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

A Comprehensive Primer

Khalid A. Mughal
Rolf W. Rasmussen

Addison-Wesley
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

| Figures   | xix |
| Tables    | xxi |
| Examples  | xxiii |
| Foreword  | xxvii |
| Preface   | xxix |

1 Basics of Java Programming  
2 Language Fundamentals  
3 Declarations  
4 Access Control  
5 Operators and Expressions  
6 Control Flow  
7 Object-Oriented Programming  
8 Fundamental Classes  
9 Object Lifetime  
10 The ArrayList<E> Class and Lambda Expressions  
11 Date and Time
A  Taking the Java SE 8 Programmer I Exam  507
B  Exam Topics: Java SE 8 Programmer I  515
C  Annotated Answers to Review Questions  519
D  Solutions to Programming Exercises  553
E  Mock Exam: Java SE 8 Programmer I  571
F  Annotated Answers to Mock Exam I  605

Index  619
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figures</td>
<td>xix</td>
</tr>
<tr>
<td>Tables</td>
<td>xxi</td>
</tr>
<tr>
<td>Examples</td>
<td>xxiii</td>
</tr>
<tr>
<td>Foreword</td>
<td>xxvii</td>
</tr>
<tr>
<td>Preface</td>
<td>xxix</td>
</tr>
</tbody>
</table>

## 1 Basics of Java Programming

1.1 Introduction

1.2 Classes

- Declaring Members: Fields and Methods

1.3 Objects

- Class Instantiation, Reference Values, and References
- Object Aliases

1.4 Instance Members

- Invoking Methods

1.5 Static Members

1.6 Inheritance

1.7 Associations: Aggregation and Composition

1.8 Tenets of Java

**Review Questions**

1.9 Java Programs

1.10 Sample Java Application

- Essential Elements of a Java Application
- Compiling and Running an Application

1.11 Program Output

- Formatted Output
## CONTENTS

1.12  The Java Ecosystem  
Object-Oriented Paradigm  
Interpreted: The JVM  
Architecture-Neutral and Portable Bytecode  
Simplicity  
Dynamic and Distributed  
Robust and Secure  
High Performance and Multithreaded  

Review Questions  
Chapter Summary  
Programming Exercise  

2  Language Fundamentals  
2.1  Basic Language Elements  
Lexical Tokens  
Identifiers  
Keywords  
Separators  
Literals  
Integer Literals  
Floating-Point Literals  
Underscores in Numerical Literals  
Boolean Literals  
Character Literals  
String Literals  
Whitespace  
Comments  

Review Questions  

2.2  Primitive Data Types  
The Integer Types  
The char Type  
The Floating-Point Types  
The boolean Type  

Review Questions  

2.3  Variable Declarations  
Declaring and Initializing Variables  
Reference Variables  

Review Questions  

2.4  Initial Values for Variables  
Default Values for Fields  
Initializing Local Variables of Primitive Data Types  
Initializing Local Reference Variables  
Lifetime of Variables  

Review Questions  
Chapter Summary  
Programming Exercise
3 Declarations

3.1 Class Declarations

3.2 Method Declarations

3.3 Constructors

3.4 Arrays

3.5 Parameter Passing

3.6 Variable Arity Methods

3.7 The main() Method

3.8 Enumerated Types

Review Questions

Chapter Summary

Programming Exercise

4 Access Control

4.1 Java Source File Structure

4.2 Packages

Review Questions

Chapter Summary

Programming Exercise
Running Code from Packages 106

4.3 Searching for Classes 107
Review Questions 110

4.4 Scope Rules 114
Class Scope for Members 114
Block Scope for Local Variables 117

4.5 Accessibility Modifiers for Top-Level Type Declarations 118

4.6 Non-Accessibility Modifiers for Classes 120
abstract Classes 120
final Classes 122
Review Questions 123

4.7 Member Accessibility Modifiers 123
public Members 124
protected Members 126
Default Accessibility for Members 127
private Members 128
Review Questions 129

4.8 Non-Accessibility Modifiers for Members 131
static Members 132
final Members 133
abstract Methods 136
synchronized Methods 136
native Methods 137
temporary Fields 138
volatile Fields 139
Review Questions 140
Chapter Summary 142
Programming Exercise 142

5 Operators and Expressions 143

5.1 Conversions 144
Widening and Narrowing Primitive Conversions 144
Widening and Narrowing Reference Conversions 145
Boxing and Unboxing Conversions 145
Other Conversions 146

5.2 Type Conversion Contexts 147
Assignment Context 147
Method Invocation Context 148
Casting Context of the Unary Type Cast Operator: (type) 148
Numeric Promotion Context 149

5.3 Precedence and Associativity Rules for Operators 150

5.4 Evaluation Order of Operands 152
Left-Hand Operand Evaluation First 152
Operand Evaluation before Operation Execution 153
Left-to-Right Evaluation of Argument Lists 154
6 Control Flow

6.1 Overview of Control Flow Statements 200
6.2 Selection Statements 200
   The Simple if Statement 200
   The if-else Statement 201
   The switch Statement 203
   Review Questions 210
6.3 Iteration Statements 213
   The while Statement 213
   The do-while Statement 214
   The for(;;) Statement 215
   The for(:) Statement 217
6.4 Transfer Statements 219
   Labeled Statements 220
   The break Statement 221
   The continue Statement 223
   The return Statement 224
   Review Questions 226
6.5 Stack-Based Execution and Exception Propagation 230
6.6 Exception Types 233
   The Exception Class 235
   The RuntimeException Class 236
   The Error Class 237
   Checked and Unchecked Exceptions 237
   Defining Customized Exceptions 238
6.7 Exception Handling: try, catch, and finally 238
   The try Block 240
   The catch Clause 240
   The finally Clause 245
6.8 The throw Statement 249
6.9 The throws Clause 251
   Overriding the throws Clause 253
6.10 Advantages of Exception Handling 254
    Review Questions 255
    Chapter Summary 258
    Programming Exercises 258

7 Object-Oriented Programming

7.1 Single Implementation Inheritance 264
   Relationships: is-a and has-a 266
   The Supertype–Subtype Relationship 267
7.2 Overriding Methods 268
   Instance Method Overriding 268
   Covariant return in Overriding Methods 273
   Overriding versus Overloading 273
## CONTENTS

7.3 **Hiding Members**  
   Field Hiding  
   Static Method Hiding  
7.4 **The Object Reference super**  
   Review Questions  
7.5 **Chaining Constructors Using this() and super()**  
   The this() Constructor Call  
   The super() Constructor Call  
   Review Questions  
7.6 **Interfaces**  
   Defining Interfaces  
   Abstract Methods in Interfaces  
   Implementing Interfaces  
   Extending Interfaces  
   Interface References  
   Default Methods in Interfaces  
   Static Methods in Interfaces  
   Constants in Interfaces  
   Review Questions  
7.7 **Arrays and Subtyping**  
   Arrays and Subtype Covariance  
   Array Store Check  
7.8 **Reference Values and Conversions**  
7.9 **Reference Value Assignment Conversions**  
7.10 **Method Invocation Conversions Involving References**  
   Overloaded Method Resolution  
7.11 **Reference Casting and the instanceof Operator**  
   The Cast Operator  
   The instanceof Operator  
   Review Questions  
7.12 **Polymorphism and Dynamic Method Lookup**  
7.13 **Inheritance versus Aggregation**  
7.14 **Basic Concepts in Object-Oriented Design**  
   Encapsulation  
   Cohesion  
   Coupling  
   Review Questions  
   Chapter Summary  
   Programming Exercises  

8 **Fundamental Classes**  
8.1 **Overview of the java.lang Package**  
8.2 **The object Class**  
   Review Questions
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.3</td>
<td>The Wrapper Classes</td>
<td>346</td>
</tr>
<tr>
<td></td>
<td>Common Wrapper Class Constructors</td>
<td>347</td>
</tr>
<tr>
<td></td>
<td>Common Wrapper Class Utility Methods</td>
<td>348</td>
</tr>
<tr>
<td></td>
<td>Numeric Wrapper Classes</td>
<td>351</td>
</tr>
<tr>
<td></td>
<td>The Character Class</td>
<td>354</td>
</tr>
<tr>
<td></td>
<td>The Boolean Class</td>
<td>355</td>
</tr>
<tr>
<td></td>
<td>Review Questions</td>
<td>355</td>
</tr>
<tr>
<td>8.4</td>
<td>The String Class</td>
<td>357</td>
</tr>
<tr>
<td></td>
<td>Immutability</td>
<td>357</td>
</tr>
<tr>
<td></td>
<td>Creating and Initializing Strings</td>
<td>357</td>
</tr>
<tr>
<td></td>
<td>The CharSequence Interface</td>
<td>360</td>
</tr>
<tr>
<td></td>
<td>Reading Characters from a String</td>
<td>361</td>
</tr>
<tr>
<td></td>
<td>Comparing Strings</td>
<td>363</td>
</tr>
<tr>
<td></td>
<td>Character Case in a String</td>
<td>364</td>
</tr>
<tr>
<td></td>
<td>Concatenation of Strings</td>
<td>364</td>
</tr>
<tr>
<td></td>
<td>Joining of CharSequence Objects</td>
<td>365</td>
</tr>
<tr>
<td></td>
<td>Searching for Characters and Substrings</td>
<td>367</td>
</tr>
<tr>
<td></td>
<td>Extracting Substrings</td>
<td>369</td>
</tr>
<tr>
<td></td>
<td>Converting Primitive Values and Objects to Strings</td>
<td>369</td>
</tr>
<tr>
<td></td>
<td>Formatted Strings</td>
<td>370</td>
</tr>
<tr>
<td></td>
<td>Review Questions</td>
<td>371</td>
</tr>
<tr>
<td>8.5</td>
<td>The StringBuilder and StringBuffer Classes</td>
<td>374</td>
</tr>
<tr>
<td></td>
<td>Thread-Safety</td>
<td>374</td>
</tr>
<tr>
<td></td>
<td>Mutability</td>
<td>374</td>
</tr>
<tr>
<td></td>
<td>Constructing String Builders</td>
<td>374</td>
</tr>
<tr>
<td></td>
<td>Reading and Changing Characters in String Builders</td>
<td>375</td>
</tr>
<tr>
<td></td>
<td>Constructing Strings from String Builders</td>
<td>375</td>
</tr>
<tr>
<td></td>
<td>Appending, Inserting, and Deleting Characters in String Builders</td>
<td>376</td>
</tr>
<tr>
<td></td>
<td>Controlling StringBuilder Capacity</td>
<td>378</td>
</tr>
<tr>
<td></td>
<td>Review Questions</td>
<td>379</td>
</tr>
<tr>
<td></td>
<td>Chapter Summary</td>
<td>382</td>
</tr>
<tr>
<td></td>
<td>Programming Exercises</td>
<td>382</td>
</tr>
<tr>
<td>9</td>
<td>Object Lifetime</td>
<td>383</td>
</tr>
<tr>
<td>9.1</td>
<td>Garbage Collection</td>
<td>384</td>
</tr>
<tr>
<td>9.2</td>
<td>Reachable Objects</td>
<td>384</td>
</tr>
<tr>
<td>9.3</td>
<td>Facilitating Garbage Collection</td>
<td>386</td>
</tr>
<tr>
<td>9.4</td>
<td>Object Finalization</td>
<td>390</td>
</tr>
<tr>
<td>9.5</td>
<td>Finalizer Chaining</td>
<td>391</td>
</tr>
<tr>
<td>9.6</td>
<td>Invoking Garbage Collection Programatically</td>
<td>393</td>
</tr>
<tr>
<td></td>
<td>Review Questions</td>
<td>396</td>
</tr>
<tr>
<td>9.7</td>
<td>Initializers</td>
<td>399</td>
</tr>
<tr>
<td>9.8</td>
<td>Field Initializer Expressions</td>
<td>400</td>
</tr>
<tr>
<td></td>
<td>Declaration Order of Initializer Expressions</td>
<td>401</td>
</tr>
<tr>
<td>9.9</td>
<td>Static Initializer Blocks</td>
<td>402</td>
</tr>
</tbody>
</table>
CONTENTS

