Java

The Java® Tutorial

A Short Course on the Basics

Fifth Edition

Sharon Biocca Zakhour, Sowmya Kannan, Raymond Gallardo

ORACLE

The Java® Tutorial

Fifth Edition

This page intentionally left blank

The Java® Tutorial

A Short Course on the Basics

Fifth Edition

Sharon Biocca Zakhour
Sowmya Kannan
Raymond Gallardo

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Cataloging-in-Publication Data is on file with the Library of Congress.

ISBN-13: 978-0-13-276169-7
ISBN-10: 0-13-276169-6

Text printed in the United States on recycled paper at Edwards Brothers Malloy in Ann Arbor, Michigan
First printing, March 2013

Contents

Preface xxi
About the Authors xxv

Chapter1 Getting Started 1
The Java Technology Phenomenon 1
The Java Programming Language 2
The Java Platform 2
What Can Java Technology Do? 4
How Will Java Technology Change My Life? 4
The "Hello World!" Application 5
"Hello World!" for the NetBeans IDE 6
"Hello World!" for Microsoft Windows 14
"Hello World!" for Solaris and Linux 18
A Closer Look at the "Hello World!" Application 22
Source Code Comments 23
The HelloWorldApp Class Definition 23
The main Method 24
Common Problems (and Their Solutions) 25
Compiler Problems 25
Runtime Problems 27
Questions and Exercises: Getting Started 29

	Questions	29
	Exercises	29
	Answers	30
Chapter 2	Object-Oriented Programming Concepts	31
	What Is an Object?	32
	What Is a Class?	34
	What Is Inheritance?	36
	What Is an Interface?	37
	What Is a Package?	38
	Questions and Exercises: Object-Oriented Programming Concepts	38
	Questions	38
	Exercises	39
	Answers	39
Chapter 3	Language Basics	41
	Variables	42
	Naming	43
	Primitive Data Types	44
	Arrays	49
	Summary of Variables	54
	Questions and Exercises: Variables	54
	Operators	55
	Assignment, Arithmetic, and Unary Operators	56
	Equality, Relational, and Conditional Operators	59
	Bitwise and Bit Shift Operators	62
	Summary of Operators	62
	Questions and Exercises: Operators	64
	Expressions, Statements, and Blocks	65
	Expressions	65
	Statements	66
	Blocks	67
	Questions and Exercises: Expressions, Statements, and Blocks	67
	Control Flow Statements	68
	The if-then and if-then-else Statements	68
	The switch Statement	70
	The while and do-while Statements	75
	The for Statement	76
	Branching Statements	78
	Summary of Control Flow Statements	82
	Questions and Exercises: Control Flow Statements	82

Chapter 4 Classes and Objects 85
Classes 86
Declaring Classes 87
Declaring Member Variables 88
Defining Methods 90
Providing Constructors for Your Classes 92
Passing Information to a Method or a Constructor 94
Objects 98
Creating Objects 99
Using Objects 103
More on Classes 106
Returning a Value from a Method 106
Using the this Keyword 108
Controlling Access to Members of a Class 110
Understanding Instance and Class Members 111
Initializing Fields 115
Summary of Creating and Using Classes and Objects 117
Questions and Exercises: Classes 118
Questions and Exercises: Objects 119
Nested Classes 120
Why Use Nested Classes? 121
Static Nested Classes 121
Inner Classes 122
Inner Class Example 123
Summary of Nested Classes 125
Questions and Exercises: Nested Classes 125
Enum Types 126
Question and Exercises: Enum Types 129
Annotations 129
Documentation 130
Annotations Used by the Compiler 132
Annotation Processing 133
Questions and Exercises: Annotations 133
Chapter 5 Interfaces and Inheritance 135
Interfaces 135
Interfaces in Java 136
Interfaces as APIs 137
Interfaces and Multiple Inheritance 137
Defining an Interface 138
Implementing an Interface 139
Using an Interface as a Type 141

- Rewriting Interfaces 142
- Summary of Interfaces 142
- Questions and Exercises: Interfaces 142

Inheritance 143
- The Java Platform Class Hierarchy 144
- An Example of Inheritance 144
- What You Can Do in a Subclass 146
- Private Members in a Superclass 146
- Casting Objects 146
- Overriding and Hiding Methods 148
- Polymorphism 150
- Hiding Fields 152
- Using the Keyword super 152
- Object as a Superclass 154
- Writing Final Classes and Methods 158
- Abstract Methods and Classes 158
- Summary of Inheritance 161
- Questions and Exercises: Inheritance 162

Chapter 6 Generics 163

Why Use Generics? 164

Generic Types 164
- A Simple Box Class 164
- A Generic Version of the Box Class 165
- Type Parameter Naming Conventions 165
- Invoking and Instantiating a Generic Type 166
- The Diamond 166
- Multiple Type Parameters 167
- Parameterized Types 168
- Raw Types 168

Generic Methods 170

Bounded Type Parameters 171
- Multiple Bounds 172
- Generic Methods and Bounded Type Parameters 173

Generics, Inheritance, and Subtypes 173
- Generic Classes and Subtyping 175

Type Inference 176
- Type Inference and Generic Methods 176
- Type Inference and Instantiation of Generic Classes 177
- Type Inference and Generic Constructors of Generic and Nongeneric Classes 178

Wildcards 179

Upper-Bounded Wildcards 179
Unbounded Wildcards 180
Lower-Bounded Wildcards 182
Wildcards and Subtyping 182
Wildcard Capture and Helper Methods 184
Guidelines for Wildcard Use 187
Type Erasure 188
Erasure of Generic Types 188
Erasure of Generic Methods 190
Effects of Type Erasure and Bridge Methods 191
Nonreifiable Types 193
Restrictions on Generics 196
Cannot Instantiate Generic Types with Primitive Types 196
Cannot Create Instances of Type Parameters 197
Cannot Declare Static Fields Whose Types Are Type Parameters 197
Cannot Use Casts or instanceof with Parameterized Types 198
Cannot Create Arrays of Parameterized Types 199
Cannot Create, Catch, or Throw Objects of Parameterized Types 199
Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type 200
Questions and Exercises: Generics 200
Questions 200
Exercises 202
Answers 202
Chapter 7 Packages 203
Creating a Package 205
Naming a Package 206
Naming Conventions 206
Using Package Members 207
Referring to a Package Member by Its Qualified Name 207
Importing a Package Member 208
Importing an Entire Package 208
Apparent Hierarchies of Packages 209
Name Ambiguities 209
The Static Import Statement 210
Managing Source and Class Files 211
Setting the CLASSPATH System Variable 213
Summary of Packages 213
Questions and Exercises: Creating and Using Packages 214

Questions 214
Exercises 214
Answers 215
Chapter 8 Numbers and Strings 217
Numbers 217
The Numbers Classes 218
Formatting Numeric Print Output 220
Beyond Basic Arithmetic 224
Autoboxing and Unboxing 230
Summary of Numbers 232
Questions and Exercises: Numbers 233
Characters 234
Escape Sequences 235
Strings 236
Creating Strings 236
String Length 237
Concatenating Strings 238
Creating Format Strings 239
Converting between Numbers and Strings 240
Manipulating Characters in a String 242
Comparing Strings and Portions of Strings 247
The StringBuilder Class 247
Summary of Characters and Strings 253
Questions and Exercises: Characters and Strings 254
Chapter 9 Exceptions 257
What Is an Exception? 258
The Catch or Specify Requirement 258
The Three Kinds of Exceptions 259
Bypassing Catch or Specify 260
Catching and Handling Exceptions 261
The try Block 262
The catch Blocks 263
The finally Block 264
The try-with-resources Statement 266
Putting It All Together 269
Specifying the Exceptions Thrown by a Method 272
How to Throw Exceptions 273
The throw Statement 274
Throwable Class and Its Subclasses 274
Error Class 275
Exception Class 275

