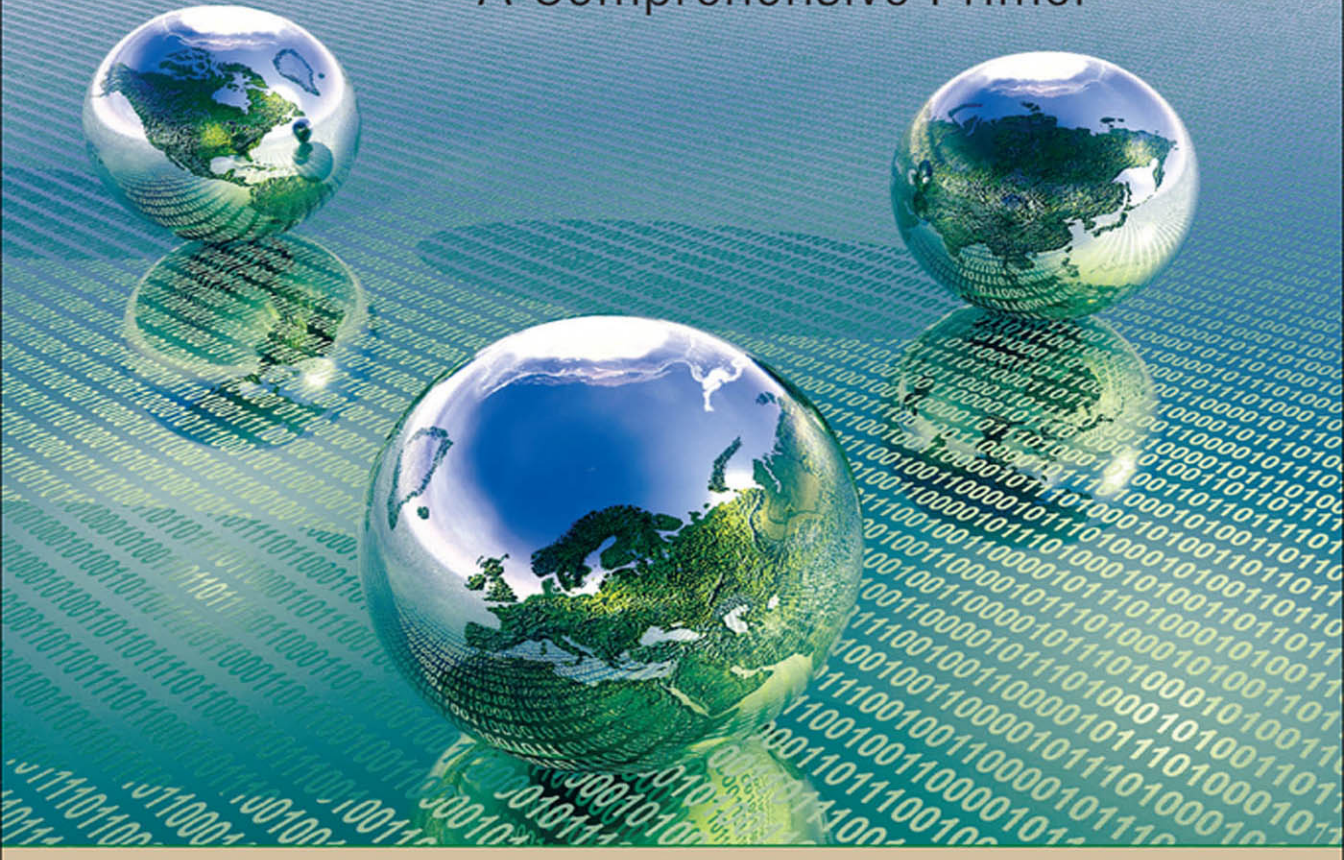
EXAM 1Z0-808

A Programmer's Guide to Java® SE 8 Oracle Certified Associate (OCA)

A Comprehensive Primer



Khalid A. Mughal • Rolf W. Rasmussen

A Programmer's Guide to

Java® SE 8

Oracle Certified Associate (OCA)

This page intentionally left blank

A Programmer's Guide to
Java® SE 8
Oracle Certified Associate (OCA)

A Comprehensive Primer

Khalid A. Mughal
Rolf W. Rasmussen

Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Library of Congress Control Number: 2016937073

ISBN-13: 978-0-13-293021-5
ISBN-10: 0-13-293021-8
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, July 2016

To the loving memory of my mother, Zubaida Begum,
and my father, Mohammed Azim
—K.A.M.

For Olivia E. Rasmussen and
Louise J. Dahlmo
—R.W.R.

This page intentionally left blank

Contents Overview

Contents

Figures

Tables

Examples

This page intentionally left blank

Foreword

Java is now over twenty years old and the current release, JDK 8, despite its name, is really the eleventh significant release of the platform. Whilst staying true to the original ideas of the platform, there have been numerous developments adding a variety of features to the language syntax as well as a huge number of APIs to the core class libraries. This has enabled developers to become substantially more productive and has helped to eliminate a variety of common situations that can easily result in bugs.

Java has continued to grow in popularity, which is in large part attributable to the continued evolution of the platform, which keeps it fresh and addresses things that developers want. According to some sources, there are more than nine million Java programmers across the globe and this number looks set to continue to grow as most universities use Java as a primary teaching language.

With so many Java programmers available to employers, how do they ensure that candidates have the necessary skills to develop high-quality, reliable code? The answer is certification: a standardized test of a developer's knowledge about the wide variety of features and techniques required to use Java efficiently and effectively. Originally introduced by Sun Microsystems, the certification process and exam has been updated to match the features of each release of Java. Oracle has continued this since acquiring Sun in 2010.

Taking and passing the exams is not a simple task. To ensure that developers meet a high standard of knowledge about Java, the candidate must demonstrate the ability to understand a wide variety of programming techniques, a clear grasp of the Java syntax, and a comprehensive knowledge of the standard class library APIs. With the release of JDK 8, not only do Java developers need to understand the details of imperative and object-oriented programming, they now need to have a grasp of functional programming so they can effectively use the key new features: lambda expressions and the Streams API.

Which is why, ultimately, you need this book to help you prepare for the exam. The authors have done a great job of presenting the material you need to know to pass

the exam in an approachable and easy-to-grasp way. The book starts with the fundamental concepts and language syntax and works its way through what you need to know about object-oriented programming before addressing more complex topics like generic types. The latter part of the book addresses the most recent changes in JDK 8, that of lambda expressions, the Streams API, and the new Date and Time API.

Having worked with Java almost since it was first released, both at Sun Microsystems and then at Oracle Corporation, I think you will find this book an invaluable guide to help you pass the Oracle Certified Associate Exam for Java SE 8. I wish you the best of luck!

—Simon Ritter
Deputy CTO, Azul Systems

Preface

Writing This Book

Dear Reader, what you hold in your hand is the result of a meticulous high-tech operation that took many months and required inspecting many parts, removing certain parts, retrofitting some old parts, and adding many new parts to our previous book on an earlier Java programmer certification exam, until we were completely satisfied with the result. After you have read the book and passed the exam, we hope that you will appreciate the TLC (tender loving care) that has gone into this operation. This is how it all came about.

Learning the names of Java certifications and the required exams is the first item on the agenda. This book provides coverage for the exam to earn *Oracle Certified Associate (OCA), Java SE 8 Programmer Certification* (also know as OCAJP8). The exam required for this certification has the name *Java SE 8 Programmer I Exam* (*Exam number 1Z0-808*). It is the first of two exams required to obtain *Oracle Certified Professional (OCP), Java SE 8 Programmer Certification* (also known as OCPJP8). The second exam required for this professional certification has the name *Java SE 8 Programmer II Exam* (*Exam number 1Z0-809*). To reiterate, this book covers only the topics for the *Java SE 8 Programmer I Exam* that is required to obtain OCAJP8 certification.

A book on the new Java SE 8 certification was a long time coming. The mantle of Java had been passed on to Oracle and Java 7 had hit the newsstand. We started out to write a book to cover the topics for the two exams required to earn the *Oracle Certified Professional, Java SE 7 Programmer Certification*. Soon after the release of Java 8, Oracle announced the certification for Java SE 8. We decided to switch to the new version. It was not a difficult decision to make. Java 8 marks a watershed when the language went from being a pure object-oriented language to one that also incorporates features of functional-style programming. As the saying goes, Java 8 changed the whole ballgame. Java passed its twentieth birthday in 2015. Java 8, released a year earlier, represented a significant milestone in its history. There was little reason to dwell on earlier versions.

The next decision concerned whether it would be best to provide coverage for the two Java SE 8 Programmer Certification exams in one or two books. Pragmatic reasons dictated two books. It would take far too long to complete a book that covered both exams, mainly because the second exam was largely revamped and would require a lot of new material. We decided to complete the book for the first exam. Once that decision was made, our draft manuscript went back on the operating table.

Our approach to writing this book has not changed from the one we employed for our previous books, mainly because it has proved successful. No stones were left unturned to create this book, as we explain here.

The most noticeable changes in the exam for OCAJP8 are the inclusion of the core classes in the new Date and Time API and the writing of predicates using lambda expressions. The emphasis remains on analyzing code scenarios, rather than individual language constructs. The exam continues to require actual experience with the language, not just mere recitation of facts. We still claim that proficiency in the language is the key to success.

Since the exam emphasizes the core features of Java, this book provides in-depth coverage of topics related to those features. As in our earlier books, supplementary topics are also included to aid in mastering the exam topics.

This book is no different from our previous books in one other important aspect: It is a one-stop guide, providing a mixture of theory and practice that enables readers to prepare for the exam. It can be used to learn Java and to prepare for the exam. After the exam is passed, it can also come in handy as a language guide.

Apart from including coverage of the new topics, our discussions of numerous topics from the previous exam were extensively revised. All elements found in our previous books (e.g., sections, examples, figures, tables, review questions, mock exam questions) were closely scrutinized. New examples, figures, tables, and review questions were specifically created for the new topics as well as for the revised ones. We continue to use UML (Unified Modeling Language) extensively to illustrate concepts and language constructs, and all numbered examples continue to be complete Java programs ready for experimenting.

Feedback from readers regarding our previous books was invaluable in shaping this book. Every question, suggestion, and comment received was deliberated upon. We are grateful for every single email we have received over the years; that input proved invaluable in improving this book.

Dear Reader, we wish you all the best should you decide to go down the path of Java certification. May your loops terminate and your exceptions get caught!

About This Book

This book provides extensive coverage of the core features of the Java programming language and its core application programming interface (API), with particular

emphasis on its syntax and usage. The book is primarily intended for professionals who want to prepare for the *Java SE 8 Programmer I* exam, but it is readily accessible to any programmer who wants to master the language. For both purposes, it provides in-depth coverage of essential features of the language and its core API.

The demand for well-trained and highly skilled Java programmers remains unabated. Oracle offers many Java certifications that professionals can take to validate their skills (see `http://education.oracle.com`). The certification provides members of the IT industry with a standard to use when hiring such professionals, and it allows professionals to turn their Java skills into credentials that are important for career advancement.

The book provides extensive coverage of all the objectives defined by Oracle for the *Java SE 8 Programmer I* exam. The exam objectives are selective, however, and do not include many of the essential features of Java. This book covers many additional topics that every Java programmer should master to be truly proficient. In this regard, the book is a comprehensive primer for learning the Java programming language. After mastering the language by working through this book, the reader can confidently sit for the exam.

This book is *not* a complete reference for Java, as it does not attempt to list every member of every class from the Java SE platform API documentation. The purpose is not to document the Java SE platform API. The emphasis is more on the Java programming language features—their syntax and correct usage through code examples—and less on teaching programming techniques.

The book assumes little background in programming. We believe the exam is accessible to any programmer who works through the book. A Java programmer can easily skip over material that is well understood and concentrate on parts that need reinforcing, whereas a programmer new to Java will find the concepts explained from basic principles.

Each topic is explained and discussed thoroughly with examples, and backed by review questions and exercises to reinforce the concepts. The book is not biased toward any particular platform, but provides platform-specific details where necessary.

Using This Book

The reader can choose a linear or a nonlinear route through the book, depending on his or her programming background. Non-Java programmers wishing to migrate to Java can read Chapter 1, which provides a short introduction to object-oriented programming concepts, and the procedure for compiling and running Java applications. For those preparing for *Java SE 8 Programmer I* exam, the book has a separate appendix (Appendix A) providing all the pertinent information on preparing for and taking the exam.

Cross-references are provided where necessary to indicate the relationships among the various constructs of the language. To understand a language construct, all

pertinent details are provided where the construct is covered, but in addition, cross-references are provided to indicate its relationship to other constructs. Sometimes it is necessary to postpone discussion of certain aspects of a topic if they depend on concepts that have not yet been covered in the book. A typical example is the consequences of object-oriented programming concepts (for example, inheritance) on the member declarations that can occur in a class. This approach can result in forward references in the initial chapters of the book.

The table of contents; listings of tables, examples, and figures; and a comprehensive index facilitate locating topics discussed in the book.

In particular, we draw attention to the following features of the book:

Programmer I Exam Objectives

0.1 Exam objectives are stated clearly at the beginning of every chapter.
0.2 The number in front of the objective identifies the exam objective, as defined by Oracle, and can be found in Appendix B.
0.3 The objectives are organized into major sections, detailing the curriculum for the exam.
0.4 The objectives for the *Java SE 8 Programmer I* exam are reproduced verbatim in Appendix B, where for each section of the syllabus, references are included to point the reader to relevant topics in the book.

Supplementary Objectives

- Supplementary objectives cover topics that are *not* on the exam, but which we believe are important for mastering the topics that *are* on the exam.
- Any supplementary objective is listed as a bullet at the beginning of the chapter.

Review Questions

Review questions are provided after every major topic to test and reinforce the material. The review questions predominantly reflect the kind of multiple-choice questions that can be asked on the actual exam. On the exam, the exact number of answers to choose for each question is explicitly stated. The review questions in this book follow that practice.

