gibson

Proofreader: Scott Klemp, Word One New York Indexer: Ted Laux

Project Coordinator, Cover: Patrick Redmond Cover Designer: Wiley

Cover Image: ©Getty Images Inc./Jeremy Woodhouse

Copyright © 2015 by John Wiley & Sons, Inc., Indianapolis, Indiana Published simultaneously in Canada

ISBN: 978-1-118-95740-0

ISBN: 978-1-118-95741-7 (ebk.)

ISBN: 978-1-118-95742-4 (ebk.)

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permit- ted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646- 8600. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at [http://www.wiley.com/go/permissions.](http://www.wiley.com/go/permissions)

Limit of Liability/Disclaimer of Warranty: The publisher and the author make no representations or war- ranties with respect to the accuracy or completeness of the contents of this work and specifically disclaim all warranties, including without limitation warranties of fitness for a particular purpose. No warranty may be created or extended by sales or promotional materials. The advice and strategies contained herein may not be suitable for every situation. This work is sold with the understanding that the publisher is not engaged in rendering legal, accounting, or other professional services. If professional assistance is required, the services of a competent professional person should be sought. Neither the publisher nor the author shall be liable for damages arising herefrom. The fact that an organization or Web site is referred to in this work as a citation and/or a potential source of further information does not mean that the author or the publisher endorses the information the organization or Web site may provide or recommendations it may make. Further, readers should be aware that Internet Web sites listed in this work may have changed or disappeared between when this work was written and when it is read.

For general information on our other products and services or to obtain technical support, please contact our Customer Care Department within the U.S. at (877) 762-2974, outside the U.S. at (317) 572-3993 or fax (317) 572-4002.

Wiley publishes in a variety of print and electronic formats and by print-on-demand. Some material included with standard print versions of this book may not be included in e-books or in print-on-demand. If this book refers to media such as a CD or DVD that is not included in the version you purchased, you may download this material at [http://booksupport.wiley.com.](http://booksupport.wiley.com/) For more information about Wiley products, visit [www.wiley.com.](http://www.wiley.com/)

**Library of Congress Control Number:** 2014954685

TRADEMARKS: Wiley, the Wiley logo, and the Sybex logo are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. Oracle and Java are registered trademarks of Oracle America, Inc. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc. is not associated with any product or vendor mentioned in this book.

10 9 8 7 6 5 4 3 2 1



Dear Reader,

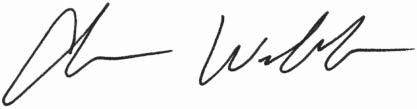
Thank you for choosing *OCA: Oracle Certified Associate Java SE 8 Programmer I Study Guide.* This book is part of a family of premium-quality Sybex books, all of which are written by outstanding authors who combine practical experience with a gift for teaching.

Sybex was founded in 1976. More than 30 years later, we’re still committed to producing consistently exceptional books. With each of our titles, we’re working hard to set a new standard for the industry. From the paper we print on, to the authors we work with, our goal is to bring you the best books available.

I hope you see all that reflected in these pages. I’d be very interested to hear your com- ments and get your feedback on how we’re doing. Feel free to let me know what you think about this or any other Sybex book by sending me an email at contactus@wiley

.com. If you think you’ve found a technical error in this book, please visit [http://sybex](http://sybex/)

.custhelp.com. Customer feedback is critical to our efforts at Sybex.

Best regards,

Chris Webb Associate Publisher

Sybex, an Imprint of Wiley

*To the programmers on FIRST robotics team 694.*

*—Jeanne*

*To my wife and the two little bundles of joy she is carrying.*

*—Scott*

#### Acknowledgments

Jeanne and Scott would like to thank numerous individuals for their contribution to this book. Thank you to Developmental Editor Alexa Murphy for teaching us about Wiley’s publishing process and making the book better in so many ways. Thank you to Ernest Friedman-Hill for being our Technical Editor as we wrote our first book. Ernest pointed out many subtle errors in addition to the big ones. And thank you to Matt Dalen for being our Technical Proofer and finding the errors we managed to sneak by Ernest. This book also wouldn’t be possible without many people at Wiley, including Jeff Kellum, Kenyon Brown, Pete Gaughan, Rebecca Anderson, and so many others.

Jeanne would personally like to thank Chris Kreussling for knowing almost a decade ago that she would someday write a book. Erik Kariyev motivated her to write her first table of contents ever. Countless CodeRanch.com moderators warned Jeanne about how much work writing a book is to get her to the point where she was ready. Michael Ernest gave her extra advice on the Wiley process. Bert Bates let Jeanne dip her toe in by contrib- uting to his Java 7 book and she learned a ton in the process. Scott was a great co-author and was available to bounce ideas off of or remind her to follow her own advice. Finally, Jeanne would like to thank all of the new programmers at CodeRanch.com and FIRST robotics team 694 for the constant reminders of how new programmers think.

Scott could not have reached this point without the help of a small army of people, led by his perpetually understanding wife Patti, without whose love and support this book would never have been possible. Professor Johannes Gehrke of Cornell University always believed in him and knew he would excel in his career. Jeanne’s patience and guidance

as co-author was invaluable while Scott adjusted to the learning curve of writing a book. Matt Dalen has been a wonderful friend and sounding board over the last year. Joel McNary introduced him to CodeRanch.com and encouraged him to post regularly, a step that changed his life. Finally, Scott would like to thank his mother and retired teacher Barbara Selikoff for teaching him the value of education and his father Mark Selikoff, for instilling in him the benefits of working hard.

#### About the Authors

**Jeanne Boyarsky** has worked as a Java developer for over 12 years at a bank in New York City where she develops, mentors, and conducts training. Besides being a senior moderator at CodeRanch.com in her free time, she works on the forum codebase. Jeanne also men- tors the programming division of a FIRST robotics team, where she works with students just getting started with Java.

Jeanne got her Bachelor of Arts in 2002 and her Master’s in Computer Information Technology in 2005. She enjoyed getting her Master’s degree in an online program while working full time. This was before online education was cool! Jeanne is also a Distinguished Toastmaster and a Scrum Master. You can find out more about Jeanne at [www.coderanch.com/how-to/java/BioJeanneBoyarsky.](http://www.coderanch.com/how-to/java/BioJeanneBoyarsky)

**Scott Selikoff** is a professional software consultant, author, and owner of Selikoff Solutions, LLC, which provides software development solutions to businesses in the

tri-state New York City area. Skilled in a plethora of software languages and platforms, Scott specializes in database-driven systems, web-based applications, and service-oriented architectures.

A native of Toms River, NJ, Scott achieved his Bachelor of Arts from Cornell University in Mathematics and Computer Science in 2002, after 3 years of study. In 2003, he received his Master’s of Engineering in Computer Science, also from Cornell University.

As someone with a deep love of education, Scott has always enjoyed teaching others new concepts. He’s given lectures at Cornell University and Rutgers University, as well as confer- ences including The Server Side Java Symposium. Scott lives in New Jersey with his loving wife and two very playful dogs, a Siberian husky named Webby and standard poodle named Georgette. You can find out more about Scott at [www.linkedin.com/in/selikoff.](http://www.linkedin.com/in/selikoff)

Jeanne and Scott are both moderators on the CodeRanch.com forums and can be reached there for questions and comments. They also co-author a technical blog called Down Home Country Coding at [www.selikoff.net.](http://www.selikoff.net/)

#### Contents at a Glance

*Introduction xxi*

*Assessment Test xxxi*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Chapter** | **1** |  | Java Building Blocks | 1 |
| **Chapter** | **2** |  | Operators and Statements | 51 |
| **Chapter** | **3** |  | Core Java APIs | 101 |
| **Chapter** | **4** |  | Methods and Encapsulation | 165 |
| **Chapter** | **5** |  | Class Design | 233 |
| **Chapter** | **6** |  | Exceptions | 299 |
| **Appendix** |  | **A** | Answers to Review Questions | 333 |
| **Appendix** |  | **B** | Study Tips | 353 |
| *Index* |  |  |  | *367* |

|  |  |  |
| --- | --- | --- |
| Contents |  | |
| *Introduction* |  | *xxi* |
| *Assessment Test* |  | *xxxi* |
| **Chapter 1** | **Java Building Blocks** | **1** |
|  | Understanding the Java Class Structure | 2 |
|  | Fields and Methods | 2 |
|  | Comments | 4 |
|  | Classes vs. Files | 5 |
|  | Writing a *main()* Method | 6 |
|  | Understanding Package Declarations and Imports | 9 |
|  | Wildcards | 10 |
|  | Redundant Imports | 11 |
|  | Naming Conflicts | 12 |
|  | Creating a New Package | 13 |
|  | Code Formatting on the Exam | 16 |
|  | Creating Objects | 16 |
|  | Constructors | 17 |
|  | Reading and Writing Object Fields | 18 |
|  | Instance Initializer Blocks | 18 |
|  | Order of Initialization | 19 |
|  | Distinguishing Between Object References and Primitives | 20 |
|  | Primitive Types | 20 |
|  | Reference Types | 24 |
|  | Key Differences | 25 |
|  | Declaring and Initializing Variables | 25 |
|  | Declaring Multiple Variables | 26 |
|  | Identifiers | 27 |
|  | Understanding Default Initialization of Variables | 29 |
|  | Local Variables | 29 |
|  | Instance and Class Variables | 30 |
|  | Understanding Variable Scope | 31 |
|  | Ordering Elements in a Class | 34 |
|  | Destroying Objects | 36 |
|  | Garbage Collection | 36 |
|  | *finalize()* | 38 |
|  | Benefits of Java | 39 |
|  | Summary | 40 |
|  | Exam Essentials | 41 |
|  | Review Questions | 42 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Chapter** | **2** | **Operators and Statements** | **51** |
|  |  | Understanding Java Operators | 52 |
|  |  | Working with Binary Arithmetic Operators | 53 |
|  |  | Arithmetic Operators | 53 |
|  |  | Numeric Promotion | 55 |
|  |  | Working with Unary Operators | 57 |
|  |  | Logical Complement and Negation Operators | 57 |
|  |  | Increment and Decrement Operators | 58 |
|  |  | Using Additional Binary Operators | 60 |
|  |  | Assignment Operators | 60 |
|  |  | Compound Assignment Operators | 62 |
|  |  | Relational Operators | 63 |
|  |  | Logical Operators | 64 |
|  |  | Equality Operators | 65 |
|  |  | Understanding Java Statements | 66 |
|  |  | The *if-then* Statement | 67 |
|  |  | The *if-then*-*else* Statement | 68 |
|  |  | The *switch* Statement | 72 |
|  |  | The *while* Statement | 76 |
|  |  | The *do-while* Statement | 78 |
|  |  | The *for* Statement | 80 |
|  |  | Understanding Advanced Flow Control | 86 |
|  |  | Nested Loops | 87 |
|  |  | Adding Optional Labels | 87 |
|  |  | The *break* Statement | 88 |
|  |  | The *continue* Statement | 90 |
|  |  | Summary | 92 |
|  |  | Exam Essentials | 92 |
|  |  | Review Questions | 94 |
| **Chapter** | **3** | **Core Java APIs** | **101** |
|  |  | Creating and Manipulating Strings | 102 |
|  |  | Concatenation | 102 |
|  |  | Immutability | 104 |
|  |  | The String Pool | 105 |
|  |  | Important String Methods | 105 |
|  |  | Method Chaining | 110 |
|  |  | Using the *StringBuilder* Class | 111 |
|  |  | Mutability and Chaining | 112 |
|  |  | Creating a *StringBuilder* | **113** |
|  |  | Important *StringBuilder* Methods | 114 |
|  |  | *StringBuilder* vs. *StringBuffer* | **117** |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | Understanding Equality | 117 |
|  |  | Understanding Java Arrays | 119 |
|  |  | Creating an Array of Primitives | 119 |
|  |  | Creating an Array with Reference Variables | 121 |
|  |  | Using an Array | 123 |
|  |  | Sorting | 124 |
|  |  | Searching | 125 |
|  |  | Varargs | 126 |
|  |  | Multidimensional Arrays | 126 |
|  |  | Understanding an *ArrayList* | 129 |
|  |  | Creating an *ArrayList* | **129** |
|  |  | Using an *ArrayList* | **130** |
|  |  | Wrapper Classes | 134 |
|  |  | Autoboxing | 136 |
|  |  | Converting Between array and *List* | **136** |
|  |  | Sorting | 138 |
|  |  | Working with Dates and Times | 138 |
|  |  | Creating Dates and Times | 138 |
|  |  | Manipulating Dates and Times | 142 |
|  |  | Working with Periods | 145 |
|  |  | Formatting Dates and Times | 148 |
|  |  | Parsing Dates and Times | 151 |
|  |  | Summary | 151 |
|  |  | Exam Essentials | 152 |
|  |  | Review Questions | 153 |
| **Chapter** | **4** | **Methods and Encapsulation** | **165** |
|  |  | Designing Methods | 166 |
|  |  | Optional Specifiers | 168 |
|  |  | Return Type | 169 |
|  |  | Method Name | 170 |
|  |  | Parameter List | 171 |
|  |  | Optional Exception List | 171 |
|  |  | Method Body | 171 |
|  |  | Working with Varargs | 172 |
|  |  | Applying Access Modifiers | 173 |
|  |  | Private Access | 173 |
|  |  | Default (Package Private) Access | 175 |
|  |  | Protected Access | 176 |
|  |  | Public Access | 180 |
|  |  | Designing Static Methods and Fields | 181 |
|  |  | Calling a Static Variable or Method | 182 |
|  |  | Static vs. Instance | 183 |
|  |  | Static Variables | 185 |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | Static Initialization | 186 |
|  |  | Static Imports | 187 |
|  |  | Passing Data Among Methods | 188 |
|  |  | Overloading Methods | 191 |
|  |  | Creating Constructors | 196 |
|  |  | Default Constructor | 197 |
|  |  | Overloading Constructors | 199 |
|  |  | Final Fields | 202 |
|  |  | Order of Initialization | 202 |
|  |  | Encapsulating Data | 205 |
|  |  | Creating Immutable Classes | 207 |
|  |  | Writing Simple Lambdas | 208 |
|  |  | Lambda Example | 209 |
|  |  | Lambda Syntax | 211 |
|  |  | Predicates | 214 |
|  |  | Summary | 215 |
|  |  | Exam Essentials | 216 |
|  |  | Review Questions | 218 |
| **Chapter** | **5** | **Class Design** | **233** |
|  |  | Introducing Class Inheritance | 234 |
|  |  | Extending a Class | 235 |
|  |  | Applying Class Access Modifiers | 237 |
|  |  | Creating Java Objects | 237 |
|  |  | Defining Constructors | 238 |
|  |  | Calling Inherited Class Members | 244 |
|  |  | Inheriting Methods | 246 |
|  |  | Inheriting Variables | 257 |
|  |  | Creating Abstract Classes | 259 |
|  |  | Defining an Abstract Class | 260 |
|  |  | Creating a Concrete Class | 262 |
|  |  | Extending an Abstract Class | 263 |
|  |  | Implementing Interfaces | 266 |
|  |  | Defining an Interface | 267 |
|  |  | Inheriting an Interface | 269 |
|  |  | Interface Variables | 273 |
|  |  | Default Interface Methods | 274 |
|  |  | Static Interface Methods | 278 |
|  |  | Understanding Polymorphism | 279 |
|  |  | Object vs. Reference | 281 |
|  |  | Casting Objects | 282 |
|  |  | Virtual Methods | 284 |
|  |  | Polymorphic Parameters | 285 |
|  |  | Polymorphism and Method Overriding | 287 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | | Summary | 288 |
| Exam Essentials | 289 |
| Review Questions | 291 |
| **Chapter** | **6** |  | **Exceptions** | **299** |
|  |  |  | Understanding Exceptions | 300 |
|  |  |  | The Role of Exceptions | 300 |
|  |  |  | Understanding Exception Types | 302 |
|  |  |  | Throwing an Exception | 304 |
|  |  |  | Using a *try* Statement | 305 |
|  |  |  | Adding a *finally* Block | 307 |
|  |  |  | Catching Various Types of Exceptions | 309 |
|  |  |  | Throwing a Second Exception | 311 |
|  |  |  | Recognizing Common Exception Types | 313 |
|  |  |  | Runtime Exceptions | 314 |
|  |  |  | Checked Exceptions | 317 |
|  |  |  | Errors | 317 |
|  |  |  | Calling Methods That Throw Exceptions | 318 |
|  |  |  | Subclasses | 319 |
|  |  |  | Printing an Exception | 321 |
|  |  |  | Summary | 323 |
|  |  |  | Exam Essentials | 324 |
|  |  |  | Review Questions | 325 |
| **Appendix** |  | **A** | **Answers to Review Questions** | **333** |
|  |  |  | Chapter 1: Java Building Blocks | 334 |
|  |  |  | Chapter 2: Operators and Statements | 336 |
|  |  |  | Chapter 3: Core Java APIs | 339 |
|  |  |  | Chapter 4: Methods and Encapsulation | 342 |
|  |  |  | Chapter 5: Class Design | 346 |
|  |  |  | Chapter 6: Exceptions | 349 |
| **Appendix** |  | **B** | **Study Tips** | **353** |
|  |  |  | Studying for the Test | 354 |
|  |  |  | Creating a Study Plan | 354 |
|  |  |  | Creating and Running Sample Applications | 355 |
|  |  |  | Taking the Test | 359 |
|  |  |  | Understanding the Question | 359 |
|  |  |  | Applying Process of Elimination | 362 |
|  |  |  | Optimizing Your Time | 364 |
|  |  |  | Getting a Good Night’s Rest | 366 |
| *Index* |  |  |  | *367* |

#### Introduction

Java, “born” in 1995, is now just about 20 years old. As with anything 20 years old, there is a good amount of history and variation between versions of Java. Over the years, the cer- tification exams have changed to cover different topics. The names of the exams have even changed. This book covers the Java 8 Oracle Certified Associate (OCA) exam.