Declaration Order of Static Initializers 403
9.10 Instance Initializer Blocks 404
   Declaration Order of Instance Initializers 405
9.11 Constructing Initial Object State 406
Review Questions 409
Chapter Summary 411

10 The ArrayList<E> Class and Lambda Expressions 413
10.1 The ArrayList<E> Class 414
   Lists 414
   Declaring References and Constructing ArrayLists 415
   Modifying an ArrayList 419
   Querying an ArrayList 422
   Traversing an ArrayList 423
   Converting an ArrayList to an Array 424
   Sorting an ArrayList 425
   Arrays versus ArrayList 425
Review Questions 430
Chapter Summary 438
Programming Exercise 438

11 Date and Time 461
11.1 Basic Date and Time Concepts 462
11.2 Working with Temporal Classes 462
   Creating Temporal Objects 464
   Querying Temporal Objects 468
   Comparing Temporal Objects 470
   Creating Modified Copies of Temporal Objects 470
   Temporal Arithmetic 474
11.3 Working with Periods 476
   Creating Periods 476
   Querying Periods 478
   Creating Modified Copies of Periods 479
   More Temporal Arithmetic 479
Review Questions 483
11.4 Formatting and Parsing 486
   Default Formatters 487
Contents

- Predefined Formatters 488
- Localized Formatters 490
- Customized Formatters 495
- Review Questions 500
- Chapter Summary 502
- Programming Exercise 503

A Taking the Java SE 8 Programmer I Exam 507
  A.1 Preparing for the Exam 507
  A.2 Registering for the Exam 508
    - Contact Information 509
    - Obtaining an Exam Voucher 509
    - Signing Up for the Test 509
    - After Taking the Exam 509
  A.3 How the Exam Is Conducted 510
    - The Testing Locations 510
    - Utilizing the Allotted Time 510
    - The Exam Program 510
    - The Exam Result 511
  A.4 The Questions 511
    - Assumptions about the Exam Questions 511
    - Types of Questions Asked 512
    - Types of Answers Expected 512
    - Topics Covered by the Questions 513

B Exam Topics: Java SE 8 Programmer I 515

C Annotated Answers to Review Questions 519

D Solutions to Programming Exercises 553

E Mock Exam: Java SE 8 Programmer I 571

F Annotated Answers to Mock Exam I 605

Index 619
Figures

1.1 UML Notation for Classes ............................................... 3
1.2 UML Notation for Objects ............................................. 5
1.3 Aliases ........................................................................... 6
1.4 Class Diagram Showing Static Members of a Class ............. 8
1.5 Members of a Class .......................................................... 9
1.6 Class Diagram Depicting Inheritance Relationship .......... 10
1.7 Class Diagram Depicting Associations ............................. 12
1.8 Class Diagram Depicting Composition .............................. 13
2.1 Primitive Data Types in Java ........................................... 37
3.1 Array of Arrays ............................................................... 67
3.2 Parameter Passing: Primitive Data Values ......................... 75
3.3 Parameter Passing: Reference Values ............................... 76
3.4 Parameter Passing: Arrays ............................................... 78
4.1 Java Source File Structure ............................................... 96
4.2 Package Hierarchy ........................................................... 97
4.3 File Hierarchy ................................................................. 107
4.4 Searching for Classes ..................................................... 108
4.5 Block Scope .................................................................. 117
4.6 Public Accessibility for Members .................................... 124
4.7 Protected Accessibility for Members ................................. 127
4.8 Default Accessibility for Members ................................... 128
4.9 Private Accessibility for Members ................................... 129
5.1 Widening Primitive Conversions .................................... 144
5.2 Converting among Binary, Octal, and Hexadecimal Numbers 158
5.3 Overflow and Underflow in Floating-Point Arithmetic ....... 166
5.4 Numeric Promotion in Arithmetic Expressions ................. 170
6.1 Activity Diagram for if Statements .................................. 201
6.2 Activity Diagram for a switch Statement ......................... 204
6.3 Activity Diagram for the while Statement ......................... 213
6.4 Activity Diagram for the do-while Statement ................. 214
6.5 Activity Diagram for the for Statement ......................... 215
6.6 Enhanced for Statement ............................................... 217
6.7 Method Execution ............................................................ 231
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.8</td>
<td>Exception Propagation</td>
<td>233</td>
</tr>
<tr>
<td>6.9</td>
<td>Partial Exception Inheritance Hierarchy</td>
<td>234</td>
</tr>
<tr>
<td>6.10</td>
<td>The try-catch-finally Construct</td>
<td>239</td>
</tr>
<tr>
<td>6.11</td>
<td>Exception Handling (Scenario 1)</td>
<td>241</td>
</tr>
<tr>
<td>6.12</td>
<td>Exception Handling (Scenario 2)</td>
<td>243</td>
</tr>
<tr>
<td>6.13</td>
<td>Exception Handling (Scenario 3)</td>
<td>245</td>
</tr>
<tr>
<td>7.1</td>
<td>Inheritance Hierarchy</td>
<td>267</td>
</tr>
<tr>
<td>7.2</td>
<td>Inheritance Hierarchy for Example 7.2 and Example 7.3</td>
<td>269</td>
</tr>
<tr>
<td>7.3</td>
<td>Inheritance Hierarchies</td>
<td>295</td>
</tr>
<tr>
<td>7.4</td>
<td>Inheritance Relationships for Interface Constants</td>
<td>303</td>
</tr>
<tr>
<td>7.5</td>
<td>Reference Type Hierarchy: Arrays and Subtype Covariance</td>
<td>310</td>
</tr>
<tr>
<td>7.6</td>
<td>Type Hierarchy That Illustrates Polymorphism</td>
<td>330</td>
</tr>
<tr>
<td>7.7</td>
<td>Implementing Data Structures by Inheritance and Aggregation</td>
<td>332</td>
</tr>
<tr>
<td>8.1</td>
<td>Partial Inheritance Hierarchy in the java.lang Package</td>
<td>342</td>
</tr>
<tr>
<td>8.2</td>
<td>Converting Values among Primitive, Wrapper, and String Types</td>
<td>347</td>
</tr>
<tr>
<td>9.1</td>
<td>Memory Organization at Runtime</td>
<td>386</td>
</tr>
<tr>
<td>10.1</td>
<td>Partial ArrayList Inheritance Hierarchy</td>
<td>415</td>
</tr>
</tbody>
</table>
# Tables