	Chained Exceptions	276
	Creating Exception Classes	277
	Unchecked Exceptions: The Controversy	279
	Advantages of Exceptions	280
	Advantage 1: Separating Error-Handling Code from "Regular" Code	280
	Advantage 2: Propagating Errors Up the Call Stack	282
	Advantage 3: Grouping and Differentiating Error Types	283
	Summary of Exceptions	285
	Questions and Exercises: Exceptions	285
	Questions	285
	Exercises	286
	Answers	287
Chapter 10	Basic I/O and NIO.2	289
	I/O Streams	289
	Byte Streams	291
	Character Streams	293
	Buffered Streams	295
	Scanning and Formatting	296
	I/O from the Command Line	302
	Data Streams	305
	Object Streams	307
	File I/O (Featuring NIO.2)	309
	What Is a Path? (And Other File System Facts)	309
	The Path Class	312
	File Operations	320
	Checking a File or Directory	324
	Deleting a File or Directory	325
	Copying a File or Directory	325
	Moving a File or Directory	326
	Managing Metadata (File and File Store Attributes)	327
	Reading, Writing, and Creating Files	335
	Random Access Files	342
	Creating and Reading Directories	343
	Links, Symbolic or Otherwise	347
	Walking the File Tree	349
	Finding Files	354
	Watching a Directory for Changes	357
	Other Useful Methods	363
	Legacy File I/O Code	365
	Summary of Basic I/O and NIO.2	366

Questions and Exercises: Basic I/O and NIO.2 368
- Questions 368
- Exercises 369
- Answers 369

Chapter 11 Collections 371

Introduction to Collections 372
- What Is a Collections Framework? 372
- Benefits of the Java Collections Framework 373

Interfaces 374
- The Collection Interface 376
- Traversing Collections 377
- Collection Interface Bulk Operations 378
- Collection Interface Array Operations 379
- The Set Interface 380
- The List Interface 384
- The Queue Interface 394
- The Map Interface 397
- Object Ordering 405
- The SortedSet Interface 412
- The SortedMap Interface 415
- Summary of Interfaces 417
- Questions and Exercises: Interfaces 417

Implementations 418
- Set Implementations 421
- List Implementations 422
- Map Implementations 424
- Queue Implementations 425
- Wrapper Implementations 427
- Convenience Implementations 429
- Summary of Implementations 431
- Questions and Exercises: Implementations 432

Algorithms 432
- Sorting 433
- Shuffling 435
- Routine Data Manipulation 435
- Searching 436
- Composition 436
- Finding Extreme Values 437

Custom Collection Implementations 437
- Reasons to Write an Implementation 437
- How to Write a Custom Implementation 438

	Interoperability	440
	Compatibility	441
	API Design	443
Chapter 12	Concurrency	445
	Processes and Threads	446
	Processes	446
	Threads	446
	Thread Objects	447
	Defining and Starting a Thread	447
	Pausing Execution with Sleep	448
	Interrupts	449
	Joins	451
	The SimpleThreads Example	451
	Synchronization	453
	Thread Interference	453
	Memory Consistency Errors	454
	Synchronized Methods	455
	Intrinsic Locks and Synchronization	457
	Atomic Access	459
	Liveness	459
	Deadlock	460
	Starvation and Livelock	461
	Guarded Blocks	461
	Immutable Objects	465
	A Synchronized Class Example	466
	A Strategy for Defining Immutable Objects	467
	High-Level Concurrency Objects	469
	Lock Objects	470
	Executors	472
	Concurrent Collections	477
	Atomic Variables	478
	Concurrent Random Numbers	480
	Questions and Exercises: Concurrency	480
	Question	480
	Exercises	480
	Answers	481
Chapter 13	Regular Expressions	483
	Introduction	484
	What Are Regular Expressions?	484
	How Are Regular Expressions Represented in This Package?	485

Test Harness 485
String Literals 486
  Metacharacters 487
Character Classes 488
  Simple Classes 488
Predefined Character Classes 492
Quantifiers 495
  Zero-Length Matches 495
  Capturing Groups and Character Classes with Quantifiers 498
  Differences among Greedy, Reluctant, and Possessive Quantifiers 499
Capturing Groups 500
  Numbering 501
  Backreferences 502
Boundary Matchers 502
Methods of the Pattern Class 504
  Creating a Pattern with Flags 504
  Embedded Flag Expressions 506
  Using the matches(String,CharSequence) Method 507
  Using the split(String) Method 507
  Other Utility Methods 508
  Pattern Method Equivalents in java.lang.String 509
Methods of the Matcher Class 509
  Index Methods 510
  Study Methods 510
  Replacement Methods 510
  Using the start and end Methods 511
  Using the matches and lookingAt Methods 512
  Using replaceFirst(String) and replaceAll(String) 513
  Using appendReplacement(StringBuffer,String) and appendTail(StringBuffer) 514
  Matcher Method Equivalents in java.lang.String 515
Methods of the PatternSyntaxException Class 515
Unicode Support 517
  Matching a Specific Code Point 517
  Unicode Character Properties 518
Additional Resources 518
Questions and Exercises: Regular Expressions 519
  Questions 519

	Exercise	519
	Answers	519
Chapter 14	The Platform Environment	521
	Configuration Utilities	521
	Properties	522
	Command-Line Arguments	526
	Environment Variables	527
	Other Configuration Utilities	529
	System Utilities	529
	Command-Line I/O Objects	530
	System Properties	530
	The Security Manager	533
	Miscellaneous Methods in System	535
	PATH and CLASSPATH	535
	Update the PATH Environment Variable (Microsoft Windows)	536
	Update the PATH Variable (Solaris and Linux)	538
	Checking the CLASSPATH Variable (All Platforms)	539
	Questions and Exercises: The Platform Environment	540
	Question	540
	Exercise	540
	Answers	540
Chapter 15	Packaging Programs in JAR Files	541
	Using JAR Files: The Basics	542
	Creating a JAR File	542
	Viewing the Contents of a JAR File	546
	Extracting the Contents of a JAR File	548
	Updating a JAR File	550
	Running JAR Packaged Software	552
	Working with Manifest Files: The Basics	553
	Understanding the Default Manifest	554
	Modifying a Manifest File	554
	Setting an Application's Entry Point	555
	Adding Classes to the JAR File's Class Path	557
	Setting Package Version Information	558
	Sealing Packages within a JAR File	559
	Sealing JAR Files	560
	Signing and Verifying JAR Files	560
	Understanding Signing and Verification	561
	Signing JAR Files	564

Verifying Signed JAR Files 566
Using JAR-Related APIs 567
An Example: The JarRunner Application 567
Questions: Packaging Programs in JAR Files 573
Questions 573
Answers 573
Chapter 16 Java Web Start 575
Developing a Java Web Start Application 576
Creating the Top JPanel Class 577
Creating the Application 578
The Benefits of Separating Core Functionality from the Final Deployment Mechanism 578
Retrieving Resources 579
Deploying a Java Web Start Application 579
Setting Up a Web Server 581
Displaying a Customized Loading Progress Indicator 581
Developing a Customized Loading Progress Indicator 582
Specifying a Customized Loading Progress Indicator for a Java Web Start Application 584
Running a Java Web Start Application 585
Running a Java Web Start Application from a Browser 585
Running a Java Web Start Application from the Java Cache Viewer 585
Running a Java Web Start Application from the Desktop 586
Java Web Start and Security 586
Dynamic Downloading of HTTPS Certificates 587
Common Java Web Start Problems 587
"My Browser Shows the JNLP File for My Application as Plain Text" 588
"When I Try to Launch My JNLP File, I Get an Error" 588
Questions and Exercises: Java Web Start 588
Questions 588
Exercises 589
Answers 589
Chapter 17 Applets 591
Getting Started with Applets 591
Defining an Applet Subclass 592
Methods for Milestones 593
Life Cycle of an Applet 594
Applet's Execution Environment 596
Developing an Applet 597
Deploying an Applet 600