If you read about “the exam” on the Web, you may see information about the older names for the exam. We’ve showed the changes in name. Here’s what happened. Sun Microsystems used to have two exams. The SCJP (Sun Certified Java Programmer) was meant for programmers and the SCJA (Sun Certified Java Associate) was meant for those who wanted broader knowledge. When Oracle bought Sun Microsystems, they changed all the names from Sun to Oracle, giving us the OCJP and OCJA.

SCJA 5-6

Renamed

OCJA 6

OCA 7-8

SCJP 1-6

Renamed

OCJP 6

OCP 7-8

Then Oracle made two strategic decisions with Java 7. They decided to stop updating the OCJA exam. They also decided to cover more on in the programmer space and split it into two exams. Now you first take the OCAJP (Oracle Certified Associate Java Programmer), also known as Java Programmer I, or OCA. That’s what this book is about. Then you

take the OCPJP (Oracle Certified Professional Java Programmer), also known as Java Programmer II, or OCP. There’s also an upgrade exam in case you took an older version of the SCJP or OCPJP and want to upgrade. Most people refer to the current exams as OCA 8, OCP 8, and the Java 8 upgrade exam. We mention when a topic is split between the OCA and OCP so you know which parts are more advanced.

We try to keep the history to a minimum in this book. There are some places on the exam where you need to know both an “old way” and a “new way” of doing things. When that happens, we will be sure to tell you what version of Java introduced it. We will also let you know about topics that are not on the exam anymore in case you see questions in the older free online mock exams.

#### The OCA Exam

All you need to do to earn the Oracle Certified Associate Java SE 8 Programmer certifica- tion is to pass the exam! That’s it.

Oracle has a tendency to fiddle with the length of the exam and the passing score once it comes out. Since it’s pretty much a guarantee that whatever we tell you here will become obsolete, we will give you a feel for the range of variation. The OCA exam has varied between 60 and 90 questions since it was introduced. The score to pass has varied between 60 percent and 80 percent. The time allowed to take the exam has varied from two hours to two-and-a-half hours.

Oracle has a tendency to “tweak” the exam objectives over time as well. They do make minor additions and removals from what is covered on the exam. Although this tends to affect the OCP exam more than the OCA exam, there are a few topics that were added to the OCA for Java 8. It wouldn’t be a surprise for Oracle to make changes.

Although there will likely be minor changes to the scope of the exam, it certainly isn’t a secret. We’ve created a book page on our blog: [www.selikoff.net/oca.](http://www.selikoff.net/oca) If there are any changes to the topics on the exam after this book is published, we will note them there.

That book page also contains a link to the official exam page so that you can check the length and passing score that Oracle has chosen for the moment.

#### Scheduling the Exam

The exam is administered by Pearson VUE and can be taken at any Pearson VUE testing center. To find a testing center or register for the exam, go to [www.pearsonvue.com.](http://www.pearsonvue.com/) Choose IT and then Oracle. If you haven’t been to the test center before, we recommend visiting in advance. Some testing centers are nice and professionally run. Others stick you in a closet with lots of people talking around you. You don’t want to be taking the test with someone complaining about their broken laptop nearby!

At this time, you can reschedule the exam without penalty until up to 24 hours before. This means that you can register for a convenient time slot well in advance, knowing that you can delay if you aren’t ready by that time. Rescheduling is easy and can be done on the Pearson VUE website. This may change, so check the rules before paying.

#### The Day of the Exam

When you go to take the exam, remember to bring two forms of ID, including one that is government issued. See Pearson’s list of what is acceptable ID at http://www.pearsonvue

.com/policies/1S.pdf. Try not to bring too much extra with you as it will not be allowed

into the exam room. While you will be allowed to check your belongings, it is better to leave extra items at home or in the car.

You will not be allowed to bring paper, your phone, and so forth into the exam room with you. Some centers are stricter than others. At one center, tissues were even taken away from us! Most centers allow keeping your ID and money. They watch you taking the exam, though, so don’t even think about writing notes on money.

The exam center will give you writing materials to use during the exam. These are used as scratch paper during the exam to figure out answers and keep track of your thought pro- cess. The exam center will dispose of them at the end. Notice how we said “writing materi- als” rather than “pen and paper.” Some centers still give pen and paper. Most give a small erasable board and a dry erase marker. If you have a preference to which you receive, call the testing center in advance to inquire.

#### Finding Out Your Score

In the past, you would find out right after finishing the exam if you passed. Now you have to wait nervously until you can check your score online.

If you go onto the Pearson VUE website, it will just have a status of “Taken” rather than your result. Oracle uses a separate system for scores. You’ll need to go to http:// certview.oracle.com to find out whether you passed and your score. It doesn’t update immediately upon taking the test, but we haven’t heard of it taking more than an hour. In addition to your score, you’ll also see objectives for which you got a question wrong and instructions on how to get a hardcopy certificate.

At some point, you’ll get an electronic certificate and some more time after that you’ll receive your printed certificate. Sound vague? It is. The times reported to receive certificates vary widely.

#### Exam Questions

The OCA exam consists of multiple-choice questions. There are typically five or six possi- ble answers. If a question has more than one answer, the question specifically states exactly how many correct answers there are. This book does not do that. We say “choose all that apply” to make the questions harder. This means the questions in this book are generally harder than those on the exam. The idea is to give you more practice so you can spot the correct answer more easily on the real exam.

Note that exam questions will sometimes have line numbers that begin with numbers higher than 1. This is to indicate that you are looking at a code snippet rather than a com- plete class. We follow this convention as well to get you used to it.

If you read about older versions of the exam online, you might see references to drag- and-drop questions. These questions had you do a puzzle on how to complete a piece of

code. There was also a bug in the exam software that caused your answers to get lost if you reviewed them again. Luckily, these are no longer on the exam.

#### Getting Started

We recommend reading Appendix B, “Study Tips,” before diving into the technical mate- rial in this book. Knowing how to approach studying will help you make better use of your study time.

Next, make sure you have downloaded version 8 of the JDK. If you learned Java some time ago, you might have version 7 or even earlier. There have been both big and small changes to the language. You could get a question wrong if you study with the wrong version.

Also, please check our book page to make sure Oracle hasn’t changed the objectives.

For example, if Oracle decided that lambdas weren’t on the exam, you’d want to know that before studying. We will post any updates that you should know about at www.selikoff

.net/oca.

#### Getting Help

Both of the authors are moderators at CodeRanch.com. CodeRanch.com is a very large and active programming forum that is very friendly toward Java beginners. It has a forum just for this exam called OCAJP. It also has a forum called Beginning Java for non-exam- specific questions. As you read the book, feel free to ask your questions in either of those forums. It could be you are having trouble compiling a class or that you are just plain con- fused about something. You’ll get an answer from a knowledgeable Java programmer. It might even be one of us.

#### Who Should Buy This Book

If you want to become certified as a Java programmer, this book is definitely for you. If you want to acquire a solid foundation in Java and your goal is to prepare for the exam, this book is also for you. You’ll find clear explanations of the concepts you need to grasp and plenty of help to achieve the high level of professional competency you need in order to suc- ceed in your chosen field.

This book is intended to be understandable to anyone who has a tiny bit of Java knowl- edge. If you’ve never read a Java book before, we recommend starting with a book that teaches programming from the beginning and then returning to this study guide.

This book is for anyone from high school students to those beginning their program- ming journey to experienced professionals who need a review for the certification.

#### How This Book Is Organized

This book consists of six chapters, plus supplementary information: a glossary, this intro- duction, three appendices, and the assessment test after the introduction. You might have noticed that there are more than six exam objectives. We split up what you need to know to make it easy to learn and remember. Each chapter begins with a list of the objectives that are covered in that chapter.

The chapters are organized as follows:

* Chapter 1, “Java Building Blocks,” covers the basics of Java such as scoping variables and how to run a program. It also includes calling methods and types of variables.
* Chapter 2, “Operators and Statements,” focuses on the core logical constructs such as conditionals and loops. It also talks about the meaning and precedence of operators.
* Chapter 3, “Core Java APIs,” introduces you to array, ArrayList, String, String- Builder, and various date classes.
* Chapter 4, “Methods and Encapsulation,” explains how to write methods, including access modifiers. It also shows how to call lambdas.
* Chapter 5, “Class Design,” adds interfaces and superclasses. It also includes casting and polymorphism.
* Chapter 6, “Exceptions,” shows the different types of exception classes and how to use them.

At the end of each chapter, you’ll find a few elements you can use to prepare for the exam:

**Summary** This section reviews the most important topics that were covered in the chapter and serves as a good review.

**Exam Essentials** This section summarizes highlights that were covered in the chapter. You should be readily familiar with the key points of each chapter and be able to explain them in detail.

**Review Questions** Each chapter concludes with at least 20 review questions. You should answer these questions and check your answers against the ones provided in Appendix A.

If you can’t answer at least 80 percent of these questions correctly, go back and review the chapter, or at least those sections that seem to be giving you difficulty.

The review questions, assessment test, and other testing elements included in this book are *not* derived from the real exam questions, so don’t memorize the answers to these questions and assume that doing so will enable you to pass the exam. You should learn the underlying topic, as described in the text of the book. This will let you answer the questions provided with this book *and* pass the exam. Learning the underlying topic

is also the approach that will serve you best in the workplace—the ultimate goal of a certification.

To get the most out of this book, you should read each chapter from start to finish before going to the chapter-end elements. They are most useful for checking and reinforcing your understanding. Even if you’re already familiar with a topic, you should skim the chap- ter. There are a number of subtleties to Java that you could easily not encounter even when working with Java, even for years.

#### Free Online Learning Environment

This book provides a free online interactive learning environment and test bank with sev- eral additional elements. The online test bank includes:

**Sample Tests** All of the questions in this book, including the 20-question assessment test at the end of this introduction and over 130 questions that make up the Review Question sections for each chapter. In addition, there are three 60-question Practice Exams to test your knowledge of the material. The online test bank runs on multiple devices.

**Electronic Flashcards** Over 200 questions in flashcard format (a question followed by a single correct answer). You can use these to reinforce your learning and provide last-minute test prep before the exam.

**Glossary** The key terms from this book and their definitions are available as a fully searchable PDF.



Go to [www.sybex.com/go/ocajavase8](http://www.sybex.com/go/ocajavase8) to register and gain access to this comprehensive study tool package.

#### Conventions Used in This Book

This book uses certain typographic styles in order to help you quickly identify important information and to avoid confusion about the meaning of words, such as onscreen prompts. In particular, look for the following styles:

* *Italicized text* indicates key terms that are described at length for the first time in a chapter. (Italics are also used for emphasis.)
* A monospaced font indicates code or command-line text.
* *Italicized monospaced text* indicates a variable.

In addition to these text conventions, which can apply to individual words or entire paragraphs, a few conventions highlight segments of text:



A note indicates information that’s useful or interesting. It is often some- thing to pay special attention to for the exam.

A sidebar is like a note but longer. The information in a sidebar is useful, but it doesn’t ﬁt into the main ﬂow of the text.

**Sidebars**



A real world scenario describes a task or an example that’s particularly grounded in the real world. Although interesting, the scenario will not show up on the exam.

**Real World Scenario**

#### OCA Exam Objectives

*OCA: Oracle Certified Associate Java SE 8 Programmer I Study Guide: Exam 1Z0-808* has been written to cover every OCA exam objective. The following table provides a break- down of this book’s exam coverage, showing you the chapter where each objective or

sub-objective is covered:

**Exam Objective Chapter**

* **Java Basics**

Define the scope of variables 1

Define the structure of a Java class 1

Create executable Java applications with a main method; run a Java program 1 from the command line, including console output

Import other Java packages to make them accessible in your code 1

Compare and contrast the features and components of Java such as platform 1 independence, object orientation, encapsulation, etc.

* **Working with Java Data Types**

Declare and initialize variables (including casting of primitive data types) 1

Differentiate between object reference variables and primitive variables 1

Know how to read or write to object fields 1

Explain an Object’s Lifecycle (creation, “dereference by reassignment,” and 1

garbage collection)

Develop code that uses wrapper classes such as Boolean, Double, and Integer 1

* **Using Operators and Decision Constructs**

Use Java operators, including parentheses to override operator precedence 2

Test equality between Strings and other objects using == and equals () 3

Create if and if/else and ternary constructs 2

Use a switch statement 2

* **Creating and Using Arrays**

Declare, instantiate, initialize, and use a one-dimensional array 3

Declare, instantiate, initialize, and use multi-dimensional array 3

**Exam Objective Chapter**

* + **Using Loop Constructs**

Create and use while loops 2

Create and use for loops including the enhanced for loop 2

Create and use do/while loops 2

Compare loop constructs 2

Use break and continue 2

* + **Working with Methods and Encapsulation**

Create methods with arguments and return values, including overloaded 4

methods

Apply the static keyword to methods and fields 4

Create and overload constructors, including impact on default constructors 4

Apply access modifiers 4

Apply encapsulation principles to a class 4

Determine the effect upon object references and primitive values when they 4

are passed into methods that change the values

* + **Working with Inheritance**

Describe inheritance and its benefits 5

Develop code that demonstrates the use of polymorphism, including 5

overriding and object type versus reference type

Determine when casting is necessary 5

Use super and this to access objects and constructors 5

Use abstract classes and interfaces 5

* + **Handling Exceptions**

Differentiate among checked exceptions, unchecked exceptions, and Errors 6

Create a try-catch block and determine how exceptions alter normal 6

program flow

*(continued)*

**Exam Objective Chapter**

Describe the advantages of Exception handling 6

Create and invoke a method that throws an exception 6

Recognize common exception classes (such as NullPointerException, Arith- 6

meticException, ArrayIndexOutOfBoundsException, ClassCastException)

* **Working with Selected Classes from the Java API**

Manipulate data using the StringBuilder class and its methods 3

Creating and manipulating Strings 3

Create and manipulate calendar data using classes from java.time.LocalDate- 3 Time, java.time.LocalDate, java.time.LocalTime, java.time.format.DateTime- Formatter, java.time.Period

Declare and use an ArrayList of a given type 3

Write a simple Lambda expression that consumes a Lambda Predicate 4

expression

#### Assessment Test

1. What is the result of the following class? (Choose all that apply)

1: public class \_C {

2: private static int $;

3: public static void main(String[] main) { 4: String a\_b;

5: System.out.print($);

6: System.out.print(a\_b);

7: } }

* 1. Compiler error on line 1.
  2. Compiler error on line 2.
  3. Compiler error on line 4.
  4. Compiler error on line 5.
  5. Compiler error on line 6.
  6. 0null
  7. nullnull

1. What is the result of the following code?

String s1 = "Java"; String s2 = "Java";

StringBuilder sb1 = new StringBuilder(); sb1.append("Ja").append("va"); System.out.println(s1 == s2); System.out.println(s1.equals(s2)); System.out.println(sb1.toString() == s1); System.out.println(sb1.toString().equals(s1));

* 1. true is printed out exactly once.
  2. true is printed out exactly twice.
  3. true is printed out exactly three times.
  4. true is printed out exactly four times.
  5. The code does not compile.