Many questions on the actual exam contain code snippets with line numbers to indicate that complete implementation is not provided, and that the necessary missing code to compile and run the code snippets can be assumed. The review questions in this book provide complete code implementations where possible, so that the code can be readily compiled and run.

Annotated answers to the review questions are provided in Appendix C.

Example 0.1 *Example Source Code*

We encourage readers to experiment with the code examples to reinforce the material from the book. These examples can be downloaded from the book website (see p. xxxiv).

Java code is presented in a `monospaced font`. Lines of code in the examples or in code snippets are referenced in the text by a number, which is specified by using a single-line comment in the code. For example, in the following code snippet, the call to the method `doSomethingInteresting()` at (1) does something interesting:

```
// ...
doSomethingInteresting();                                    // (1)
// ...
```

Names of classes and interfaces start with an uppercase letter. Names of packages, variables, and methods start with a lowercase letter. Constants are in all uppercase letters. Interface names begin with the prefix `I`, when it makes sense to distinguish them from class names. Coding conventions are followed, except when we have had to deviate from these conventions in the interest of space or clarity.

Chapter Summary

Each chapter concludes with a summary of the topics covered in the chapter, pointing out the major concepts that were introduced.

Programming Exercises

Programming exercises at the end of each chapter provide the opportunity to put concepts into practice. Solutions to the programming exercises are provided in Appendix D.

Mock Exam

The mock exam in Appendix E should be attempted when the reader feels confident about the topics on the exam. It is highly recommended to read Appendix A before attempting the mock exam, as Appendix A contains pertinent information about the questions to expect on the actual exam. Each multiple-choice question in the mock exam explicitly states how many answers are applicable for a given question, as is the case on the actual exam. Annotated answers to the questions in the mock exam are provided in Appendix F.

Java SE Platform API Documentation

A vertical gray bar is used to highlight methods and fields found in the classes of the Java SE Platform API.

Any explanation following the API information is also similarly highlighted.

To obtain the maximum benefit from using this book in preparing for the *Java SE 8 Programmer I* exam, we strongly recommend installing the latest version (Release 8 or newer) of the JDK and its accompanying API documentation. The book focuses solely on Java 8, and does not acknowledge previous versions.

Book Website

This book is backed by a website providing auxiliary material:

```
www.ii.uib.no/~khalid/ocajp8/
```

The contents of the website include the following:

- Source code for all the examples in the book
- Solutions to the programming exercises in the book
- Annotated answers to the reviews questions in the book
- Annotated answers to the mock exam in the book
- Table of contents, sample chapter, and index from the book
- Errata for the book
- Links to miscellaneous Java resources (e.g., certification, discussion groups, tools)

Information about the Java Standard Edition (SE) and its documentation can be found at the following website:

```
www.oracle.com/technetwork/java/javase/overview/index.html
```

The current authoritative technical reference for the Java programming language, *The Java® Language Specification: Java SE 8 Edition* (also published by Addison-Wesley), can be found at this website:

```
http://docs.oracle.com/javase/specs/index.html
```

Request for Feedback

Considerable effort has been made to ensure the accuracy of the content of this book. All code examples (including code fragments) have been compiled and tested on various platforms. In the final analysis, any errors remaining are the sole responsibility of the authors.

Any questions, comments, suggestions, and corrections are welcome. Let us know whether the book was helpful (or not) for your purpose. Any feedback is valuable. The principal author can be reached at the following email address:

```
khalid.mughal@uib.no
```

Register your copy of *A Programmer's Guide to Java® SE 8 Oracle Certified Associate (OCA)* at informit.com for convenient access to downloads, updates, and corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN (9780132930215) and click Submit. Once the process is complete, you will find any available bonus content under "Registered Products."

About the Authors

Khalid A. Mughal

Khalid A. Mughal is an associate professor at the Department of Informatics at the University of Bergen, Norway, where he has been responsible for designing and implementing various courses in informatics. Over the years, he has taught programming (primarily Java), software engineering (object-oriented system development), databases (data modeling and database management systems), compiler techniques, web application development, and software security courses. For 15 years, he was responsible for developing and running web-based programming courses in Java, which were offered to off-campus students. He has also given numerous courses and seminars at various levels in object-oriented programming and system development using Java and Java-related technologies, both at the University of Bergen and for the IT industry.

Mughal is the principal author and solely responsible for the contents of this book. He is also the principal author of three books on previous versions of the Java programmer certification—*A Programmer's Guide to Java™ SCJP Certification: A Comprehensive Primer, Third Edition* (0321556054); *A Programmer's Guide to Java™ Certification: A Comprehensive Primer, Second Edition* (0201728281); and *A Programmer's Guide to Java™ Certification* (0201596148)—and three introductory textbooks on programming in Java: *Java Actually: A First Course in Programming* (1844804186); *Java Actually: A Comprehensive Primer in Java Programming* (1844809331); and *Java som første programmeringsspråk/Java as First Programming Language, Third Edition* (8202245540).

Mughal currently works on security issues related to mobile data collection systems for delivering health services in low- and middle-income countries.

Rolf W. Rasmussen

Rolf W. Rasmussen is a system development manager at Vizrt, a company that develops solutions for the TV broadcast industry, including real-time 3D graphic renderers, and content and control systems. Rasmussen works mainly on control and automation systems, video processing, typography, and real-time visualization. He has worked on clean-room implementations of the Java class libraries in the past and is a contributor to the Free Software Foundation.

Over the years, Rasmussen has worked both academically and professionally with numerous programming languages, including Java. He was primarily responsible for developing the review questions and answers, the programming exercises and their solutions, the mock exam, and all the practical aspects related to taking the exam in our three previous books on Java programmer certification. Selected earlier content has been utilized in this book. Together with Mughal, he is also a co-author of three introductory textbooks on programming in Java.

Acknowledgments

At Addison-Wesley, Greg Doench was again our editor, who effectively managed the process of publishing this book. Regular dialog with him in recent months helped to keep this project on track. Julie Nahil was the in-house contact at Addison-Wesley, who professionally managed the production of the book. Anna Popick was the project editor, who diligently handled the day-to-day project management for this book. Jill Hobbs did a truly marvelous job copy editing the book. The folks at The CIP Group performed the typesetting wizardry necessary to materialize the book. We would like to extend our sincere thanks to Greg, Julie, Anna, Jill, the folks at The CIP Group, and all those behind the scenes at Addison-Wesley, who helped to put this publication on the bookshelf.

For the technical review of the book, we were lucky that Roel De Nijs agreed to take on the task. If you drop in on `CodeRanch.com`, you are bound to find him executing his duties as a Sheriff, especially helping greenhorns find their bearing in the Java certification corrals. He is a freelance Java developer with many IT companies as clients and a multitude of Java certification accolades under his belt (SCJA, SCJP, SCJD, OCAJP7). And not least, he is a Technical Reviewer Par Excellence. Without doubt, Roel has a meticulous eye for detail. It is no exaggeration to say that his exhaustive feedback has been invaluable in improving the quality of this book at all levels. Roel, you have our most sincere thanks for your many excellent comments and suggestions, and above all, for weeding out numerous pesky errors in the manuscript.

Over the years, we have also been lucky to have our own personal manuscript quality controller: Marit Seljeflot Mughal. As diligently as with our previous books, she tirelessly proofread several chapter drafts for this book, and put her finger on many unmentionable mistakes and errors in the manuscript. Her valuable comments and suggestions have also been instrumental in improving the quality of this book. If Marit, who has no IT background, could make sense of the Java jargon we wrote, then we were confident our readers would as well. Our most sincere thanks.

Great effort has been made to eliminate mistakes and errors in this book. We accept full responsibility for any remaining oversights. We hope that when our Dear Readers find any, they will bring them to our attention.

Many family occasions have been missed while working on this book. Without family support, this book would not have seen the light of day. Khalid is ever grateful to his family for their love, support, and understanding—but especially when he is working on a book. Now that this book is out the door, he is off to play with his three grandchildren.

—Khalid A. Mughal

17 May 2016
Bergen, Norway

This page intentionally left blank

Declarations

3

Programmer I Exam Objectives	
[1.2] Define the structure of a Java class	*§3.1, p. 48*
[4.1] Declare, instantiate, initialize, and use a one-dimensional array	*§3.4, p. 58*
[4.2] Declare, instantiate, initialize, and use multi-dimensional array	*§3.4, p. 64*
[6.1] Create methods with arguments and return values; including overloaded methods ❍ *For return statement, see §6.4, p. 224.* ❍ *For covariant return, see §7.2, p. 273.* ❍ *For overloaded method resolution, see §7.10, p. 316.* ❍ *For overriding methods, see §7.2, p. 268.*	*§3.2, p. 49* *§3.5, p. 72*
[6.3] Create and overload constructors; including impact on default constructors ❍ *For constructor chaining, see §7.5, p. 282.*	*§3.3, p. 53*
[6.6] Determine the effect upon object references and primitive values when they are passed into methods that change the values ❍ *For conversions in assignment and method invocation contexts, see §5.2, p. 147.*	*§3.5, p. 72*
[7.4] Use super and this to access objects and constructors ❍ *For the* `super` *reference, see §7.4, p. 276.* ❍ *For using* `this()` *and* `super()`*, see §7.5, p. 282.*	*§3.2, p. 50*
Supplementary Objectives	
• Sorting and searching arrays	*§3.4, p. 68* *§3.4, p. 69*
• Declaring and calling methods with a variable arity parameter	*§3.6, p. 81*
• Passing values to a Java application as program arguments on the command line	*§3.7, p. 85*
• Basic introduction to declaring and using enum types	*§3.8, p. 87*

3.1 Class Declarations

A class declaration introduces a new reference type. For the purpose of this book, we will use the following simplified syntax of a class declaration:

```
class_modifiers class class_name
                      extends_clause
                      implements_clause // Class header
{ // Class body
     field_declarations
     method_declarations
     constructor_declarations
}
```

In the class header, the name of the class is preceded by the keyword `class`. In addition, the class header can specify the following information:

- An *accessibility modifier* (§4.5, p. 118)
- Additional *class modifiers* (§4.6, p. 120)
- Any class it *extends* (§7.1, p. 264)
- Any interfaces it *implements* (§7.6, p. 290)

The class body, enclosed in braces (`{}`), can contain *member declarations*. In this book, we discuss the following two kinds of member declarations:

- *Field declarations* (§2.3, p. 40)
- *Method declarations* (§3.2, p. 49)

Members declared `static` belong to the class and are called *static members*. Non-static members belong to the objects of the class and are called *instance members*. In addition, the following declarations can be included in a class body:

- *Constructor declarations* (§3.3, p. 53)

The declarations can appear in any order in the class body. The only mandatory parts of the class declaration syntax are the keyword `class`, the class name, and the class body braces (`{}`), as exemplified by the following class declaration:

```
class X { }
```

To understand which code can be legally declared in a class, we distinguish between *static context* and *non-static context*. A static context is defined by static methods, static field initializers, and static initializer blocks. A non-static context is defined by instance methods, non-static field initializers, instance initializer blocks, and constructors. By *static code*, we mean expressions and statements in a static context; by *non-static code*, we mean expressions and statements in a non-static context. One crucial difference between the two contexts is that static code can refer only to other static members.

3.2 Method Declarations

For the purpose of this book, we will use the following simplified syntax of a method declaration:

method_modifiers return_type method_name
(*formal_parameter_list*) *throws_clause* `// Method header`

```
{ // Method body
    local_variable_declarations
    statements
}
```

In addition to the name of the method, the method header can specify the following information:

- Scope or *accessibility modifier* (§4.7, p. 123)
- Additional *method modifiers* (§4.8, p. 131)
- The *type* of the *return value*, or `void` if the method does not return any value (§6.4, p. 224)
- A *formal parameter list*
- Any *exceptions* thrown by the method, which are specified in a `throws` clause (§6.9, p. 251)

The *formal parameter list* is a comma-separated list of parameters for passing information to the method when the method is invoked by a *method call* (§3.5, p. 72). An empty parameter list must be specified by `()`. Each parameter is a simple variable declaration consisting of its type and name:

optional_parameter_modifier type parameter_name

The parameter names are local to the method (§4.4, p. 117). The *optional parameter modifier* `final` is discussed in §3.5, p. 80. It is recommended to use the `@param` tag in a Javadoc comment to document the formal parameters of a method.

The *signature* of a method comprises the name of the method and the types of the formal parameters only.

The method body is a *block* containing the *local variable declarations* (§2.3, p. 40) and the *statements* of the method.

The mandatory parts of a method declaration are the return type, the method name, and the method body braces (`{}`), as exemplified by the following method declaration:

```
void noAction() {}
```

Like member variables, member methods can be characterized as one of two types:

- *Instance methods*, which are discussed later in this section
- *Static methods*, which are discussed in §4.8, p. 132

Statements

Statements in Java can be grouped into various categories. Variable declarations with explicit initialization of the variables are called *declaration statements* (§2.3, p. 40, and §3.4, p. 60). Other basic forms of statements are *control flow statements* (§6.1, p. 200) and *expression statements*.

An *expression statement* is an expression terminated by a semicolon. Any value returned by the expression is discarded. Only certain types of expressions have meaning as statements:

- Assignments (§5.6, p. 158)
- Increment and decrement operators (§5.9, p. 176)
- Method calls (§3.5, p. 72)
- Object creation expressions with the `new` operator (§5.17, p. 195)

A solitary semicolon denotes the *empty statement*, which does nothing.

A block, `{}`, is a *compound* statement that can be used to group zero or more local declarations and statements (§4.4, p. 117). Blocks can be nested, since a block is a statement that can contain other statements. A block can be used in any context where a simple statement is permitted. The compound statement that is embodied in a block begins at the left brace, `{`, and ends with a matching right brace, `}`. Such a block must not be confused with an array initializer in declaration statements (§3.4, p. 60).

Labeled statements are discussed in §6.4 on page 220.