Doing More with Applets 603
Finding and Loading Data Files 603
Defining and Using Applet Parameters 604
Displaying Short Status Strings 606
Displaying Documents in the Browser 607
Invoking JavaScript Code from an Applet 608
Invoking Applet Methods from JavaScript Code 611
Handling Initialization Status with Event Handlers 614
Manipulating DOM of an Applet's Web Page 616
Displaying a Customized Loading Progress Indicator 618
Developing a Customized Loading Progress Indicator 618
Specifying a Loading Progress Indicator for an Applet 621
Integrating the Loading Progress Indicator with an Applet's User Interface 622
Writing Diagnostics to Standard Output and Error Streams 623
Developing Draggable Applets 623
Communicating with Other Applets 626
Working with a Server-Side Application 628
What Applets Can and Cannot Do 630
Solving Common Applet Problems 632
"My Applet Does Not Display" 632
"The Java Console Log Displays java.lang.ClassNotFoundException" 633
"I Was Able to Build the Code Once, but Now the Build Fails Even Though There Are No Compilation Errors" 633
"When I Try to Load a Web Page That Has an Applet, My Browser Redirects Me to www.java.com without Any Warning" 633
"I Fixed Some Bugs and Rebuilt My Applet's Source Code, but When I Reload the Applet's Web Page, My Fixes Are Not Showing Up" 633
Questions and Exercises: Applets 633
Questions 633
Exercises 634
Answers 634
Chapter 18 Doing More with Java Rich Internet Applications 635
Setting Trusted Arguments and Secure Properties 635
System Properties 637
JNLP API 638
Accessing the Client Using JNLP API 639

	Cookies	643
	Types of Cookies	643
	Cookie Support in RIAs	643
	Accessing Cookies	644
	Customizing the Loading Experience	646
	Security in RIAs	646
	Questions and Exercises: Doing More with Java Rich Internet Applications	647
	Questions	647
	Exercise	648
	Answers	648
Chapter 19	Deployment in Depth	649
	Deployment Toolkit	649
	Location of Deployment Toolkit Script	650
	Deploying an Applet	650
	Deploying a Java Web Start Application	655
	Checking the Client JRE Software Version	658
	Java Network Launch Protocol	659
	Structure of the JNLP File	659
	Deployment Best Practices	667
	Reducing the Download Time	667
	Avoiding Unnecessary Update Checks	669
	Signing JAR Files Only When Necessary	671
	Ensuring the Presence of the JRE Software	672
	Questions and Exercises: Deployment In Depth	673
	Questions	673
	Exercise	673
	Answers	674
Appendix	Preparing for Java Programming Language Certification	675
	Programmer Level I Exam	675
	Section 1: Java Basics	675
	Section 2: Working with Java Data Types	676
	Section 3: Using Operators and Decision Constructs	677
	Section 4: Creating and Using Arrays	678
	Section 5: Using Loop Constructs	678
	Section 6: Working with Methods and Encapsulation	678
	Section 7: Working with Inheritance	679
	Section 8: Handling Exceptions	680

Programmer Level II Exam 680
- Section 1: Java Class Design 681
- Section 2: Advanced Class Design 681
- Section 3: Object-Oriented Design Principles 682
- Section 4: Generics and Collections 682
- Section 5: String Processing 684
- Section 6: Exceptions and Assertions 684
- Section 7: Java I/O Fundamentals 685
- Section 8: Java File I/O (NIO.2) 685
- Section 9: Building Database Applications with JDBC 686
- Section 10: Threads 686
- Section 11: Concurrency 687
- Section 12: Localization 687

Java SE 7 Upgrade Exam 688
- Section 1: Language Enhancements 688
- Section 2: Design Patterns 689
- Section 3: Database Applications with JDBC 689
- Section 4: Concurrency 690
- Section 5: Localization 691
- Section 6: Java File I/O (NIO.2) 692

Index 693

This page intentionally left blank

Preface

Since the acquisition of Sun Microsystems by Oracle Corporation in early 2010, it has been an exciting time for the Java language. As evidenced by the activities of the Java Community Process program, the Java language continues to evolve. The publication of this fifth edition of *The Java® Tutorial* reflects release 7 of the Java Platform Standard Edition (Java SE) and references the Application Programming Interface (API) of that release.

This edition introduces new features added to the platform since the publication of the fourth edition (under release 6), such as a section on NIO.2, the new file I/O API, and information on migrating legacy code to the new API. The deployment coverage has been expanded with new chapters on "Doing More with Java Rich Internet Applications" (Chapter 18) and "Deployment in Depth" (Chapter 19). A section on the Fork/Join feature has been added to the "Concurrency" chapter (Chapter 12). Information reflecting Project Coin developments has been added where appropriate, including the new `try`-with-resources statement, the ability to catch more than one type of exception with a single exception handler, support for binary literals, and diamond syntax, which results in cleaner generics code.

In addition to coverage of new features, previous chapters have been rewritten to include new information. For example, "Generics" (Chapter 6), "Java Web Start" (Chapter 16), and "Applets" (Chapter 17) have been updated. The appendix for the Java Certification exam has also been completely replaced.

If you plan to take one of the Java SE 7 certification exams, this book can help. The appendix, "Preparing for Java Programming Language Certification," lists the three exams that are available, detailing the items covered on each exam, cross-referenced to places in the book where you can find more information about each topic. Note that this is one source, among others, that you will want to use to prepare for your exam.

All the material has been thoroughly reviewed by members of Oracle Java engineering to ensure that the information is accurate and up to date. This book is based on the online tutorial hosted on Oracle's web site at the following URL:

```
http://docs.oracle.com/javase/tutorial
```

The information in this book, often referred to as "the core tutorial," is essential for most beginning to intermediate programmers. Once you have mastered this material, you can explore the rest of the Java platform documentation on the web site. If you are interested in developing sophisticated Rich Internet Applications (RIAs), check out JavaFX, the new cutting-edge Java graphical user interface (GUI) toolkit. As of the release of Java SE 7 update 5, you automatically get the JavaFX Software Development Kit (SDK) when you download the JDK. To learn more, see the JavaFX documentation at the following URL:

```
http://docs.oracle.com/javafx
```

As always, our goal is to create an easy-to-read, practical programmers' guide to help you learn how to use the rich environment provided by Java to build applications, applets, and components. Go forth and program!

Who Should Read This Book?

This book is geared toward both novice and experienced programmers.

- *New programmers* can benefit most from reading the book from beginning to end, including the step-by-step instructions for compiling and running your first program in Chapter 1, "Getting Started."
- *Programmers experienced with procedural languages* such as C may want to start with the material on object-oriented concepts and features of the Java programming language.
- *Experienced programmers* may want to jump feet first into the more advanced topics, such as generics, concurrency, or deployment.

This book contains information to address the learning needs of programmers with various levels of experience.

How to Use This Book

This book is designed so you can read it straight through or skip around from topic to topic. The information is presented in a logical order, and forward references are avoided wherever possible.

The examples in this tutorial are compiled against the JDK 7 release. *You need to download this release (or later) in order to compile and run most examples.*

Some material referenced in this book is available online (e.g., the downloadable examples, the solutions to the questions and exercises, the JDK 7 guides, and the API specification). You will see footnotes like the following:

```
7/docs/api/java/lang/Class.html
```

and

```
tutorial/java/generics/examples/BoxDemo.java
```

The Java Tutorials are also available in two e-book formats:

- mobi e-book files for Kindle
- ePub e-book files for iPad, Nook, and other devices that support the ePub format

Each e-book contains a single trail, which is equivalent to several related chapters in this book. You can download the e-books via the link "In Book Form" on the home page for the Java Tutorials:

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

We welcome feedback on this edition. To contact us, please see the tutorial feedback page:

```
http://docs.oracle.com/javase/feedback.html
```

This book would not be what it is without the Oracle Java engineering team who tirelessly reviews the technical content of our writing. For this edition of the book, we especially want to thank Alan Bateman, Alex Buckley, Calvin Cheung, Maurizio Cimadamore, Joe Darcy, Andy Herrick, Stuart Marks, Igor Nekrestyanov, Thomas Ng, Nam Nguyen, and Daniel Smith.