1. What is the output of the following code? (Choose all that apply)

1: interface HasTail { int getTailLength(); } 2: abstract class Puma implements HasTail {

3: protected int getTailLength() {return 4;}

4: }

5: public class Cougar extends Puma {

6: public static void main(String[] args) { 7: Puma puma = new Puma();

8: System.out.println(puma.getTailLength());

9: }

10:

11: public int getTailLength(int length) {return 2;} 12: }

* 1. 2
  2. 4
  3. The code will not compile because of line 3.
  4. The code will not compile because of line 5.
  5. The code will not compile because of line 7.
  6. The code will not compile because of line 11.
  7. The output cannot be determined from the code provided.

1. What is the output of the following program?

1: public class FeedingSchedule {

2: public static void main(String[] args) { 3: boolean keepGoing = true;

4: int count = 0;

5: int x = 3;

6: while(count++ < 3) {

7: int y = (1 + 2 \* count) % 3;

8: switch(y) {

9: default:

10: case 0: x -= 1; break;

11: case 1: x += 5;

12: }

13: }

14: System.out.println(x);

15: } }

* 1. 4
  2. 5
  3. 6
  4. 7
  5. 13
  6. The code will not compile because of line 7.

1. What is the output of the following code snippet?

13: System.out.print("a");

14: try {

15: System.out.print("b");

16: throw new IllegalArgumentException(); 17: } catch (RuntimeException e) {

18: System.out.print("c"); 19: } finally {

20: System.out.print("d");

21: }

22: System.out.print("e");

* 1. abe
  2. abce
  3. abde
  4. abcde
  5. The code does not compile.
  6. An uncaught exception is thrown.

1. What is the result of the following program?

1: public class MathFunctions {

2: public static void addToInt(int x, int amountToAdd) { 3: x = x + amountToAdd;

4: }

5: public static void main(String[] args) { 6: int a = 15;

7: int b = 10;

8: MathFunctions.addToInt(a, b); 9: System.out.println(a); } }

* 1. 10
  2. 15
  3. 25
  4. Compiler error on line 3.
  5. Compiler error on line 8.
  6. None of the above.

1. What is the result of the following code?

int[] array = {6,9,8};

List<Integer> list = new ArrayList<>();

list.add(array[0]);

list.add(array[2]);

list.set(1, array[1]); list.remove(0); System.out.println(list); **A.** [8]

**B.** [9]

1. Something like [Ljava.lang.String;@160bc7c0
2. An exception is thrown.
3. The code does not compile.
4. What is the output of the following code?

1: public class Deer {

2: public Deer() { System.out.print("Deer"); }

3: public Deer(int age) { System.out.print("DeerAge"); } 4: private boolean hasHorns() { return false; }

5: public static void main(String[] args) { 6: Deer deer = new Reindeer(5);

7: System.out.println(","+deer.hasHorns());

8: }

9: }

10: class Reindeer extends Deer {

11: public Reindeer(int age) { System.out.print("Reindeer"); } 12: public boolean hasHorns() { return true; }

13: }

* 1. DeerReindeer,false
  2. DeerReindeer,true
  3. ReindeerDeer,false
  4. ReindeerDeer,true
  5. DeerAgeReindeer,false
  6. DeerAgeReindeer,true
  7. The code will not compile because of line 7.
  8. The code will not compile because of line 12.

1. Which of the following statements are true? (Choose all that apply)
   1. Checked exceptions are intended to be thrown by the JVM (and not the programmer).
   2. Checked exceptions are required to be caught or declared.
   3. Errors are intended to be thrown by the JVM (and not the programmer).
   4. Errors are required to be caught or declared.
   5. Runtime exceptions are intended to be thrown by the JVM (and not the programmer).
   6. Runtime exceptions are required to be caught or declared.
2. Which are true of the following code? (Choose all that apply)

1: import java.util.\*;

2: public class Grasshopper {

3: public Grasshopper(String n) { 4: name = n;

5: }

6: public static void main(String[] args) { 7: Grasshopper one = new Grasshopper("g1"); 8: Grasshopper two = new Grasshopper("g2"); 9: one = two;

10: two = null; 11: one = null; 12: }

13: private String name; }

* 1. Immediately after line 9, no grasshopper objects are eligible for garbage collection.
  2. Immediately after line 10, no grasshopper objects are eligible for garbage collection.
  3. Immediately after line 9, only one grasshopper object is eligible for garbage collection.
  4. Immediately after line 10, only one grasshopper object is eligible for garbage collection.
  5. Immediately after line 11, only one grasshopper object is eligible for garbage collection.
  6. The code compiles.
  7. The code does not compile.

1. What is the output of the following program?

1: public class FeedingSchedule {

2: public static void main(String[] args) { 3: int x = 5, j = 0;

4: OUTER: for(int i=0; i<3; ) 5: INNER: do {

6: i++; x++;

7: if(x > 10) break INNER;

8: x += 4;

9: j++;

10: } while(j <= 2);

11: System.out.println(x);

12: } }

* 1. 10
  2. 12
  3. 13
  4. 17
  5. The code will not compile because of line 4.
  6. The code will not compile because of line 6.

1. What is the result of the following program?

1: public class Egret {

2: private String color; 3: public Egret() {

4: this("white");

5: }

6: public Egret(String color) { 7: color = color;

8: }

9: public static void main(String[] args) { 10: Egret e = new Egret();

11: System.out.println("Color:" + e.color); 12: }

13: }

* 1. Color:
  2. Color:null
  3. Color:White
  4. Compiler error on line 4.
  5. Compiler error on line 10.
  6. Compiler error on line 11.

1. What is the output of the following program?

1: public class BearOrShark {

2: public static void main(String[] args) { 3: int luck = 10;

4: if((luck>10 ? luck++: --luck)<10) { 5: System.out.print("Bear");

6: } if(luck<10) System.out.print("Shark"); 7: } }

* 1. Bear
  2. Shark
  3. BearShark
  4. The code will not compile because of line 4.
  5. The code will not compile because of line 6.
  6. The code compiles without issue but does not produce any output.

1. Assuming we have a valid, non-null HenHouse object whose value is initialized by the blank line shown here, which of the following are possible outputs of this application? (Choose all that apply)

1: class Chicken {}

2: interface HenHouse { public java.util.List<Chicken> getChickens(); } 3: public class ChickenSong {

4: public static void main(String[] args) { 5: HenHouse house =

6: Chicken chicken = house.getChickens().get(0); 7: for(int i=0; i<house.getChickens().size();

8: chicken = house.getChickens().get(i++)) { 9: System.out.println("Cluck");

10: } } }

* 1. The code will not compile because of line 6.
  2. The code will not compile because of lines 7–8.
  3. The application will compile but not produce any output.
  4. The application will output Cluck exactly once.
  5. The application will output Cluck more than once.
  6. The application will compile but produce an exception at runtime.

1. Which of the following statements can be inserted in the blank line so that the code will compile successfully? (Choose all that apply)

public interface CanSwim {}

public class Amphibian implements CanSwim {} class Tadpole extends Amphibian {}

public class FindAllTadPole {

public static void main(String[] args) { List<Tadpole> tadpoles = new ArrayList<Tadpole>(); for(Amphibian amphibian : tadpoles) {

tadpole = amphibian;

} } }

* 1. CanSwim
  2. Long
  3. Amphibian
  4. Tadpole
  5. Object

1. What individual changes, if any, would allow the following code to compile? (Choose all that apply)

1: public interface Animal { public default String getName() { return null; } } 2: interface Mammal { public default String getName() { return null; } }

3: abstract class Otter implements Mammal, Animal {}

* 1. The code compiles without issue.
  2. Remove the default method modifier and method implementation on line 1.
  3. Remove the default method modifier and method implementation on line 2.
  4. Remove the default method modifier and method implementation on lines 1 and 2.
  5. Change the return value on line 1 from null to "Animal".
  6. Override the getName() method with an abstract method in the Otter class.
  7. Override the getName() method with a concrete method in the Otter class.

1. Which of the following lines can be inserted at line 11 to print true? (Choose all that apply)

10: public static void main(String[] args) { 11: // INSERT CODE HERE

12: }

13: private static boolean test(Predicate<Integer> p) { 14: return p.test(5);

15: }

* 1. System.out.println(test(i -> i == 5));
  2. System.out.println(test(i -> {i == 5;}));
  3. System.out.println(test((i) -> i == 5));
  4. System.out.println(test((int i) -> i == 5);
  5. System.out.println(test((int i) -> {return i == 5;}));
  6. System.out.println(test((i) -> {return i == 5;}));

1. Which of the following print out a date representing April 1, 2015? (Choose all that apply)
   1. System.out.println(LocalDate.of(2015, Calendar.APRIL, 1));
   2. System.out.println(LocalDate.of(2015, Month.APRIL, 1));
   3. System.out.println(LocalDate.of(2015, 3, 1));
   4. System.out.println(LocalDate.of(2015, 4, 1));
   5. System.out.println(new LocalDate(2015, 3, 1));
   6. System.out.println(new LocalDate(2015, 4, 1));
2. Bytecode is in a file with which extension?
   1. .bytecode
   2. .bytes
   3. .class
   4. .exe
   5. .javac
   6. .java
3. Which of the following are checked exceptions? (Choose all that apply)
   1. Exception
   2. IllegalArgumentException
   3. IOException
   4. NullPointerException
   5. NumberFormatException
   6. StackOverflowError

#### Answers to Assessment Test

1. E. Option E is correct because local variables require assignment before referencing them. Option D is incorrect because class and instance variables have default values and allow referencing. a\_b defaults to a null value. Options A, B, and C are incorrect because identifiers may begin with a letter, underscore, or dollar sign. Options F and G are incorrect because the code does not compile. If a\_b was an instance variable, the code would compile and output 0null. For more information, see Chapter 1.
2. C. String literals are used from the string pool. This means that *s1* and *s2* refer to the same object and are equal. Therefore, the first two print statements print true. The third print statement prints false because toString() uses a method to compute the value and it is not from the string pool. The final print statement again prints true because equals() looks at the values of String objects. For more information, see Chapter 3.
3. C, D, E. First, the method getTailLength() in the interface HasTail is assumed to be public, since it is part of an interface. The implementation of the method on line 3 is therefore an invalid override, as protected is a more restrictive access modifier than public, so option C is correct. Next, the class Cougar implements an overloaded ver- sion of getTailLength(), but since the declaration in the parent class Puma is invalid, it needs to implement a public version of the method. Since it does not, the declaration of Puma is invalid, so option D is correct. Option E is incorrect, since Puma is marked abstract and cannot be instantiated. The overloaded method on line 11 is declared correctly, so option F is not correct. Finally, as the code has multiple compiler errors, options A, B, and G can be eliminated. For more information, see Chapter 5.
4. C. The code compiles and runs without issue; therefore, option F is incorrect. This type of problem is best examined one loop iteration at a time:
   * The loop continues as count loop expression evaluates to 0 < 3, which is true, with y taking a new value of 1. The value of y is set to:

y = (1 + 2 \* 1) % 3

= (1 + 2) % 3

= 3 % 3

= 0

* + The first case block is called and the value of x is then set to:

x = 3 - 1 = 2

* + The loop continues as count loop expression evaluates to 1 < 3, which is true, with y taking a new value of 2. The value of y is set to:

y = (1 + 2 \* 2) % 3

= (1 + 4) % 3

= 4 % 3

= 2

* The default block is called and the value of x is set to:

x = 2 - 1 = 1

* The loop continues as the count loop expression evaluates to 2 < 3, which is true, with y taking a new value of 3. The value of y is set to:

y = (1 + 2 \* 3) % 3

= (1 + 6) % 3

= 7 % 3

= 1

* The second case block is called and the value of x is then set to:

x = 1 + 5 = 6

* The loop ends as the count loop expression evaluates to 3 < 3, with y also taking a new value of 4. The most recent value of x, 6, is output, so the answer is option C. For more information, see Chapter 2.

1. D. The code starts running and prints a and b on lines 13 and 15. Line 16 throws an exception, which is caught on line 17. After line 18 prints c, the finally block is run and d is printed. Then the try statement ends and e is printed on line 22. For more information, see Chapter 6.
2. B. The code compiles successfully, so options D and E are incorrect. The value of *a* cannot be changed by the addToInt method, no matter what the method does, because only a copy of the variable is passed into the parameter *x*. Therefore, *a* does not change and the output on line 9 is 15. For more information, see Chapter 4.
3. B. The array is allowed to use an anonymous initializer because it is in the same line as the declaration. The ArrayList uses the diamond operator allowed since Java 7. This specifies the type matches the one on the left without having to re-type it. After adding the two elements, *list* contains [6, 8]. We then replace the element at index 1 with 9, resulting in [6, 9]. Finally, we remove the element at index 0, leaving [9]. Option C is incorrect because arrays output something like that rather than an ArrayList. For more information, see Chapter 3.
4. A. The code compiles and runs without issue, so options G and H are incorrect. First, the Reindeer object is instantiated using the constructor that takes an int value. Since there is no explicit call to the parent constructor, the default no-argument super()

is inserted as the first line of the constructor. The output is then Deer, followed by Reindeer in the child constructor, so only options A and B can be correct. Next, the method hasHorns() looks like an overridden method, but it is actually a hidden method since it is declared private in the parent class. Because the hidden method is referenced in the parent class, the parent version is used, so the code outputs false, and option A is the correct answer.

1. B, C. Only checked exceptions are required to be handled (caught) or declared. Run- time exceptions are commonly thrown by both the JVM and programmer code. Checked exceptions are usually thrown by programmer code. Errors are intended to be

thrown by the JVM. While a programmer could throw one, this would be a horrible practice. For more information, see Chapter 6.

1. C, D, F. Immediately after line 9, only Grasshopper g1 is eligible for garbage collection since both one and two point to Grasshopper g2. Immediately after line 10, we still only have Grasshopper g1 eligible for garbage collection. Reference one points to g1 and reference two is null. Immediately after line 11, both Grasshopper objects are eli- gible for garbage collection since both one and two point to null. The code does com- pile. Although it is traditional to declare instance variables early in the class, you don’t have to. For more information, see Chapter 1.
2. B. The code compiles and runs without issue; therefore, options E and F are incorrect. This type of problem is best examined one loop iteration at a time:
   * On the first iteration of the outer loop i is 0, so the loop continues.
   * On the first iteration of the inner loop, i is updated to 1 and x to 6. The if-then

statement branch is not executed, and x is increased to 10 and j to 1.

* + On the second iteration of the inner loop (since j = 1 and 1 <= 2), i is updated to 2 and x to 11. At this point, the if-then branch will evaluate to true for the remainder of the program run, which causes the flow to break out of the inner loop each time it is reached.
  + On the second iteration of the outer loop (since i = 2), i is updated to 3 and x to

1. As before, the inner loop is broken since x is still greater than 10.
   * On the third iteration of the outer loop, the outer loop is broken, as i is already not less than 3. The most recent value of x, 12, is output, so the answer is option B. For more information, see Chapter 2.
2. B. Line 10 calls the constructor on lines 3–5. That constructor calls the other construc- tor. However, the constructor on lines 6–8 assigns the method parameter to itself, which leaves the *color* instance variable on line 2 set to its default value of *null*. For more information, see Chapter 4.
3. C. The code compiles and runs without issue, so options D and E are correct. Remem- ber that only one of the right-hand ternary expressions will be evaluated at runtime. Since luck is not less than 10, the second expression, --luck, will be evaluated, and since the pre-increment operator was used, the value returned will be 9, which is less than 10. So the first if-then statement will be visited and Bear will be output. Notice there is no else statement on line 6. Since luck is still less than 10, the second if-then statement will also be reached and Shark will be output; therefore, the correct answer is option C. For more information, see Chapter 2.
4. D, E, F. The code compiles without issue, so options A and B are incorrect. If house

.getChickens() returns an array of one element, the code will output Cluck once, so option D is correct. If house.getChickens() returns an array of multiple elements, the code will output Cluck once for each element in the array, so option E is correct. Alter- natively, if house.getChickens() returns an array of zero elements, then the code will

throw an IndexOutOfBoundsException on the call to house.getChickens().get(0); therefore, option C is not possible and option F is correct. The code will also throw an exception if the array returned by house.getChickens() is null, so option F is possible under multiple circumstances. For more information, see Chapter 2.