1.1 Terminology for Class Members ................................................. 9
1.2 Format Specifier Examples ..................................................... 20
2.1 Keywords in Java ................................................................... 29
2.2 Reserved Literals in Java ......................................................... 29
2.3 Reserved Keywords Not Currently in Use ................................. 29
2.4 Separators in Java .................................................................. 29
2.5 Examples of Literals ............................................................... 30
2.6 Examples of Decimal, Binary, Octal, and Hexadecimal Literals .... 30
2.7 Examples of Character Literals ................................................. 33
2.8 Escape Sequences .................................................................. 33
2.9 Examples of Escape Sequence \ddd .......................................... 34
2.10 Range of Integer Values ....................................................... 38
2.11 Range of Character Values .................................................... 38
2.12 Range of Floating-Point Values ............................................. 39
2.13 Boolean Values .................................................................... 39
2.14 Summary of Primitive Data Types ........................................... 39
2.15 Default Values ..................................................................... 42
3.1 Parameter Passing by Value .................................................... 73
4.1 Accessing Members within a Class ........................................... 115
4.2 Summary of Accessibility Modifiers for Top-Level Types .......... 120
4.3 Summary of Non-Accessibility Modifiers for Classes ............... 122
4.4 Summary of Accessibility Modifiers for Members .................... 129
4.5 Summary of Non-Accessibility Modifiers for Members ............ 139
5.1 Selected Conversion Contexts and Conversion Categories ......... 147
5.2 Operator Summary ................................................................. 151
5.3 Representing Signed byte Values Using Two’s Complement ....... 155
5.4 Examples of Truncated Values ............................................... 162
5.5 Arithmetic Operators ............................................................. 164
5.6 Examples of Arithmetic Expression Evaluation ....................... 169
5.7 Arithmetic Compound Assignment Operators ....................... 172
5.8 Relational Operators ............................................................. 180
5.9 Primitive Data Value Equality Operators ............................... 181
5.10 Reference Equality Operators ................................................. 182
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.11</td>
<td>Truth Values for Boolean Logical Operators</td>
<td>184</td>
</tr>
<tr>
<td>5.12</td>
<td>Boolean Logical Compound Assignment Operators</td>
<td>185</td>
</tr>
<tr>
<td>5.13</td>
<td>Conditional Operators</td>
<td>186</td>
</tr>
<tr>
<td>5.14</td>
<td>Truth Values for Conditional Operators</td>
<td>186</td>
</tr>
<tr>
<td>5.15</td>
<td>Integer Bitwise Operators</td>
<td>189</td>
</tr>
<tr>
<td>5.16</td>
<td>Result Table for Bitwise Operators</td>
<td>190</td>
</tr>
<tr>
<td>5.17</td>
<td>Examples of Bitwise Operations</td>
<td>190</td>
</tr>
<tr>
<td>5.18</td>
<td>Bitwise Compound Assignment Operators</td>
<td>192</td>
</tr>
<tr>
<td>6.1</td>
<td>The <code>return</code> Statement</td>
<td>225</td>
</tr>
<tr>
<td>7.1</td>
<td>Overriding versus Overloading</td>
<td>274</td>
</tr>
<tr>
<td>7.2</td>
<td>Same Signature for Subclass and Superclass Method</td>
<td>276</td>
</tr>
<tr>
<td>7.3</td>
<td>Types and Values</td>
<td>309</td>
</tr>
<tr>
<td>10.1</td>
<td>Summary of Arrays versus ArrayLists</td>
<td>426</td>
</tr>
<tr>
<td>10.2</td>
<td>Selected Functional Interfaces from the <code>java.util.function</code> Package</td>
<td>444</td>
</tr>
<tr>
<td>11.1</td>
<td>Selected Common Method Prefix of the Temporal Classes</td>
<td>463</td>
</tr>
<tr>
<td>11.2</td>
<td>Selected ISO-Based Predefined Formatters for Date and Time</td>
<td>489</td>
</tr>
<tr>
<td>11.3</td>
<td>Format Styles for Date and Time</td>
<td>490</td>
</tr>
<tr>
<td>11.4</td>
<td>Combination of Format Styles and Localized Formatters</td>
<td>491</td>
</tr>
<tr>
<td>11.5</td>
<td>Selected Date/Time Pattern Letters</td>
<td>496</td>
</tr>
</tbody>
</table>
Examples

1.1 Basic Elements of a Class Declaration 4
1.2 Static Members in Class Declaration 8
1.3 Defining a Subclass 11
1.4 An Application 16
1.5 Formatted Output 21
2.1 Default Values for Fields 42
2.2 Flagging Uninitialized Local Variables of Primitive Data Types 43
2.3 Flagging Uninitialized Local Reference Variables 44
3.1 Using the this Reference 51
3.2 Namespaces 54
3.3 Using Arrays 62
3.4 Using Anonymous Arrays 64
3.5 Using Multidimensional Arrays 67
3.6 Passing Primitive Values 74
3.7 Passing Reference Values 75
3.8 Passing Arrays 77
3.9 Array Elements as Primitive Data Values 79
3.10 Array Elements as Reference Values 79
3.11 Calling a Variable Arity Method 83
3.12 Passing Program Arguments 86
3.13 Using Enums 88
4.1 Defining Packages and Using Type Import 99
4.2 Single Static Import 102
4.3 Avoiding the Interface Constant Antipattern 102
4.4 Importing Enum Constants 103
4.5 Shadowing Static Import 104
4.6 Conflict in Importing Static Method with the Same Signature 105
4.7 Class Scope 116
4.8 Accessibility Modifiers for Classes and Interfaces 118
4.9 Abstract Classes 121
4.10 Public Accessibility of Members 125
4.11 Accessing Static Members 132
4.12 Using final Modifier 134
4.13 Synchronized Methods

5.1 Evaluation Order of Operands and Arguments

5.2 Numeric Promotion in Arithmetic Expressions

5.3 Short-Circuit Evaluation Involving Conditional Operators

5.4 Bitwise Operations

6.1 Fall-Through in a switch Statement

6.2 Using break in a switch Statement

6.3 Nested switch Statement

6.4 Strings in switch Statement

6.5 Enums in switch Statement

6.6 The break Statement

6.7 Labeled break Statement

6.8 continue Statement

6.9 Labeled continue Statement

6.10 The return Statement

6.11 Method Execution

6.12 The try-catch Construct

6.13 Exception Propagation

6.14 The try-catch-finally Construct

6.15 The try-finally Construct

6.16 The finally Clause and the return Statement

6.17 Throwing Exceptions

6.18 The throws Clause

7.1 Extending Classes: Inheritance and Accessibility

7.2 Overriding, Overloading, and Hiding

7.3 Using the super Keyword

7.4 Constructor Overloading

7.5 The this() Constructor Call

7.6 The super() Constructor Call

7.7 Implementing Interfaces

7.8 Default Methods in Interfaces

7.9 Default Methods and Multiple Inheritance

7.10 Static Methods in Interfaces

7.11 Constants in Interfaces

7.12 Inheriting Constants in Interfaces

7.13 Assigning and Passing Reference Values

7.14 Choosing the Most Specific Method (Simple Case)

7.15 Overloaded Method Resolution

7.16 The instanceof and Cast Operators

7.17 Using the instanceof Operator

7.18 Polymorphism and Dynamic Method Lookup

7.19 Implementing Data Structures by Inheritance and Aggregation

8.1 Methods in the Object Class

8.2 String Representation of Integers

8.3 String Construction and Equality

8.4 Reading Characters from a String
<table>
<thead>
<tr>
<th>Examples</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1 Garbage Collection Eligibility</td>
<td>388</td>
</tr>
<tr>
<td>9.2 Using Finalizers</td>
<td>392</td>
</tr>
<tr>
<td>9.3 Invoking Garbage Collection</td>
<td>394</td>
</tr>
<tr>
<td>9.4 Initializer Expression Order and Method Calls</td>
<td>402</td>
</tr>
<tr>
<td>9.5 Static Initializers and Forward References</td>
<td>403</td>
</tr>
<tr>
<td>9.6 Instance Initializers and Forward References</td>
<td>405</td>
</tr>
<tr>
<td>9.7 Object State Construction</td>
<td>406</td>
</tr>
<tr>
<td>9.8 Initialization Anomaly under Object State Construction</td>
<td>408</td>
</tr>
<tr>
<td>10.1 Using an ArrayList</td>
<td>427</td>
</tr>
<tr>
<td>10.2 Implementing Customized Methods for Filtering an ArrayList</td>
<td>434</td>
</tr>
<tr>
<td>10.3 Implementing an Interface for Filtering an ArrayList</td>
<td>436</td>
</tr>
<tr>
<td>10.4 User-Defined Functional Interface for Filtering an ArrayList</td>
<td>439</td>
</tr>
<tr>
<td>10.5 Using the Predicate&lt;T&gt; Functional Interface for Filtering an ArrayList</td>
<td>441</td>
</tr>
<tr>
<td>10.6 Accessing Members in an Enclosing Object</td>
<td>447</td>
</tr>
<tr>
<td>10.7 Accessing Local Variables in an Enclosing Method</td>
<td>449</td>
</tr>
<tr>
<td>10.8 Filtering an ArrayList</td>
<td>452</td>
</tr>
<tr>
<td>11.1 Creating Temporal Objects</td>
<td>467</td>
</tr>
<tr>
<td>11.2 Using Temporal Objects</td>
<td>472</td>
</tr>
<tr>
<td>11.3 Temporal Arithmetic</td>
<td>475</td>
</tr>
<tr>
<td>11.4 Period-Based Loop</td>
<td>481</td>
</tr>
<tr>
<td>11.5 More Temporal Arithmetic</td>
<td>482</td>
</tr>
<tr>
<td>11.6 Using Default Date and Time Formatters</td>
<td>488</td>
</tr>
<tr>
<td>11.7 Using Predefined Format Styles with Time-Based Values</td>
<td>492</td>
</tr>
<tr>
<td>11.8 Using Predefined Format Styles with Date-Based Values</td>
<td>493</td>
</tr>
<tr>
<td>11.9 Using Predefined Format Styles with Date and Time-Based Values</td>
<td>494</td>
</tr>
<tr>
<td>11.10 Formatting and Parsing with Letter Patterns</td>
<td>497</td>
</tr>
<tr>
<td>11.11 Formatting with Date/Time Letter Patterns</td>
<td>499</td>
</tr>
</tbody>
</table>
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
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 known 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, 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:

<table>
<thead>
<tr>
<th>Programmer I Exam Objectives</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1 Exam objectives are stated clearly at the beginning of every chapter.</td>
</tr>
<tr>
<td>0.2 The number in front of the objective identifies the exam objective, as defined by Oracle, and can be found in Appendix B.</td>
</tr>
<tr>
<td>0.3 The objectives are organized into major sections, detailing the curriculum for the exam.</td>
</tr>
<tr>
<td>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.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Supplementary Objectives</th>
</tr>
</thead>
<tbody>
<tr>
<td>• 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.</td>
</tr>
<tr>
<td>• Any supplementary objective is listed as a bullet at the beginning of the chapter.</td>
</tr>
</tbody>
</table>

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 \texttt{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 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
# Declarations