Instance Methods and the Object Reference `this`

Instance methods belong to every object of the class and can be invoked only on objects. All members defined in the class, both static and non-static, are accessible in the context of an instance method. The reason is that all instance methods are passed an implicit reference to the *current object*—that is, the object on which the method is being invoked. The current object can be referenced in the body of the instance method by the keyword `this`. In the body of the method, the `this` reference can be used like any other object reference to access members of the object. In fact, the keyword `this` can be used in any non-static context. The `this` reference can be used as a normal reference to reference the current object, but the reference cannot be modified—it is a `final` reference (§4.8, p. 133).

The `this` reference to the current object is useful in situations where a local variable hides, or *shadows*, a field with the same name. In Example 3.1, the two parameters `noOfWatts` and `indicator` in the constructor of the `Light` class have the same names as the fields in the class. The example also declares a local variable `location`, which has the same name as one of the fields. The reference `this` can be used to distinguish the fields from the local variables. At (1), the `this` reference is used to identify the field `noOfWatts`, which is assigned the value of the parameter `noOfWatts`. Without the `this` reference at (2), the value of the parameter `indicator` is assigned back to

this parameter, and not to the field by the same name, resulting in a logical error. Similarly at (3), without the `this` reference, it is the local variable `location` that is assigned the value of the parameter `site`, and not the field with the same name.

Example 3.1 *Using the* `this` *Reference*

```
public class Light {
  // Fields:
  int     noOfWatts;      // Wattage
  boolean indicator;      // On or off
  String  location;       // Placement

  // Constructor
  public Light(int noOfWatts, boolean indicator, String site) {
    String location;

    this.noOfWatts = noOfWatts;   // (1) Assignment to field
    indicator = indicator;        // (2) Assignment to parameter
    location = site;              // (3) Assignment to local variable
    this.superfluous();           // (4)
    superfluous();                // equivalent to call at (4)
  }

  public void superfluous() {
    System.out.printf("Current object: %s%n", this); // (5)
  }

  public static void main(String[] args) {
    Light light = new Light(100, true, "loft");
    System.out.println("No. of watts: " + light.noOfWatts);
    System.out.println("Indicator:    " + light.indicator);
    System.out.println("Location:     " + light.location);
  }
}
```

Probable output from the program:

```
Current object: Light@1bc4459
Current object: Light@1bc4459
No. of watts: 100
Indicator:    false
Location:     null
```

If a member is not shadowed by a local declaration, the simple name `member` is considered a short-hand notation for `this.member`. In particular, the `this` reference can be used explicitly to invoke other methods in the class. This usage is illustrated at (4) in Example 3.1, where the method `superfluous()` is called.

If, for some reason, a method needs to pass the current object to another method, it can do so using the `this` reference. This approach is illustrated at (5) in Example 3.1, where the current object is passed to the `printf()` method. The `printf()` method

prints the string representation of the current object (which comprises the name of the class of the current object and the hexadecimal representation of the current object's hash code). (The *hash code* of an object is an `int` value that can be used to store and retrieve the object from special data structures called *hash tables*.)

Note that the `this` reference cannot be used in a static context, as static code is not executed in the context of any object.

Method Overloading

Each method has a *signature*, which comprises the name of the method plus the types and order of the parameters in the formal parameter list. Several method implementations may have the same name, as long as the method signatures differ. This practice is called *method overloading*. Because overloaded methods have the same name, their parameter lists must be different.

Rather than inventing new method names, method overloading can be used when the same logical operation requires multiple implementations. The Java SE platform API makes heavy use of method overloading. For example, the class `java.lang.Math` contains an overloaded method `min()`, which returns the minimum of two numeric values.

```
public static double min(double a, double b)
public static float min(float a, float b)
public static int min(int a, int b)
public static long min(long a, long b)
```

In the following examples, five implementations of the method `methodA` are shown:

```
void methodA(int a, double b) { /* ... */ }      // (1)
int  methodA(int a)           { return a; }      // (2)
int  methodA()                { return 1; }      // (3)
long methodA(double a, int b) { return b; }      // (4)
long methodA(int x, double y) { return x; }      // (5) Not OK.
```

The corresponding signatures of the five methods are as follows:

```
methodA(int, double)          1'
methodA(int)                  2': Number of parameters
methodA()                     3': Number of parameters
methodA(double, int)          4': Order of parameters
methodA(int, double)          5': Same as 1'
```

The first four implementations of the method named `methodA` are overloaded correctly, each time with a different parameter list and, therefore, different signatures. The declaration at (5) has the same signature `methodA(int, double)` as the declaration at (1) and, therefore, is not a valid overloading of this method.

```
void bake(Cake k)  { /* ... */ }                 // (1)
void bake(Pizza p) { /* ... */ }                 // (2)

int     halfIt(int a) { return a/2; }            // (3)
double  halfIt(int a) { return a/2.0; }          // (4) Not OK. Same signature.
```

The method named `bake` is correctly overloaded at (1) and (2), with two different parameter lists. In the implementation, changing just the return type (as shown at (3) and (4) in the preceding example), is not enough to overload a method, and will be flagged as a compile-time error. The parameter list in the declarations must be different.

Only methods declared in the same class and those that are inherited by the class can be overloaded. Overloaded methods should be considered to be individual methods that just happen to have the same name. Methods with the same name are allowed, since methods are identified by their signature. At compile time, the right implementation of an overloaded method is chosen, based on the signature of the method call. Details of method overloading resolution can be found in §7.10 on page 316. Method overloading should not be confused with *method overriding* (§7.2, p. 268).

3.3 Constructors

The main purpose of constructors is to set the initial state of an object, when the object is created by using the `new` operator.

For the purpose of this book, we will use the following simplified syntax of a constructor:

```
accessibility_modifier class_name (formal_parameter_list)
                                    throws_clause // Constructor header
{ // Constructor body
    local_variable_declarations
    statements
}
```

Constructor declarations are very much like method declarations. However, the following restrictions on constructors should be noted:

- Modifiers other than an accessibility modifier are not permitted in the constructor header. For accessibility modifiers for constructors, see §4.7, p. 123.
- Constructors cannot return a value and, therefore, do not specify a return type, not even `void`, in the constructor header. But their declaration can use the `return` statement that does not return a value in the constructor body (§6.4, p. 224).
- The constructor name must be the same as the class name.

Class names and method names exist in different *namespaces*. Thus, there are no name conflicts in Example 3.2, where a method declared at (2) has the same name as the constructor declared at (1). A method must always specify a return type, whereas a constructor does not. However, using such naming schemes is strongly discouraged.

A constructor that has no parameters, like the one at (1) in Example 3.2, is called a *no-argument constructor*.

Example 3.2 *Namespaces*

```
public class Name {

  Name() {                      // (1) No-argument constructor
    System.out.println("Constructor");
  }

  void Name() {                 // (2) Instance method
    System.out.println("Method");
  }

  public static void main(String[] args) {
    new Name().Name();          // (3) Constructor call followed by method call
  }
}
```

Output from the program:

```
Constructor
Method
```

The Default Constructor

If a class does not specify *any* constructors, then a *default constructor* is generated for the class by the compiler. The default constructor is equivalent to the following implementation:

class_name`() { super(); }   // No parameters. Calls superclass constructor.`

A default constructor is a no-argument constructor. The only action taken by the default constructor is to call the superclass constructor. This ensures that the inherited state of the object is initialized properly (§7.5, p. 282). In addition, all instance variables in the object are set to the default value of their type, barring those that are initialized by an initialization expression in their declaration.

In the following code, the class `Light` does not specify any constructors:

```
class Light {
  // Fields:
  int     noOfWatts;       // Wattage
  boolean indicator;       // On or off
  String  location;        // Placement

  // No constructors
  //...
}

class Greenhouse {
  // ...
  Light oneLight = new Light();     // (1) Call to default constructor
}
```

In this code, the following default constructor is called when a `Light` object is created by the object creation expression at (1):

```
Light() { super(); }
```

Creating an object using the `new` operator with the default constructor, as at (1), will initialize the fields of the object to their default values (that is, the fields `noOfWatts`, `indicator`, and `location` in a `Light` object will be initialized to `0`, `false`, and `null`, respectively).

A class can choose to provide its own constructors, rather than relying on the default constructor. In the following example, the class `Light` provides a no-argument constructor at (1).

```
class Light {
  // ...
  Light() {                          // (1) No-argument constructor
    noOfWatts = 50;
    indicator = true;
    location  = "X";
  }
  //...
}

class Greenhouse {
  // ...
  Light extraLight = new Light();   // (2) Call of explicit default constructor
}
```

The no-argument constructor ensures that any object created with the object creation expression `new Light()`, as at (2), will have its fields `noOfWatts`, `indicator`, and `location` initialized to `50`, `true`, and `"X"`, respectively.

If a class defines *any* constructor, it can no longer rely on the default constructor to set the state of its objects. If such a class requires a no-argument constructor, it must provide its own implementation, as in the preceding example. In the next example the class `Light` does not provide a no-argument constructor, but rather includes a non-zero argument constructor at (1). It is called at (2) when an object of the class `Light` is created with the `new` operator. Any attempt to call the default constructor will be flagged as a compile-time error, as shown at (3).

```
class Light {
  // ...
  // Only non-zero argument constructor:
  Light(int noOfWatts, boolean indicator, String location) {          // (1)
    this.noOfWatts = noOfWatts;
    this.indicator = indicator;
    this.location  = location;
  }
  //...
}

class Greenhouse {
  // ...
  Light moreLight  = new Light(100, true, "Greenhouse");// (2) OK
  Light firstLight = new Light();                       // (3) Compile-time error
}
```

Overloaded Constructors

Like methods, constructors can be overloaded. Since the constructors in a class all have the same name as the class, their signatures are differentiated by their parameter lists. In the following example, the class `Light` now provides explicit implementation of the no-argument constructor at (1) and that of a non-zero argument constructor at (2). The constructors are overloaded, as is evident by their signatures. The non-zero argument constructor at (2) is called when an object of the class `Light` is created at (3), and the no-argument constructor is likewise called at (4). Overloading of constructors allows appropriate initialization of objects on creation, depending on the constructor invoked (see chaining of constructors in §7.5, p. 282). It is recommended to use the `@param` tag in a Javadoc comment to document the formal parameters of a constructor.

```
class Light {
  // ...
  // No-argument constructor:
  Light() {                                                    // (1)
    noOfWatts = 50;
    indicator = true;
    location  = "X";
  }

  // Non-zero argument constructor:
  Light(int noOfWatts, boolean indicator, String location) { // (2)
    this.noOfWatts = noOfWatts;
    this.indicator = indicator;
    this.location  = location;
  }
  //...
}

class Greenhouse {
  // ...
  Light moreLight  = new Light(100, true, "Greenhouse");     // (3) OK
  Light firstLight = new Light();                            // (4) OK
}
```

Review Questions

3.1 Which one of these declarations is a valid method declaration?

Select the one correct answer.

(a) `void method1         { /* ... */ }`
(b) `void method2()       { /* ... */ }`
(c) `void method3(void)   { /* ... */ }`
(d) `method4()            { /* ... */ }`
(e) `method5(void)        { /* ... */ }`

3.2 Which statements, when inserted at (1), will not result in compile-time errors?

```
public class ThisUsage {
  int planets;
  static int suns;

  public void gaze() {
    int i;
    // (1) INSERT STATEMENT HERE
  }
}
```

Select the three correct answers.

(a) `i = this.planets;`
(b) `i = this.suns;`
(c) `this = new ThisUsage();`
(d) `this.i = 4;`
(e) `this.suns = planets;`

3.3 Given the following pairs of method declarations, which statements are true?

```
void fly(int distance) {}
int  fly(int time, int speed) { return time*speed; }

void fall(int time) {}
int  fall(int distance) { return distance; }

void glide(int time) {}
void Glide(int time) {}
```

Select the two correct answers.

(a) The first pair of methods will compile, and overload the method name `fly`.
(b) The second pair of methods will compile, and overload the method name `fall`.
(c) The third pair of methods will compile, and overload the method name `glide`.
(d) The first pair of methods will not compile.
(e) The second pair of methods will not compile.
(f) The third pair of methods will not compile.

3.4 Given a class named `Book`, which one of these constructor declarations is valid for the class `Book`?

Select the one correct answer.

(a) `Book(Book b) {}`
(b) `Book Book() {}`
(c) `private final Book() {}`
(d) `void Book() {}`
(e) `public static void Book(String[] args) {}`
(f) `abstract Book() {}`

3.5 Which statements are true?

Select the two correct answers.

(a) A class must define a constructor.
(b) A constructor can be declared `private`.
(c) A constructor can return a value.
(d) A constructor must initialize all fields when a class is instantiated.
(e) A constructor can access the non-static members of a class.

3.6 What will be the result of compiling the following program?

```
public class MyClass {
  long var;

  public void MyClass(long param) { var = param; }  // (1)

  public static void main(String[] args) {
    MyClass a, b;
    a = new MyClass();                              // (2)
    b = new MyClass(5);                             // (3)
  }
}
```

Select the one correct answer.

(a) A compile-time error will occur at (1).
(b) A compile-time error will occur at (2).
(c) A compile-time error will occur at (3).
(d) The program will compile without errors.

3.4 Arrays

An *array* is a data structure that defines an indexed collection of a fixed number of homogeneous data elements. This means that all elements in the array have the same data type. A position in the array is indicated by a non-negative integer value called the *index*. An element at a given position in the array is accessed using the index. The size of an array is fixed and cannot be changed after the array has been created.

In Java, arrays are objects. Arrays can be of primitive data types or reference types. In the former case, all elements in the array are of a specific primitive data type. In the latter case, all elements are references of a specific reference type. References in the array can then denote objects of this reference type or its subtypes. Each array object has a `public final` field called `length`, which specifies the array size (i.e., the number of elements the array can accommodate). The first element is always at index 0 and the last element at index $n - 1$, where n is the value of the `length` field in the array.