1. A, C, E. The for-each loop automatically casts each Tadpole object to an Amphibian reference, which does not require an explicit cast because Tadpole is a subclass of Amphibian. From there, any parent class or interface that Amphibian inherits from is permitted without an explicit cast. This includes CanSwim, the interface Amphibian implements, and Object, which all classes extend from, so options A and E are correct. Option C is also correct since the reference is being cast to the same type, so no explicit cast is required. Option B is incorrect, since Long is not a parent of Amphibian. Option D is incorrect as well, although an explicit cast to Tadpole on the right-hand side of the expression would be required to allow the code to compile. For more information, see Chapter 5.
2. D, F, G. The code does not compile, since a class cannot inherit two interfaces that both define default methods with the same signature, unless the class implementing the interfaces overrides it with an abstract or concrete method. Therefore, option A is incorrect and options F and G are correct. The alternate approach is to make the getName() method abstract in the interfaces, because an interface may inherit two abstract methods with the same signature. The change must be made to both inter- faces, though, so options B and C are incorrect if taken individually, and option D is correct since the changes are taken together. For more information, see Chapter 5.
3. A, C, F. The only functional programming interface you need to memorize for the exam is Predicate. It takes a single parameter and returns a boolean. Lambda expres- sions with one parameter are allowed to omit the parentheses around the parameter list, making options A and C correct. The return statement is optional when a single statement is in the body, making option F correct. Option B is incorrect because a return statement must be used if braces are included around the body. Options D and E are incorrect because the type is Integer in the predicate and int in the lambda. Autoboxing works for collections not inferring predicates. If these two were changed to Integer, they would be correct. For more information, see Chapter 4.
4. B, D. The new date APIs added in Java 8 use static methods rather than a constructor to create a new date, making options E and F incorrect. The months are indexed start- ing with 1 in these APIs, making options A and C incorrect. Option A uses the old Calendar constants which are indexed from 0. Therefore, options B and D are correct. For more information, see Chapter 3.
5. C. Files with the .java extension contain the Java source code and are compiled to files with the .class extension that contain the bytecode. For more information, see Chapter 1.
6. A, C. Option A is the exception base class, which is a checked exception. Options B, D, and E extend RuntimeException directly or indirectly and therefore are unchecked exceptions. Option F is a throwable and not an exception, and so should not be caught or declared. For more information, see Chapter 6.

# OCA: Oracle®

## Certified Associate Java® SE 8 Programmer I

**Study Guide Exam 1Z0-808**



**Chapter**

**1**

**Java Building Blocks**

###### OCA EXAM OBJECTIVES COVERED IN THIS CHAPTER:

* + Java Basics
    - Define the scope of variables
    - Define the structure of a Java class
    - Create executable Java applications with a main method; run a Java program from the command line; including console output
    - Import other Java packages to make them accessible in your code
    - Compare and contrast the features and components of Java such as platform independence, object orientation, encapsu- lation, etc.
  + Working with Java Data Types
    - Declare and initialize variables (including casting or primitive types)
    - Differentiate between object reference variables and primi- tive variables
    - Know how to read or write to object fields
    - Explain an Object’s Lifecycle (creation, “dereference by reassignment” and garbage collection

Welcome to the beginning of your journey to become certified on Java. We assume this isn’t the first Java programming book you’ve read. Although we do talk about the basics, we do so

only because we want to make sure you have all the terminology and detail you’ll need for the OCA exam. If you’ve never written a Java program before, we recommend you pick up an introductory book on any version of Java—something like *Head First Java*, *2nd Edition* (O’Reilly Media, 2005); *Java for Dummies* (For Dummies, 2014), or *Thinking in Java, 4th Edition* (Prentice Hall, 2006). (It’s okay if the book covers an older version of Java—even Java 1.3 is fine.) Then come back to this certification study guide.

This chapter covers the fundamentals of Java. You’ll see how to define and run a Java class, and learn about packages, variables, and the object life cycle.

Understanding the Java Class Structure

In Java programs, classes are the basic building blocks. When defining a *class*, you describe all the parts and characteristics of one of those building blocks. To use most classes, you have to create objects. An *object* is a runtime instance of a class in memory. All the various objects of all the different classes represent the state of your program.

In the following sections, we’ll look at fields, methods, and comments. We’ll also explore the relationship between classes and files.

##### Fields and Methods

Java classes have two primary elements: *methods*, often called functions or procedures in other languages, and *fields*, more generally known as variables. Together these are called the *members* of the class. Variables hold the state of the program, and methods operate on that state. If the change is important to remember, a variable stores that change. That’s all classes really do. It’s the programmer who creates and arranges these elements in such a way that the resulting code is useful and, ideally, easy for other programmers to understand.

Other building blocks include interfaces, which you’ll learn about in Chapter 5, “Class Design,” and enums, which you’ll learn about when you start studying for the OCP exam.

The simplest Java class you can write looks like this:

1: public class Animal { 2: }

Java calls a word with special meaning a *keyword*. The public keyword on line 1 means the class can be used by other classes. The class keyword indicates you’re defining a class. Animal gives the name of the class. Granted, this isn’t a very interesting class, so add your first field:

1: public class Animal { 2: String name;

3: }



The line numbers aren’t part of the program; they’re just there to make the code easier to talk about.

On line 2, we define a variable named name. We also define the type of that variable to be a String. A String is a value that we can put text into, such as "**this is a string**". String is also a class supplied with Java. Next you can add methods:

1: public class Animal { 2: String name;

3: public String getName() { 4: return name;

5: }

6: public void setName(String *newName*) { 7: name = *newName*;

8: }

9: }

On lines 3–5, you’ve defined your first method. A method is an operation that can be called. Again, public is used to signify that this method may be called from other classes. Next comes the return type—in this case, the method returns a String. On lines 6–8 is another method. This one has a special return type called *void*. void means that no value at all is returned. This method requires information be supplied to it from the calling method; this information is called a *parameter*. setName has one parameter named *newName*, and it

is of type String. This means the caller should pass in one String parameter and expect nothing to be returned.

The full declaration of a method is called a *method signature*. In this example, can you identify the return type and parameters?

public int numberVisitors(int *month*)

The return type is int, which is a numeric type. There’s one parameter named *month*, which is of type int as well.

##### Comments

Another common part of the code is called a *comment*. Because comments aren’t execut- able code, you can place them anywhere. Comments make your code easier to read. You won’t see many comments on the exam—the exam creators are trying to make the code difficult to read—but you’ll see them in this book as we explain the code. And we hope you use them in your own code. There are three types of comments in Java. The first is called a single-line comment:

// comment until end of line

A single-line comment begins with two slashes. Anything you type after that on the same line is ignored by the compiler. Next comes the multiple-line comment:

/\* Multiple

\* line comment

\*/

A multiple-line comment (also known as a multiline comment) includes anything starting from the symbol /\* until the symbol \*/. People often type an asterisk (\*) at the beginning of each line of a multiline comment to make it easier to read, but you don’t have to. Finally, we have a Javadoc comment:

/\*\*

* Javadoc multiple-line comment
* @author Jeanne and Scott

\*/

This comment is similar to a multiline comment except it starts with /\*\*. This special syntax tells the Javadoc tool to pay attention to the comment. Javadoc comments have a specific structure that the Javadoc tool knows how to read. You won’t see a Javadoc com- ment on the exam—just remember it exists so you can read up on it online when you start writing programs for others to use.

As a bit of practice, can you identify which type of comment each of these five words is in? Is it a single-line or a multiline comment?

/\*

* // anteater

\*/

// bear

// // cat

// /\* dog \*/

/\* elephant \*/

/\*

\* /\* ferret \*/

\*/

Did you look closely? Some of these are tricky. Even though comments technically aren’t on the exam, it is good to practice to look at code carefully.

Okay, on to the answers. anteater is in a multiline comment. Everything between /\* and \*/ is part of a multiline comment—even if it includes a single-line comment within it! bear is your basic single-line comment. cat and dog are also single-line comments.

Everything from // to the end of the line is part of the comment, even if it is another type of comment. elephant is your basic multiline comment.

The line with ferret is interesting in that it doesn’t compile. Everything from the first /\*

to the first \*/ is part of the comment, which means the compiler sees something like this:

/\* \*/ \*/

We have a problem. There is an extra \*/. That’s not valid syntax—a fact the compiler is happy to inform you about.

##### Classes vs. Files

Most of the time, each Java class is defined in its own \*.java file. It is usually public, which means any code can call it. Interestingly, Java does not require that the class be public. For example, this class is just fine:

1: class Animal { 2: String name;

3: }

You can even put two classes in the same file. When you do so, at most one of the classes in the file is allowed to be public. That means a file containing the following is also fine:

1: public class Animal { 2: private String name; 3: }

4: class Animal2 { 5: }

If you do have a public class, it needs to match the filename. public class Animal2 would not compile in a file named Animal.java. In Chapter 5, we will discuss what non- public access means.

Writing a *main()* Method

A Java program begins execution with its main() *method*. A main() method is the gateway between the startup of a Java process, which is managed by the *Java Virtual Machine* (JVM), and the beginning of the programmer’s code. The JVM calls on the underlying system to allocate memory and CPU time, access files, and so on.

The main() method lets us hook our code into this process, keeping it alive long enough to do the work we’ve coded. The simplest possible class with a main() method looks like this:

1: public class Zoo {

2: public static void main(String[] args) { 3:

4: }

5:}

This code doesn’t do anything useful (or harmful). It has no instructions other than to declare the entry point. It does illustrate, in a sense, that what you can put in a main() method is arbitrary. Any legal Java code will do. In fact, the only reason we even need a

class structure to start a Java program is because the language requires it. To compile and execute this code, type it into a file called Zoo.java and execute the following:

$ javac Zoo.java

$ java Zoo

If you don’t get any error messages, you were successful. If you do get error messages, check that you’ve installed a Java Development Kit (JDK) and not a Java Runtime Environment (JRE), that you have added it to the PATH, and that you didn’t make any typos in the example. If you have any of these problems and don’t know what to do, post a question with the error message you received in the Beginning Java forum at CodeRanch ([www.coderanch.com/](http://www.coderanch.com/) forums/f-33/java).

To compile Java code, the file must have the extension .java. The name of the file must

match the name of the class. The result is a file of *bytecode* by the same name, but with a .class filename extension. Bytecode consists of instructions that the JVM knows how to execute. Notice that we must omit the .class extension to run Zoo.java because the period has a reserved meaning in the JVM.

The rules for what a Java code file contains, and in what order, are more detailed than what we have explained so far (there is more on this topic later in the chapter). To keep things simple for now, we’ll follow a subset of the rules:

* Each file can contain only one class.
* The filename must match the class name, including case, and have a .java extension.

Suppose we replace line 3 in Zoo.java with System.out.println("Welcome!");. When we compile and run the code again, we’ll get the line of output that matches what’s between the quotes. In other words, the program will output Welcome!.

Let’s first review the words in the main() method’s signature, one at a time. The key- word public is what’s called an *access modifier.* It declares this method’s level of exposure to potential callers in the program. Naturally, public means anyplace in the program.

You’ll learn about access modifiers in Chapter 4, “Methods and Encapsulation.”

The keyword *static* binds a method to its class so it can be called by just the class name, as in, for example, Zoo.main(). Java doesn’t need to create an object to call the main() method—which is good since you haven’t learned about creating objects yet! In fact, the JVM does this, more or less, when loading the class name given to it. If a main() method isn’t present in the class we name with the .java executable, the process will throw an error and terminate. Even if a main() method is present, Java will throw an exception if it isn’t static. A nonstatic main() method might as well be invisible from the point of view of the JVM. We’ll see static again in Chapter 4.

The keyword *void* represents the *return type*. A method that returns no data returns control to the caller silently. In general, it’s good practice to use void for methods that change an object’s state. In that sense, the main() method changes the program state from started to finished. We will explore return types in Chapter 4 as well. Excited for Chapter 4 yet?

Finally we arrive at the main() method’s parameter list, represented as an array of java. lang.String objects. In practice, you can write String[] args, String args[] or String... args; the compiler accepts any of these. The variable name *args* hints that this list contains values that were read in (arguments) when the JVM started. You can use any name you like, though. The characters [] are brackets and represent an array. An array is a fixed-size list of items that are all of the same type. The characters ... are called varargs (variable argument lists). You will learn about String in Chapter 2, “Operators and Statements.” Arrays and varargs will follow in Chapter 3, “Core Java APIs.”

Let’s see how to use the *args* parameter. First we modify the Zoo program to print out the first two arguments passed in:

public class Zoo {

public static void main(String[] args) { System.out.println(args[0]); System.out.println(args[1]);

} }

args[0] accesses the first element of the array. That’s right: array indexes begin with 0 in Java. To run it, type this:

$ **javac Zoo.java**

$ java Zoo Bronx Zoo

The output is what you might expect:

Bronx Zoo

The program correctly identifies the first two “words” as the arguments. Spaces are used to separate the arguments. If you want spaces inside an argument, you need to use quotes as in this example:

$ javac Zoo.java

$ java Zoo "San Diego" Zoo

Now we have a space in the output:

San Diego Zoo

All command-line arguments are treated as String objects, even if they represent another data type:

$ javac Zoo.java

$ java Zoo Zoo 2

No matter. You still get the values output as Strings. In Chapter 2, you’ll learn how to convert Strings to numbers.

Zoo 2

Finally, what happens if you don’t pass in enough arguments?

$ javac Zoo.java

$ java Zoo Zoo

Reading args[0] goes fine and Zoo is printed out. Then Java panics. There’s no sec- ond argument! What to do? Java prints out an exception telling you it has no idea what to do with this argument at position 1. (You’ll learn about exceptions in Chapter 6, “Exceptions.”)

ZooException in thread "main" java.lang.ArrayIndexOutOfBoundsException: 1

at mainmethod.Zoo.main(Zoo.java:7)

To review, you need to have a JDK to compile because it includes a compiler. You do not need to have a JDK to run the code—a JRE is enough. Java class files run on the JVM and therefore run on any machine with Java rather than just the machine or operating system they happened to have been compiled on.

#### Understanding Package Declarations and Imports

Java comes with thousands of built-in classes, and there are countless more from developers like you. With all those classes, Java needs a way to organize them. It handles this in a way similar to a file cabinet. You put all your pieces of paper in folders. Java puts classes in *packages*. These are logical groupings for classes.

We wouldn’t put you in front of a file cabinet and tell you to find a specific paper.

Instead, we’d tell you which folder to look in. Java works the same way. It needs you to tell it which packages to look in to find code.

Suppose you try to compile this code:

public class ImportExample {

public static void main(String[] args) {

Random r = new Random(); // DOES NOT COMPILE System.out.println(r.nextInt(10));

}

}

The Java compiler helpfully gives you an error that looks like this:

Random cannot be resolved to a type

This error could mean you made a typo in the name of the class. You double-check and discover that you didn’t. The other cause of this error is omitting a needed *import* state- ment. Import statements tell Java which packages to look in for classes. Since you didn’t tell Java where to look for Random, it has no clue.

Trying this again with the import allows you to compile:

import java.util.Random; // import tells us where to find Random public class ImportExample {

public static void main(String[] args) { Random r = new Random();

System.out.println(r.nextInt(10)); // print a number between 0 and 9

}

}

Now the code runs; it prints out a random number between 0 and 9. Just like arrays,

Java likes to begin counting with 0.

Java classes are grouped into packages. The import statement tells the compiler which package to look in to find a class. This is similar to how mailing a letter works.

Imagine you are mailing a letter to 123 Main St., Apartment 9. The mail carrier first brings the letter to 123 Main St. Then she looks for the mailbox for apartment number 9. The address is like the package name in Java. The apartment number is like the class name in Java. Just as the mail carrier only looks at apartment numbers in the building, Java only looks for class names in the package.

Package names are hierarchical like the mail as well. The postal service starts with the top level, looking at your country first. You start reading a package name at the begin- ning too. If it begins with java or javax, this means it came with the JDK. If it starts with something else, it likely shows where it came from using the website name in reverse. From example, com.amazon.java8book tells us the code came from amazon.com. After the web- site name, you can add whatever you want. For example, com.amazon.java8.my.name also came from amazon.com. Java calls more detailed packages *child packages*. com.amazon

.java8book is a child package of com.amazon. You can tell because it’s longer and thus more specific.