## Programmer I Exam Objectives

<table>
<thead>
<tr>
<th>Objective</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1.2] Define the structure of a Java class</td>
<td>§3.1, p. 48</td>
</tr>
<tr>
<td>[4.1] Declare, instantiate, initialize, and use a one-dimensional array</td>
<td>§3.4, p. 58</td>
</tr>
<tr>
<td>[4.2] Declare, instantiate, initialize, and use multi-dimensional array</td>
<td>§3.4, p. 64</td>
</tr>
<tr>
<td>[6.1] Create methods with arguments and return values; including overloaded methods</td>
<td>§3.2, p. 49</td>
</tr>
<tr>
<td>⊳ For return statement, see §6.4, p. 224.</td>
<td></td>
</tr>
<tr>
<td>⊳ For covariant return, see §7.2, p. 273.</td>
<td></td>
</tr>
<tr>
<td>⊳ For overloaded method resolution, see §7.10, p. 316.</td>
<td></td>
</tr>
<tr>
<td>⊳ For overriding methods, see §7.2, p. 268.</td>
<td></td>
</tr>
<tr>
<td>[6.3] Create and overload constructors; including impact on default constructors</td>
<td>§3.3, p. 53</td>
</tr>
<tr>
<td>⊳ For constructor chaining, see §7.5, p. 282.</td>
<td></td>
</tr>
<tr>
<td>[6.6] Determine the effect upon object references and primitive values when they are passed into methods that change the values</td>
<td>§3.5, p. 72</td>
</tr>
<tr>
<td>⊳ For conversions in assignment and method invocation contexts, see §5.2, p. 147.</td>
<td></td>
</tr>
<tr>
<td>[7.4] Use super and this to access objects and constructors</td>
<td>§3.2, p. 50</td>
</tr>
<tr>
<td>⊳ For the super reference, see §7.4, p. 276.</td>
<td></td>
</tr>
<tr>
<td>⊳ For using this() and super(), see §7.5, p. 282.</td>
<td></td>
</tr>
</tbody>
</table>

## Supplementary Objectives

<table>
<thead>
<tr>
<th>Objective</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Sorting and searching arrays</td>
<td>§3.4, p. 68</td>
</tr>
<tr>
<td>• Declaring and calling methods with a variable arity parameter</td>
<td>§3.6, p. 81</td>
</tr>
<tr>
<td>• Passing values to a Java application as program arguments on the command line</td>
<td>§3.7, p. 85</td>
</tr>
<tr>
<td>• Basic introduction to declaring and using enum types</td>
<td>§3.8, p. 87</td>
</tr>
</tbody>
</table>
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:

\[
\text{method_modifiers return_type method_name} \\
\quad \text{(formal_parameter_list) throws_clause // Method header} \\
\{ // Method body \\
\quad \text{local_variable_declarations} \\
\quad \text{statements} \\
\}
\]

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:

\[
\text{optional_parameter_modifier type parameter_name}
\]

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:

\[
\text{void noAction() \{} \\
\}
\]

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

```java
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': Number of parameters
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 cor-
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 declara-
tion 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

```java
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:

```java
    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:

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

    // No constructors
    //...
}
```

In the code, the class Greenhouse does not specify any constructors:

```java
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):

```java
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).

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

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).

```java
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 extraLight = new Light();   // (2) Call of explicit default constructor
}
```

```java
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.

```java
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?

```java
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?

```java
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?
```java
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:

\[
elment\_type[] \ array\_name;
\]

or

\[
elment\_type \ array\_name[];
\]

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.

```java
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 \textit{array\_size} is 0; in other words zero-length arrays can be constructed in Java. If the array size is negative, a \texttt{NegativeArraySizeException} is thrown at runtime.

Given the declarations

\begin{verbatim}
int anIntArray[], oneInteger;
Pizza[] mediumPizzas, largePizzas;
\end{verbatim}

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

\begin{verbatim}
anIntArray    = new int[10];          // array for 10 integers
mediumPizzas = new Pizza[5];         // array of 5 pizzas
largePizzas  = new Pizza[3];         // array of 3 pizzas
\end{verbatim}

The array declaration and construction can be combined.

\begin{verbatim}
element\_type\_1[] array\_name = new element\_type\_2[array\_size];
\end{verbatim}

In the preceding syntax, the array type \texttt{element\_type\_1[]} must be \texttt{assignable} to the array type \texttt{element\_type\_2[]} (§7.7, p. 309). When the array is constructed, all of its elements are initialized to the default value for \texttt{element\_type}. 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 \texttt{anIntArray} get the value 0, and all elements of the array \texttt{mediumPizzas} get the value \texttt{null} when the arrays are constructed.

\begin{verbatim}
int[] anIntArray = new int[10];                  // Default element value: 0
Pizza[] mediumPizzas = new Pizza[5];             // Default element value: null
\end{verbatim}

The value of the field \texttt{length} in each array is set to the number of elements specified during the construction of the array; for example, \texttt{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.

\section*{Initializing an Array}

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

\begin{verbatim}
element\_type[] array\_name = \{ array\_initialize\_list \};
\end{verbatim}

This form of initialization applies to fields as well as to local arrays. The \texttt{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.

\begin{verbatim}
int[] anIntArray = \{13, 49, 267, 15, 215\};
\end{verbatim}

In this declaration statement, the variable \texttt{anIntArray} is declared as a reference to an array of \texttt{ints}. 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.

```java
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:

```java
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`.

```java
// 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 `i`th 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 \textit{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:

\begin{verbatim}
char letter = "AHA".toCharArray()[1];     // 'H'
\end{verbatim}

The array operator [] is used to declare array types (\texttt{Topping[]}), specify the array size (\texttt{new Topping[3]}), and access array elements (\texttt{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 \texttt{for(;;)} loop at (3) in the \texttt{main()} method initializes the local array \texttt{trialArray} declared at (2) five times with pseudo-random numbers (from 0.0 to 100.0), by calling the method \texttt{randomize()} declared at (5). The minimum value in the array is found by calling the method \texttt{findMinimum()} declared at (6), and is stored in the array \texttt{storeMinimum} declared at (1). Both of these methods also use a \texttt{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 \texttt{for(():)} loop at (4) in the \texttt{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.

\begin{itemize}
\item \textbf{Example 3.3} \textit{Using Arrays}
\end{itemize}

\begin{verbatim}
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);
            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;
    }
}
\end{verbatim}
```java
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>[] array_name = new <element_type> [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:

```
n<element_type>[] array_name = {array_initialize_list}
n<new element_type>[] {array_initialize_list}
n<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.

```java
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**

```java
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:

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

or

```java
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:

```java
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.

```java
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 ≤ i < 4`. Each element in the `i`th row, `mXnArray[i]`, is accessed by `mXnArray[i][j]`, where `0 ≤ 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 ≤ 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:

```java
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:

```java
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 (0.00). 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
executed \((m \times n) \times (m \times n)\) times, 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**

```java
class MultiArrays {
    public static void main(String[] args) {
        // Declare and construct the M X N matrix.
        int[][] mXnArray = {
            {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.

```java
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:

```java
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.

```java
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**.

```java
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:

```java
// 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:

```java
// 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:

```java
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:

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

```java
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 `String`S 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][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?

```java
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 `null` 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?

```java
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.
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:

```java
int i = 4;
```

the method call

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

is effectively the same as

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

and not the same as

```java
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

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

with the following code:

```java
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

```java
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
3.5: PARAMETER PASSING

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

```java
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**

```java
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

```java
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

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*

```java
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*

```java
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:

```java
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 Strings.

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:

```java
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:

```java
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:

```java
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

```java
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,
                                                                     //                   Integer.valueOf(year)}
                                        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:

```bash
$ 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
> java VarargsDemo To arg or not to arg
Type: [Ljava.lang.Object;  No. of elements: 0
Element values:
Type: [Ljava.lang.Object;  No. of elements: 1
Element values: 13
Type: [Ljava.lang.Object;  No. of elements: 2
Element values: 13 August
Type: [Ljava.lang.Object;  No. of elements: 3
Element values: 13 August 2009
Type: [Ljava.lang.Object;  No. of elements: 3
Element values: 13 August 2009
Type: [Ljava.lang.Object;  No. of elements: 1
Element values: [Ljava.lang.Object;@1eed786
Type: [Ljava.lang.Object;  No. of elements: 1
Element values: [Ljava.lang.String;@187aeca
Type: [Ljava.lang.String;  No. of elements: 6
Element values: To arg or not to arg
Type: [Ljava.lang.Object;  No. of elements: 1
Element values: [Ljava.lang.String;@187aeca
Type: [Ljava.lang.String;  No. of elements: 6
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:

```
fexiPrint((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.
3.7: THE `main()` METHOD

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:

```java
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):

```java
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):

```java
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):

```java
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:

```java
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 \texttt{main()} method can also be overloaded like any other method (§3.2, p. 52). The JVM ensures that the \texttt{main()} method having the previously mentioned method header is the starting point of program execution.

\section*{Program Arguments}

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

\begin{verbatim}
>java Colors red green blue
\end{verbatim}

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

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

In Example 3.12, the three arguments \texttt{red}, \texttt{green}, and \texttt{blue} can be accessed in the \texttt{main()} method of the \texttt{Colors} class as \texttt{args[0]}, \texttt{args[1]}, and \texttt{args[2]}, respectively. The total number of arguments is given by the field \texttt{length} of the \texttt{String} array \texttt{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 \texttt{String} elements is created and passed to the \texttt{main()} method. Thus the reference value of the formal parameter in the \texttt{main()} method is never \texttt{null}.

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

\begin{example}[Passing Program Arguments]

\begin{verbatim}
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.");
    }
}
\end{verbatim}

Running the program:

\begin{verbatim}
>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.
\end{verbatim}

\end{example}
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:

```java
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:

```java
public enum MachineState { // Enum header
    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:

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

System.out.println("Current machine state: " + state // (4) Printing the enum name.
);


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

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:

```java
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?

```java
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;
    }
}
```
3.8: ENUMERATED TYPES

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?

```java
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?