Simple arrays are *one-dimensional arrays*—that is, a simple list of values. Since arrays can store reference values, the objects referenced can also be array objects. Thus, multidimensional arrays are implemented as *array of arrays*.

Passing array references as parameters is discussed in §3.5, p. 72. Type conversions for array references on assignment and on method invocation are discussed in §7.7, p. 309.

Declaring Array Variables

A one-dimensional array variable declaration has either of the following syntaxes:

element_type`[]` *array_name*`;`

or

element_type array_name`[];`

where *element_type* can be a primitive data type or a reference type. The array variable *array_name* has the type *element_type*`[]`. Note that the array size is not specified. As a consequence, the array variable *array_name* can be assigned the reference value of an array of any length, as long as its elements have *element_type.*

It is important to understand that the declaration does not actually create an array. Instead, it simply declares a *reference* that can refer to an array object. The `[]` notation can also be specified after a variable name to declare it as an array variable, but then it applies to just that variable.

```
int anIntArray[], oneInteger;
Pizza[] mediumPizzas, largePizzas;
```

These two declarations declare `anIntArray` and `mediumPizzas` to be reference variables that can refer to arrays of `int` values and arrays of `Pizza` objects, respectively. The variable `largePizzas` can denote an array of `Pizza` objects, but the variable `oneInteger` cannot denote an array of `int` values—it is a simple variable of the type `int`.

An array variable that is declared as a field in a class, but is not explicitly initialized to any array, will be initialized to the default reference value `null`. This default initialization does *not* apply to *local* reference variables and, therefore, does not apply to local array variables either (§2.4, p. 42). This behavior should not be confused with initialization of the elements of an array during array construction.

Constructing an Array

An array can be constructed for a fixed number of elements of a specific type, using the `new` operator. The reference value of the resulting array can be assigned to an array variable of the corresponding type. The syntax of the *array creation expression* is shown on the right-hand side of the following assignment statement:

array_name `= new` *element_type*`[`*array_size*`];`

The minimum value of *array_size* is 0; in other words zero-length arrays can be constructed in Java. If the array size is negative, a `NegativeArraySizeException` is thrown at runtime.

Given the declarations

```
int anIntArray[], oneInteger;
Pizza[] mediumPizzas, largePizzas;
```

the three arrays in the declarations can be constructed as follows:

```
anIntArray   = new int[10];          // array for 10 integers
mediumPizzas = new Pizza[5];         // array of 5 pizzas
largePizzas  = new Pizza[3];         // array of 3 pizzas
```

The array declaration and construction can be combined.

element_type$_1$`[]` *array_name* `=` `new` *element_type*$_2$`[`*array_size*`]`;

In the preceding syntax, the array type *element_type*$_2$`[]` must be *assignable* to the array type *element_type*$_1$`[]` (§7.7, p. 309). When the array is constructed, all of its elements are initialized to the default value for *element_type*$_2$. This is true for both member and local arrays when they are constructed.

In the next examples, the code constructs the array, and the array elements are implicitly initialized to their default values. For example, all elements of the array `anIntArray` get the value `0`, and all elements of the array `mediumPizzas` get the value `null` when the arrays are constructed.

```
int[] anIntArray = new int[10];                 // Default element value: 0
Pizza[] mediumPizzas = new Pizza[5];            // Default element value: null
```

The value of the field `length` in each array is set to the number of elements specified during the construction of the array; for example, `mediumPizzas.length` has the value `5`.

Once an array has been constructed, its elements can also be explicitly initialized individually—for example, in a loop. The examples in the rest of this section make use of a loop to traverse the elements of an array for various purposes.

Initializing an Array

Java provides the means of declaring, constructing, and explicitly initializing an array in one declaration statement:

element_type`[]` *array_name* `=` `{` *array_initialize_list* `}`;

This form of initialization applies to fields as well as to local arrays. The *array_initialize_list* is a comma-separated list of zero or more expressions. Such an array initializer results in the construction and initialization of the array.

```
int[] anIntArray = {13, 49, 267, 15, 215};
```

In this declaration statement, the variable `anIntArray` is declared as a reference to an array of `int`s. The array initializer results in the construction of an array to hold

five elements (equal to the length of the list of expressions in the block), where the first element is initialized to the value of the first expression (13), the second element to the value of the second expression (49), and so on.

```
Pizza[] pizzaOrder = { new Pizza(), new Pizza(), null };
```

In this declaration statement, the variable pizzaOrder is declared as a reference to an array of Pizza objects. The array initializer constructs an array to hold three elements. The initialization code sets the first two elements of the array to refer to two Pizza objects, while the last element is initialized to the null reference. The reference value of the array of Pizza objects is assigned to the reference pizzaOrder. Note also that this declaration statement actually creates *three* objects: the array object with three references and the two Pizza objects.

The expressions in the *array_initialize_list* are evaluated from left to right, and the array name obviously cannot occur in any of the expressions in the list. In the preceding examples, the *array_initialize_list* is terminated by the right brace, }, of the block. The list can also be legally terminated by a comma. The following array has length 2, and not 3:

```
Topping[] pizzaToppings = { new Topping("cheese"), new Topping("tomato"), };
```

The declaration statement at (1) in the following code defines an array of four String objects, while the declaration statement at (2) shows that a String object is not the same as an array of char.

```
// Array with 4 String objects:
String[] pets = {"crocodiles", "elephants", "crocophants", "elediles"}; // (1)

// Array of 3 characters:
char[] charArray = {'a', 'h', 'a'};    // (2) Not the same as "aha"
```

Using an Array

The array object is referenced by the array name, but individual array elements are accessed by specifying an index with the [] operator. The array element access expression has the following syntax:

array_name [*index_expression*]

Each individual element is treated as a simple variable of the element type. The *index* is specified by the *index_expression*, whose value should be promotable to an int value; otherwise, a compile-time error is flagged. Since the lower bound of an array index is always 0, the upper bound is 1 less than the array size—that is, *array_name*.length-1. The ith element in the array has index (i-1). At runtime, the index value is automatically checked to ensure that it is within the array index bounds. If the index value is less than 0, or greater than or equal to *array_name*.length, an ArrayIndexOutOfBoundsException is thrown. A program can either check the index explicitly or catch the runtime exception (§6.5, p. 230), but an illegal index is typically an indication of a programming error.

In the array element access expression, the *array_name* can be any expression that returns a reference to an array. For example, the expression on the right-hand side of the following assignment statement returns the character `'H'` at index 1 in the character array returned by a call to the `toCharArray()` method of the `String` class:

```
char letter = "AHA".toCharArray()[1];     // 'H'
```

The array operator `[]` is used to declare array types (`Topping[]`), specify the array size (`new Topping[3]`), and access array elements (`toppings[1]`). This operator is not used when the array reference is manipulated, such as in an array reference assignment (§7.9, p. 312), or when the array reference is passed as an actual parameter in a method call (§3.5, p. 77).

Example 3.3 shows traversal of arrays using `for` loops (§6.3, p. 215 and p. 217). A `for(;;)` loop at (3) in the `main()` method initializes the local array `trialArray` declared at (2) five times with pseudo-random numbers (from `0.0` to `100.0`), by calling the method `randomize()` declared at (5). The minimum value in the array is found by calling the method `findMinimum()` declared at (6), and is stored in the array `storeMinimum` declared at (1). Both of these methods also use a `for(;;)` loop. The loop variable is initialized to a start value—`0` in (3) and (5), and `1` in (6). The loop condition tests whether the loop variable is less than the length of the array; this guarantees that the loop will terminate when the last element has been accessed. The loop variable is incremented after each iteration to access the next element.

A `for(:)` loop at (4) in the `main()` method is used to print the minimum values from the trials, as elements are read consecutively from the array, without keeping track of an index value.

Example 3.3 *Using Arrays*

```
public class Trials {
  public static void main(String[] args) {
    // Declare and construct the local arrays:
    double[] storeMinimum = new double[5];                // (1)
    double[] trialArray = new double[15];                 // (2)
    for (int i = 0; i < storeMinimum.length; ++i) {       // (3)
      // Initialize the array.
      randomize(trialArray);

      // Find and store the minimum value.
      storeMinimum[i] = findMinimum(trialArray);
    }

    // Print the minimum values:                             (4)
    for (double minValue : storeMinimum)
      System.out.printf("%.4f%n", minValue);
  }

  public static void randomize(double[] valArray) {       // (5)
    for (int i = 0; i < valArray.length; ++i)
      valArray[i] = Math.random() * 100.0;
  }
```

```
  public static double findMinimum(double[] valArray) {  // (6)
    // Assume the array has at least one element.
    double minValue = valArray[0];
    for (int i = 1; i < valArray.length; ++i)
      minValue = Math.min(minValue, valArray[i]);
    return minValue;
  }
}
```

Probable output from the program:

```
6.9330
2.7819
6.7427
18.0849
26.2462
```

Anonymous Arrays

As shown earlier in this section, the following declaration statement can be used to construct arrays using an array creation expression:

element_type$_1$[] *array_name* = new *element_type*$_2$[*array_size*]; // (1)

```
int[] intArray = new int[5];
```

The size of the array is specified in the array creation expression, which creates the array and initializes the array elements to their default values. By comparison, the following declaration statement both creates the array and initializes the array elements to specific values given in the array initializer:

element_type[] *array_name* = { *array_initialize_list* }; // (2)

```
int[] intArray = {3, 5, 2, 8, 6};
```

However, the array initializer is *not* an expression. Java has another array creation expression, called an *anonymous array,* which allows the concept of the array creation expression from (1) to be combined with the array initializer from (2), so as to create and initialize an array object:

new *element_type*[] { *array_initialize_list* }

```
new int[] {3, 5, 2, 8, 6}
```

This construct has enough information to create a nameless array of a specific type. Neither the name of the array nor the size of the array is specified. The construct returns the reference value of the newly created array, which can be assigned to references and passed as argument in method calls. In particular, the following declaration statements are equivalent:

```
int[] intArray = {3, 5, 2, 8, 6};                    // (1)
int[] intArray = new int[] {3, 5, 2, 8, 6};          // (2)
```

In (1), an array initializer is used to create and initialize the elements. In (2), an anonymous array expression is used. It is tempting to use the array initializer as an expression—for example, in an assignment statement, as a shortcut for assigning values to array elements in one go. However, this is illegal; instead, an anonymous array expression should be used. The concept of the anonymous array combines the definition and the creation of the array into one operation.

```
int[] daysInMonth;
daysInMonth = {31, 28, 31, 30, 31, 30,
               31, 31, 30, 31, 30, 31};                  // Compile-time error
daysInMonth = new int[] {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}; // OK
```

In Example 3.4, an anonymous array is constructed at (1), and passed as an actual parameter to the static method `findMinimum()` defined at (2). Note that no array name or array size is specified for the anonymous array.

Example 3.4 *Using Anonymous Arrays*

```
public class AnonArray {
  public static void main(String[] args) {
    System.out.println("Minimum value: " +
        findMinimum(new int[] {3, 5, 2, 8, 6}));                    // (1)
  }

  public static int findMinimum(int[] dataSeq) {                    // (2)
    // Assume the array has at least one element.
    int min = dataSeq[0];
    for (int index = 1; index < dataSeq.length; ++index)
      if (dataSeq[index] < min)
        min = dataSeq[index];
    return min;
  }
}
```

Output from the program:

```
Minimum value: 2
```

Multidimensional Arrays

Since an array element can be an object reference and arrays are objects, array elements can themselves refer to other arrays. In Java, an array of arrays can be defined as follows:

element_type`[][]...[]` *array_name*;

or

element_type array_name`[][]...[]`;

In fact, the sequence of square bracket pairs, `[]`, indicating the number of dimensions, can be distributed as a postfix to both the element type and the array name. Arrays of arrays are often called *multidimensional arrays*.

The following declarations are all equivalent:

```
int[][] mXnArray;      // 2-dimensional array
int[]   mXnArray[];    // 2-dimensional array
int     mXnArray[][];  // 2-dimensional array
```

It is customary to combine the declaration with the construction of the multidimensional array.

```
int[][] mXnArray = new int[4][5];    // 4 x 5 matrix of ints
```

The previous declaration constructs an array `mXnArray` of four elements, where each element is an array (row) of five `int` values. The concept of rows and columns is often used to describe the dimensions of a 2-dimensional array, which is often called a *matrix*. However, such an interpretation is not dictated by the Java language.

Each row in the previous matrix is denoted by `mXnArray[i]`, where $0 \leq i < 4$. Each element in the `i`th row, `mXnArray[i]`, is accessed by `mXnArray[i][j]`, where $0 \leq j < 5$. The number of rows is given by `mXnArray.length`, in this case `4`, and the number of values in the `i`th row is given by `mXnArray[i].length`, in this case `5` for all the rows, where $0 \leq i < 4$.