You’ll see package names on the exam that don’t follow this convention. Don’t be surprised to see package names like a.b.c. The rule for package names is that they are mostly letters or numbers separated by dots. Technically, you’re allowed a couple of other characters between the dots. The rules are the same as for variable names, which you’ll see later in the chapter. The exam may try to trick you with invalid variable names. Luckily, it doesn’t try to trick you by giving invalid package names.

In the following sections, we’ll look at imports with wildcards, naming conflicts with imports, how to create a package of your own, and how the exam formats code.

##### Wildcards

Classes in the same package are often imported together. You can use a shortcut to import all the classes in a package:

import java.util.\*; // imports java.util.Random among other things public class ImportExample {

public static void main(String[] args) { Random r = new Random(); System.out.println(r.nextInt(10));

}

}

In this example, we imported java.util.Random and a pile of other classes. The \* is a

wildcard that matches all classes in the package. Every class in the java.util package is available to this program when Java compiles it. It doesn’t import child packages, fields, or methods; it imports only classes. (Okay, it’s only classes for now, but there’s a special type of import called the “static import” that imports other types. You’ll learn more about that in Chapter 4.)

You might think that including so many classes slows down your program, but it doesn’t. The compiler figures out what’s actually needed. Which approach you choose is personal preference.

Listing the classes used makes the code easier to read, especially for new programmers. Using the wildcard can shorten the import list. You’ll see both approaches on the exam.

##### Redundant Imports

Wait a minute! We’ve been referring to System without an import and Java found it just fine. There’s one special package in the Java world called java.lang. This package is special in that it is automatically imported. You can still type this package in an import statement, but you don’t have to. In the following code, how many of the imports do you think are redundant?

1: import java.lang.System;

2: import java.lang.\*;

3: import java.util.Random;

4: import java.util.\*;

5: public class ImportExample {

6: public static void main(String[] args) { 7: Random r = new Random();

8: System.out.println(r.nextInt(10));

9: }

10: }

The answer is that three of the imports are redundant. Lines 1 and 2 are redundant because everything in java.lang is automatically considered to be imported. Line 4 is also redundant in this example because Random is already imported from java.util.Random.

If line 3 wasn’t present, java.util.\* wouldn’t be redundant, though, since it would cover importing Random.

Another case of redundancy involves importing a class that is in the same package as the class importing it. Java automatically looks in the current package for other classes.

Let’s take a look at one more example to make sure you understand the edge cases for imports. For this example, Files and Paths are both in the package java.nio.file. You don’t need to memorize this package for the OCA exam (but you should know it for the OCP exam). When testing your understanding of packages and imports, the OCA exam will use packages you may never have seen before. The question will let you know which package the class is in if you need to know that in order to answer the question.

What imports do you think would work to get this code to compile?

public class InputImports { public void read(Files files) {

Paths.get("name");

}

}

There are two possible answers. The shorter one is to use a wildcard to import both at the same time:

import java.nio.file.\*;

The other answer is to import both classes explicitly:

import java.nio.file.Files; import java.nio.file.Paths;

Now let’s consider some imports that don’t work:

import java.nio.\*; // NO GOOD – a wildcard only matches

//class names, not "file.\*Files"

import java.nio.\*.\*; // NO GOOD – you can only have one wildcard

//and it must be at the end

import java.nio.files.Paths.\*; // NO GOOD – you cannot import methods

//only class names

##### Naming Conflicts

One of the reasons for using packages is so that class names don’t have to be unique across all of Java. This means you’ll sometimes want to import a class that can be found in mul- tiple places. A common example of this is the Date class. Java provides implementations

of java.util.Date and java.sql.Date. This is another example where you don’t need to

know the package names for the OCA exam—they will be provided to you. What import could we use if we want the java.util.Date version?

public class Conflicts { Date date;

// some more code

}

The answer should be easy by now. You can write either **import java.util.\*;** or

**import java.util.Date;**. The tricky cases come about when other imports are present:

import java.util.\*;

import java.sql.\*; // DOES NOT COMPILE

When the class is found in multiple packages, Java gives you the compiler error:

The type Date is ambiguous

In our example, the solution is easy—remove the java.sql.Date import that we don’t need. But what do we do if we need a whole pile of other classes in the java.sql package?

import java.util.Date;

import java.sql.\*;

Ah, now it works. If you explicitly import a class name, it takes precedence over any wildcards present. Java thinks, “Okay! The programmer really wants me to assume use of the java.util.Date class.”

One more example. What does Java do with “ties” for precedence?

import java.util.Date;

import java.sql.Date;

Java is smart enough to detect that this code is no good. As a programmer, you’ve claimed to explicitly want the default to be both the java.util.Date and java.sql.Date implementations. Because there can’t be two defaults, the compiler tells you:

The import java.sql.Date collides with another import statement

Sometimes you really do want to use Date from two different packages. When this hap- pens, you can pick one to use in the import and use the other’s fully qualiﬁed class name (the package name, a dot, and the class name) to specify that it’s special. For example:

import java.util.Date;

public class Conflicts { Date date;

java.sql.Date sqlDate;

}

Or you could have neither with an import and always use the fully qualiﬁed class name:

public class Conflicts {

java.util.Date date; java.sql.Date sqlDate;

}

**If You Really Need to Use Two Classes with the Same Name…**

##### Creating a New Package

Up to now, all the code we’ve written in this chapter has been in the *default package*. This is a special unnamed package that you should use only for throwaway code. You can tell the code is in the default package, because there’s no package name. On the exam, you’ll see the default package used a lot to save space in code listings. In real life, always name your packages to avoid naming conflicts and to allow others to reuse your code.

Now it’s time to create a new package. The directory structure on your computer is related to the package name. Suppose we have these two classes:

C:\temp\packagea\ClassA.java

package packagea; public class ClassA {

}

C:\temp\packageb\ClassB.java

package packageb; import packagea.ClassA; public class ClassB {

public static void main(String[] args) { ClassA a;

System.*out*.println("Got it");

}

}

When you run a Java program, Java knows where to look for those package names. In this

case, running from C:\temp works because both packagea and packageb are underneath it.

Compiling Code with Packages

You’ll learn Java much more easily by using the command line to compile and test your examples. Once you know the Java syntax well, you can switch to an integrated development environment (IDE) like Eclipse. An IDE will save you time in coding. But for the exam, your goal is to know details about the language and not have the IDE hide them for you.

Follow this example to make sure you know how to use the command line. If you have any problems following this procedure, post a question in the Beginning Java forum at CodeRanch ([www.coderanch.com/forums/f-33/java).](http://www.coderanch.com/forums/f-33/java)) Describe what you tried and what the error said.

**Windows Setup**

Create the two ﬁles:

* C:\temp\packagea\ClassA.java
* C:\temp\packageb\ClassB.java

Then type this command:

cd C:\temp

**Mac/Linux Setup**

Create the two ﬁles:

* + /tmp/packagea/ClassA.java
  + /tmp/packageb/ClassB.java

Then type this command:

cd /tmp

**To Compile**

Type this command:

javac packagea/ClassA.java packageb/ClassB.java

If this command doesn’t work, you’ll get an error message. Check your ﬁles carefully for typos against the provided ﬁles. If the command does work, two new ﬁles will be created:

packagea/ClassA.class and packageb/ClassB.class.

**To Run**

Type this command:

java packageb.ClassB

If it works, you’ll see Got it printed. You might have noticed we typed ClassB rather than

ClassB.class. In Java you don’t pass the extension when running a program.

**Class Paths and JARs**

You can also specify the location of the other ﬁles explicitly using a class path. This tech- nique is useful when the class ﬁles are located elsewhere or in special JAR ﬁles. A JAR ﬁle is like a zip ﬁle of mainly Java class ﬁles. This goes beyond what you’ll need to do on version 8 of the exam, although it appears on older versions.

On Windows, you type the following:

java -cp ".;C:\temp\someOtherLocation;c:\temp\myJar.jar" myPackage.MyClass

And on Mac OS/Linux, you type this:

java -cp ".:/tmp/someOtherLocation:/tmp/myJar.jar" myPackage.MyClass

The dot indicates you want to include the current directory in the class path. The rest of the command says to look for loose class ﬁles (or packages) in someOtherLocation and within myJar.jar. Windows uses semicolons to separate parts of the class path; other operating systems use colons.

Finally, you can use a wildcard (\*) to match all the JARs in a directory. Here’s an example:

java -cp "C:\temp\directoryWithJars\\*" myPackage.MyClass

This command will add all the JARs to the class path that are in directoryWithJars. It won’t include any JARs in the class path that are in a subdirectory of directoryWithJars.

##### Code Formatting on the Exam

Not all questions will include the imports. If the exam isn’t asking about imports in the question, it will often omit the imports to save space. You’ll see examples with line numbers that don’t begin with 1 in this case. The question is telling you, “Don’t worry—imagine

the code we omitted is correct; just focus on what I’m giving you.” This means when you do see the line number 1 or no line numbers at all, you have to make sure imports aren’t missing. Another thing the exam does to save space is to merge code on the same line. You should expect to see code like the following and to be asked whether it compiles. (You’ll learn about ArrayList in Chapter 3—assume that part is good for now.)

6: public void method(ArrayList list) {

7: if (list.isEmpty()) { System.*out*.println("e");

8: } else { System.*out*.println("n"); 9: } }

The answer here is that it does compile because the code starts below the imports. Now, what about this one? Does it compile?

1: public class LineNumbers {

2: public void method(ArrayList list) {

3: if (list.isEmpty()) { System.*out*.println("e");

4: } else { System.*out*.println("n"); 5: } } }

For this one, you would answer “Does not compile.” Since the code begins with line 1, you don’t get to assume that valid imports were provided earlier. The exam will let

you know what package classes are in unless they’re covered in the objectives. You’ll be expected to know that ArrayList is in java.util—at least you will once you get to Chapter 3 of this book!

You’ll also see code that doesn’t have a main() method. When this happens, assume the main() method, class definition, and all necessary imports are present. You’re just being asked if the part of the code you’re shown compiles when dropped into valid sur- rounding code.

#### Creating Objects

Our programs wouldn’t be able to do anything useful if we didn’t have the ability to create new objects. Remember that an object is an instance of a class. In the following sections, we’ll look at constructors, object fields, instance initializers, and the order in which values are initialized.

##### Constructors

To create an instance of a class, all you have to do is write new before it. For example:

Random r = new Random();

First you declare the type that you’ll be creating (Random) and give the variable a name (r). This gives Java a place to store a reference to the object. Then you write new Random() to actually create the object.

Random() looks like a method since it is followed by parentheses. It’s called a *construc- tor*, which is a special type of method that creates a new object. Now it’s time to define a constructor of your own:

public class Chick { public Chick() {

System.out.println("in constructor");

}

}

There are two key points to note about the constructor: the name of the constructor

matches the name of the class, and there’s no return type. You’ll likely see a method like this on the exam:

public void Chick() { } // NOT A CONSTRUCTOR

When you see a method name beginning with a capital letter and having a return type, pay special attention to it. It is *not* a constructor since there’s a return type. It’s a regular method that won’t be called when you write new Chick().

The purpose of a constructor is to initialize fields, although you can put any code in

there. Another way to initialize fields is to do so directly on the line on which they’re declared. This example shows both approaches:

public class Chicken {

int numEggs = 0;// initialize on line String name;

public Chicken() {

name = "Duke";// initialize in constructor

} }

For most classes, you don’t have to code a constructor—the compiler will supply a “do nothing” default constructor for you. There’s one scenario that requires you to declare a constructor that you’ll learn about in Chapter 5.

##### Reading and Writing Object Fields

It’s possible to read and write instance variables directly from the caller. In this example, a mother swan lays eggs:

public class Swan {

int numberEggs;// instance variable public static void main(String[] args) {

Swan mother = new Swan(); mother.numberEggs = 1; // set variable

System.*out*.println(mother.numberEggs); // read variable

}

}

Reading a variable is known as *getting* it. The class gets *numberEggs* directly to print it

out. Writing to a variable is known as *setting* it. This class sets *numberEggs* to 1.

In Chapter 4, you’ll learn how to protect the Swan class from having someone set a nega- tive number of eggs.

You can even read and write fields directly on the line declaring them:

1: public class Name {

2: String first = "Theodore"; 3: String last = "Moose";

4: String full = first + last; 5: {

Lines 2 and 3 both write to fields. Line 4 does both. It reads the fields *first* and *last*. It then writes the field *full*.

##### Instance Initializer Blocks

When you learned about methods, you saw braces ({}). The code between the braces is called a *code block*. Sometimes this code is called being inside the braces. Anywhere you see braces is a code block.

Sometimes code blocks are inside a method. These are run when the method is called.

Other times, code blocks appear outside a method. These are called *instance initializers*. In Chapter 5, you’ll learn how to use a static initializer.

How many blocks do you see in this example? How many instance initializers do you see?

3: public static void main(String[] args) { 4: { System.*out*.println("Feathers"); } 5: }

6: { System.*out*.println("Snowy"); }

There are three code blocks and one instance initializer. Counting code blocks is easy: you just count the number of pairs of braces. If there aren’t the same number of open ({) and close (}) braces, the code doesn’t compile. It doesn’t matter that one set of braces is inside the main() method—it still counts.

When counting instance initializers, keep in mind that it does matter whether the braces are inside a method. There’s only one pair of braces outside a method. Line 6 is an instance initializer.

##### Order of Initialization

When writing code that initializes fields in multiple places, you have to keep track of the order of initialization. We’ll add some more rules to the order of initialization in Chapters 4 and 5. In the meantime, you need to remember:

* + Fields and instance initializer blocks are run in the order in which they appear in the file.
  + The constructor runs after all fields and instance initializer blocks have run. Let’s look at an example:

1: public class Chick {

2: private String name = "Fluffy";

3: { System.*out*.println("setting field"); } 4: public Chick() {

5: name = "Tiny";

6: System.out.println("setting constructor");

7: }

8: public static void main(String[] args) { 9: Chick chick = new Chick();

10: System.out.println(chick.name); } }

Running this example prints this:

setting field setting constructor Tiny

Let’s look at what’s happening here. We start with the main() method because that’s where Java starts execution. On line 9, we call the constructor of Chick. Java creates a new object. First it initializes name to "Fluffy" on line 2. Next it executes the print statement in the instance initializer on line 3. Once all the fields and instance initializers have run, Java returns to the constructor. Line 5 changes the value of name to "Tiny" and line 6 prints another statement. At this point, the constructor is done executing and goes back to the print statement on line 10.

Order matters for the fields and blocks of code. You can’t refer to a variable before it has been initialized:

{ System.out.println(name); } // DOES NOT COMPILE

private String name = "Fluffy";

You should expect to see a question about initialization on the exam. Let’s try one more.

What do you think this code prints out?

public class Egg { public Egg() { number = 5;

}

public static void main(String[] args) { Egg egg = new Egg(); System.*out*.println(egg.number);

}

private int number = 3;

{ number = 4; } }

If you answered 5, you got it right. Fields and blocks are run first in order, setting

*number* to 3 and then 4. Then the constructor runs, setting *number* to 5.

#### Distinguishing Between Object References and Primitives

Java applications contain two types of data: primitive types and reference types. In this section, we’ll discuss the differences between a primitive type and a reference type.

##### Primitive Types

Java has eight built-in data types, referred to as the Java *primitive types*. These eight data types represent the building blocks for Java objects, because all Java objects are just a com- plex collection of these primitive data types. The exam assumes you are well versed in the eight primitive data types, their relative sizes, and what can be stored in them.

Table 1.1 shows the Java primitive types together with their size in bytes and the range of values that each holds.

|  |  |  |
| --- | --- | --- |
| **TA B L E 1 .1** | Java primitive types |  |
| **Keyword** | **Type** | **Example** |
| boolean | true or false | true |
| byte | 8-bit integral value | 123 |
| short | 16-bit integral value | 123 |
| int | 32-bit integral value | 123 |
| long | 64-bit integral value | 123 |
| float | 32-bit floating-point value | 123.45f |
| double | 64-bit floating-point value | 123.456 |
| char | 16-bit Unicode value | 'a' |

There’s a lot of information in Table 1.1. Let’s look at some key points:

* + float and double are used for floating-point (decimal) values.
  + A float requires the letter f following the number so Java knows it is a float.
  + byte, short, int, and long are used for numbers without decimal points.
  + Each numeric type uses twice as many bits as the smaller similar type. For example,

short uses twice as many bits as byte does.