Illustrators Jordan Douglas and Dawn Tyler created our professional graphics, quickly and efficiently. Devika Gollapudi provided invaluable assistance by capturing our screenshots for publication.

Editors Deborah Owens and Susan Shepard provided careful and thorough copy edits of our JDK 7 work.

Thanks for the support of our team: Bhavesh Patel, Devika Gollapudi, and Santhosh La.

Last but not least, thanks for the support of our management: Alan Sommerer, who saw us through the acquisition and beyond; Barbara Ramsey; Sophia Mikulinsky; and Sowmya Kannan, recently elevated as manager of our team.

About the Authors

Sharon Biocca Zakhour is a principal technical writer on staff at Oracle Corporation and was formerly at Sun Microsystems. She has contributed to Java SE platform documentation for more than twelve years, including *The Java™ Tutorial, Fourth Edition* (Addison-Wesley, 2007), and *The JFC Swing Tutorial, Second Edition* (Addison-Wesley, 2004). She graduated from the University of California, Berkeley, with a B.A. in computer science and has worked as a programmer, developer support engineer, and technical writer for thirty years.

Sowmya Kannan was previously a principal technical writer at Oracle Corporation and Sun Microsystems. She is currently the manager of the Java SE documentation team and has more than fifteen years of experience in the design, development, and documentation of the Java platform, Java-based middleware, and web applications.

Raymond Gallardo is a technical writer for Oracle Corporation. His previous engagements include college instructor, technical writer for IBM, and bicycle courier. He obtained his B.Sc. in computer science and English from the University of Toronto and his M.A. in creative writing from the City College of New York.

This page intentionally left blank

6

Generics

Chapter Contents

Why Use Generics? 164
Generic Types 164
Generic Methods 170
Bounded Type Parameters 171
Generics, Inheritance, and Subtypes 173
Type Inference 176
Wildcards 179
Type Erasure 188
Restrictions on Generics 196
Questions and Exercises: Generics 200

In any nontrivial software project, bugs are simply a fact of life. Careful planning, programming, and testing can help reduce their pervasiveness, but somehow, somewhere, they'll always find a way to creep into your code. This becomes especially apparent as new features are introduced and your code base grows in size and complexity.

Fortunately, some bugs are easier to detect than others. Compile-time bugs, for example, can be detected early on; you can use the compiler's error messages to figure out what the problem is and fix it right then and there. Runtime bugs, however, can be much more problematic; they don't always surface immediately, and when they do, it may be at a point in the program that is far removed from the actual cause of the problem. Generics, introduced in Java SE 5.0, add stability to your code by making more of your bugs detectable at compile time.

Why Use Generics?

In a nutshell, generics enable *types* (classes and interfaces) to be parameters when defining classes, interfaces, and methods. Much like the more familiar *formal parameters* used in method declarations, type parameters provide a way for you to reuse the same code with different inputs. The difference is that the inputs to formal parameters are values, while the inputs to type parameters are types.

Code that uses generics has many benefits over nongeneric code:

- *Stronger type checks at compile time.* A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety. Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.
- *Elimination of casts.* The following nongeneric code snippet requires casting:

  ```
  List list = new ArrayList();
  list.add("hello");
  String s = (String) list.get(0);
  ```

  When rewritten using generics, the code does not require casting:

  ```
  List<String> list = new ArrayList<String>();
  list.add("hello");
  String s = list.get(0);   // no cast
  ```

- *The ability to implement generic algorithms.* By using generics, programmers can implement generic algorithms that work on collections of different types, can be customized, and are type safe and easier to read.

Generic Types

A *generic type* is a generic class or interface that is parameterized over types. The following `Box` class will be modified to demonstrate the concept.

A Simple Box Class

Begin by examining a nongeneric `Box` class that operates on objects of any type. It only needs to provide two methods: `set`, which adds an object to the box, and `get`, which retrieves it:

```
public class Box {
    private Object object;

    public void set(Object object) { this.object = object; }
    public Object get() { return object; }
}
```

Since its methods accept or return an Object, you are free to pass in whatever you want, provided that it is not one of the primitive types. At compile time, there is no way to verify how the class is used. One part of the code may place an Integer in the box and expect to get Integers out of it, while another part of the code may mistakenly pass in a String, resulting in a runtime error.

A Generic Version of the Box Class

A *generic class* is defined with the following format:

```
class name<T1, T2, . . . , Tn> { /* . . . */ }
```


The type parameter section, delimited by angle brackets (<>), follows the class name. It specifies the *type parameters* (also called *type variables*) T1, T2, . . . , and Tn.

To update the Box class to use generics, you create a *generic type declaration* by changing the code public class Box to public class Box<T>. This introduces the type variable T, which can be used anywhere inside the class.

With this change, the Box class becomes the following:

```
/**
 * Generic version of the Box class.
 * @param <T> the type of the value being boxed
 */
public class Box<T> {
    // T stands for "Type"
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

As you can see, all occurrences of Object are replaced by T. A type variable can be any nonprimitive type you specify: any class type, interface type, array type, or even another type variable. This same technique can be applied to create generic interfaces.

Type Parameter Naming Conventions

By convention, type parameter names are single, uppercase letters. This stands in sharp contrast to the variable naming conventions that you already know about, with good reason: without this convention, it would be difficult to tell the difference between a type variable and an ordinary class or interface name.

The most commonly used type parameter names are as follows:

- E—Element (used extensively by the Java Collections Framework)
- K—Key
- N—Number
- T—Type
- V—Value
- S, U, V, and so on—Second, third, and fourth types

You'll see these names used throughout the Java SE Application Programming Interface (API) and the rest of this chapter.

Invoking and Instantiating a Generic Type

To reference the generic Box class from within your code, you must perform a *generic type invocation*, which replaces T with some concrete value, such as Integer:

```
Box<Integer> integerBox;
```

You can think of a generic type invocation as being similar to an ordinary method invocation, but instead of passing an argument to a method, you are passing a *type argument* (Integer in this case) to the Box class itself.

> **Note**
> Many developers use the terms *type parameter* and *type argument* interchangeably, but these terms are not the same. In code, one provides type arguments in order to create a parameterized type. Therefore, the T in Foo<T> is a type parameter and the String in Foo<String> f is a type argument. This chapter observes this definition when using these terms.

Like any other variable declaration, this code does not actually create a new Box object. It simply declares that integerBox will hold a reference to a "Box of Integer," which is how Box<Integer> is read.

An invocation of a generic type is generally known as a *parameterized type*. To instantiate this class, use the new keyword, as usual, but place <Integer> between the class name and the parentheses:

```
Box<Integer> integerBox = new Box<Integer>();
```

The Diamond

In Java SE 7 and later, you can replace the type arguments required to invoke the constructor of a generic class with an empty set of type arguments (<>) as long as the compiler can determine, or infer, the type arguments from the context. This pair of angle brackets (<>) is informally called *the diamond*. For example, you can create an instance of Box<Integer> with the following statement:

```
Box<Integer> integerBox = new Box<>();
```

For more information on diamond notation and type inference, see the "Type Inference" section later on in this chapter.

Multiple Type Parameters

As mentioned previously, a generic class can have multiple type parameters. One example is the generic OrderedPair class, which implements the generic Pair interface:

```
public interface Pair<K, V> {
    public K getKey();
    public V getValue();
}

public class OrderedPair<K, V> implements Pair<K, V> {

    private K key;
    private V value;

    public OrderedPair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey()   { return key; }
    public V getValue() { return value; }
}
```

The following statements create two instantiations of the OrderedPair class:

```
Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);
Pair<String, String>  p2 = new OrderedPair<String, String>("hello", "world");
```

The code, new OrderedPair<String, Integer>, instantiates K as a String and V as an Integer. Therefore, the parameter types of OrderedPair's constructor are String and Integer, respectively. Due to autoboxing, it is valid to pass a String and an int to the class.

As mentioned previously, because a Java compiler can infer the K and V types from the declaration OrderedPair<String, Integer>, these statements can be shortened using diamond notation:

```
OrderedPair<String, Integer> p1 = new OrderedPair<>("Even", 8);
OrderedPair<String, String>  p2 = new OrderedPair<>("hello", "world");
```

To create a generic interface, follow the same conventions as you would to create a generic class.

Parameterized Types

You can also substitute a type parameter (e.g., K or V) with a parameterized type (e.g., List<String>). Here is an example, using OrderedPair<V, K>:

```
OrderedPair<String, Box<Integer>> p = new OrderedPair<>("primes", new
   Box<Integer>( . . . ));
```


Raw Types

A *raw type* is the name of a generic class or interface without any type arguments. Here is an example, given the generic Box class:

```
public class Box<T> {
    public void set(T t) { /* . . . */ }
    // . . .
}
```

To create a parameterized type of Box<T>, you supply an actual type argument for the formal type parameter T:

```
Box<Integer> intBox = new Box<>();
```

If the actual type argument is omitted, you create a raw type of Box<T>:

```
Box rawBox = new Box();
```

Therefore, Box is the raw type of the generic type Box<T>. However, a nongeneric class or interface type is *not* a raw type.

Raw types show up in legacy code because lots of API classes (such as the Collections classes) were not generic prior to the Java SE Development Kit (JDK) 5.0. When using raw types, you essentially get pregenerics behavior: a Box gives you Objects. For backward compatibility, assigning a parameterized type to its raw type is allowed:

```
Box<String> stringBox = new Box<>();
Box rawBox = stringBox;               // OK
```

However, if you assign a raw type to a parameterized type, you get a warning:

```
Box rawBox = new Box();           // rawBox is a raw type of Box<T>
Box<Integer> intBox = rawBox;     // warning: unchecked conversion
```

You also get a warning if you use a raw type to invoke generic methods defined in the corresponding generic type:

```
Box<String> stringBox = new Box<>();
Box rawBox = stringBox;
rawBox.set(8);  // warning: unchecked invocation to set(T)
```

The warning shows that raw types bypass generic type checks, deferring the catch of unsafe code to runtime. Therefore, you should avoid using raw types.

The "Type Erasure" section later on in this chapter has more information on how the Java compiler uses raw types.

Unchecked Error Messages

As mentioned previously, when mixing legacy code with generic code, you may encounter warning messages similar to the following:

```
Note: Example.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

This can happen when using an older API that operates on raw types, as shown in the following example:

```
public class WarningDemo {
    public static void main(String[] args){
        Box<Integer> bi;
        bi = createBox();
    }

    static Box createBox(){
        return new Box();
    }
}
```

The term *unchecked* means that the compiler does not have enough type information to perform all type checks necessary to ensure type safety. The unchecked warning is disabled, by default, though the compiler gives a hint. To see all unchecked warnings, recompile with `-Xlint:unchecked`.

Recompiling the previous example with `-Xlint:unchecked` reveals the following additional information:

```
WarningDemo.java:4: warning: [unchecked] unchecked conversion
found   : Box
required: Box<java.lang.Integer>
        bi = createBox();
                      ^
1 warning
```

To completely disable unchecked warnings, use the `-Xlint:-unchecked` flag. The `@SuppressWarnings("unchecked")` annotation suppresses unchecked warnings. If you are unfamiliar with the `@SuppressWarnings` syntax, see Chapter 4, “Annotations.”

Generic Methods

Generic methods are methods that introduce their own type parameters. This is similar to declaring a generic type, but the type parameter’s scope is limited to the method where it is declared. Static and nonstatic generic methods are allowed, as are generic class constructors.

6

The syntax for a generic method includes a type parameter, inside angle brackets, that appears before the method’s return type. For static generic methods, the type parameter section must appear before the method’s return type.

The `Util` class includes a generic method, `compare`, which compares two `Pair` objects:

```
public class Util {
    // Generic static method
    public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {
        return p1.getKey().equals(p2.getKey()) &&
               p1.getValue().equals(p2.getValue());
    }
}

public class Pair<K, V> {

    private K key;
    private V value;

    // Generic constructor
    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    // Generic methods
    public void setKey(K key) { this.key = key; }
    public void setValue(V value) { this.value = value; }
    public K getKey()   { return key; }
    public V getValue() { return value; }
}
```

The complete syntax for invoking this method is as follows:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.<Integer, String>compare(p1, p2);
```

The type has been explicitly provided, as shown in bold. Generally, this can be left out and the compiler will infer the type that is needed:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.compare(p1, p2);
```

This feature, known as *type inference*, allows you to invoke a generic method as an ordinary method, without specifying a type between angle brackets. This topic is further discussed later on, in "Type Inference."

Bounded Type Parameters

There may be times when you want to restrict the types that can be used as type arguments in a parameterized type. For example, a method that operates on numbers might only want to accept instances of `Number` or its subclasses. This is what *bounded type parameters* are for.

To declare a bounded type parameter, list the type parameter's name, followed by the `extends` keyword, followed by its *upper bound*, which in this example is `Number`. Note that, in this context, `extends` is used in a general sense to mean either *extends* (as in classes) or *implements* (as in interfaces):

```
public class Box<T> {

    private T t;

    public void set(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }

    public <U extends Number> void inspect(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.set(new Integer(10));
        integerBox.inspect("some text"); // error: this is still String!
    }
}
```

If we modify our generic method to include this bounded type parameter, compilation will now fail, since our invocation of `inspect` still includes a `String`:

```
Box.java:21: <U>inspect(U) in Box<java.lang.Integer> cannot
  be applied to (java.lang.String)
                        integerBox.inspect("10");
                                  ^
1 error
```

In addition to limiting the types you can use to instantiate a generic type, bounded type parameters allow you to invoke methods defined in the bounds:


```
public class NaturalNumber<T extends Integer> {

    private T n;

    public NaturalNumber(T n)  { this.n = n; }

    public boolean isEven() {
        return n.intValue() % 2 == 0;
    }

    // . . .
}
```

The `isEven` method invokes the `intValue` method defined in the `Integer` class through n.

Multiple Bounds

The previous example illustrates the use of a type parameter with a single bound, but a type parameter can have *multiple bounds*:

```
<T extends B1 & B2 & B3>
```

A type variable with multiple bounds is a subtype of all the types listed in the bound. If one of the bounds is a class, it must be specified first, as follows:

```
Class A { /* . . . */ }
interface B { /* . . . */ }
interface C { /* . . . */ }

class D <T extends A & B & C> { /* . . . */ }
```

If bound `A` is not specified first, you get a compile-time error:

```
class D <T extends B & A & C> { /* . . . */ }  // compile-time error
```

Generic Methods and Bounded Type Parameters

Bounded type parameters are key to the implementation of generic algorithms. Consider the following method, which counts the number of elements in an array `T[]` that are greater than a specified element `elem`:

```
public static <T> int countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray)
        if (e > elem)  // compiler error
            ++count;
    return count;
}
```


The implementation of the method is straightforward, but it does not compile because the greater than operator (>) applies only to primitive types such as `short`, `int`, `double`, `long`, `float`, `byte`, and `char`. You cannot use the greater than operator to compare objects. To fix the problem, use a type parameter bounded by the `Comparable<T>` interface:

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

The resulting code is as follows:

```
public static <T extends Comparable<T>> int countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray)
        if (e.compareTo(elem) > 0)
            ++count;
    return count;
}
```

Generics, Inheritance, and Subtypes

As you already know, it is possible to assign an object of one type to an object of another type, provided that the types are compatible. For example, you can assign an `Integer` to an `Object`, since `Object` is one of `Integer`'s supertypes:

```
Object someObject = new Object();
Integer someInteger = new Integer(10);
someObject = someInteger;   // OK
```

In object-oriented terminology, this is called an *is a* relationship. Since an `Integer` *is a* kind of `Object`, the assignment is allowed. But `Integer` is also a kind of `Number`, so the following code is valid as well:

```
public void someMethod(Number n) { /* . . . */ }

someMethod(new Integer(10));   // OK
someMethod(new Double(10.1);   // OK
```

The same is also true with generics. You can perform a generic type invocation, passing `Number` as its type argument, and any subsequent invocation of `add` will be allowed if the argument is compatible with `Number`:

```
Box<Number> box = new Box<Number>();
box.add(new Integer(10));   // OK
box.add(new Double(10.1));  // OK
```

Now consider the following method:

```
public void boxTest(Box<Number> n) { /* . . . */ }
```

What type of argument does it accept? By looking at its signature, you can see that it accepts a single argument whose type is `Box<Number>`. But what does that mean? Are you allowed to pass in `Box<Integer>` or `Box<Double>`, as you might expect? The answer is *no* because, as shown in Figure 6.1, `Box<Integer>` and `Box<Double>` are not subtypes of `Box<Number>`. This is a common misunderstanding when it comes to programming with generics and is an important concept to learn.