Multidimensional arrays can also be constructed and explicitly initialized using the array initializers discussed for simple arrays. Note that each row is an array that uses an array initializer to specify its values:

```
double[][] identityMatrix = {
  {1.0, 0.0, 0.0, 0.0 }, // 1. row
  {0.0, 1.0, 0.0, 0.0 }, // 2. row
  {0.0, 0.0, 1.0, 0.0 }, // 3. row
  {0.0, 0.0, 0.0, 1.0 }  // 4. row
}; // 4 x 4 floating-point matrix
```

Arrays in a multidimensional array need not have the same length; when they do not, they are called *ragged arrays*. The array of arrays `pizzaGalore` in the following code has five rows; the first four rows have different lengths but the fifth row is left unconstructed:

```
Pizza[][] pizzaGalore = {
  { new Pizza(), null, new Pizza() },    // 1. row is an array of 3 elements.
  { null, new Pizza()},                  // 2. row is an array of 2 elements.
  new Pizza[1],                          // 3. row is an array of 1 element.
  {},                                    // 4. row is an array of 0 elements.
  null                                   // 5. row is not constructed.
};
```

When constructing multidimensional arrays with the `new` operator, the length of the deeply nested arrays may be omitted. In such a case, these arrays are left unconstructed. For example, an array of arrays to represent a room on a floor in a hotel on a street in a city can have the type `HotelRoom[][][][]`. From left to right, the

square brackets represent indices for street, hotel, floor, and room, respectively. This 4-dimensional array of arrays can be constructed piecemeal, starting with the leftmost dimension and proceeding to the rightmost successively.

```
HotelRoom[][][][] rooms = new HotelRoom[10][5][][];  // Just streets and hotels.
```

The preceding declaration constructs the array of arrays `rooms` partially with ten streets, where each street has five hotels. Floors and rooms can be added to a particular hotel on a particular street:

```
rooms[0][0]       = new HotelRoom[3][]; // 3 floors in 1st hotel on 1st street.
rooms[0][0][0]    = new HotelRoom[8];   // 8 rooms on 1st floor in this hotel.
rooms[0][0][0][0] = new HotelRoom();    // Initializes 1st room on this floor.
```

The next code snippet constructs an array of arrays `matrix`, where the first row has one element, the second row has two elements, and the third row has three elements. Note that the outer array is constructed first. The second dimension is constructed in a loop that constructs the array in each row. The elements in the multidimensional array will be implicitly initialized to the default `double` value (`0.0D`). In Figure 3.1, the array of arrays `matrix` is depicted after the elements have been explicitly initialized.

```
double[][] matrix = new double[3][];      // Number of rows.

for (int i = 0; i < matrix.length; ++i)
  matrix[i] = new double[i + 1];          // Construct a row.
```

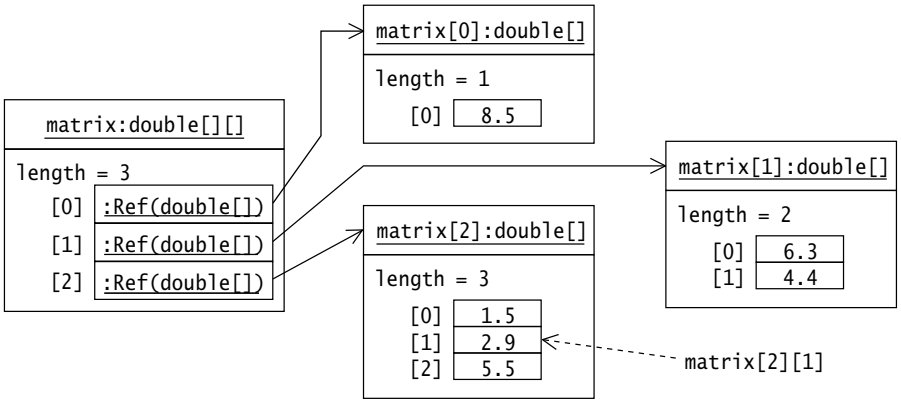
Two other ways of initializing such an array of arrays are shown next. The first approach uses array initializers, and the second uses an anonymous array of arrays.

```
double[][] matrix2 = {    // Using array initializers.
  {0.0},                  // 1. row
  {0.0, 0.0},             // 2. row
  {0.0, 0.0, 0.0}         // 3. row
};

double[][] matrix3 = new double[][] { // Using an anonymous array of arrays.
  {0.0},                  // 1. row
  {0.0, 0.0},             // 2. row
  {0.0, 0.0, 0.0}         // 3. row
};
```

The type of the variable `matrix` is `double[][]`, a two-dimensional array of `double` values. The type of the variable `matrix[i]` (where `0 ≤ i< matrix.length`) is `double[]`, a one-dimensional array of `double` values. The type of the variable `matrix[i][j]` (where `0 ≤ i< matrix.length` and `0 ≤ j< matrix[i].length`) is `double`, a simple variable of type `double`.

Nested loops are a natural match for manipulating multidimensional arrays. In Example 3.5, a rectangular 4 × 3 `int` matrix is declared and constructed at (1). The program finds the minimum value in the matrix. The outer loop at (2) traverses the rows (`mXnArray[i]`, where `0 ≤ i< mXnArray.length`), and the inner loop at (3) traverses the elements in each row in turn (`mXnArray[i][j]`, where `0 ≤ j< mXnArray[i].length`). The outer loop is executed `mXnArray.length` times, or 4 times, and the inner loop is

Figure 3.1 *Array of Arrays*



executed (`mXnArray.length`) × (`mXnArray[i].length`), or 12 times, since all rows have the same length 3.

The `for(:)` loop also provides a safe and convenient way of traversing an array. Several examples of its use are provided in §6.3, p. 217.

Example 3.5 *Using Multidimensional Arrays*

```
public class MultiArrays {

  public static void main(String[] args) {
    // Declare and construct the M X N matrix.
    int[][] mXnArray = {                                              // (1)
        {16,  7, 12}, // 1. row
        { 9, 20, 18}, // 2. row
        {14, 11,  5}, // 3. row
        { 8,  5, 10}  // 4. row
    }; // 4 x 3 int matrix

    // Find the minimum value in a M X N matrix:
    int min = mXnArray[0][0];
    for (int i = 0; i < mXnArray.length; ++i)                         // (2)
      // Find min in mXnArray[i], in the row given by index i:
      for (int j = 0; j < mXnArray[i].length; ++j)                    // (3)
        min = Math.min(min, mXnArray[i][j]);

    System.out.println("Minimum value: " + min);
  }
}
```

Output from the program:

```
Minimum value: 5
```

Sorting Arrays

Sorting implies ordering the elements according to some ranking criteria, usually based on the *values* of the elements. The values of numeric data types can be compared and ranked by using the relational operators. For comparing objects of a class, the class typically implements the `compareTo()` method of the `Comparable` interface. The ordering defined by this method is called the *natural ordering* for the objects of the class. The wrapper classes for primitive values and the `String` class implement the `compareTo()` method (§8.3, p. 350, and §8.4, p. 363, respectively).

The `java.util.Arrays` class provides many overloaded versions of the `sort()` method to sort practically any type of array.

> `void sort(`*type*`[] array)`
>
> Permitted *type* for elements includes `byte`, `char`, `double`, `float`, `int`, `long`, `short`, and `Object`. The method sorts the elements in the array according to their *natural ordering*. In the case of an array of objects being passed as argument, the *objects* must be *mutually comparable*; that is, it should be possible to compare any two objects in the array according to the natural ordering defined by the `compareTo()` method of the `Comparable` interface.

An appropriate `import` statement should be included in the source code to access the `java.util.Arrays` class. In the next code snippet, an array of strings is sorted according to natural ordering for strings—that is, based on the Unicode values of the characters in the strings:

```
String[] strArray =  {"biggest", "big", "bigger", "Bigfoot"};
Arrays.sort(strArray);    // Natural ordering: [Bigfoot, big, bigger, biggest]
```

The next examples illustrate sorting an array of primitive values (`int`) at (1), and an array of type `Object` containing mutually comparable elements (`String`) at (2). In (3), the numerical values are autoboxed into their corresponding wrapper classes (§8.3, p. 346), but the objects of different wrapper classes and the `String` class are not mutually comparable. In (4), the numerical values are also autoboxed into their corresponding wrapper classes, but again the objects of different wrapper classes are not mutually comparable. A `ClassCastException` is thrown when the elements are not mutually comparable.

```
int[] intArray = {5, 3, 7, 1};              // int
Arrays.sort(intArray);                      // (1) Natural ordering: [1, 3, 5, 7]

Object[] objArray1 = {"I", "am", "OK"};     // String
Arrays.sort(objArray1);                     // (2) Natural ordering: [I, OK, am]

Object[] objArray2 = {23, "ten", 3.14};     // Not mutually comparable
Arrays.sort(objArray2);                     // (3) ClassCastException

Number[] numbers = {23, 3.14, 10L};         // Not mutually comparable
Arrays.sort(numbers);                       // (4) ClassCastException
```

Searching Arrays

A common operation on an array is to search the array for a given element, called the *key*. The `java.util.Arrays` class provides overloaded versions of the `binarySearch()` method to search in practically any type of array that is *sorted*.

> `int binarySearch(type[] array, type key)`
>
> Permitted *type* for elements include `byte`, `char`, `double`, `float`, `int`, `long`, `short`, and `Object`. The array must be sorted in ascending order before calling this method, or the results are unpredictable. In the case where an array of objects is passed as argument, the *objects* must be sorted in ascending order according to their *natural ordering*, as defined by the `Comparable` interface.
>
> The method returns the index to the key in the sorted array, if the key exists. The index is then guaranteed to be greater or equal to 0. If the key is not found, a negative index is returned, corresponding to –(*insertion point + 1*), where *insertion point* is the index of the element where the key would have been found, if it had been in the array. If there are duplicate elements equal to the key, there is no guarantee which duplicate's index will be returned. The elements and the key must be *mutually comparable*.

An appropriate `import` statement should be included in the source code to access the `java.util.Arrays` class. In the code that follows, the return value –3 indicates that the key would have been found at index 2 had it been in the list:

```
// Sorted String array (natural ordering): [Bigfoot, big, bigger, biggest]
// Search in natural ordering:
int index1 = Arrays.binarySearch(strArray, "bigger");   // Successful:    2
int index2 = Arrays.binarySearch(strArray, "bigfeet");  // Unsuccessful: -3
int index3 = Arrays.binarySearch(strArray, "bigmouth"); // Unsuccessful: -5
```

Results are unpredictable if the array is not sorted, or if the ordering used in the search is not the same as the sort ordering. Searching in the `strArray` using natural ordering when the array is sorted in reverse natural ordering gives the wrong result:

```
// Sorted String array (inverse natural ordering): [biggest, bigger, big, Bigfoot]
// Search in natural ordering:
int index4 = Arrays.binarySearch(strArray, "big");  //  -1 (INCORRECT)
```

A `ClassCastException` is thrown if the key and the elements are not mutually comparable:

```
int index5 = Arrays.binarySearch(strArray, 4); // Key: 4 => ClassCastException
```

However, this incompatibility is caught at compile time in the case of arrays with primitive values:

```
// Sorted int array (natural ordering): [1, 3, 5, 7]
int index6 = Arrays.binarySearch(intArray, 4.5);//Key: 4.5 => compile-time error!
```

The method `binarySearch()` derives its name from the divide-and-conquer algorithm that it uses to perform the search. It repeatedly divides the remaining elements to be searched into two halves and selects the half containing the key to continue the search in, until either the key is found or there are no more elements left to search.

Review Questions

3.7 Given the following declaration, which expression returns the size of the array, assuming that the `array` reference has been properly initialized?

```
int[] array;
```

Select the one correct answer.

(a) `array[].length()`
(b) `array.length()`
(c) `array[].length`
(d) `array.length`
(e) `array[].size()`
(f) `array.size()`
(g) `array[].size`
(h) `array.size`

3.8 Is it possible to create arrays of length zero?

Select the one correct answer.

(a) Yes, you can create arrays of any type with length zero.
(b) Yes, but only for primitive data types.
(c) Yes, but only for arrays of reference types.
(d) No, you cannot create zero-length arrays, but the `main()` method may be passed a zero-length array of `Strings` when no program arguments are specified.
(e) No, it is not possible to create arrays of length zero in Java.

3.9 Which one of the following array declaration statements is not legal?

Select the one correct answer.

(a) `int []a[] = new int [4][4];`
(b) `int a[][] = new int [4][4];`
(c) `int a[][] = new int [][4];`
(d) `int []a[] = new int [4][];`
(e) `int [][]a = new int [4][4];`

3.10 Which of these array declaration statements are not legal?

Select the two correct answers.

(a) `int[] i[] = { { 1, 2 }, { 1 }, {}, { 1, 2, 3 } };`
(b) `int i[] = new int[2] {1, 2};`
(c) `int i[][] = new int[][] { {1, 2, 3}, {4, 5, 6} };`
(d) `int i[][] = { { 1, 2 }, new int[ 2 ] };`
(e) `int i[4] = { 1, 2, 3, 4 };`

3.11 What would be the result of compiling and running the following program?

```
public class MyClass {
  public static void main(String[] args) {
    int size = 20;
    int[] arr = new int[ size ];

    for (int i = 0; i < size; ++i) {
      System.out.println(arr[i]);
    }
  }
}
```

Select the one correct answer.

(a) The code will not compile, because the array type `int[]` is incorrect.
(b) The program will compile, but will throw an `ArrayIndexOutOfBoundsException` when run.
(c) The program will compile and run without error, but will produce no output.
(d) The program will compile and run without error, and will print the numbers 0 through 19.
(e) The program will compile and run without error, and will print `0` twenty times.
(f) The program will compile and run without error, and will print `null` twenty times.

3.12 What would be the result of compiling and running the following program?

```
public class DefaultValuesTest {
  int[] ia = new int[1];
  boolean b;
  int i;
  Object o;

  public static void main(String[] args) {
    DefaultValuesTest instance = new DefaultValuesTest();
    instance.print();
  }

  public void print() {
    System.out.println(ia[0] + " " + b + " " + i + " " + o);
  }
}
```

Select the one correct answer.

(a) The program will fail to compile because of uninitialized variables.
(b) The program will throw a `java.lang.NullPointerException` when run.
(c) The program will print `0 false NaN null`.
(d) The program will print `0 false 0 null`.
(e) The program will print `null 0 0 null`.
(f) The program will print `null false 0 null`.

3.5 Parameter Passing

Objects communicate by calling methods on each other. A *method call* is used to invoke a method on an object. Parameters in the method call provide one way of exchanging information between the caller object and the callee object (which need not be different).

Declaring methods is discussed in §3.2, p. 49. Invoking static methods on classes is discussed in §4.8, p. 132.