```java
public class ParameterUse {
    public 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:

```java
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
Symbols

- 169
-- 176
^ 184, 189
^= 185
_ 32
; 50
: 110
! 184
!= 181, 342
?: 194
. 7, 72, 97, 108
... 81, 85
' 32, 33
" 34
[] 59, 61, 195
{} 50, 60, 117
@FunctionalInterface 442, 443
@Override 270
@param 49, 56
@return 225
@throws 253
* 100, 163, 167
* 172
/ 167
/* and */ 35
/** and */ 36
// 35
//= 172
\ 33
& 184, 189
& & 186
&= 185
% 167, 168
%= 172
+= 169, 174
++ 176
+= 172
< 180
<= 180
er 414, 415, 416
-= 172
= 158
== 181, 342, 351, 359
-> 195, 439, 444
> 180
>= 180
| 184, 189
|= 185
|| 186
~ 189

A
ability interfaces
    see marker interfaces
abrupt method completion 232
absolute adjusters 470
abstract
    classes 120
    interfaces 290
    methods 136, 291, 442
abstract 120, 136, 290, 291, 442
abstract method declarations 442
    in interfaces 290, 291
abstraction 2, 10
accessibility 7, 17, 114
    default 118, 127
    members 114, 120, 123
    modifiers 118
    package 118
private 128
protected 126
public 124
UML notation 124
accessibility modifiers 48, 53
activation frame 384
   see method execution 230
actual parameter 72
actual parameter list 72, 315
adding to class 264
additive operators 169
aggregation 10, 12, 267
   hierarchy 267
   versus inheritance 331
aliases 6, 75, 182, 183
   see also references
ambiguous call 316
analyzing program code 512
and operator 189
annotations
   @Override 270
anonymous arrays 63, 66
   [] 63
anonymous classes 436
anonymous functions 439
API (application programming interface) 22
apostrophe 33
application 16
architecture neutral 23
argument
   see actual parameter
arguments to main method 85
arithmetic compound assignment
   operators 172
ArithmeticException 236
arity 81
array creation expression 59, 63
array initializer 60, 63, 66
array store check 311
array types
   see arrays
ArrayIndexOutOfBoundsException 61, 236
ArrayList 366, 414
   add collection 419
   add element 417, 419
   boxing 421
capacity 416
clear list 420
   comparison with arrays 425
   constructing 415
   constructors 418
   converting to array 424
   element search 423
   element type 415
   filtering 434
   import 415
   inheritance hierarchy 415
   initial capacity 417
   insertion order 414
   list of lists 417
   membership test 422
   modifying 419
   nested lists 417
   object value equality 422
   open range-view operations 414
   ordered 414
   position-based access 415
   positional index 422
   positional order 414
   positional retrieve 422
   querying 422
   references 415
   remove collection 420
   remove element 419
   replace element 419
   size 416, 422
   sorting 425
   subtype relationship 418
   textual representation 417
   traversing 423
   trim to size 420
   type-safety 416, 417, 418
   unchecked conversion warning 416
   zero-based index 414
arrays 58, 342
   [] 59, 61
   {} 60
anonymous 63, 66
array creation expression 59
array initializer list 60, 63
array initializer 60, 66
array name 59
array size 60
array store check 311
ArrayIndexOutOfBoundsException 61
bounds 61
construction 59
declarations 59
default initialization 59, 60
dynamic 415
element access expression 61
element default value 310
element type 59
elements 58, 61
index 58
index expression 61
initialization 60, 65
iterating over 217
length 58
multidimensional 63, 65
objects 342
ragged 65
reference 59, 62, 311
searching 69
sorting 68
subtype covariance 309
traverse an array 62
using 61
arrays of arrays 59, 65
multidimensional 65
ArrayStoreException 311, 418, 442
arrow -> 195, 439, 444
ASCII 32, 38
AssertionError 237
assignable 147, 314
assignment compatible 148, 314, 416
assignment conversions 147
assignment operator 5
assignments
arithmetic compound operators 172
bitwise 192
cascading 159
compound operators 185, 192
expression statement 159
implicit narrowing 160
multiple 159
numeric conversions 160
operator 151, 158
primitive values 159
references 159
widening reference conversions 267
association 12
aggregation 267
composition 267
realization 296
associativity 152
atomic values 13
attributes see properties
autoboxing 68, 348
for(:) statement 218
AutoCloseable 387
automatic garbage collection 6, 384
automatic variables see local variables
B
backslash 33
backspace 33
base 30, 349, 352
base class 264
basic for statement 215
Before Current Era (BCE) 464
behavior 433
behavior parameterization 434, 441
binary
numeric promotion 150
operators 151
binary search
arrays 69
bit mask 190
bit patterns 154
bitwise
and operator 189
assignment 192
compound and 189
compound assignment 192
operators 189
or operator 189
xor 189
bitwise AND
& 189
bitwise complement
~ 189
bitwise exclusive OR
^ 189
bitwise OR
| 189
bitwise XOR
^ 189
blank final variable 80, 134
blocks 49, 50, 117
scope 117, 448
try 240
boilerplate code 436
Boolean
condition 200
Boolean wrapper class 355
booleans 37, 39
casting 149
expressions 180
literals 32
boxing conversions 145, 146
break statement 205, 206, 221
BS see backspace
building abstractions 10
byte 30, 38
bytecode 16, 23

class hierarchy
  see inheritance hierarchy
class inheritance
  see implementation inheritance
class method 10
class modifiers 48
class path 107
  absolute pathnames 110
  entries order 110
  entry-separator character 110
  fully qualified package name 109
  path-separator character 110
  relative pathnames 110
  search in a named package 109
  searching for classes 107
  whitespace 110
class search path
  see class path
class variable 10
ClassCastException 236, 321
classes
  abstract 120
  accessibility 118
  adding to 264
  base 264
  body 48
  child 264
  cohesion 335
  concrete 121, 122, 436
  constructors 53, 282
  coupling 336
  declarations 48, 96
  definitions 2, 5
  derived 264
  diagram 8, 9
  encapsulation 335
  extending 122, 264
  final 122
  final vs. abstract 122
  fully qualified name 107
  fully qualified package name 98
generalized 266
  grouping 97
  header 48
  implementing interfaces 291
  initialization 409
  instance members 48
  instances 4
  members 7
  methods 132
  modifiers 120
  name 97
normal 121
object 342
parent 264
runtime 343
scope 114
searching for 107
specialized 266
static members 48
subclass 10, 264
superclass 10, 264
variables 132
wrappers 342, 346
ClassLoader class 342
ClassNotFoundException 235
CLASSPATH environment variable
see class path-
-classpath option
see class path
clauses
   catch 240
   extends 264
   finally 240, 245
   implements 291
   throws 251
cleaning up 386
clean-up code 245
client 7, 16
Cloneable interface 343
CloneNotSupportedException 343
cloning objects 343
code optimizations 134
code reuse 23, 264, 334
CodeRanch 508
cohesion 335
   coincidental 335
   functional 335
   high 335
coincidental cohesion 335
Collection 414
collections 414
   as single entity 414
   elements 414
   ordered 414
   sorting 414
command 17
   java 17
   javac 17
command line 17, 86
command prompt 17
comments 35
communication 7, 72
Comparable interface 350, 363, 376, 425
comparing objects 342
comparing strings 363
comparison 180
compilation unit 98
compiling Java source code 17
complement
~ 189
completes abruptly
   see exception propagation 232
composite object 10
composition 12, 267
compound object 50
compound statement 50
concatenation of strings 364
concatenation operator 174
crude concrete classes 436
crude concrete method 134
ConcurrentModificationException 424
condition
   Boolean 200
   expressions 200
   conditional 180
   and 186
   operators 186, 194
   or 186
   statements 200
   conditional expressions 194
      associativity 195
      nested 195
      precedence 194
      short-circuit evaluation 194
      side effects 194
   conditions 180
   connecting punctuation character 28
   const 29
   constant declarations 290
   constant expression 147, 160, 161, 176
   constant field values
      case labels 208
      constant string expressions 208
      constant values 30, 133
   constant variable 161
   constants 302
   constituent objects 10
   constituents 12
   constructing array 59
   constructor chaining 283, 287, 406
   constructors 3, 53, 282
      accessibility 124
      accessibility modifier 53
      body 53
      chaining 283, 287
   class name 53
<table>
<thead>
<tr>
<th>Declaration</th>
<th>Default</th>
<th>Header</th>
<th>Implicit Default</th>
<th>Local Declarations</th>
<th>No-Argument</th>
<th>Non-Zero Argument</th>
<th>Overloading</th>
<th>Superclass Constructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
<td>54</td>
<td>53</td>
<td>54</td>
<td>53, 54, 283, 287</td>
<td>55, 287, 288</td>
<td>55, 287, 288</td>
<td>56</td>
<td>54</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constructs</th>
<th>High-Level</th>
<th>Loops See Iteration Statements</th>
<th>Container</th>
<th>Contains Collections</th>
<th>Continues Statement</th>
<th>Contract</th>
<th>Control Flow</th>
<th>Break</th>
<th>Continue</th>
<th>Do-While</th>
<th>For(;;)</th>
<th>For(;)</th>
<th>If</th>
<th>If-Else</th>
<th>Iteration</th>
<th>Loops See Iteration Statements</th>
<th>Return</th>
<th>Statements</th>
<th>Switch</th>
<th>Throw</th>
<th>Transfer Statements</th>
<th>While</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Control Transfer</th>
<th>Conversion Categories</th>
<th>Conversion Contexts</th>
<th>Conversions</th>
<th>Assignment</th>
<th>Contexts</th>
<th>Identity</th>
<th>Implicit Narrowing</th>
<th>Method Invocation</th>
<th>Narrowing Reference</th>
<th>Number Systems</th>
<th>Numeric Promotions</th>
<th>Parameters</th>
<th>Reference Casting</th>
<th>String Concatenation</th>
<th>To Strings</th>
<th>Truncation</th>
<th>Type-Safe</th>
<th>Unsafe Casts</th>
<th>Widening Reference</th>
<th>Converting Number Systems</th>
<th>Converting Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>219</td>
<td>147</td>
<td>147</td>
<td>144, 311</td>
<td>147</td>
<td>147</td>
<td>172</td>
<td>173</td>
<td>148</td>
<td>320</td>
<td>157</td>
<td>149</td>
<td>73</td>
<td>320</td>
<td>175</td>
<td>369</td>