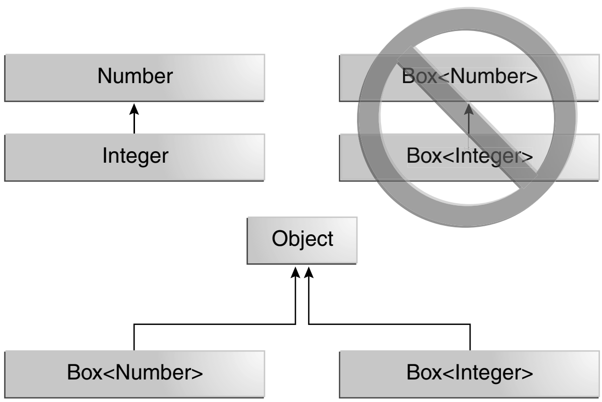


Figure 6.1 Box<Integer> Is Not a Subtype of Box<Number> Even Though Integer Is a Subtype of Number

> **Note**
>
> Given two concrete types `A` and `B` (e.g., `Number` and `Integer`), `MyClass` has no relationship to `MyClass<B>`, regardless of whether or not `A` and `B` are related. The common parent of `MyClass` and `MyClass<B>` is `Object`.
>
> For information on how to create a subtype-like relationship between two generic classes when the type parameters are related, see the section on "Wildcards and Subtyping" later on in this chapter.

Generic Classes and Subtyping

You can subtype a generic class or interface by extending or implementing it. The relationship between the type parameters of one class or interface and the type parameters of another are determined by the `extends` and `implements` clauses.

Using the `Collections` classes as an example, as shown in Figure 6.2, `ArrayList<E> implements List<E>`, and `List<E> extends Collection<E>`. So `ArrayList<String>` is a subtype of `List<String>`, which is a subtype of `Collection<String>`. So long as you do not vary the type argument, the subtyping relationship is preserved between the types.

Now imagine we want to define our own list interface, `PayloadList` (Figure 6.3), which associates an optional value of generic type `P` with each element. Its declaration might resemble the following:

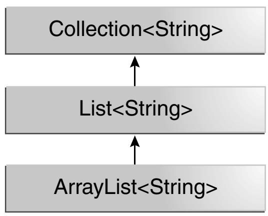


Figure 6.2 A Sample Collections Hierarchy

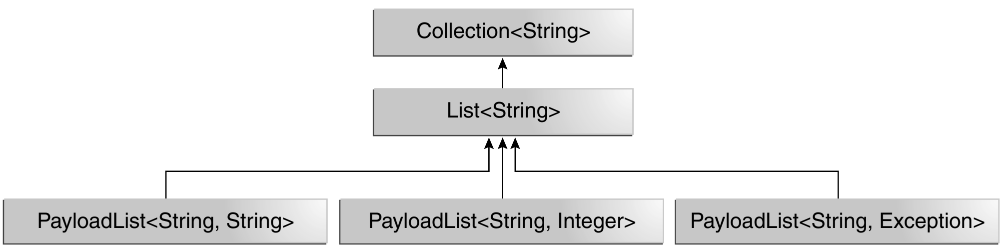


Figure 6.3 A Sample PayloadList Hierarchy

```
interface PayloadList<E,P> extends List<E> {
  void setPayload(int index, P val);
   . . .
}
```

The following parameterizations of `PayloadList` are subtypes of `List<String>`:

- `PayloadList<String,String>`
- `PayloadList<String,Integer>`
- `PayloadList<String,Exception>`

Type Inference

Type inference is a Java compiler's ability to look at each method invocation and corresponding declaration to determine the type argument (or arguments) that makes the invocation applicable. The inference algorithm determines the types of the arguments and, if available, the type that the result is being assigned or returned. Finally, the inference algorithm tries to find the *most specific* type that works with all the arguments.

To illustrate this last point, in the following example, inference determines that the second argument being passed to the `pick` method is of type `Serializable`:

```
static <T> T pick(T a1, T a2) { return a2; }
Serializable s = pick("d", new ArrayList<String>());
```

Type Inference and Generic Methods

The previous discussion of generic methods introduced you to type inference, which enables you to invoke a generic method as you would an ordinary method, without specifying a type between angle brackets. Consider the following example, `BoxDemo`, which requires the `Box` class:

```
public class BoxDemo {

  public static <U> void addBox(U u,
      java.util.List<Box<U>> boxes) {
    Box<U> box = new Box<>();
    box.set(u);
    boxes.add(box);
  }

  public static <U> void outputBoxes(java.util.List<Box<U>> boxes) {
    int counter = 0;
    for (Box<U> box: boxes) {
      U boxContents = box.get();
      System.out.println("Box #" + counter + " contains [" +
```

```
                boxContents.toString() + "]");
            counter++;
        }
    }

    public static void main(String[] args) {
        java.util.ArrayList<Box<Integer>> listOfIntegerBoxes =
            new java.util.ArrayList<>();
        BoxDemo.<Integer>addBox(Integer.valueOf(10), listOfIntegerBoxes);
        BoxDemo.addBox(Integer.valueOf(20), listOfIntegerBoxes);
        BoxDemo.addBox(Integer.valueOf(30), listOfIntegerBoxes);
        BoxDemo.outputBoxes(listOfIntegerBoxes);
    }
}
```


The following is the output from this example:

```
Box #0 contains [10]
Box #1 contains [20]
Box #2 contains [30]
```

The generic method `addBox` defines one type parameter, named `U`. Generally, a Java compiler can infer the type parameters of a generic method call. Consequently, in most cases, you do not need to specify them. For example, to invoke the generic method `addBox`, you can specify the type parameter as follows:

```
BoxDemo.<Integer>addBox(Integer.valueOf(10), listOfIntegerBoxes);
```

Alternatively, if you omit the type parameters, a Java compiler automatically infers (from the method's arguments) that the type parameter is `Integer`:

```
BoxDemo.addBox(Integer.valueOf(20), listOfIntegerBoxes);
```

Type Inference and Instantiation of Generic Classes

You can replace the type arguments required to invoke the constructor of a generic class with an empty set of type parameters (<>; informally known as *the diamond*) as long as the compiler can infer the type arguments from the context.