The syntax of a method call can be any one of the following:

object_reference`.`*method_name*`(`*actual_parameter_list*`)`

class_name`.`*static_method_name*`(`*actual_parameter_list*`)`

method_name`(`*actual_parameter_list*`)`

The *object_reference* must be an expression that evaluates to a reference value denoting the object on which the method is called. If the caller and the callee are the same, *object reference* can be omitted (see the discussion of the `this` reference in §3.2, p. 50). The *class_name* can be the *fully qualified name* (§4.2, p. 97) of the class. The *actual_parameter_list* is *comma separated* if there is more than one parameter. The parentheses are mandatory even if the actual parameter list is empty. This distinguishes the method call from field access. One can specify fully qualified names for classes and packages using the dot operator (`.`).

```
objRef.doIt(time, place);         // Explicit object reference
int i = java.lang.Math.abs(-1);   // Fully qualified class name
int j = Math.abs(-1);             // Simple class name
someMethod(ofValue);              // Object or class implicitly implied
someObjRef.make().make().make();  // make() returns a reference value
```

The dot operator (`.`) has left associativity. In the last code line, the first call of the `make()` method returns a reference value that denotes the object on which to execute the next call, and so on. This is an example of *call chaining*.

Each *actual parameter* (also called an *argument*) is an expression that is evaluated, and whose value is passed to the method when the method is invoked. Its value can vary from invocation to invocation. *Formal parameters* are parameters defined in the *method declaration* (§3.2, p. 49) and are local to the method (§2.4, p. 44).

In Java, all parameters are *passed by value*—that is, an actual parameter is evaluated and its value is assigned to the corresponding formal parameter. Table 3.1 summarizes the value that is passed depending on the type of the parameters. In the case of primitive data types, the data value of the actual parameter is passed. If the actual parameter is a reference to an object, the reference value of the denoted object is passed and not the object itself. Analogously, if the actual parameter is an array element of a primitive data type, its data value is passed, and if the array element is a reference to an object, then its reference value is passed.

Table 3.1 *Parameter Passing by Value*

Data type of the formal parameter	Value passed
Primitive data type	Primitive data value of the actual parameter
Reference type (i.e., class, interface, array, or enum type)	Reference value of the actual parameter

It should also be stressed that each invocation of a method has its own copies of the formal parameters, as is the case for any local variables in the method (§6.5, p. 230).

The order of evaluation in the actual parameter list is always *from left to right*. The evaluation of an actual parameter can be influenced by an earlier evaluation of an actual parameter. Given the following declaration:

```
int i = 4;
```

the method call

```
leftRight(i++, i);
```

is effectively the same as

```
leftRight(4, 5);
```

and not the same as

```
leftRight(4, 4);
```

An overview of the conversions that can take place in a method invocation context is provided in §5.2, p. 148. Method invocation conversions for primitive values are discussed in the next subsection (p. 73), and those for reference types are discussed in §7.10, p. 315. Calling variable arity methods is discussed in §3.6, p. 81.

For the sake of simplicity, the examples in subsequent sections primarily show method invocation on the same object or the same class. The parameter passing mechanism is no different when different objects or classes are involved.

Passing Primitive Data Values

An actual parameter is an expression that is evaluated first, with the resulting value then being assigned to the corresponding formal parameter at method invocation. The use of this value in the method has no influence on the actual parameter. In particular, when the actual parameter is a variable of a primitive data type, the value of the variable is copied to the formal parameter at method invocation. Since formal parameters are local to the method, any changes made to the formal parameter will not be reflected in the actual parameter after the call completes.

Legal type conversions between actual parameters and formal parameters of *primitive data types* are summarized here from Table 5.1, p. 147:

- Widening primitive conversion
- Unboxing conversion, followed by an optional widening primitive conversion

These conversions are illustrated by invoking the following method

```
static void doIt(long i) { /* ... */ }
```

with the following code:

```
Integer intRef = 34;
Long longRef = 34L;
doIt(34);         // (1) Primitive widening conversion: long <-- int
doIt(longRef);    // (2) Unboxing: long <-- Long
doIt(intRef);     // (3) Unboxing, followed by primitive widening conversion:
                  //     long <-- int <-- Integer
```

However, for parameter passing, there are no implicit narrowing conversions for integer constant expressions (§5.2, p. 148).

Example 3.6 *Passing Primitive Values*

```
public class CustomerOne {
  public static void main (String[] args) {
    PizzaFactory pizzaHouse = new PizzaFactory();
    int pricePrPizza = 15;
    System.out.println("Value of pricePrPizza before call: " + pricePrPizza);
    double totPrice = pizzaHouse.calcPrice(4, pricePrPizza);             // (1)
    System.out.println("Value of pricePrPizza after call: " + pricePrPizza);
  }
}

class PizzaFactory {
  public double calcPrice(int numberOfPizzas, double pizzaPrice) {       // (2)
    pizzaPrice = pizzaPrice / 2.0;       // Changes price.
    System.out.println("Changed pizza price in the method: " + pizzaPrice);
    return numberOfPizzas * pizzaPrice;
  }
}
```

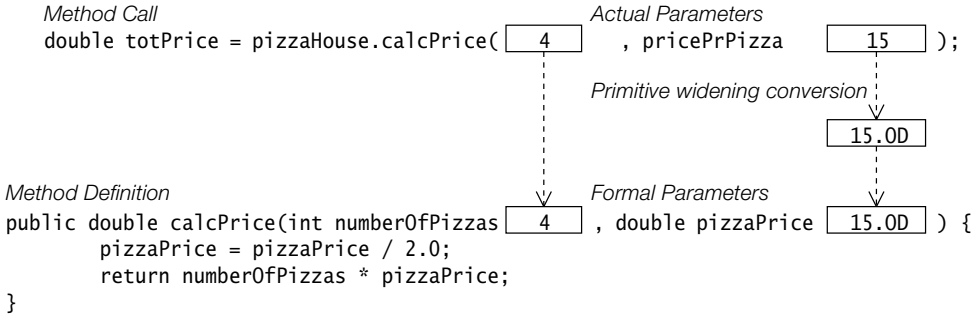
Output from the program:

```
Value of pricePrPizza before call: 15
Changed pizza price in the method: 7.5
Value of pricePrPizza after call: 15
```

In Example 3.6, the method `calcPrice()` is defined in the class `PizzaFactory` at (2). It is called from the `CustomerOne.main()` method at (1). The value of the first actual parameter, `4`, is copied to the `int` formal parameter `numberOfPizzas`. Note that the second actual parameter `pricePrPizza` is of the type `int`, while the corresponding formal parameter `pizzaPrice` is of the type `double`. Before the value of the actual parameter `pricePrPizza` is copied to the formal parameter `pizzaPrice`, it is implicitly widened to a `double`. The passing of primitive values is illustrated in Figure 3.2.

The value of the formal parameter `pizzaPrice` is changed in the `calcPrice()` method, but this does not affect the value of the actual parameter `pricePrPizza` on

Figure 3.2 *Parameter Passing: Primitive Data Values*



return: It still has the value 15. The bottom line is that the formal parameter is a local variable, and changing its value does not affect the value of the actual parameter.

Passing Reference Values

If the actual parameter expression evaluates to a reference value, the resulting reference value is assigned to the corresponding formal parameter reference at method invocation. In particular, if an actual parameter is a reference to an object, the reference value stored in the actual parameter is passed. Consequently, both the actual parameter and the formal parameter are aliases to the object denoted by this reference value during the invocation of the method. In particular, this implies that changes made to the object via the formal parameter *will* be apparent after the call returns.

Type conversions between actual and formal parameters of reference types are discussed in §7.10, p. 315.

In Example 3.7, a `Pizza` object is created at (1). Any object of the class `Pizza` created using the class declaration at (5) always results in a beef pizza. In the call to the `bake()` method at (2), the reference value of the object referenced by the actual parameter `favoritePizza` is assigned to the formal parameter `pizzaToBeBaked` in the declaration of the `bake()` method at (3).

Example 3.7 *Passing Reference Values*

```
public class CustomerTwo {
  public static void main (String[] args) {
    Pizza favoritePizza = new Pizza();              // (1)
    System.out.println("Meat on pizza before baking: " + favoritePizza.meat);
    bake(favoritePizza);                            // (2)
    System.out.println("Meat on pizza after baking: " + favoritePizza.meat);
  }
```

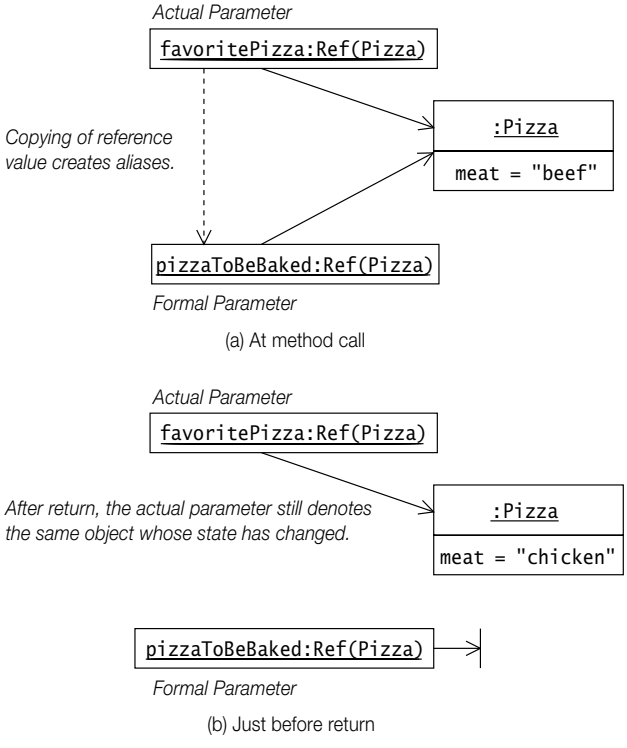
```
  public static void bake(Pizza pizzaToBeBaked) {   // (3)
    pizzaToBeBaked.meat = "chicken";  // Change the meat on the pizza.
    pizzaToBeBaked = null;                          // (4)
  }
}

class Pizza {                                       // (5)
  String meat = "beef";
}
```

Output from the program:

```
Meat on pizza before baking: beef
Meat on pizza after baking: chicken
```

One particular consequence of passing reference values to formal parameters is that any changes made to the object via formal parameters will be reflected back in the calling method when the call returns. In this case, the reference `favoritePizza` will show that chicken has been substituted for beef on the pizza. Setting the formal parameter `pizzaToBeBaked` to `null` at (4) does not change the reference value in the actual parameter `favoritePizza`. The situation at method invocation, and just before the return from method `bake()`, is illustrated in Figure 3.3.

Figure 3.3 *Parameter Passing: Reference Values*



In summary, the formal parameter can only change the *state* of the object whose reference value was passed to the method.

The parameter passing strategy in Java is *call by value* and not *call by reference*, regardless of the type of the parameter. Call by reference would have allowed values in the actual parameters to be changed via formal parameters; that is, the value in `pricePrPizza` would be halved in Example 3.6 and `favoritePizza` would be set to `null` in Example 3.7. However, this cannot be directly implemented in Java.

Passing Arrays

The discussion of passing reference values in the previous section is equally valid for arrays, as arrays are objects in Java. Method invocation conversions for array types are discussed along with those for other reference types in §7.10, p. 315.

In Example 3.8, the idea is to repeatedly swap neighboring elements in an integer array until the largest element in the array *percolates* to the last position in the array.

Example 3.8 *Passing Arrays*

```
public class Percolate {

  public static void main (String[] args) {
    int[] dataSeq = {8,4,6,2,1};    // Create and initialize an array.

    // Write array before percolation:
    printIntArray(dataSeq);

    // Percolate:
    for (int index = 1; index < dataSeq.length; ++index)
      if (dataSeq[index-1] > dataSeq[index])
        swap(dataSeq, index-1, index);                    // (1)

    // Write array after percolation:
    printIntArray(dataSeq);
  }

  public static void swap(int[] intArray, int i, int j) { // (2)
    int tmp = intArray[i]; intArray[i] = intArray[j]; intArray[j] = tmp;
  }

  public static void swap(int v1, int v2) {               // (3) Logical error!
    int tmp = v1; v1 = v2; v2 = tmp;
  }

  public static void printIntArray(int[] array) {         // (4)
    for (int value : array)
      System.out.print(" " + value);
    System.out.println();
  }
}
```

Output from the program:

```
8 4 6 2 1
4 6 2 1 8
```

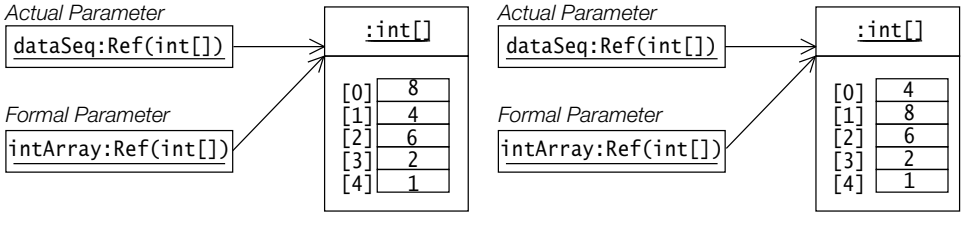
Note that in the declaration of the method `swap()` at (2), the formal parameter `intArray` is of the array type `int[]`. The `swap()` method is called in the `main()` method at (1), where one of the actual parameters is the array variable `dataSeq`. The reference value of the array variable `dataSeq` is assigned to the array variable `intArray` at method invocation. After return from the call to the `swap()` method, the array variable `dataSeq` will reflect the changes made to the array via the corresponding formal parameter. This situation is depicted in Figure 3.4 at the first call and return from the `swap()` method, indicating how the values of the elements at indices `0` and `1` in the array have been swapped.

However, the declaration of the `swap()` method at (3) will *not* swap two values. The method call

```
swap(dataSeq[index-1], dataSeq[index]);
```

will have no effect on the array elements, as the swapping is done on the values of the formal parameters.

Figure 3.4 *Parameter Passing: Arrays*



(a) At first call to the `swap()` method (b) Just before first return from the `swap()` method

The method `printIntArray()` at (4) also has a formal parameter of array type `int[]`. Note that the formal parameter is specified as an array reference using the `[]` notation, but this notation is not used when an array is passed as an actual parameter.