<td>348, 349, 350, 352, 353, 355, 369</td>
<td>215</td>
<td>336</td>
<td>336</td>
<td>321</td>
<td>267, 320</td>
</tr>
</tbody>
</table>
declarations
arrays 59, 196
classes 48, 96
interfaces 96
local 50
main method 85
methods 49
multidimensional arrays 63
packages 96, 98
statements 50
variable arity method 81
declared type 268, 274, 275, 315
declared-type parameters 445
declaring see declarations
decoupling 330
decrement operator 176
deep copying 343
default
accessibility 118, 124, 127
constructor 54
exception handler 232
method 297
values 42, 400, 406
default 297
label 204
method 297
default constructor 54
default method 297, 442, 443
multiple inheritance 298
overriding 298
default package 98
deferred execution 451
definitions
inherence 296
interfaces 290
delegating requests 334
derived class 264
destination directory 106
destination stream 18
destroying objects 390
diagrams
class 3
object 5
see also UML
diamond operator (∈) 416
dictionary order 363
distributed 23
divide-and-conquer algorithm 69
dividend 168
division
floating-point 167
integer 167
division operator
/ 167
divisor 168
documentation 35
documentation comment 35, 36
tags 36
documenting see documentation
dot 97
double 31, 39
double quote 33
do-while statement 214
downcasting 145
duplicating objects 343
duration class 476
time-based 476
dynamic 23
dynamic arrays 415
dynamic binding
see dynamic method lookup
dynamic method lookup 277, 329, 330
dynamic type 268, 274, 275

E
effectively final 448
element type 59, 415
elements 58, 414
eligible for garbage collection 385
dashes 81
else clause matching 203
embedded applications 22
empty statement 50
empty string 358
encapsulation 22, 97, 335
encapsulation of implementation 334
ends with characters 368
enhanced for loop 213
enterprise applications 22
denum constant 87
symbolic names 87
values 87
denum types 87, 103, 209, 303
declaring 87
finalization 391
named constants 87
natural order 90
ordinal value 90, 209
switch expression 204
using 88
enumerated types
see enum types
EOFException 235
equality 181, 342
  equals method 183, 342
  object value 183
  objects 183
  primitive values 181
  reference values 182
  equals method 183, 342
Error 237
escape sequences 33
evaluation order 152, 187
  arithmetic expressions 164
evaluation short-circuits 187
exam 507
  multiple-choice 513
  program 510
  questions 511
  registration 508
  result 511
  testing locations 510
  voucher 509
exam objectives
  OCAJP8 515
Exception class 236
exception handler 230
  see also exceptions
exception handling
  advantages 254
exceptions 230, 239
  customized 238
  default handler 232
  handler 230
  ignored 390
  propagation 230
  situations 235
  throw 249
  throwing see throwing exceptions
  thrown by JVM 235
  thrown by method 49
  thrown programmatically 235
  throws 251
types 233
  uncaught 232
  unchecked 237
exchanging information 72
explicit
  garbage collection 393
  explicit traversal 452
  exponent 31
expression statements 50, 159, 177, 216, 217, 446
expressions 205
  actual parameters 72
boolean 180
  case labels 205
  conditional 194
deterministic evaluation 150
label 205
return 224
  statements 50
  throw 249
extending
  classes 264
  interfaces 294
extends clause
  see extending
extensions
  .class 16
  .java 16
external libraries 403
extracting substrings 369
F
fall-through 204, 205
false literal 32
FF see form feed
field declarations 48
field hiding 275
field initialization 406
fields 2
file name 96
file path 105
  separator character 105
filtering 434
final
  classes 122
  members 133
  parameters 80
finalization 385
finalization mechanism 385
finalize method 343, 390
finalizer chaining 391
finalizer see finalize method
finally clause 240, 245
fixed arity method 81
fixed arity method call 84
float 31, 39
floating-point 37
  double 39
  float 39
  literals 31
floating-point arithmetic 165
  strictfp 166
floating-point data types 31
floating-point division 167
floating-point remainder 169
flow control see control flow
for(;;) statement 215
  backward 216
  forward 215
  traverse array 62
for(;) statement 217
  traverse array 62
for-each loop 213
form feed 33, 35
formal parameters 49, 53, 72, 117, 315
  modifier 49
  name 49
  type 49
formal type parameter 290
format specifications 18, 370
format specifier 19
format styles 486, 490
FormatStyle enum type 486, 490
formatted output 18
  format specifier 19
  formatted string 370
formatting 35, 462, 486
forward reference 400, 401, 403, 405, 406
fractional signed numbers 37
fully qualified class name 107
fully qualified package name 97, 98, 100
fully qualified type name 97, 101
function 451
  function type 450
  functional cohesion 335
functional interface 438
  @FunctionalInterface 442, 443
  abstract method 442
  function type 450
  functional method 442
  general-purpose 443
  generic 441
  Predicate<T> 440, 451
  primitive values 444
  target type 450
  see also lambda expressions
functional method 442
  functional programming 24
  functionality 433
  functional-style programming 433

G
  garbage collection 387, 389, 390, 393
  automatic 384
facilitate 387
  general abstractions 266
  general loops 215
  generalization 10
  generalized classes 266
  generic method 423
  generic type 414
  goto 29, 220
  grammar rules 28
  grouping 97

H
  handles see references
  has-a relationship 267
  hash code 52, 343
  hash tables 52
  heap 384
  hiding internals 335
  high cohesion 335
  high-performance 24
  horizontal tab 33
  hotspots 24
  HT see horizontal tab

I
  IDE (integrated development environment) 508
  identifiers 28
    predefined 29
    reserved 29
    variable 40
  identity conversion 146, 172
  identity of object 5
  IEEE 754-1985 38
  if block 200
  if-else statement 201
  ignored exceptions 390
  IllegalArgumentException 236, 495
  immediate superclass 285
  immutable 462
  immutable objects 346, 357
  immutable strings 357
  implementation inheritance 264
  implementations 2, 266, 335
    inheritance hierarchy 122
    implementing interfaces 291
    implements clause 291
    implicit
    inheritance 264
narrowing conversions 173
implicit default constructor 54
import
declaration 100
see also static import
single-type-import declaration 100
statement 96
type-import-on-demand declaration 100
importing
enum constants 103
reference types 99
static members 101
increment operator 176
index 58
index expression 61
IndexOutOfBoundsException 361, 369, 375, 376, 419, 422
individual array elements 61
inequality 181
see also equality
inferred-type parameters 445
infinite loop 217
infinity 165, 349
negative 165
positive 165
information hiding 335
inheritance 10, 267
hierarchy 266
supertype–subtype relationship 267
initial capacity 417
initial state of object 406
initialization
arrays 60, 65
code 60
default values 42
for statement 215
objects 5
references 41
variables 41
initializer 399
declaration-before-reading rule 401
static 400, 401, 405, 409
initializer block
instance 404
static 402
initializer expression 400
initializers
non-static block 48
non-static field 48
static block 48
static field 48
initializing see initialization
insertion order 414
insertion point 69
instance
members 9, 48
methods 9, 49, 50
variable initialization 42
variables 9, 44
see also object
instance initializer block 404
instance methods 6
instance variables 6, 406
instanceof operator 195, 320, 321
instantiation 4
int 30, 38
integer arithmetic 165
integer bitwise operators 189
integer constant expressions 148
integer data types 30
integer division 167
integer remainder operation 168
integers 38
and operator 189
byte 38
complement 189
data types 38
int 38
literals 30
long 38
or operator 189
representation 154
short 38
types 38
xor 189
integral types 37, 38, 144
interface constant antipattern 102
interfaces 290
abstract 290
abstract methods 291
accessibility 118
body 290
constants 302
declarations 96
default methods 297
extending 294
header 290
implementing 291
initialization 409
marker 291
realization 296
references 296
static methods 300
subinterfaces 294
superinterfaces 294
UML 295
variables 302
internal traversal 452
interned strings 358, 359
interned values 351
interpackage accessibility 335
interpreter 17
intraclass dependencies 336
invocation stack
  see JVM stack
invoker 224
invoking garbage collection 393
IOException 235
is-a relationship 266, 267, 334
ISO Latin-1 32, 38
ISO standard 486, 487, 488
Iterable interface 366, 424
iteration 215
iteration statements 213
  next iteration 223
  termination 213, 222
iterators 414, 424
J
Java
  Native Interface see JNI
java 17
Java bytecode 16
Java Collections Framework 414
Java compiler 17
Java Development Kit (JDK) 21
Java ecosystem 21
Java EE (Enterprise Edition) 22
Java ME (Micro Edition) 22
Java Native Interface see JNI
Java Platforms 22
Java Runtime Environment (JRE) 22
Java SE (Standard Edition) 22
Java Virtual Machine see JVM
java.time package 462
java.time.format package 462
java.util package 414
java.util.function<T> package 444
javac 17
Javadoc comment 35
  @param tag 49, 56
  @return tag 225
  @throws tag 253
javac utility 36
JDK 17, 508
JNI 137
just-in-time (JIT) 23
JVM 17, 22, 384, 393
JVM stack 230, 384
K
key 69
keywords 29
  abstract 120, 136, 291, 442
  boolean 39
  break statement 221
  byte 38
  case 203
  catch 240
  char 38
  class 48, 290
  const 29
  continue 223
  default 204, 297
  do 214
  double 39
  else 201
  extends 264
  final 80, 122, 133
  finally 245
  float 39
  for 215, 217
  if 200
  implements 291
  import 100
  instanceof 195, 320, 321
  int 38
  interface 290
  long 38
  native 137
  new see new operator
  null 149, 183, 320
  package 98
  private 128
  protected 126
  public 124
  reserved words 29
  return 224
  short 38
  static 17, 101, 132, 300, 402
  strictfp 166
  super 54, 272, 276, 285, 299
  switch 203
  synchronized 136
  this 50
  throw 249
INDEX

throws 251
transient 138
try 240
unused words 29
void 17, 347
volatile 139
while 213, 214

labeled break statement 222
labels 220, 222
break 222
case 203
default 204