For example, consider the following variable declaration:

```
Map<String, List<String>> myMap = new HashMap<String, List<String>>();
```

You can substitute the parameterized type of the constructor with an empty set of type parameters (<>):

```
Map<String, List<String>> my Map = new HashMap<>();
```

In order to take advantage of type inference during generic class instantiation, you must place notation inside the diamond. In the following example, the compiler generates an unchecked conversion warning because the HashMap() constructor refers to the HashMap raw type, not the Map<String, List<String>> type:

```
Map<String, List<String>> myMap = new HashMap(); // unchecked conversion warning
```

Java supports limited type inference for generic instance creation; you can only use type inference if the parameterized type of the constructor is obvious from the context. For example, the following code does not compile:

```
List<String> list = new ArrayList<>();
list.add("A");

// The following statement should fail since addAll expects
// Collection<? extends String>
list.addAll(new ArrayList<>());
```

Note that the diamond often works in method calls; however, for greater clarity, it is suggested that you use the diamond primarily to initialize a variable where it is declared. Note that the following example can successfully compile:

```
List<? extends String> list2 = new ArrayList<>();
list.addAll(list2);
```

Type Inference and Generic Constructors of Generic and Nongeneric Classes

Note that constructors can be generic (i.e., declare their own formal type parameters) in both generic and nongeneric classes. Consider the following example:

```
class MyClass<X> {
  <T> MyClass(T t) {
    // . . .
  }
}
```

Now consider the following instantiation of the class MyClass:

```
new MyClass<Integer>("")
```

This statement creates an instance of the parameterized type MyClass<Integer>; the statement explicitly specifies the type Integer for the formal type parameter, X, of the generic class MyClass<X>. Note that the constructor for this generic class contains a formal type parameter, `T`. The compiler infers the type `String` for the formal type parameter, `T`, of the constructor of this generic class (because the actual parameter of this constructor is a `String` object).

Compilers from releases prior to Java SE 7 are able to infer the actual type parameters of generic constructors, similar to generic methods. However, compilers in Java SE 7 and later can infer the actual type parameters of the generic class being instantiated if you use the diamond (<>). Consider the following example:

```
MyClass<Integer> myObject = new MyClass<>("");
```

In this example, the compiler infers the type `Integer` for the formal type parameter, `X`, of the generic class `MyClass<X>`. It infers the type `String` for the formal type parameter, `T`, of the constructor of this generic class.

> **Note**
> It is important to note that the inference algorithm uses only invocation arguments and, possibly, an obvious expected return type to infer types. The inference algorithm does not use results from later in the program.

Wildcards

In generic code, the question mark (`?`), called the *wildcard*, represents an unknown type. The wildcard can be used in a variety of situations: as the type of a parameter, field, or local variable, or sometimes as a return type (though it is better programming practice to be more specific). The wildcard is never used as a type argument for a generic method invocation, a generic class instance creation, or a supertype.

The following sections discuss wildcards in more detail, including upper-bounded wildcards, lower-bounded wildcards, and wildcard capture.

Upper-Bounded Wildcards

You can use an upper-bounded wildcard to relax the restrictions on a variable. For example, say you want to write a method that works on `List<Integer>`, `List<Double>`, *and* `List<Number>`; you can achieve this by using an upper-bounded wildcard.

To declare an upper-bounded wildcard, use the wildcard character (`?`), followed by the `extends` keyword, followed by its *upper bound*. Note that, in this context, `extends` is used in a general sense to mean either *extends* (as in classes) or *implements* (as in interfaces).

To write the method that works on lists of `Number` and the subtypes of `Number`, such as `Integer`, `Double`, and `Float`, you would specify `List<? extends Number>`. The term `List<Number>` is more restrictive than `List<? extends Number>` because the former matches a list of type `Number` only, whereas the latter matches a list of type `Number` or any of its subclasses.

Consider the following `process` method:

```
public static void process(List<? extends Foo> list) { /* . . . */ }
```

The upper-bounded wildcard, `<? extends Foo>`, where `Foo` is any type, matches `Foo` and any subtype of `Foo`. The `process` method can access the list elements as type `Foo`:

```
public static void process(List<? extends Foo> list) {
    for (Foo elem : list) {
        // . . .
    }
}
```

In the `foreach` clause, the `elem` variable iterates over each element in the list. Any method defined in the `Foo` class can now be used on `elem`.

The `sumOfList` method returns the sum of the numbers in a list:

```
public static double sumOfList(List<? extends Number> list) {
    double s = 0.0;
    for (Number n : list)
        s += n.doubleValue();
    return s;
}
```

The following code, using a list of `Integer` objects, prints `sum = 6.0`:

```
List<Integer> li = Arrays.asList(1, 2, 3);
System.out.println("sum = " + sumOfList(li));
```

A list of `Double` values can use the same `sumOfList` method. The following code prints `sum = 7.0`:

```
List<Double> ld = Arrays.asList(1.2, 2.3, 3.5);
System.out.println("sum = " + sumOfList(ld));
```

Unbounded Wildcards

The unbounded wildcard type is specified using the wildcard character (`?`)—for example, `List<?>`. This is called a *list of unknown type*. There are two scenarios where an unbounded wildcard is a useful approach:

- It is useful if you are writing a method that can be implemented using functionality provided in the `Object` class.
- It is useful when the code is using methods in the generic class that don't depend on the type parameter (e.g., `List.size` or `List.clear`). In fact, `Class<?>` is often used because most of the methods in `Class<T>` do not depend on `T`.

Consider the following method, `printList`:

```
public static void printList(List<Object> list) {
    for (Object elem : list)
        System.out.println(elem + " ");
    System.out.println();
}
```

The goal of `printList` is to print a list of any type, but it fails to achieve that goal: It prints only a list of `Object` instances. It cannot print `List<Integer>`, `List<String>`, `List<Double>`, and so on. This is because they are not subtypes of `List<Object>`. To write a generic `printList` method, use `List<?>`:

```
public static void printList(List<?> list) {
    for (Object elem: list)
        System.out.print(elem + " ");
    System.out.println();
}
```

For any concrete type `A`, `List` is a subtype of `List<?>`. Thus you can use `printList` to print a list of any type:

```
List<Integer> li = Arrays.asList(1, 2, 3);
List<String>  ls = Arrays.asList("one", "two", "three");
printList(li);
printList(ls);
```

It's important to note that `List<Object>` and `List<?>` are not the same. You can insert an `Object`, or any subtype of `Object`, into a `List<Object>`. But you can only insert `null` into a `List<?>`. The "Guidelines for Wildcard Use" section has more information on how to determine what kind of wildcard, if any, should be used in a given situation.

> **Note**
>
> The `Arrays.asList`[1] method is used in examples throughout this chapter. This static factory method converts the specified array and returns a fixed-size list.

1. `7/docs/api/java/util/Arrays.html#asList(T...)`

Lower-Bounded Wildcards

The "Upper-Bounded Wildcards" section shows that an upper-bounded wildcard restricts the unknown type to be a specific type or a subtype of that type and is represented using the extends keyword. In a similar way, a lower-bounded wildcard restricts the unknown type to be a specific type or a supertype of that type. A lower-bounded wildcard is expressed using the wildcard character (?), followed by the super keyword, followed by its *lower bound*: <? super A>.

> **Note**
> You can specify either an upper bound or a lower bound for a wildcard; you cannot specify both.

Say you want to write a method that puts Integer objects into a list. To maximize flexibility, you would like the method to work on List<Integer>, List<Number>, and List<Object>—anything that can hold Integer values.

To write the method that works on lists of Integer and the supertypes of Integer, such as Integer, Number, and Object, you would specify List<? super Integer>. The term List<Integer> is more restrictive than List<? super Integer> because the former matches a list of type Integer only, whereas the latter matches a list of any type that is a supertype of Integer.

The following code adds the numbers 1 through 10 to the end of a list:

```
public static void addNumbers(List<? super Integer> list) {
    for (int i = 1; i <= 10; i++) {
        list.add(i);
    }
}
```

The "Guidelines for Wildcard Use" section in this chapter provides guidance on when to use upper-bounded wildcards and when to use lower-bounded wildcards.

Wildcards and Subtyping

As described previously in "Generics, Inheritance, and Subtypes," generic classes or interfaces are not related merely because there is a relationship between their types. However, you can use wildcards to create a relationship between generic classes or interfaces.

Consider the following two regular (nongeneric) classes:

```
class A { /* . . . */ }
```

```
class B extends A { /* . . . */ }
```

For these classes, it would be reasonable to write the following code:

```
B b = new B();
A a = b;
```

This example shows that inheritance of regular classes follows the rule of subtyping: class B is a subtype of class A if B extends A. This rule does not apply to generic types:


```
List<B> lb = new ArrayList<>();
List la = lb;   // compile-time error
```

Given that Integer is a subtype of Number, what is the relationship between List<Integer> and List<Number>? Although Integer is a subtype of Number, List<Integer> is not a subtype of List<Number> and, in fact, these two types are not related. The common parent of List<Number> and List<Integer> is List<?> (Figure 6.4).