Array Elements as Actual Parameters

Array elements, like other variables, can store values of primitive data types or reference values of objects. In the latter case, they can also be arrays—that is, arrays of arrays (§3.4, p. 63). If an array element is of a primitive data type, its data value is passed; if it is a reference to an object, the reference value is passed. The method invocation conversions apply to the values of array elements as well.

Example 3.9 *Array Elements as Primitive Data Values*

```
public class FindMinimum {

  public static void main(String[] args) {
    int[] dataSeq = {6,4,8,2,1};

    int minValue = dataSeq[0];
    for (int index = 1; index < dataSeq.length; ++index)
      minValue = minimum(minValue, dataSeq[index]);            // (1)

    System.out.println("Minimum value: " + minValue);
  }

  public static int minimum(int i, int j) {                    // (2)
    return (i <= j) ? i : j;
  }
}
```

Output from the program:

```
Minimum value: 1
```

In Example 3.9, the value of all but one element of the array `dataSeq` is retrieved and passed consecutively at (1) to the formal parameter `j` of the `minimum()` method defined at (2). The discussion in §3.5, p. 73, on passing primitive values also applies to array elements that have primitive values.

In Example 3.10, the formal parameter `seq` of the `findMinimum()` method defined at (4) is an array variable. The variable `matrix` denotes an array of arrays declared at (1) simulating a multidimensional array, which has three rows, where each row is a simple array. The first row, denoted by `matrix[0]`, is passed to the `findMinimum()` method in the call at (2). Each remaining row is passed by its reference value in the call to the `findMinimum()` method at (3).

Example 3.10 *Array Elements as Reference Values*

```
public class FindMinimumMxN {

  public static void main(String[] args) {
    int[][] matrix = { {8,4},{6,3,2},{7} };                    // (1)

    int min = findMinimum(matrix[0]);                          // (2)
    for (int i = 1; i < matrix.length; ++i) {
      int minInRow = findMinimum(matrix[i]);                   // (3)
      min = Math.min(min, minInRow);
    }
    System.out.println("Minimum value in matrix: " + min);
  }
```

```
  public static int findMinimum(int[] seq) {                // (4)
    int min = seq[0];
    for (int i = 1; i < seq.length; ++i)
      min = Math.min(min, seq[i]);
    return min;
  }
}
```

Output from the program:

```
Minimum value in matrix: 2
```

`final` Parameters

A formal parameter can be declared with the keyword `final` preceding the parameter declaration in the method declaration. A `final` parameter is also known as a *blank final variable*; that is, it is blank (uninitialized) until a value is assigned to it, (e.g., at method invocation) and then the value in the variable cannot be changed during the lifetime of the variable (see also the discussion in §4.8, p. 133). The compiler can treat `final` variables as constants for code optimization purposes. Declaring parameters as `final` prevents their values from being changed inadvertently. A formal parameter's declaration as `final` does not affect the caller's code.

The declaration of the method `calcPrice()` from Example 3.6 is shown next, with the formal parameter `pizzaPrice` declared as `final`:

```
public double calcPrice(int numberOfPizzas, final double pizzaPrice) {  // (2')
  pizzaPrice = pizzaPrice/2.0;                          // (3) Not allowed
  return numberOfPizzas * pizzaPrice;
}
```

If this declaration of the `calcPrice()` method is compiled, the compiler will not allow the value of the `final` parameter `pizzaPrice` to be changed at (3) in the body of the method.

As another example, the declaration of the method `bake()` from Example 3.7 is shown here, with the formal parameter `pizzaToBeBaked` declared as `final`:

```
public static void bake(final Pizza pizzaToBeBaked) { // (3)
  pizzaToBeBaked.meat = "chicken";                     // (3a) Allowed
  pizzaToBeBaked = null;                               // (4) Not allowed
}
```

If this declaration of the `bake()` method is compiled, the compiler will not allow the reference value of the `final` parameter `pizzaToBeBaked` to be changed at (4) in the body of the method. Note that this applies to the reference value in the `final` parameter, but not to the object denoted by this parameter. The state of the object can be changed as before, as shown at (3a).

3.6 Variable Arity Methods

A *fixed arity* method must be called with the same number of actual parameters (also called *arguments*) as the number of formal parameters specified in its declaration. If the method declaration specifies two formal parameters, every call of this method must specify exactly two arguments. We say that the arity of this method is 2. In other words, the arity of such a method is fixed, and it is equal to the number of formal parameters specified in the method declaration.

Java also allows declaration of *variable arity* methods, meaning that the number of arguments in its call can be *varied*. As we shall see, invocations of such a method may contain more actual parameters than formal parameters. Variable arity methods are heavily employed in formatting text representation of values, as demonstrated by the variable arity method `System.out.printf()` that is used in many examples for this purpose.

The *last* formal parameter in a variable arity method declaration is declared as follows:

type`...` *formal_parameter_name*

The ellipsis (`...`) is specified between the *type* and the *formal_parameter_name*. The *type* can be a primitive type, a reference type, or a type parameter. Whitespace can be specified on both sides of the ellipsis. Such a parameter is usually called a *variable arity parameter* (also known as *varargs*).

Apart from the variable arity parameter, a variable arity method is identical to a fixed arity method. The method `publish()` is a variable arity method:

```
public static void publish(int n, String... data) {      // (int, String[])
  System.out.println("n: " + n + ", data size: " + data.length);
}
```

The variable arity parameter in a variable arity method is always interpreted as having an array type:

type`[]`

In the body of the `publish()` method, the variable arity parameter `data` has the type `String[]`, so it is a simple array of `String`s.

Only *one* variable arity parameter is permitted in the formal parameter list, and it is always the *last* parameter in the formal parameter list. Given that the method declaration has n formal parameters, and the method call has k actual parameters, k must be equal to or greater than $n - 1$. The last $k - n + 1$ actual parameters are evaluated and stored in an array whose reference value is passed as the value of the actual parameter. In the case of the `publish()` method, n is equal to 2, so k can be 1, 2, 3, and so on. The following invocations of the `publish()` method show which arguments are passed in each method call:

```
publish(1);                 // (1, new String[] {})
publish(2, "two");          // (2, new String[] {"two"})
publish(3, "two", "three"); // (3, new String[] {"two", "three"})
```

Each method call results in an implicit array being created and passed as an argument. This array can contain zero or more argument values that do *not* correspond to the formal parameters preceding the variable arity parameter. This array is referenced by the variable arity parameter `data` in the method declaration. The preceding calls would result in the `publish()` method printing the following output:

```
n: 1, data size: 0
n: 2, data size: 1
n: 3, data size: 2
```

To overload a variable arity method, it is not enough to change the type of the variable arity parameter to an explicit array type. The compiler will complain if an attempt is made to overload the method `transmit()`, as shown in the following code:

```
public static void transmit(String... data) {  }  // Compile-time error!
public static void transmit(String[] data)  {  }  // Compile-time error!
```

These declarations would result in two methods with equivalent signatures in the same class, which is not permitted.

Overloading and overriding of methods with variable arity are discussed in §7.10, p. 316.

Calling a Variable Arity Method

Example 3.11 illustrates various aspects of calling a variable arity method. The method `flexiPrint()` in the `VarargsDemo` class has a variable arity parameter:

```
public static void flexiPrint(Object... data) { // Object[]
  //...
}
```

The variable arity method prints the name of the `Class` object representing the *actual array* that is passed at runtime. It prints the number of elements in this array as well as the text representation of each element in the array.

The method `flexiPrint()` is called in the `main()` method. First with the values of primitive types and `Strings` ((1) to (8)), then it is called with the program arguments (p. 85) supplied in the command line, ((9) to (11)).

Compiling the program results in a *warning* at (9), which we ignore for the time being. The program can still be run, as shown in Example 3.11. The numbers at the end of the lines in the output relate to numbers in the code, and are not printed by the program.

Example 3.11 *Calling a Variable Arity Method*

```
public class VarargsDemo {
  public static void flexiPrint(Object... data) { // Object[]
    // Print the name of the Class object for the varargs parameter.
    System.out.print("Type: " + data.getClass().getName());

    System.out.println("  No. of elements: " + data.length);

    System.out.print("Element values: ");
    for(Object element : data)
      System.out.print(element + " ");
    System.out.println();
  }

  public static void main(String... args) {
    int    day       = 13;
    String monthName = "August";
    int    year      = 2009;

    // Passing primitives and non-array types:
    flexiPrint();                      // (1) new Object[] {}
    flexiPrint(day);                   // (2) new Object[] {Integer.valueOf(day)}
    flexiPrint(day, monthName);        // (3) new Object[] {Integer.valueOf(day),
                                       //                   monthName}
    flexiPrint(day, monthName, year);  // (4) new Object[] {Integer.valueOf(day),
                                       //                   monthName,
                                       //                   Integer.valueOf(year)}
    System.out.println();

    // Passing an array type:
    Object[] dateInfo = {day,          // (5) new Object[] {Integer.valueOf(day),
                         monthName,    //                   monthName,
                         year};        //                   Integer.valueOf(year)}
    flexiPrint(dateInfo);              // (6) Non-varargs call
    flexiPrint((Object) dateInfo);     // (7) new Object[] {(Object) dateInfo}
    flexiPrint(new Object[]{dateInfo});// (8) Non-varargs call
    System.out.println();

    // Explicit varargs or non-varargs call:
    flexiPrint(args);                  // (9) Warning!
    flexiPrint((Object) args);         // (10) Explicit varargs call
    flexiPrint((Object[]) args);       // (11) Explicit non-varargs call
  }
}
```

Compiling the program:

```
>javac VarargsDemo.java
VarargsDemo.java:41: warning: non-varargs call of varargs method with inexact
argument type for last parameter;
    flexiPrint(args);                  // (9) Warning!
               ^
  cast to Object for a varargs call
  cast to Object[] for a non-varargs call and to suppress this warning
1 warning
```

Running the program:

```
>java VarargsDemo To arg or not to arg
Type: [Ljava.lang.Object;  No. of elements: 0                (1)
Element values:
Type: [Ljava.lang.Object;  No. of elements: 1                (2)
Element values: 13
Type: [Ljava.lang.Object;  No. of elements: 2                (3)
Element values: 13 August
Type: [Ljava.lang.Object;  No. of elements: 3                (4)
Element values: 13 August 2009

Type: [Ljava.lang.Object;  No. of elements: 3                (6)
Element values: 13 August 2009
Type: [Ljava.lang.Object;  No. of elements: 1                (7)
Element values: [Ljava.lang.Object;@1eed786
Type: [Ljava.lang.Object;  No. of elements: 1                (8)
Element values: [Ljava.lang.Object;@1eed786

Type: [Ljava.lang.String;  No. of elements: 6                (9)
Element values: To arg or not to arg
Type: [Ljava.lang.Object;  No. of elements: 1                (10)
Element values: [Ljava.lang.String;@187aeca
Type: [Ljava.lang.String;  No. of elements: 6                (11)
Element values: To arg or not to arg
```

Variable Arity and Fixed Arity Method Calls

The calls in (1) to (4) in Example 3.11 are all *variable arity calls*, as an implicit `Object` array is created, in which the values of the actual parameters are stored. The reference value of this array is passed to the method. The printout shows that the type of the parameter is actually an array of `Objects` (`[Ljava.lang.Object;`).

The call at (6) differs from the previous calls, in that the actual parameter is an array that has the *same* type (`Object[]`) as the variable arity parameter, without having to create an implicit array. In such a case, *no* implicit array is created, and the reference value of the array `dateInfo` is passed to the method. See also the result from this call at (6) in the output. The call at (6) is a *fixed arity call* (also called a *non-varargs call*), where no implicit array is created:

```
flexiPrint(dateInfo);              // (6) Non-varargs call
```

However, if the actual parameter is cast to the type `Object` as in (7), a *variable arity* call is executed:

```
flexiPrint((Object) dateInfo);     // (7) new Object[] {(Object) dateInfo}
```

The type of the actual argument is now *not* the same as that of the variable arity parameter, resulting in an array of the type `Object[]` being created, in which the array `dateInfo` is stored as an element. The printout at (7) shows that only the text representation of the `dateInfo` array is printed, and not its elements, as it is the sole element of the implicit array.

The call at (8) is a *fixed arity* call, for the same reason as the call in (6). Now, however, the array `dateInfo` is explicitly stored as an element in an array of the type `Object[]` that matches the type of the variable arity parameter:

```
flexiPrint(new Object[]{dateInfo});// (8) Non-varargs call
```

The output from (8) is the same as the output from (7), where the array `dateInfo` was passed as an element in an implicitly created array of type `Object[]`.

The compiler issues a *warning* for the call at (9):

```
flexiPrint(args);                  // (9) Warning!
```

The actual parameter `args` is an array of the type `String[]`, which is a *subtype* of `Object[]`—the type of the variable arity parameter. The array `args` can be passed in a fixed arity call as an array of the type `String[]`, or in a variable arity call as *an element* in an implicitly created array of the type `Object[]`. *Both* calls are feasible and valid in this case. Note that the compiler chooses a fixed arity call rather than a variable arity call, but also issues a warning. The result at (9) confirms this course of action.

The array `args` of the type `String[]` is explicitly passed as an `Object` in a variable arity call at (10), similar to the call at (7):

```
flexiPrint((Object) args);         // (10) Explicit varargs call
```

The array `args` of type `String[]` is explicitly passed as an array of the type `Object[]` in a fixed arity call at (11). This call is equivalent to the call at (9), where the widening reference conversion is implicit, but now without a warning at compile time. The two calls print the same information, as is evident from the output at (9) and (11):

```
flexiPrint((Object[]) args);       // (11) Explicit non-varargs call
```

3.7 The `main()` Method

The mechanics of compiling and running Java applications using the JDK are outlined in §1.10, p. 16. The `java` command executes a method called `main` in the class specified on the command line. Any class can have a `main()` method, but only the `main()` method of the class specified in the `java` command starts the execution of a Java application.