expressions 205
labeled statement 220
switch statement 203
lambda body 439, 444, 445
lambda expressions 433, 438, 444
access class members 446
anonymous functions 439
arrow -> 439, 444
as values 439
blocks
  scope 448
declared-type parameters 445
default 204
expression 445
expression statements 446
function 451
inferred-type parameters 445
lambda body 439, 444, 445
lambda parameters 445
void return 445
parameter list 439, 444
single expression 439, 445
statement block 439, 446
target type 450
target typing 451
type checking 450
variable capture 449
void return 445
lambda parameters 445
late binding
  see dynamic method lookup
least significant bit 155
left associativity 152
legal assignments 314
length method 361
letter pattern 495
lexical scope

see blocks: scope
lexical tokens 28
lexicographical ordering 363, 425
LF see linefeed
libraries 403
lifetime 385
  see scope 44
line separator 19
line terminator 35
linear implementation inheritance 266
linefeed 33
LinkageError 237
LinkedList 417
List 414
lists
  see ArrayList
literals 30
  boolean 32
  character 32
default type 30, 31
double 31
  escape sequences 33
false 32
float 31
floating-point 31
integer 30
null 30
predefined 29
prefix 30
quoting 32
scientific notation 31
string 34
suffix 30, 31
true 32
litmus test
design by inheritance 266
local 43
  chaining of constructors 283, 406
  variables 44, 117
local declarations 49, 50
local variables 53
  blocks
  scope 448
LocalDate class
  see temporal objects
LocalDateTime class
  see temporal objects
locale 364, 490, 492
localizing information 335
LocalTime class
  see temporal objects
locations
see class path
logical AND
& 184
logical complement
! 184
logical exclusive OR
^ 184
logical inclusive OR
| 184
logical XOR
^ 184
long 30, 38
suffix 30
loop body 213, 215
loop condition 213, 215
loops see iteration statements
loose coupling 336
loss of precision 144

M
magnitude 144
main method 17, 18, 85
arguments 86
modifiers 85
manifest constant 134
marker interfaces 291
Math class 52
MAX_VALUE constant 351
member declarations 48, 290
members 3, 114
access 50
accessibility 120, 123
default values 42
final 133
inheritance 264
instance 48
modified 264
modifiers 131
of objects 7
scope 114
short-hand 51
static 7, 48, 132
terminology 9
variables see fields
memory management 384
memory organization 384
message
receiver 7
method call 7, 49, 72
chaining 376, 378
fixed arity 84
variable arity 84
method chaining 471, 474, 479
method declaration 48
method header 136, 137
method invocation conversions 148, 315
method modifiers 49
method overloading 52, 273
method overriding 268, 273, 407
method signature 49, 269
method type 450
methods 3
@override 270
abstract 136, 291, 442
abstract method declarations 291
accessibility 49
ambiguous call 316
automatic variables see local variables
behavior 433
blocks 49
body 49, 117
call chaining 72
call see method call
calling variable arity method 82
classified 365
clone 343
classification 134
declaration 49, 72
default 297
dynamic lookup 330
equals 183, 342
exceptions 49
final 134
finalize 343, 390
fixed arity 81
functional 442
getClass 343
header 49
implementation 136
invocation see method call
local declarations 49
local variables
main see main method
method invocation conversions 315
method type 450
modifiers 49
most specific 316, 422
name 72
native 137, 251
objects 50
overloaded resolution 316
overloading see method overloading
overriding see method overriding
overriding vs. overloading 273
parameters 49
recursive 237
return 224
return value 49
signature 49, 52, 273
static 132, 300
synchronized 136
termination 224
throws clause 251
toString 343
variable arity 81
MIN_VALUE constant 351
minimizing overhead 386
mobile applications 22
modifiers
abstract 120, 136, 291
accessibility 118, 123
classes 120
default 297
final 133
members 131
native 137
static 132, 300
strictfp 166
synchronized 136
transient 138
volatile 139
Month enum type 465
most specific method 316, 422
multicore 441
multicore architectures 24
multidimensional arrays 63, 65
multiple assignments 159
multiple catch clauses 239
multiple implementation inheritance 290
multiple inheritance 298
multiple interface inheritance 290
multiple-line comment 35
multiplication operator
* 167
multiplicative operators 167
multithreaded 24
mutable character sequences 374
mutually comparable 68, 69
mutually exclusive
actions 202
MVC 335

N
ame 28

named constants 134
namespaces 53
NaN 166, 349
narrower range 144
narrowing conversions
primitive 144
reference 145
narrowing reference conversions 320
native libraries 403
native methods 137, 251
header 137
natural ordering 68, 69, 425
negative zero 165
nested lists 417
nested loops 66
new operator 5, 53, 59, 195, 406
newline see linefeed
NL see newline
no-argument constructor 53, 54, 283, 287
non-associativity 151
non-static code 48
see non-static context 48
non-static context 48
non-static field 9
non-static field initializers 48
non-static initializer block 48
non-varargs call
see fixed arity call
void return 445
non-zero argument constructor 55, 287, 288
normal class 121
normal execution 232
notifying threads 344
null reference 30
casting 320
null reference literal
casting 149
equality comparison 183
nulling references 387
NullPointerException 236
NumberFormatException 236
Number class 351
number systems
base 30
converting 157
decimal 30
hexadecimal 30
octal 30
radix 30
NumberFormatException 236, 347, 348
numeric promotions 149
assignment 160
INDEX

binary 150
unary 149
numeric wrapper classes 351
numerical literals
  using underscore 32

O

object 4
Object class 266, 342
object hierarchy 267
object references 4, 40
object state 6, 53, 77, 406
object-oriented design 334
  cohesion 335
object-oriented paradigm 22
object-oriented programming 2
objects 13
  aggregate 12
  alive 385
  arrays 58
  callee 72
  caller 72
  Class class 343
  cleaning up 386
  cloning 343
  communication 72
  comparing 342
  composite 385
  constituent 12, 385
  constructing 406
  contract 335
  decoupling 330
  destroying 390
  eligible 387
  equality 183, 342
  exchanging information 72
  finalization 385
  garbage collection 384
  identity 5
  immutable 346
  implementation 335
  initial state 406
  initialization 5, 53
  initializer block 404
  internals 335
  lifetime 385
  members 7
  methods 50
  Object class 342
  persistence 138
  reachable 384, 385
  resurrection 385
  services 335
  state 133
  see object state
  value equality 183

OCAJP 507
  exam objectives 515
  exam question assumptions 511

OCPJP 507
  one-dimensional arrays 59
  operands 148
  evaluation order 152
  operations 2
  operators 150
    - 163, 169
    -- 176
    ^ 184, 189
    &= 185, 192
    ! 184
    != 181, 182
    ? : 194
    . 7, 97
    [] 61, 195
    * 163, 167
    *= 172
    / 163, 167
    /= 172
    & 184, 189
    && 186
    %= 185, 192
    % 163, 167, 168
    %= 172
    + 163, 169, 174
    ++ 176
    += 172
    < 180
    <= 180
    => 172
    = 158
    == 181, 182
    -> 195
    > 180
    >= 180
    | 184, 189
    |= 185, 192
    || 186
    ~ 189
  arithmetic compound assignment 172
  assignment 151, 158
  associativity 150
  binary 151
  bitwise 189
boolean 180, 181, 184
cast 151
comparisons 180
compound assignment 185, 192
conditional 186, 194
decrement 176
dot 7
equality 181
execution order 152
floating-point 165
floating-point division 167
floating-point remainder 169
increment 176
instanceof 195, 320, 321
integer 189
integer arithmetic 165
integer division 167
integer remainder 168
logical 184
multiplicative 167
new see new operator
overflow 165
overloaded 164, 167
postfix 151
precedence 150
relational 180
short-circuited 186
string concatenation 174
ternary 151
unary 150, 151, 167
unary - 167
unary + 167
optimizations 24
or operator 189
Oracle University 509
ordinal value 90, 209
OutOfMemoryException 395
output 18
overflow 155, 165
overloaded 164
overloaded method resolution 316
overloading
    constructors 56
    method resolution 316
    methods 52, 273
overloading vs. overriding 273
overriding 253
    equals 183
    finalizers 390
    methods 268, 273
toString 175
overriding methods
covariant return 273
overriding vs. overloading 273
ownership 12

P
package accessibility 118, 124
package directory 106
package statement 96, 98
packages 97
    accessibility see package accessibility
declaration 96
definition 98
destination directory 106
hierarchy 97
java.lang 342
members 97
naming scheme 98
package directory 106
running code from 106
short-hand 100
statement see package statement
subpackages 97
unnamed 98
using 99
palindromes 382, 434
parallel code 441
parameter
    variable arity 81
parameter list 439, 444
parameter list see formal parameters
parameter passing
    by value 72
    variable arity 81
parameters 49
    actual 72
array elements 78
finally 80
fixed arity 81
formal see formal parameters
implicit 50
main method 86
passing 72
primitives 73
program 86
references 75
this 50
variable arity 81
parent class 264
parentheses 150
parseType method 352
parsing 462, 486
INDEX

parsing numeric values 352
partial implementation 293
pass by value 72
passing
  parameters 72
  references 75
  variable arity parameter 81
paths
  see class path
path-separator character 110
pattern letters 486, 495, 496
Pearson VUE 509
performance 24
period 462, 476
  creating 476
  date-based 476
  equality 478
  get methods 478
  immutable 476
  normalization 479
  parsing 477
  period-based loop 481
  plus/minus methods 479
  querying 478
  temporal arithmetic 479
  textual representation 477
  thread-safe 476
  with methods 479
Period class
  see period
persistent objects 138
polymorphism 311, 329, 334
portability 23
positional order 414
positive zero 165
postfix operators 151
precedence rules 151
precision 160
predefined identifiers 29
predefined literals 29
predicate 436
Predicate<T> 440, 451
prefix 30
  0 30
  0x 30
primitive data types
  see primitive types
primitive types 13, 144
  autoboxing 348
  unboxing 350
  see also primitive values
primitive values
  assignment 159