You won’t be asked about the exact sizes of most of these types. You should know that a byte can hold a value from –128 to 127. So you aren’t stuck memorizing this, let’s look at how Java gets that. A byte is 8 bits. A bit has two possible values. (These are basic com- puter science definitions that you should memorize.) 28 is 2 × 2 = 4 × 2 = 8 × 2 = 16 × 2 =

32 × 2 = 64 × 2 = 128 × 2 = 256. Since 0 needs to be included in the range, Java takes it away from the positive side. Or if you don’t like math, you can just memorize it.

The number of bits is used by Java when it figures out how much memory to reserve for your variable. For example, Java allocates 32 bits if you write this:

int num;



You do not have to know this for the exam, but the maximum number an int can hold is

2,147,483,647. How do we know this? One way is to have Java tell us:

System.out.println(Integer.MAX\_VALUE);

The other way is with math. An int is 32 bits. 232 is 4,294,967,296. Divide that by 2 and you get 2,147,483,648. Then subtract 1 as we did with bytes and you get 2,147,483,647. It’s easier to just ask Java to print the value, isn’t it?

**What Is the Largest int?**

There are a few more things you should know about numeric primitives. When a number is present in the code, it is called a *literal*. By default, Java assumes you are defining an int value with a literal. In this example, the number listed is bigger than what fits in an int.

Remember, you aren’t expected to memorize the maximum value for an int. The exam will include it in the question if it comes up.

long max = 3123456789; // DOES NOT COMPILE

Java complains the number is out of range. And it is—for an int. However, we don’t have an int. The solution is to add the character L to the number:

long max = 3123456789L; // now Java knows it is a long

Alternatively, you could add a lowercase l to the number. But please use the uppercase L. The lowercase l looks like the number 1.

Another way to specify numbers is to change the “base.” When you learned how to count, you studied the digits 0–9. This numbering system is called base 10 since there are 10 numbers. It is also known as the decimal number system. Java allows you to specify dig- its in several other formats:

* octal (digits 0–7), which uses the number 0 as a prefix—for example, 017
* hexadecimal (digits 0–9 and letters A–F), which uses the number 0 followed by x or X as a prefix—for example, 0xFF
* binary (digits 0–1), which uses the number 0 followed by b or B as a prefix—for exam- ple, 0b10

You won’t need to convert between number systems on the exam. You’ll have to recog- nize valid literal values that can be assigned to numbers.

Converting Back to Binary

Although you don’t need to convert between number systems on the exam, we’ll look at one example in case you’re curious:

System.out.println(56); // 56

System.out.println(0b11); // 3

System.out.println(017); // 15

System.out.println(0x1F); // 31

First we have our normal base 10 value. We know you already know how to read that, but bear with us. The rightmost digit is 6, so it’s “worth” 6. The second-to-rightmost digit is 5, so it’s “worth” 50 (5 times 10.) Adding these together, we get 56.

Next we have binary, or base 2. The rightmost digit is 1 and is “worth” 1. The second-to- rightmost digit is also 1. In this case, it’s “worth” 2 (1 times 2) because the base is 2. Add- ing these gets us 3.

Then comes octal, or base 8. The rightmost digit is 7 and is “worth” 7. The second-to- rightmost digit is 1. In this case, it’s “worth” 8 (1 times 8) because the base is 8. Adding these gets us 15.

Finally, we have hexadecimal, or base 16, which is also known as hex. The rightmost “digit” is F and it’s “worth” 15 (9 is “worth” 9, A is “worth” 10, B is “worth” 11, and so forth). The second-to-rightmost digit is 1. In this case, it’s “worth” 16 (1 times 16) because the base is 16. Adding these gets us 31.

The last thing you need to know about numeric literals is a feature added in Java 7. You can have underscores in numbers to make them easier to read:

int million1 = 1000000; int million2 = 1\_000\_000;

We’d rather be reading the latter one because the zeroes don’t run together. You can add underscores anywhere except at the beginning of a literal, the end of a literal, right before a decimal point, or right after a decimal point. Let’s look at a few examples:

double notAtStart = \_1000.00; // DOES NOT COMPILE double notAtEnd = 1000.00\_; // DOES NOT COMPILE double notByDecimal = 1000\_.00; // DOES NOT COMPILE double annoyingButLegal = 1\_00\_0.0\_0; // this one compiles

##### Reference Types

A *reference type* refers to an object (an instance of a class). Unlike primitive types that hold their values in the memory where the variable is allocated, references do not hold the value of the object they refer to. Instead, a reference “points” to an object by storing the memory address where the object is located, a concept referred to as a *pointer*. Unlike other languages, Java does not allow you to learn what the physical memory address is. You can only use the reference to refer to the object.

Let’s take a look at some examples that declare and initialize reference types. Suppose we declare a reference of type java.util.Date and a reference of type String:

java.util.Date today;

String greeting;

The *today* variable is a reference of type Date and can only point to a Date object. The *greeting* variable is a reference that can only point to a String object. A value is assigned to a reference in one of two ways:

* A reference can be assigned to another object of the same type.
* A reference can be assigned to a new object using the new keyword.

For example, the following statements assign these references to new objects:

today = new java.util.Date(); greeting = "How are you?";

The *today* reference now points to a new Date object in memory, and *today* can be used to access the various fields and methods of this Date object. Similarly, the *greeting* refer- ence points to a new String object, "How are you?". The String and Date objects do not have names and can be accessed only via their corresponding reference. Figure 1.1 shows how the reference types appear in memory.

**F I G U R E 1 .1** An object in memory can be accessed only via a reference.

A Date reference

today

A Date object

|  |
| --- |
| day  29 |
| month  7 |
| year  2011 |

A String object

A String reference greeting

How are you?

##### Key Differences

There are a few important differences you should know between primitives and reference types. First, reference types can be assigned null, which means they do not currently refer to an object. Primitive types will give you a compiler error if you attempt to assign them null. In this example, *value* cannot point to null because it is of type int:

int value = null; // DOES NOT COMPILE String s = null;

Next, reference types can be used to call methods when they do not point to null.

Primitives do not have methods declared on them. In this example, we can call a method on *reference* since it is of a reference type. You can tell length is a method because it has () after it. The following line is gibberish. No methods exist on *len* because it is an int primi- tive. Primitives do not have methods.

String reference = "hello"; int len = reference.length();

int bad = len.length(); // DOES NOT COMPILE

Finally, notice that all the primitive types have lowercase type names. All classes that come with Java begin with uppercase. You should follow this convention for classes you create as well.

#### Declaring and Initializing Variables

We’ve seen some variables already. A *variable* is a name for a piece of memory that stores data. When you declare a variable, you need to state the variable type along with giving it a name. For example, the following code declares two variables. One is named *zooName* and is of type String. The other is named *numberAnimals* and is of type int.

String zooName; int numberAnimals;

Now that we’ve declared a variable, we can give it a value. This is called initializing a variable. To initialize a variable, you just type the variable name followed by an equal sign, followed by the desired value:

zooName = "The Best Zoo"; numberAnimals = 100;

Since you often want to initialize a variable right away, you can do so in the same state- ment as the declaration. For example, here we merge the previous declarations and initial- izations into more concise code:

String zooName = "The Best Zoo"; int numberAnimals = 100;

In the following sections, we’ll look at how to declare multiple variables in one-line and legal identifiers.

##### Declaring Multiple Variables

You can also declare and initialize multiple variables in the same statement. How many variables do you think are declared and initialized in the following two lines?

String s1, s2;

String s3 = "yes", s4 = "no";

Four String variables were declared: *s1*, *s2*, *s3*, and *s4*. You can declare many variables in the same declaration as long as they are all of the same type. You can also initialize any or all of those values inline. In the previous example, we have two initialized variables: *s3* and *s4*. The other two variables remain declared but not yet initialized.

This is where it gets tricky. Pay attention to tricky things! The exam will attempt to trick you. Again, how many variables do you think are declared and initialized in this code?

int i1, i2, i3 = 0;

As you should expect, three variables were declared: *i1*, *i2*, and *i3*. However, only one of those values was initialized: *i3*. The other two remain declared but not yet initialized. That’s the trick. Each snippet separated by a comma is a little declaration of its own. The initialization of *i3* only applies to *i3*. It doesn’t have anything to do with *i1* or *i2* despite being in the same statement.

Another way the exam could try to trick you is to show you code like this line:

int num, String value; // DOES NOT COMPILE

This code doesn’t compile because it tries to declare multiple variables of *different* types in the same statement. The shortcut to declare multiple variables in the same statement only works when they share a type.

To make sure you understand this, see if you can figure out which of the following are legal declarations. “Legal,” “valid,” and “compiles” are all synonyms in the Java exam world. We try to use all the terminology you could encounter on the exam.

boolean b1, b2; String s1 = "1", s2; double d1, double d2; int i1; int i2;

int i3; i4;

The first statement is legal. It declares two variables without initializing them. The second statement is also legal. It declares two variables and initializes only one of them.

The third statement is *not* legal. Java does not allow you to declare two different types in the same statement. Wait a minute! Variables *d1* and *d2* are the same type. They are both

of type double. Although that’s true, it still isn’t allowed. If you want to declare multiple variables in the same statement, they must share the same type declaration and not repeat it. double d1, d2; would have been legal.

The fourth statement is legal. Although int does appear twice, each one is in a separate statement. A semicolon (;) separates statements in Java. It just so happens there are two completely different statements on the same line. The fifth statement is *not* legal. Again, we have two completely different statements on the same line. The second one is not a

valid declaration because it omits the type. When you see an oddly placed semicolon on the exam, pretend the code is on separate lines and think about whether the code compiles that way. In this case, we have the following:

int i1; int i2; int i3;

i4;// DOES NOT COMPILE

Looking at the last line on its own, you can easily see that the declaration is invalid. And yes, the exam really does cram multiple statements onto the same line—partly to try to trick you and partly to fit more code on the screen. In the real world, please limit

yourself to one declaration per statement and line. Your teammates will thank you for the readable code.

##### Identifiers

It probably comes as no surprise that Java has precise rules about *identifier* names. Luckily, the same rules for identifiers apply to anything you are free to name, including variables, methods, classes, and fields.

There are only three rules to remember for legal identifiers:

* + The name must begin with a letter or the symbol $ or \_.
  + Subsequent characters may also be numbers.
  + You cannot use the same name as a Java *reserved word*. As you might imagine, a reserved word is a keyword that Java has reserved so that you are not allowed to use it. Remember that Java is case sensitive, so you can use versions of the keywords that only differ in case. Please don’t, though.

Don’t worry—you won’t need to memorize the full list of reserved words. The exam will only ask you about ones you’ve already learned, such as class. The following is a list of all the reserved words in Java. const and goto aren’t actually used in Java. They are reserved so that people coming from other languages don’t use them by accident—and in theory, in case Java wants to use them one day.

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

Prepare to be tested on these rules. The following examples are legal:

okidentifier

$OK2Identifier

\_alsoOK1d3ntifi3r

SStillOkbutKnotsonice$

These examples are not legal:

3DPointClass // identifiers cannot begin with a number hollywood@vine // @ is not a letter, digit, $ or \_

\*$coffee // \* is not a letter, digit, $ or \_ public // public is a reserved word

Although you can do crazy things with identifier names, you shouldn’t. Java has con- ventions so that code is readable and consistent. This consistency includes CamelCase. In CamelCase, each word begins with an uppercase letter. This makes multiple-word variable names easier to read. Which would you rather read: *Thisismyclass name* or *ThisIsMyClass name*? The exam will mostly use common conventions for identifiers, but not always. When you see a nonstandard identifier, be sure to check if it is legal. If not, you get to mark the answer “does not compile” and skip analyzing everything else in the question.



Most Java developers follow these conventions for identiﬁer names:

* Method and variables names begin with a lowercase letter followed by CamelCase.
* Class names begin with an uppercase letter followed by CamelCase. Don’t start any identiﬁers with $. The compiler uses this symbol for some ﬁles.

Also, know that valid letters in Java are not just characters in the English alphabet. Java supports the Unicode character set, so there are more than 45,000 characters that can start a legal Java identiﬁer. A few hundred more are non-Arabic numerals that may appear after the ﬁrst character in a legal identiﬁer. Luckily, you don’t have to worry about memorizing those for the exam. If you are in a country that doesn’t use the English alpha- bet, this is useful to know for a job.

**Identiﬁers in the Real World**

#### Understanding Default Initialization of Variables

Before you can use a variable, it needs a value. Some types of variables get this value set automatically, and others require the programmer to specify it. In the following sections, we’ll look at the differences between the defaults for local, instance, and class variables.

##### Local Variables

A *local variable* is a variable defined within a method. Local variables must be initialized before use. They do not have a default value and contain garbage data until initialized. The compiler will not let you read an uninitialized value. For example, the following code generates a compiler error:

4: public int notValid() { 5: int y = 10;

6: int x;

7: int reply = x + y; // DOES NOT COMPILE 8: return reply;

9: }

*y* is initialized to 10. However, because *x* is not initialized before it is used in the expres- sion on line 7, the compiler generates the following error:

Test.java:5: variable x might not have been initialized int reply = x + y;

^

Until *x* is assigned a value, it cannot appear within an expression, and the compiler will gladly remind you of this rule. The compiler knows your code has control of what happens inside the method and can be expected to initialize values.

The compiler is smart enough to recognize variables that have been initialized after their declaration but before they are used. Here’s an example:

public int valid() { int y = 10;

int x; // x is declared here

x = 3; // and initialized here int reply = x + y;

return reply;

}

The compiler is also smart enough to recognize initializations that are more complex.

In this example, there are two branches of code. *answer* is initialized in both of them so the compiler is perfectly happy. *onlyOneBranch* is only initialized if *check* happens to be

true. The compiler knows there is the possibility for *check* to be false, resulting in uninitial- ized code, and gives a compiler error. You’ll learn more about the if statement in the next chapter.

public void findAnswer(boolean check) { int answer;

int onlyOneBranch; if (check) {

onlyOneBranch = 1;

answer = 1;

} else { answer = 2;

}

System.*out*.println(answer); System.*out*.println(onlyOneBranch); // DOES NOT COMPILE

}

##### Instance and Class Variables

Variables that are not local variables are known as *instance variables* or *class variables.*

Instance variables are also called fields. Class variables are shared across multiple objects.

You can tell a variable is a class variable because it has the keyword static before it. You’ll learn about this in Chapter 4. For now, just know that a variable is a class variable if it has the static keyword in its declaration.

Instance and class variables do not require you to initialize them. As soon as you declare these variables, they are given a default value. You’ll need to memorize everything in table

1.2 except the default value of char. To make this easier, remember that the compiler doesn’t know what value to use and so wants the simplest type it can give the value: null for an object and 0/false for a primitive.

**TA B L E 1 . 2** Default initialization values by type

**Variable type Default initialization value**

boolean false

byte, short, int, long 0 (in the type’s bit-length)

float, double 0.0 (in the type’s bit-length)

char '\u0000' (NUL)

All object references (everything else) null

#### Understanding Variable Scope

You’ve learned that local variables are declared within a method. How many local variables do you see in this example?

public void eat(int piecesOfCheese) { int bitesOfCheese = 1;

}

There are two local variables in this method. *bitesOfCheese* is declared inside the

method. *piecesOfCheese* is called a method parameter. It is also local to the method. Both of these variables are said to have a *scope* local to the method. This means they cannot be used outside the method.

Local variables can never have a scope larger than the method they are defined in.

However, they can have a smaller scope. Consider this example:

3: public void eatIfHungry(boolean hungry) { 4: if (hungry) {

5: int bitesOfCheese = 1;

6: } // bitesOfCheese goes out of scope here

7: System.*out*.println(bitesOfCheese);// DOES NOT COMPILE 8: }

*hungry* has a scope of the entire method. *bitesOfCheese* has a smaller scope. It is only available for use in the if statement because it is declared inside of it. When you see a set of braces ({ }) in the code, it means you have entered a new block of code. Each block of code has its own scope. When there are multiple blocks, you match them from the inside out.