The `main()` method must have `public` accessibility so that the JVM can call this method (§4.7, p. 123). It is a `static` method belonging to the class, so that no object of the class is required to start the execution (§4.8, p. 132). It does not return a value; that is, it is declared as `void` (§6.4, p. 224). It always has an array of `String` objects as its only formal parameter. This array contains any arguments passed to the program on the command line (see the next subsection). The following method header declarations fit the bill, and any one of them can be used for the `main()` method:

```
public static void main(String[] args)    // Method header
public static void main(String... args)   // Method header
```

The three modifiers can occur in any order in the method header. The requirements given in these examples do not exclude specification of additional modifiers (§4.8, p. 131) or any `throws` clause (§6.9, p. 251). The `main()` method can also be overloaded like any other method (§3.2, p. 52). The JVM ensures that the `main()` method having the previously mentioned method header is the starting point of program execution.

Program Arguments

Any arguments passed to the program on the command line can be accessed in the `main()` method of the class specified on the command line:

```
>java Colors red green blue
```

These arguments are called *program arguments*. Note that the command name, `java`, and the class name `Colors` are not passed to the `main()` method of the class `Colors`, nor are any other options that are specified on the command line passed to this method.

Since the formal parameter of the `main()` method is an array of `String` objects, individual `String` elements in the array can be accessed by using the `[]` operator.

In Example 3.12, the three arguments `red`, `green`, and `blue` can be accessed in the `main()` method of the `Colors` class as `args[0]`, `args[1]`, and `args[2]`, respectively. The total number of arguments is given by the field `length` of the `String` array `args`. Note that program arguments can be passed only as strings, and must be explicitly converted to other values by the program, if necessary.

When no arguments are specified on the command line, an array of zero `String` elements is created and passed to the `main()` method. Thus the reference value of the formal parameter in the `main()` method is never `null`.

Program arguments supply information to the application, which can be used to tailor the runtime behavior of the application according to user requirements.

Example 3.12 *Passing Program Arguments*

```
public class Colors {
  public static void main(String[] args) {
    System.out.println("No. of program arguments: " + args.length);
    for (int i = 0; i < args.length; i++)
      System.out.println("Argument no. " + i + " (" + args[i] + ") has " +
                         args[i].length() + " characters.");
  }
}
```

Running the program:

```
>java Colors red green blue
No. of program arguments: 3
Argument no. 0 (red) has 3 characters.
Argument no. 1 (green) has 5 characters.
Argument no. 2 (blue) has 4 characters.
```

3.8 Enumerated Types

In this section we provide a basic introduction to enumerated types. An *enumerated type* defines *a finite set of symbolic names and their values*. These symbolic names are usually called *enum constants* or *named constants*.

One way to define constants is to declare them as `final`, `static` variables in a class (or interface) declaration:

```
public class MachineState {
  public static final int BUSY = 1;
  public static final int IDLE = 0;
  public static final int BLOCKED = -1;
}
```

Such constants are not type-safe, as *any* `int` value can be used where we need to use a constant declared in the `MachineState` class. Such a constant must be qualified by the class (or interface) name, unless the class is extended (or the interface is implemented). When such a constant is printed, only its value (for example, `0`), and not its name (for example, `IDLE`), is printed. A constant also needs recompiling if its value is changed, as the values of such constants are compiled into the client code.

An enumerated type in Java is a special kind of class type that is much more powerful than the approach outlined earlier for defining collections of named constants.

Declaring Type-safe Enums

The canonical form of declaring an *enum type* is shown here:

```
public enum MachineState        // Enum header
{                               // Enum body
  BUSY, IDLE, BLOCKED           // Enum constants
}
```

The keyword `enum` is used to declare an enum type, as opposed to the keyword `class` for a class declaration. The basic notation requires the *enum type name* in enum header, and *a comma-separated list of enum constants* can be specified in the enum body. Optionally, an access modifier can also be specified in the enum header, as for a (top-level) class. In the example enum declaration, the name of the enum type is `MachineState`. It defines three enum constants with explicit names. An enum constant can be any legal Java identifier, but the convention is to use uppercase letters in the name. Essentially, an enum declaration defines a *reference type* that has a *finite number of permissible values* referenced by the enum constants, and the compiler ensures they are used in a type-safe manner.

Other member declarations can be specified in the body of an enum type, but the canonical form suffices for the purpose of this book. Analogous to a class declaration, an enum type is compiled to Java bytecode that is placed in a separate class file.

The enum types `java.time.Month` and `java.time.DayOfWeek` are two examples of enum types from the Java SE platform API. As we would expect, the `Month` enum type represents the months from `JANUARY` to `DECEMBER`, and the `DayOfWeek` enum type represents the days of the week from `MONDAY` to `SUNDAY`. Examples of their usage can be found in §11.2, p. 462.

Some additional examples of enum types follow:

```
public enum MarchingOrders { LEFT, RIGHT }

public enum TrafficLightState { RED, YELLOW, GREEN }

enum MealType { BREAKFAST, LUNCH, DINNER }
```

Using Type-safe Enums

Example 3.13 illustrates the use of enum constants. An enum type is essentially used in the same way as any other reference type. Enum constants are actually `public`, `static`, `final` fields of the enum type, and they are implicitly initialized with instances of the enum type when the enum type is loaded at runtime. Since the enum constants are static members, they can be accessed using the name of the enum type—analogous to accessing static members in a class or an interface.

Example 3.13 shows a machine client that uses a machine whose state is an enum constant. In this example, we see that an enum constant can be passed as an argument, as shown as (1), and we can declare references whose type is an enum type, as shown as (3), but we *cannot* create new constants (that is, objects) of the enum type `MachineState`. An attempt to do so, at (5), results in a compile-time error.

The string representation of an enum constant is its name, as shown at (4). Note that it is not possible to pass a type of value other than a `MachineState` enum constant in the call to the method `setState()` of the `Machine` class, as shown at (2).

Example 3.13 *Using Enums*

```
// File: MachineState.java
public enum MachineState { BUSY, IDLE, BLOCKED }
```

```
// File: Machine.java
public class Machine {

  private MachineState state;

  public void setState(MachineState state) { this.state = state; }
  public MachineState getState() { return this.state; }
}
```

```
// File: MachineClient.java
public class MachineClient {
  public static void main(String[] args) {
```

```
        Machine machine = new Machine();
        machine.setState(MachineState.IDLE);            // (1) Passed as a value.
        // machine.setState(1);                         // (2) Compile-time error!

        MachineState state = machine.getState();        // (3) Declaring a reference.
        System.out.println(
            "Current machine state: " + state           // (4) Printing the enum name.
        );

        // MachineState newState = new MachineState();  // (5) Compile-time error!

        System.out.println("All machine states:");
        for (MachineState ms : MachineState.values()) { // (6) Traversing over enum
          System.out.println(ms + ":" + ms.ordinal());  //     contants.
        }

        System.out.println("Comparison:");
        MachineState state1 = MachineState.BUSY;
        MachineState state2 = state1;
        MachineState state3 = MachineState.BLOCKED;

        System.out.println(state1 + " == " + state2 + ": " +
                           (state1 == state2));                          // (7)
        System.out.println(state1 + " is equal to " + state2 + ": " +
                           (state1.equals(state2)));                     // (8)
        System.out.println(state1 + " is less than " + state3 + ": " +
                           (state1.compareTo(state3) < 0));              // (9)
      }
    }
```

Output from the program:

```
Current machine state: IDLE
All machine states:
BUSY:0
IDLE:1
BLOCKED:2
Comparison:
BUSY == BUSY: true
BUSY is equal to BUSY: true
BUSY is less than BLOCKED: true
```

Selected Methods for Enum Types

All enum types implicitly have the following useful method:

> `static` *`EnumTypeName`*`[] values()`
>
> Returns an array containing the enum constants of this enum type, *in the order they are specified*.

The loop at (6) in Example 3.13 illustrates traversing over all the `MachineState` enum constants in the order they are specified. An array containing all the `MachineState` constants is obtained by calling the static method `values()` on the enum type.

All enum types are subtypes of the `java.lang.Enum` class, which provides the default behavior. All enum types inherit the following selected methods from the `java.lang.Enum` class:

> `final boolean equals(Object other)`
>
> This method returns `true` if the specified object is equal to this enum constant.
>
> `final int compareTo(E other)`
>
> The *natural order* of the enum constants in an enum type is based on their *ordinal values* (see the `ordinal()` method next). The `compareTo()` method of the `Comparable` interface returns the value zero if this enum constant is equal to the other enum constant, a value less than zero if this enum constant is less than the other enum constant, or a value greater than zero if this enum constant is greater than the other enum constant.
>
> `final int ordinal()`
>
> This method returns the *ordinal value* of this enum constant (that is, its position in its enum type declaration). The first enum constant is assigned an ordinal value of zero. If the ordinal value of an enum constant is less than the ordinal value of another enum constant of the same enum type, the former occurs before the latter in the enum type declaration.

Note that the equality test implemented by the `equals()` method is based on reference equality (`==`) of the enum constants, not on value equality. An enum type has a finite number of distinct objects. Comparing two enum references for equality means determining whether they store the reference value of the same enum constant—in other words, whether the references are aliases. Thus, for any two enum references `state1` and `state2`, the expressions `state1.equals(state2)` and `state1 == state2` are equivalent, as shown at (7) and (8) in Example 3.13.

The ordinal value of the constants in an enum type determines the result of comparing such constants with the `compareTo()` method, as shown at (9) in Example 3.13.

Review Questions

3.13 What will the following program print when run?

```
public class ParameterPass {
  public static void main(String[] args) {
    int i = 0;
    addTwo(i++);
    System.out.println(i);
  }

  static void addTwo(int i) {
    i += 2;
  }
}
```

Select the one correct answer.

(a) 0
(b) 1
(c) 2
(d) 3

3.14 What will be the result of compiling and running the following program?

```
public class Passing {
  public static void main(String[] args) {
    int a = 0; int b = 0;
    int[] bArr = new int[1]; bArr[0] = b;

    inc1(a); inc2(bArr);

    System.out.println("a=" + a + " b=" + b + " bArr[0]=" + bArr[0]);
  }

  public static void inc1(int x) { x++; }

  public static void inc2(int[] x) { x[0]++; }
}
```

Select the one correct answer.

(a) The code will fail to compile, since `x[0]++;` is not a legal statement.
(b) The code will compile and will print `a=1 b=1 bArr[0]=1` at runtime.
(c) The code will compile and will print `a=0 b=1 bArr[0]=1` at runtime.
(d) The code will compile and will print `a=0 b=0 bArr[0]=1` at runtime.
(e) The code will compile and will print `a=0 b=0 bArr[0]=0` at runtime.

3.15 Which statements, when inserted at (1), will result in a compile-time error?

```
public class ParameterUse {
  static void main(String[] args) {
    int a = 0;
    final int b = 1;
    int[] c = { 2 };
    final int[] d = { 3 };
    useArgs(a, b, c, d);
  }

  static void useArgs(final int a, int b, final int[] c, int[] d) {
    // (1) INSERT STATEMENT HERE.
  }
}
```

Select the two correct answers.

(a) `a++;`
(b) `b++;`
(c) `b = a;`
(d) `c[0]++;`
(e) `d[0]++;`
(f) `c = d;`

3.16 Which of the following method declarations are valid declarations?

Select the three correct answers.

(a) `void compute(int... is) { }`
(b) `void compute(int is...) { }`
(c) `void compute(int... is, int i, String... ss) { }`
(d) `void compute(String... ds) { }`
(e) `void compute(String... ss, int len) { }`
(f) `void compute(char[] ca, int... is) { }`

3.17 Given the following code:

```
public class RQ810A40 {
  static void print(Object... obj) {
    System.out.println("Object...: " + obj[0]);
  }
  public static void main(String[] args) {
    // (1) INSERT METHOD CALL HERE.
  }
}
```

Which method call, when inserted at (1), will not result in the following output from the program:

```
Object...: 9
```

Select the one correct answer.

(a) `print("9", "1", "1");`
(b) `print(9, 1, 1);`
(c) `print(new int[] {9, 1, 1});`
(d) `print(new Integer[] {9, 1, 1});`
(e) `print(new String[] {"9", "1", "1"});`
(f) `print(new Object[] {"9", "1", "1"});`
(g) None of the above.

Chapter Summary

The following topics were covered in this chapter:

- An overview of declarations that can be specified in a class
- Declaration of methods, usage of the `this` reference in an instance method, and method overloading
- Declaration of constructors, usage of the default constructor, and overloading of constructors
- Explanation of declaration, construction, initialization, and usage of both one-dimensional and multidimensional arrays, including anonymous arrays
- Sorting and searching arrays

- Parameter passing, both primitive values and object references, including arrays and array elements; and declaring `final` parameters
- Declaring and calling methods with variable arity
- Declaration of the `main()` method whose execution starts the application
- Passing program arguments to the `main()` method
- Declaring and using simple enum types

Programming Exercise

3.1 Write a program to grade a short multiple-choice quiz. The correct answers for the quiz are

```
1. C    5. B
2. A    6. C
3. B    7. C
4. D    8. A
```

Assume that the passing marks are at least 5 out of 8. The program stores the correct answers in an array. The submitted answers are specified as program arguments. Let `X` represent a question that was not answered on the quiz. Use an enum type to represent the result of answering a question.

Example of running the program:

```
>java QuizGrader C B B D B C A X
Question  Submitted Ans. Correct Ans.  Result
    1         C              C        CORRECT
    2         B              A          WRONG
    3         B              B        CORRECT
    4         D              D        CORRECT
    5         B              B        CORRECT
    6         C              C        CORRECT
    7         A              C          WRONG
    8         X              A     UNANSWERED
No. of correct answers:      5
No. of wrong answers:        2
No. of questions unanswered: 1
The candidate PASSED.
```

This page intentionally left blank

Index

Symbols

A

B

D

E

F

G

H

I

J

K

M

P

Q

R

S

T

U

V

W

X

Z