  equality 181
  passing 73
  printing values 18
  private 11
  private members 128
  process of elimination 510
program
  application 16
  arguments 86
  compiling 17
  formatting 35
  running 17
program arguments 86
program output 18
programming to an interface 417
proleptic year 464
promotion 149
properties 2
  see also class members
protected 11
protected members 126
public 17
public members 124
punctuators 29

Q
quotation mark 33, 34
quotient 168

R
radix
  prefix 30
  see base 349
ragged arrays 65
range
  character values 38
  floating-point values 39
  integer values 38
range of date-based values 464
range of time-based values 464
ranking criteria 414
realization 296
reclaiming memory 384
reducing complexity 335
reference types 41, 267
  classes 48
  enum types 87
  reference values 4
reference variables 40
references 4, 9, 40, 41, 72
abstract types 121
aliases 75, 183
array 59, 62, 311
assignment 159
casting 149, 320
dangling 384
declared type 268
downcasting 145
dynamic type 268
equality 182
field 385
interface type 296
local 384
narrowing conversions 145
nu11 see nu11 reference
passing 75
reachable 384, 385
super 276
this 50
upcasting 145
widening conversions 145
relational operators 180
relative adjusters 474
reliability 24
remainder 168
remainder operator 168
remove whitespace 369
replacing characters 367
reserved identifiers 29
reserved keywords 29
const 29
goto 220
reserved literals
false 32
nu11 see nu11 reference
ture 32
resources 387
resurrecting objects 385, 391
return statement 224
@return tag 225
return type
covariant
return value 7
reuse of code 264, 334
right associativity 152
rightmost bit 155
ripple effect 334
robustness 24, 254
role relationship 334
root
see inheritance hierarchy
running a Java application 17
runtime
bounds checking 61
runtime checks 148, 418
Runtime class 342, 393
runtime class 343
runtime environment 384
runtime stack
see JVM stack
RuntimeException 236
S
scientific notation 31
scope 114
block 117
catch clause 244
class 114
disjoint 118
searching
arrays 69
searching in string 367
secure 24
SecurityManager class 342
selection statements 200
semantic definition 28
semicolon 50
separators 29, 151
serialization 138
services 335
shadowing 446
shallow copying 343
short 30, 38
short-circuit 186
evaluation 187
signature 52, 273
simple
assignment operator 158
if 200
statement 50
simple type name 97
simplicity 23
single expression 439
single implementation inheritance 266, 290, 296
single quote (’) 32, 33
single static import 101
single-line comment 3, 35
skeletal source file 96
sorting arrays 68
source
  file 15, 98
  file name 96
  file structure 96
spaces 35
special character values 33
specialization 10
specialized classes 266
stack 3
stack frame
  see method execution
stack trace 232, 235
  see method execution
StackOverflowError 237
standard error stream 235
standard out 18
starts with characters 368
state
  see object state
statement block 439, 446
statements 50
  break 221
  compound 50
  conditional 200
  continue 223
  control flow 50, 200
  control transfer 219
  declaration flow 50, 200
  declarations 50
  do-while 214
  empty 50
  expression 50, 177
  for(;;) 215
  for(:) 217
  if 200
  if-else 201
  iteration 213
  labeled 220
  return 224
  selection 200
  simple 50
  simple if 200
  switch 203
  throw 249
  transfer 219
  try 240
  while 213
static
  members see static members
  methods 7, 10, 49
  variable initialization 42
  variables see static variables
static 101, 132, 300
static code
  see static context 48
static context 48
static field 10
static field initializers 48
static import 101
  conflicts 104
  on demand 101
  shadow static members 103
  single static import 101
static initializer block 48, 137, 402
static keyword 402
static members 7, 9, 10, 48
static type
  see declared type
static variables 7, 10, 44
storing objects 138
strictfp 166
string builders 176
  appending 376
  capacity 374, 378
  constructing 374
  deleting 376
  differences with strings 376
  individual characters 375
  inserting 376
  joining 366
  length 375
  thread-safety 374
String class
  see strings
string conversion 146, 175
string conversions 370
string literal pool 358
  interned 358
string literals 357
  case labels 208
  hash value 208
  interned 358
StringBuffer class 374
  see string builders
  thread-safe 374
StringBuilder class 374, 434
  see string builders
strings
  appending 376
  buffers 374
  builders 374
  capacity 378
  changing case 364
  compareTo 363
  comparing 363
concatenation 174, 364
concatenation operator + 176
constructing 374
contains 368
conversions 370
convert to character array 361
copying characters 361
creating 357
deleting 376
differences with string builders 376
empty 358
ends with 368
equals 363
extracting substrings 369
finding index 367
formatted 370
ignoring case in comparison 363
immutable 357
individual characters 361, 375
initializing 357
inserting 376
interned 358
joining 365, 366
length 361, 375
lexicographical ordering 363
literals 34, 357
mutable 374
read character at index 361
replacing 367
searching 367
starts with 368
string literal pool 358
substrings 369
trimming 369
strongly typed language 148
subclass 10, 11, 264
subinterface 294
subpackages 97
subsequence 361
substring searching 367
substrings 367, 369
subtype covariance 309, 310
subtype relationship 418
subtypes 293
subtype–supertype relationship 145
suffix
D 31
F 31
L 30
super 299
construct 285
keyword 272, 276
reference 276
superclass 10, 11, 264
superclass constructor 54
superclass–subclass relationship 266
superinterfaces 294
supertypes 293
supertype–subtype relationship 267
supplementary characters 357
suppressed exceptions 235
switch statement 203
break 205, 206
default clause 204
enum types 209
using strings 208
synchronized
methods 136
textually legal 28
System
out 18
System class 342
system clock 466

T
TAB see horizontal tab
tabs 35
tabulators 35
tags 36
target type 450
target typing 451
telephone directory order 363
temporal arithmetic 474, 479
temporal objects
before/after methods 469
combining date and time 466
common method prefix 463
comparing 470
creating with factory methods 464
date 462
date units 474
date-based values 462
date-time 462
formatting 486
get methods 468
immutable 462
method naming convention 463
parsing 486
plus/minus methods 474
querying 468
range of date-based values 464
range of time-based values 464
temporal arithmetic 474, 479, 480
temporal values 464
thread-safe 462
time 462
time units 474
time-based values 462
with methods 470
temporal values 464
TemporalAmount interface 479
terminating loops 221
ternary conditional expressions
  see also conditional expressions 194
ternary conditional operator 192
  151, 194
textual representation 343
this
  reference 50
this() constructor call 282, 406
ThreadDeath 237
threads 24, 342, 384
dead 232
exception propagation 232
JVM stack 285
live 384
notifying 344
synchronization 136
waiting 344
thread-safe 357, 374, 415, 462
throw statement 280
Throwble 233, 342
throw-and-catch paradigm 230
throwing exceptions 230
throws clause 251
time
  see temporal objects
time units 474
time-based values 462
tokens 28
toString method 343, 349
transfer statements 219
transient variables 138
transitive relation 267
trim method 369
ture literal 32
truth-values 32, 39
try block 240
try-catch-finally construct 238
two's complement 154
type
  declared 274
dynamic 274
type cast 148
type cast expression 320
type checking 450
type declarations 96
type hierarchy 145, 267
type import
  see import
type parameter 290, 414, 441
types
  boolean 37, 39
  byte 30, 38
  casting 148
  char 38
classes see classes
  comparing 321
  compatibility 148
double 39
exceptions 233
float 39
floating-point 37, 38
int 30, 38
integers 38
integral types 37
interface 290
long 30, 38
parsing 352
short 30, 38
wrappers 346
  see also classes
type-safe 315
type-safety 416, 417, 418
typeValue method 350, 352

U

UML 2
  accessibility 124
  aggregation 12
  associations 12
classes 3
  composition 12
  inheritance 10
  see also diagrams
unary arithmetic operators 167
unary numeric promotion 149
unary operators 150, 151
unboxing 350
do-while statement 214
for(;;) statement 215
for(:) statement 218
if statement 200
if-else statement 202
switch statement 204
while statement 213
unboxing conversions 145, 146
uncaught exceptions 232
unchangeable variables 134
unchecked conversion warning 416
unchecked conversions 146
unchecked exceptions 237
unchecked warnings 145
underflow 155, 165
uni-catch clause 239
Unicode 32, 38, 354, 357, 363, 434
Unified Modeling Language see UML
unreachable code 244
unsafe casts 321
unsigned integer 353
UnsupportedTemporalTypeException 480, 490
unused keywords 29
upcasting 145
update expression 215
using arrays 61
using packages 99
using variables 41
UTF-16 357
  supplementary characters 357

V
valueOf method 348, 349, 369
values 155
  constants 30
  overflow 155
  underflow 155
  wrap-around 155
  see also variables
varargs 81
variable arity call 84
variable arity method 81
variable arity parameter 81
variable capture 449
variable declarations 41, 117
variable initialization 8, 43
variables 4, 41
  blank final 80, 134
  constant values 133
  default values 42
  effectively final 448
  final 133
  identifiers 40
  in interfaces 302
  initialization see variable initialization
  lifetime 44
  local 117
  parameters 49, 72
  reference variable 41
  references 41
  static 7
  storing 138
  transient 138
  volatile 139
virtual method invocation
  see dynamic method lookup
VirtualMachineError 237
void 17, 224, 347
void return 445
Void wrapper class 346
volatile variables 139
voucher 509

W
waiting threads 344
while statement 213
whitespace 35, 369
whole–part relationship 267
widening conversions
  primitive 144
  references 145
widening reference conversions 267, 320
wider range 144
withers 470
wrapper classes 38, 342, 343, 346, 347
  interned values 351
  wrapper type 146, 164, 177

X
xor 189

Z
zero
  negative 165
  positive 165
zero-based index 414