In our case, the if statement block begins at line 4 and ends at line 6. The method’s block begins at line 3 and ends at line 8.

Since *bitesOfCheese* is declared in such a block, the scope is limited to that block. When the compiler gets to line 7, it complains that it doesn’t know anything about this *bitesOf- Cheese* thing and gives an error:

bitesOfCheese cannot be resolved to a variable

Remember that blocks can contain other blocks. These smaller contained blocks can ref- erence variables defined in the larger scoped blocks, but not vice versa. For example:

16: public void eatIfHungry(boolean hungry) { 17: if (hungry) {

18: int bitesOfCheese = 1;

19: {

20: boolean teenyBit = true;

21: System.*out*.println(bitesOfCheese);

22: }

23: }

24: System.out.println(teenyBit); // DOES NOT COMPILE 25: }

The variable defined on line 18 is in scope until the block ends on line 23. Using it in the smaller block from lines 19 to 22 is fine. The variable defined on line 20 goes out of scope on line 22. Using it on line 24 is not allowed.

The exam may attempt to trick you with questions on scope. You’ll probably see a ques- tion that appears to be about something complex and fails to compile because one of the variables is out of scope. Let’s try one. Don’t worry if you aren’t familiar with if state- ments or while loops yet. It doesn’t matter what the code does since we are talking about scope. See if you can figure out on which line each of the five local variables goes into and out of scope:

11: public void eatMore(boolean hungry, int amountOfFood) { 12: int roomInBelly = 5;

13: if (hungry) {

14: boolean timeToEat = true; 15: while (amountOfFood > 0) {

16: int amountEaten = 2;

17: roomInBelly = roomInBelly - amountEaten; 18: amountOfFood = amountOfFood - amountEaten; 19: }

20: }

21: System.*out*.println(amountOfFood);

22: }

The first step in figuring out the scope is to identify the blocks of code. In this case, there are three blocks. You can tell this because there are three sets of braces. Starting from the innermost set, we can see where the while loop’s block starts and ends. Repeat this as we go out for the if statement block and method block. Table 1.3 shows the line numbers that each block starts and ends on.

|  |  |  |  |
| --- | --- | --- | --- |
| **TA B L E** | **1 . 3** | Blocks for scope |  |
| **Line** |  | **First line in block** | **Last line in block** |
| while |  | 15 | 19 |
| if |  | 13 | 20 |
| Method |  | 11 | 22 |

You’ll want to practice this skill a lot. Identifying blocks needs to be second nature for the exam. The good news is that there are lots of code examples to practice on. You can look at any code example in this book on any topic and match up braces.

Now that we know where the blocks are, we can look at the scope of each variable. *hungry* and *amountOfFood* are method parameters, so they are available for the entire method. This means their scope is lines 11 to 22. *roomInBelly* goes into scope on line 12 because that is where it is declared. It stays in scope for the rest of the method and so goes out of scope on line 22. *timeToEat* goes into scope on line 14 where it is declared. It goes out of scope on line 20 where the if block ends. *amountEaten* goes into scope on line 16 where it is declared. It goes out of scope on line 19 where the while block ends.

All that was for local variables. Luckily the rule for instance variables is easier: they are available as soon as they are defined and last for the entire lifetime of the object itself. The rule for class (static) variables is even easier: they go into scope when declared like the other variables types. However, they stay in scope for the entire life of the program.

Let’s do one more example to make sure you have a handle on this. Again, try to figure out the type of the four variables and when they go into and out of scope.

1: public class Mouse {

2: static int *MAX\_LENGTH* = 5;

3: int length;

4: public void grow(int inches) { 5: if (length < *MAX\_LENGTH*) {

6: int newSize = length + inches; 7: length = newSize;

8: }

9: }

10: }

In this class, we have one class variable (*MAX\_LENGTH*), one instance variable (*length*), and two local variables (*inches* and *newSize*.) *MAX\_LENGTH* is a class variable because it has the static keyword in its declaration. *MAX\_LENGTH* goes into scope on line 2 where it is declared. It stays in scope until the program ends. *length* goes into scope on line 3 where it is declared. It stays in scope as long as this Mouse object exists. *inches* goes into scope where it is declared on line 4. It goes out of scope at the end of the method on line 9. *newSize* goes into scope where it is declared on line 6. Since it is defined inside the if state- ment block, it goes out of scope when that block ends on line 8.

Got all that? Let’s review the rules on scope:

* Local variables—in scope from declaration to end of block
* Instance variables—in scope from declaration until object garbage collected
* Class variables—in scope from declaration until program ends

#### Ordering Elements in a Class

Now that you’ve seen the most common parts of a class, let’s take a look at the correct order to type them into a file. Comments can go anywhere in the code. Beyond that, you need to memorize the rules in Table 1.4.

**TA B L E 1 . 4** Elements of a class

|  |  |  |  |
| --- | --- | --- | --- |
| **Element** | **Example** | **Required?** | **Where does it go?** |
| Package declaration | package abc; | No | First line in the file |
| Import statements | import java.util.\*; | No | Immediately after the package |
| Class declaration | public class C | Yes | Immediately after the import |
| Field declarations | int value; | No | Anywhere inside a class |
| Method declarations | void method() | No | Anywhere inside a class |

Let’s look at a few examples to help you remember this. The first example contains one of each element:

package structure; // package must be first non-comment import java.util.\*; // import must come after package public class Meerkat { // then comes the class

double weight; // fields and methods can go in either order public double getWeight() {

return weight; }

double height; // another field – they don't need to be together

}

So far so good. This is a common pattern that you should be familiar with. How about this one?

/\* header \*/ package structure;

// class Meerkat

public class Meerkat { }

Still good. We can put comments anywhere, and imports are optional. In the next example, we have a problem:

import java.util.\*;

package structure; // DOES NOT COMPILE String name; // DOES NOT COMPILE

public class Meerkat { }

There are two problems here. One is that the package and import statements are reversed. Though both are optional, package must come before import if present. The other issue is that a field attempts declaration outside a class. This is not allowed. Fields and methods must be within a class.

Got all that? Think of the acronym PIC (picture): package, import, and class. Fields and methods are easier to remember because they merely have to be inside of a class.

You need to know one more thing about class structure for the OCA exam: multiple classes can be defined in the same file, but only one of them is allowed to be public. The public class matches the name of the file. For example, these two classes must be in a file named Meerkat.java:

1: public class Meerkat { } 2: class Paw { }

A file is also allowed to have neither class be public. As long as there isn’t more than one public class in a file, it is okay. On the OCP exam, you’ll also need to understand inner classes, which are classes within a class.

#### Destroying Objects

Now that we’ve played with our objects, it is time to put them away. Luckily, Java auto- matically takes care of that for you. Java provides a garbage collector to automatically look for objects that aren’t needed anymore.

All Java objects are stored in your program memory’s *heap*. The heap, which is also referred to as the free store*,* represents a large pool of unused memory allocated to your Java application. The heap may be quite large, depending on your environment, but there is always a limit to its size. If your program keeps instantiating objects and leaving them on the heap, eventually it will run out of memory.

In the following sections, we’ll look at garbage collection and the finalize() method.

##### Garbage Collection

Garbage collection refers to the process of automatically freeing memory on the heap by deleting objects that are no longer reachable in your program. There are many different algorithms for garbage collection, but you don’t need to know any of them for the exam. You *do* need to know that System.gc() is not guaranteed to run, and you should be able to recognize when objects become eligible for garbage collection.

Let’s start with the first one. Java provides a method called System.gc(). Now you might think from the name that this tells Java to run garbage collection. Nope! It meekly *suggests* that now might be a good time for Java to kick off a garbage collection run. Java is free to ignore the request.

The more interesting part of garbage collection is when the memory belonging to an object can be reclaimed. Java waits patiently until the code no longer needs that memory. An object will remain on the heap until it is no longer reachable. An object is no longer reachable when one of two situations occurs:

* The object no longer has any references pointing to it.
* All references to the object have gone out of scope.

Do not confuse a reference with the object that it refers to; they are two different enti- ties. The reference is a variable that has a name and can be used to access the contents of an object. A reference can be assigned to another reference, passed to a method, or returned from a method. All references are the same size, no matter what their type is.

An object sits on the heap and does not have a name. Therefore, you have no way to access an object except through a reference. Objects come in all different shapes and sizes and consume varying amounts of memory. An object cannot be assigned to another

**Objects vs. References**

object, nor can an object be passed to a method or returned from a method. It is the object that gets garbage collected, not its reference.

The Heap

A Reference name

An Object

A reference may or may not be created on the heap. All references are the same size, no matter what their

data type is, and are accessed by their variable name.

Objects are always on the heap.

They have no name and can only be accessed via a reference. Objects vary in size depending on their class definition.

Realizing the difference between a reference and an object goes a long way toward understanding garbage collection, the new operator, and many other facets of the Java language. Look at this code and see if you can figure out when each object first becomes eligible for garbage collection:

1: public class Scope {

2: public static void main(String[] args) { 3: String one, two;

4: one = new String("a"); 5: two = new String("b"); 6: one = two;

7: String three = one; 8: one = null;

9: } }

When you get asked a question about garbage collection on the exam, we recommend you draw what’s going on. There’s a lot to keep track of in your head and it’s easy to make a silly mistake trying to keep it all in your memory. Let’s try it together now. Really. Get a pencil and paper. We’ll wait.

Got that paper? Okay, let’s get started. On line 3, we write *one* and *two*. Just the words.

No need for boxes or arrows yet since no objects have gone on the heap yet. On line 4, we have our first object. Draw a box with the string "a" in it and draw an arrow from the

word *one* to that box. Line 5 is similar. Draw another box with the string "b" in it this time

and an arrow from the word *two*. At this point, your work should look like Figure 1.2.

**F I G U R E 1 . 2** Your drawing after line 5

one

"a"

two

"b"

On line 6, the variable *one* changes to point to "b". Either erase or cross out the arrow from *one* and draw a new arrow from *one* to "b". On line 7, we have a new variable, so write the word *three* and draw an arrow from *three* to "b". Notice that *three* points to what *one* is pointing to right now and not what it was pointing to at the beginning. This is why we are drawing pictures. It’s easy to forget something like that. At this point, your work should look like Figure 1.3.

**F I G U R E 1 . 3** Your drawing after line 7

one

"a"

two

"b"

three

Finally, cross out the line between *one* and "b" since line 8 sets this variable to null.

Now, we were trying to find out when the objects were first eligible for garbage collection. On line 6, we got rid of the only arrow pointing to "a", making that object eligible for garbage collection. "b" has arrows pointing to it until it goes out of scope. This means "b" doesn’t go out of scope until the end of the method on line 9.

***finalize()***

Java allows objects to implement a method called finalize() that might get called. This method gets called if the garbage collector tries to collect the object. If the garbage collector doesn’t run, the method doesn’t get called. If the garbage collector fails to collect the object and tries to run it again later, the method doesn’t get called a second time.

In practice, this means you are highly unlikely to use it in real projects. Luckily, there isn’t much to remember about finalize() for the exam. Just keep in mind that it might not get called and that it definitely won’t be called twice.

With that said, this call produces no output when we run it:

public class Finalizer { protected void finalize() {

System.out.println("Calling finalize");

}

public static void main(String[] args) { Finalizer f = new Finalizer();

} }

The reason is that the program exits before there is any need to run the garbage collec- tor. While *f* is eligible for garbage collection, Java has better things to do than take out the trash constantly. For the exam, you need to know that this finalize() call could run zero or one time. Now for a more interesting example:

public class Finalizer {

private static List *objects* = new ArrayList(); protected void finalize() {

objects.add(this); // Don't do this

} }

Remember, finalize() is only run when the object is eligible for garbage collection. The problem here is that by the end of the method, the object is no longer eligible for garbage collection because a static variable is referring to it and static variables stay in scope until the program ends. Java is smart enough to realize this and aborts the attempt to throw out the object. Now suppose later in the program *objects* is set to null. Oh, good, we can finally remove the object from memory. Java remembers already running finalize() on this object and will not do so again. The lesson is that the finalize() call could run zero or one time. This is the exact same lesson as the simple example—that’s why it’s so easy to remember.

#### Benefits of Java

Java has some key benefits that you’ll need to know for the exam:

**Object Oriented** Java is an object-oriented language, which means all code is defined in classes and most of those classes can be instantiated into objects. We’ll discuss this more throughout the book. Many languages before Java were procedural, which meant there were routines or methods but no classes. Another common approach is functional programming. Java allows for functional programming within a class, but object oriented is still the main organization of code.

**Encapsulation** Java supports access modifiers to protect data from unintended access and modification. Most people consider encapsulation to be an aspect of object-oriented languages. Since the exam objectives call attention to it specifically, so do we.

**Platform Independent** Java is an interpreted language because it gets compiled to bytecode. A key benefit is that Java code gets compiled once rather than needing to be

recompiled for different operating systems. This is known as “write once, run everywhere.” On the OCP exam, you’ll learn that it is possible to write code that does not run every- where. For example, you might refer to a file in a specific directory. If you get asked on the OCA exam, the answer is that the same class files run everywhere.

**Robust** One of the major advantages of Java over C++ is that it prevents memory leaks. Java manages memory on its own and does garbage collection automatically. Bad memory management in C++ is a big source of errors in programs.

**Simple** Java was intended to be simpler than C++. In addition to eliminating pointers, it got rid of operator overloading. In C++, you could write a + b and have it mean almost anything.

**Secure** Java code runs inside the JVM. This creates a sandbox that makes it hard for Java code to do evil things to the computer it is running on.

#### Summary

In this chapter, you saw that Java classes consist of members called fields and methods. An object is an instance of a Java class. There are three styles of comment: a single-line com- ment (//), a multiline comment (/\* \*/), and a Javadoc comment (/\*\* \*/).

Java begins program execution with a main() method. The most common signature for

this method run from the command line is public static void main(String[] args). Arguments are passed in after the class name, as in java NameOfClass firstArgument. Arguments are indexed starting with 0.

Java code is organized into folders called packages. To reference classes in other pack- ages, you use an import statement. A wildcard ending an import statement means you want to import all classes in that package. It does not include packages that are inside that one. java.lang is a special package that does not need to be imported.

Constructors create Java objects. A constructor is a method matching the class name and

omitting the return type. When an object is instantiated, fields and blocks of code are initialized first. Then the constructor is run.

Primitive types are the basic building blocks of Java types. They are assembled into reference types. Reference types can have methods and be assigned to null. In addition to “normal” numbers, numeric literals are allowed to begin with 0 (octal), 0x (hex), 0X (hex), 0b (binary), or 0B (binary). Numeric literals are also allowed to contain underscores as long as they are directly between two other numbers.

Declaring a variable involves stating the data type and giving the variable a name.

Variables that represent fields in a class are automatically initialized to their correspond- ing “zero” or null value during object instantiation. Local variables must be specifically initialized. Identifiers may contain letters, numbers, $, or \_. Identifiers may not begin with numbers.

Scope refers to that portion of code where a variable can be accessed. There are three kinds of variables in Java, depending on their scope: instance variables, class variables, and

local variables. Instance variables are the nonstatic fields of your class. Class variables are the static fields within a class. Local variables are declared within a method.

For some class elements, order matters within the file. The package statement comes first if present. Then comes the import statements if present. Then comes the class declaration. Fields and methods are allowed to be in any order within the class.

Garbage collection is responsible for removing objects from memory when they can never be used again. An object becomes eligible for garbage collection when there are no more references to it or its references have all gone out of scope. The finalize() method will run once for each object if/when it is first garbage collected.

Java code is object oriented, meaning all code is defined in classes. Access modifiers allow classes to encapsulate data. Java is platform independent, compiling to bytecode. It is robust and simple by not providing pointers or operator overloading. Finally, Java is secure because it runs inside a virtual machine.

#### Exam Essentials

**Be able to write code using a main() method.** A main() method is usually written as public static void main(String[] args). Arguments are referenced starting with args[0]. Accessing an argument that wasn’t passed in will cause the code to throw an exception.

**Understand the effect of using packages and imports.** Packages contain Java classes. Classes can be imported by class name or wildcard. Wildcards do not look at subdirecto- ries. In the event of a conflict, class name imports take precedence.

**Be able to recognize a constructor.** A constructor has the same name as the class. It looks like a method without a return type.

**Be able to identify legal and illegal declarations and initialization.** Multiple variables can be declared and initialized in the same statement when they share a type. Local variables require an explicit initialization; others use the default value for that type. Identifiers may contain letters, numbers, $, or \_. Identifiers may not begin with numbers. Numeric literals may contain underscores between two digits and begin with 1–9, 0, 0x, 0X, 0b, and 0B.

**Be able to determine where variables go into and out of scope.** All variables go into scope when they are declared. Local variables go out of scope when the block they are declared in ends. Instance variables go out of scope when the object is garbage collected. Class vari- ables remain in scope as long as the program is running.

**Be able to recognize misplaced statements in a class.** Package and import statements are optional. If present, both go before the class declaration in that order. Fields and methods are also optional and are allowed in any order within the class declaration.

**Know how to identify when an object is eligible for garbage collection.** Draw a diagram to keep track of references and objects as you trace the code. When no arrows point to a box (object), it is eligible for garbage collection.

#### Review Questions

1. Which of the following are valid Java identifiers? (Choose all that apply)
   1. A$B
   2. \_helloWorld
   3. true
   4. java.lang
   5. Public

**F.** 1980\_s

1. What is the output of the following program?

1: public class WaterBottle { 2: private String brand;

3: private boolean empty;

4: public static void main(String[] args) { 5: WaterBottle wb = new WaterBottle();

6: System.*out*.print("Empty = " + wb.empty); 7: System.*out*.print(", Brand = " + wb.brand); 8: } }

* 1. Line 6 generates a compiler error.
  2. Line 7 generates a compiler error.
  3. There is no output.
  4. Empty = false, Brand = null
  5. Empty = false, Brand =
  6. Empty = null, Brand = null

1. Which of the following are true? (Choose all that apply)

4: short numPets = 5;

5: int numGrains = 5.6;

6: String name = "Scruffy"; 7: numPets.length();

8: numGrains.length();

9: name.length();

* 1. Line 4 generates a compiler error.
  2. Line 5 generates a compiler error.
  3. Line 6 generates a compiler error.
  4. Line 7 generates a compiler error.
  5. Line 8 generates a compiler error.
  6. Line 9 generates a compiler error.
  7. The code compiles as is.

1. Given the following class, which of the following is true? (Choose all that apply)

1: public class Snake { 2:

3: public void shed(boolean time) { 4:

5: if (time) { 6:

7: }

8: System.*out*.println(result);

9:

10: }

11: }

* 1. If String result = "done"; is inserted on line 2, the code will compile.
  2. If String result = "done"; is inserted on line 4, the code will compile.
  3. If String result = "done"; is inserted on line 6, the code will compile.
  4. If String result = "done"; is inserted on line 9, the code will compile.
  5. None of the above changes will make the code compile.

1. Given the following classes, which of the following can independently replace INSERT IMPORTS HERE to make the code compile? (Choose all that apply)

package aquarium; public class Tank { }

package aquarium.jellies; public class Jelly { }

package visitor;

INSERT IMPORTS HERE

public class AquariumVisitor {

public void admire(Jelly jelly) { } }

* 1. import aquarium.\*;
  2. import aquarium.\*.Jelly;
  3. import aquarium.jellies.Jelly;
  4. import aquarium.jellies.\*;
  5. import aquarium.jellies.Jelly.\*;
  6. None of these can make the code compile.

1. Given the following classes, what is the maximum number of imports that can be removed and have the code still compile?

package aquarium; public class Water { }

package aquarium; import java.lang.\*;

import java.lang.System; import aquarium.Water; import aquarium.\*; public class Tank {

public void print(Water water) { System.*out*.println(water); } }

* 1. 0
  2. 1
  3. 2
  4. 3
  5. 4
  6. Does not compile.

1. Given the following classes, which of the following snippets can be inserted in place of

INSERT IMPORTS HERE and have the code compile? (Choose all that apply)

package aquarium; public class Water {

boolean salty = false;

}

package aquarium.jellies; public class Water {

boolean salty = true;

}

package employee;

INSERT IMPORTS HERE

public class WaterFiller { Water water;

}

* 1. import aquarium.\*;
  2. import aquarium.Water; import aquarium.jellies.\*;
  3. import aquarium.\*;

import aquarium.jellies.Water;

* 1. import aquarium.\*;

import aquarium.jellies.\*;

* 1. import aquarium.Water;

import aquarium.jellies.Water;

* 1. None of these imports can make the code compile.

1. Given the following class, which of the following calls print out Blue Jay? (Choose all that apply)

public class BirdDisplay {

public static void main(String[] name) { System.*out*.println(name[1]);

} }

* 1. java BirdDisplay Sparrow Blue Jay
  2. java BirdDisplay Sparrow "Blue Jay"
  3. java BirdDisplay Blue Jay Sparrow
  4. java BirdDisplay "Blue Jay" Sparrow
  5. java BirdDisplay.class Sparrow "Blue Jay"
  6. java BirdDisplay.class "Blue Jay" Sparrow
  7. Does not compile.

1. Which of the following legally fill in the blank so you can run the main() method from the command line? (Choose all that apply)

public static void main( )

* 1. String[] \_names
  2. String[] 123
  3. String abc[]
  4. String \_Names[]
  5. String... $n
  6. String names
  7. None of the above.

1. Which of the following are legal entry point methods that can be run from the command line? (Choose all that apply)
   1. private static void main(String[] args)
   2. public static final main(String[] args)
   3. public void main(String[] args)
   4. public static void test(String[] args)
   5. public static void main(String[] args)
   6. public static main(String[] args)
   7. None of the above.
2. Which of the following are true? (Choose all that apply)
   1. An instance variable of type double defaults to null.
   2. An instance variable of type int defaults to null.
   3. An instance variable of type String defaults to null.
   4. An instance variable of type double defaults to 0.0.
   5. An instance variable of type int defaults to 0.0.
   6. An instance variable of type String defaults to 0.0.
   7. None of the above.
3. Which of the following are true? (Choose all that apply)
   1. A local variable of type boolean defaults to null.
   2. A local variable of type float defaults to 0.
   3. A local variable of type Object defaults to null.
   4. A local variable of type boolean defaults to false.
   5. A local variable of type boolean defaults to true.
   6. A local variable of type float defaults to 0.0.
   7. None of the above.
4. Which of the following are true? (Choose all that apply)
   1. An instance variable of type boolean defaults to false.
   2. An instance variable of type boolean defaults to true.
   3. An instance variable of type boolean defaults to null.
   4. An instance variable of type int defaults to 0.
   5. An instance variable of type int defaults to 0.0.
   6. An instance variable of type int defaults to null.
   7. None of the above.
5. Given the following class in the file /my/directory/named/A/Bird.java:

INSERT CODE HERE

public class Bird { }

Which of the following replaces INSERT CODE HERE if we compile from /my/directory**?**

(Choose all that apply)

* 1. package my.directory.named.a;
  2. package my.directory.named.A;
  3. package named.a;
  4. package named.A;
  5. package a;
  6. package A;
  7. Does not compile.

1. Which of the following lines of code compile? (Choose all that apply)

**A.** int i1 = 1\_234;

**B.** double d1 = 1\_234\_.0;

**C.** double d2 = 1\_234.\_0;

**D.** double d3 = 1\_234.0\_;

**E.** double d4 = 1\_234.0;

**F.** None of the above.

1. Given the following class, which of the following lines of code can replace INSERT CODE HERE to make the code compile? (Choose all that apply)

public class Price {

public void admission() {

INSERT CODE HERE

System.*out*.println(amount);

} }

* 1. int amount = 9L;
  2. int amount = 0b101;
  3. int amount = 0xE;
  4. double amount = 0xE;
  5. double amount = 1\_2\_.0\_0;
  6. int amount = 1\_2\_;
  7. None of the above.

1. Which of the following are true? (Choose all that apply)

public class Bunny {

public static void main(String[] args) { Bunny bun = new Bunny();

} }

* 1. Bunny is a class.
  2. bun is a class.
  3. main is a class.
  4. Bunny is a reference to an object.
  5. bun is a reference to an object.
  6. main is a reference to an object.
  7. None of the above.

1. Which represent the order in which the following statements can be assembled into a pro- gram that will compile successfully? (Choose all that apply)

A: class Rabbit {}

B: import java.util.\*;

C: package animals;

* 1. A, B, C
  2. B, C, A
  3. C, B, A
  4. B, A
  5. C, A
  6. A, C
  7. A, B

1. Suppose we have a class named Rabbit. Which of the following statements are true? (Choose all that apply)

1: public class Rabbit {

2: public static void main(String[] args) { 3: Rabbit one = new Rabbit();

4: Rabbit two = new Rabbit(); 5: Rabbit three = one;

6: one = null;

7: Rabbit four = one; 8: three = null;

9: two = null;

10: two = new Rabbit(); 11: System.gc();

12: } }

* 1. The Rabbit object from line 3 is first eligible for garbage collection immediately following line 6.
  2. The Rabbit object from line 3 is first eligible for garbage collection immediately following line 8.
  3. The Rabbit object from line 3 is first eligible for garbage collection immediately following line 12.
  4. The Rabbit object from line 4 is first eligible for garbage collection immediately following line 9.
  5. The Rabbit object from line 4 is first eligible for garbage collection immediately following line 11.
  6. The Rabbit object from line 4 is first eligible for garbage collection immediately following line 12.

1. What is true about the following code? (Choose all that apply)

public class Bear {

protected void finalize() { System.*out*.println("Roar!");

}

public static void main(String[] args) { Bear bear = new Bear();

bear = null;

System.*gc*();

} }

* 1. finalize() is guaranteed to be called.
  2. finalize() might or might not be called
  3. finalize() is guaranteed not to be called.
  4. Garbage collection is guaranteed to run.
  5. Garbage collection might or might not run.
  6. Garbage collection is guaranteed not to run.
  7. The code does not compile.

1. What does the following code output?

1: public class Salmon { 2: int count;

3: public void Salmon() { 4: count = 4;

5: }

6: public static void main(String[] args) { 7: Salmon s = new Salmon();

8: System.*out*.println(s.count);

9: } }

* 1. 0
  2. 4
  3. Compilation fails on line 3.
  4. Compilation fails on line 4.
  5. Compilation fails on line 7.
  6. Compilation fails on line 8.

1. Which of the following are true statements? (Choose all that apply)
   1. Java allows operator overloading.
   2. Java code compiled on Windows can run on Linux.
   3. Java has pointers to specific locations in memory.
   4. Java is a procedural language.
   5. Java is an object-oriented language.
   6. Java is a functional programming language.
2. Which of the following are true? (Choose all that apply)
   1. javac compiles a .class file into a .java file.
   2. javac compiles a .java file into a .bytecode file.
   3. javac compiles a .java file into a .class file.
   4. Java takes the name of the class as a parameter.
   5. Java takes the name of the .bytecode file as a parameter.
   6. Java takes the name of the .class file as a parameter.



**Chapter**

**2**

### Operators and Statements

###### OCA EXAM OBJECTIVES COVERED IN THIS CHAPTER:

* + - Using Operators and Decision Constructs
      * Use Java operators; including parentheses to override opera- tor precedence
      * Create if and if/else and ternary constructs
      * Use a switch statement
    - Using Loop Constructs
      * Create and use while loops
      * Create and use for loops including the enhanced for loop
      * Create and use do/while loops
      * Compare loop constructs
      * Use break and continue

Like many programming languages, Java is composed primar- ily of variables, operators, and statements put together in some logical order. In the previous chapter, we discussed variables

and gave some examples; in this chapter we’ll discuss the various operators and statements available to you within the language. This knowledge will allow you to build complex func- tions and class structures that you’ll see in later chapters.

Understanding Java Operators

A Java *operator* is a special symbol that can be applied to a set of variables, values, or literals—referred to as operands—and that returns a result. Three flavors of operators are available in Java: unary, binary, and ternary. These types of operators can be applied to one, two, or three operands, respectively. For the OCA exam, you’ll need know a specific subset of Java operators, how to apply them, and the order in which they should be applied.

Java operators are not necessarily evaluated from left-to-right order. For example, the following Java expression is actually evaluated from right-to-left given the specific opera- tors involved:

int y = 4;

double x = 3 + 2 \* --y;

In this example, you would first decrement y to 3, and then multiply the resulting value by 2, and finally add 3. The value would then be automatically upcast from 9 to 9.0 and assigned to x. The final values of x and y would be 9.0 and 3, respectively. If you didn’t follow that evaluation, don’t worry. By the end of this chapter, solving problems like this should be second nature.

Unless overridden with parentheses, Java operators follow *order of operation*, listed in Table 2.1, by decreasing order of *operator precedence*. If two operators have the same level of precedence, then Java guarantees left-to-right evaluation. You need to know only those operators in bold for the OCA exam.

**TA B L E 2 .1** Order of operator precedence

**Operator Symbols and examples**

Post-unary operators *expression***++**, *expression***--**

Pre-unary operators **++***expression*, **--***expression*

Working with Binary Arithmetic Operators **53**

**Operator Symbols and examples**

Other unary operators +, **-**, **!** Multiplication/Division/Modulus **\***, **/**, **%** Addition/Subtraction **+**, **-**

Shift operators <<, >>, >>>

Relational operators **<**, **>**, **<=**, **>=**, instanceof

Equal to/not equal to **==**, **!=**

Logical operators **&**, **^**, **|**

Short-circuit logical operators **&&**, **||**

Ternary operators boolean expression **?** expression1 **:** expres- sion2

Assignment operators **=**, **+=**, **-=**, \*=, /=, %=, &=, ^=, !=, <<=, >>=, >>>=

We’ll spend the first half of this chapter discussing many of the operators in this list as well as how operator precedence determines which operators should be applied first. Note that you won’t be tested on some operators, although we recommend that you be aware of their existence.

#### Working with Binary Arithmetic Operators

We’ll begin our discussion with *binary operators*, by far the most common operators in the Java language. They can be used to perform mathematical operations on variables, create logical expressions, as well as perform basic variable assignments. Binary operators are commonly combined in complex expressions with more than two variables; therefore, operator precedence is very important in evaluating expressions.

##### Arithmetic Operators

*Arithmetic operators* are often encountered in early mathematics and include addition (+), subtraction (-), multiplication (\*), division (/), and modulus (%). They also include the

unary operators, ++ and --, although we cover them later in this chapter. As you may have

**54** Chapter 2 ■ Operators and Statements

noticed in Table 2.1, the *multiplicative* operators (\*, /, %) have a higher order of precedence than the *additive* operators (+, -). That means when you see an expression such as this:

int x = 2 \* 5 + 3 \* 4 - 8;

you first evaluate the 2 \* 5 and 3 \* 4, which reduces the expression to the following:

int x = 10 + 12 - 8;

Then, you evaluate the remaining terms in left-to-right order, resulting in a value of x of

*14*. Make sure you understand why the result is *24* as you’ll likely see this kind of operator precedence question on the exam.

Notice that we said “Unless overridden with parentheses…” prior to Table 2.1. That’s because you can change the order of operation explicitly by wrapping parentheses around the sections you want evaluated first. Compare the previous example with the following one containing the same values and operators, in the same order, but with two sets of parentheses:

int x = 2 \* ((5 + 3) \* 4 – 8);

This time you would evaluate the addition operator 10 + 3, which reduces the expres- sion to the following:

int x = 2 \* (8 \* 4 – 8);

You can further reduce this expression by multiplying the first two values within the parentheses:

int x = 2 \* (32 – 8);

Next, you subtract the values within the parentheses before applying terms outside the parentheses:

int x = 2 \* 24;

Finally, you would multiply the result by 2, resulting in a value of 48 for x.

All of the arithmetic operators may be applied to any Java primitives, except boolean and String. Furthermore, only the addition operators + and += may be applied to String values, which results in String concatenation.

Although we are sure you have seen most of the arithmetic operators before, the modu- lus operator, %, may be new to you. The modulus, or remainder operator, is simply the remainder when two numbers are divided. For example, 9 divided by 3 divides evenly and has no remainder; therefore, the remainder, or 9 % 3, is 0. On the other hand, 11 divided by 3 does not divide evenly; therefore, the remainder, or 11 % 3, is 2.

Be sure to understand the difference between arithmetic division and modulus. For inte- ger values, division results in the floor value of the nearest integer that fulfills the operation, whereas modulus is the remainder value. The following examples illustrate this distinction:

System.out.print(9 / 3); // Outputs 3

System.out.print(9 % 3); // Outputs 0