In order to create a relationship between these classes so that the code can access Number's methods through List<Integer>'s elements, use an upper-bounded wildcard:

```
List<? extends Integer> intList = new ArrayList<>();
// OK. List<? extends Integer> is a subtype of List<? extends Number>
List<? extends Number>  numList = intList;
```

Because Integer is a subtype of Number, and numList is a list of Number objects, a relationship now exists between intList (a list of Integer objects) and numList. Figure 6.5 shows the relationships between several List classes declared with both upper- and lower-bounded wildcards.

The "Guidelines for Wildcard Use" section later on in this chapter has more information about the ramifications of using upper- and lower-bounded wildcards.

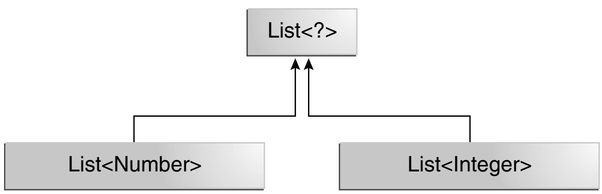
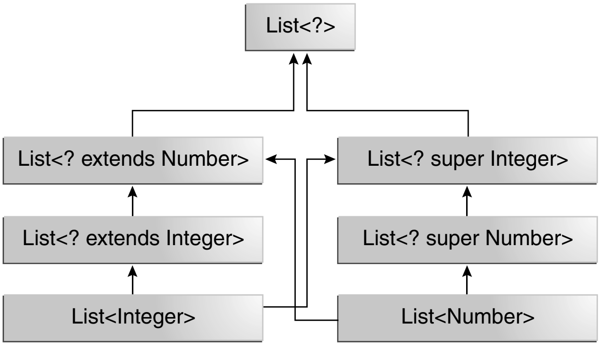


Figure 6.4 The Common Parent Is List<?>



6

Figure 6.5 A Hierarchy of Several Generic List Class Declarations

Wildcard Capture and Helper Methods

In some cases, the compiler infers the type of a wildcard. For example, a list may be defined as `List<?>`, but when evaluating an expression, the compiler infers a particular type from the code. This scenario is known as *wildcard capture*. For the most part, you don't need to worry about wildcard capture, except when you see an error message that contains the phrase `capture of`.

The `WildcardError` example produces a capture error when compiled:

```
import java.util.List;

public class WildcardError {

    void foo(List<?> i) {
        i.set(0, i.get(0));
    }
}
```

In this example, the compiler processes the `i` input parameter as being of type `Object`. When the `foo` method invokes `List.set(int, E)`[2], the compiler is not able to confirm the type of object that is being inserted into the list and an error is produced. When this type of error occurs, it typically means that the compiler believes that you are assigning the wrong type to a variable. Generics were added to the Java language for this reason—to enforce type safety at compile time.

The `WildcardError` example generates the following error when compiled by Oracle's JDK 7 `javac` implementation:

```
WildcardError.java:6: error: method set in interface List<E> cannot
   be applied to given types;
```

2. `7/docs/api/java/util/List.html#set(int,E)`

```
    i.set(0, i.get(0));
     ^
  required: int,CAP#1
  found: int,Object
  reason: actual argument Object cannot be converted to CAP#1 by method
     invocation conversion
  where E is a type-variable:
    E extends Object declared in interface List
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object from capture of ?
1 error
```


In this example, the code is attempting to perform a safe operation, so how can you work around the compiler error? You can fix it by writing a *private helper method*, which captures the wildcard. In this case, you can work around the problem by creating the private helper method, `fooHelper`, as shown in `WildcardFixed`:

```
public class WildcardFixed {

    void foo(List<?> i) {
        fooHelper(i);
    }

    // Helper method created so that the wildcard can be captured
    // through type inference.
    private <T> void fooHelper(List<T> l) {
        l.set(0, l.get(0));
    }
}
```

Thanks to the helper method, the compiler uses inference to determine that `T` is `CAP#1`, the capture variable, in the invocation. The example now compiles successfully.

By convention, helper methods are generally named *`originalMethodNameHelper`*. Now consider a more complex example, `WildcardErrorBad`:

```
import java.util.List;

public class WildcardErrorBad {

    void swapFirst(List<? extends Number> l1, List<? extends Number> l2) {
      Number temp = l1.get(0);
      l1.set(0, l2.get(0)); // expected a CAP#1 extends Number,
                            // got a CAP#2 extends Number;
                            // same bound, but different types
      l2.set(0, temp);      // expected a CAP#1 extends Number,
                            // got a Number
    }
}
```

In this example, the code is attempting an unsafe operation. For example, consider the following invocation of the `swapFirst` method:

```
List<Integer> li = Arrays.asList(1, 2, 3);
List<Double>  ld = Arrays.asList(10.10, 20.20, 30.30);
swapFirst(li, ld);
```

While `List<Integer>` and `List<Double>` both fulfill the criteria of `List<? extends Number>`, it is clearly incorrect to take an item from a list of `Integer` values and attempt to place it into a list of `Double` values.

Compiling the code with Oracle's JDK `javac` compiler produces the following error:


```
WildcardErrorBad.java:7: error: method set in interface List<E> cannot be
   applied to given types;
      l1.set(0, l2.get(0)); // expected a CAP#1 extends Number,
        ^
  required: int,CAP#1
  found: int,Number
  reason: actual argument Number cannot be converted to CAP#1 by method
     invocation conversion
  where E is a type-variable:
    E extends Object declared in interface List
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Number from capture of ? extends Number
WildcardErrorBad.java:10: error: method set in interface List<E> cannot
   be applied to given types;
      l2.set(0, temp);      // expected a CAP#1 extends Number,
        ^
  required: int,CAP#1
  found: int,Number
  reason: actual argument Number cannot be converted to CAP#1 by method
     invocation conversion
  where E is a type-variable:
    E extends Object declared in interface List
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Number from capture of ? extends Number
WildcardErrorBad.java:15: error: method set in interface List<E> cannot
   be applied to given types;
        i.set(0, i.get(0));
         ^
  required: int,CAP#1
  found: int,Object
  reason: actual argument Object cannot be converted to CAP#1 by method
     invocation conversion
  where E is a type-variable:
    E extends Object declared in interface List
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object from capture of ?
3 errors
```

There is no helper method to work around the problem because the code is fundamentally wrong.

Guidelines for Wildcard Use

One of the more confusing aspects when learning to program with generics is determining when to use an upper-bounded wildcard and when to use a lower-bounded wildcard. This section provides some guidelines to follow when designing your code.

For purposes of this discussion, it is helpful to think of variables as serving one of two functions:

- An *in* variable serves up data to the code. Imagine a copy method with two arguments: `copy(src, dest)`. The `src` argument provides the data to be copied, so it is the *in* parameter.
- An *out* variable holds data for use elsewhere. In the copy example, `copy(src, dest)`, the `dest` argument accepts data, so it is the *out* parameter.

Of course, some variables are used for both in and out purposes, as discussed later. You can use the in and out principles when deciding whether to use a wildcard and what type of wildcard is appropriate. The following list provides the guidelines that you should follow:

- An invariable is defined with an upper-bounded wildcard, using the `extends` keyword.
- An out variable is defined with a lower-bounded wildcard, using the `super` keyword.
- In the case where the in variable can be accessed using methods defined in the `Object` class, use an unbounded wildcard.
- In the case where the code needs to access the variable as both an in and an out variable, do not use a wildcard.

These guidelines do not apply to a method's return type. Using a wildcard as a return type should be avoided because it forces programmers using the code to deal with wildcards.

A list defined by `List<? extends . . . >` can be informally thought of as read only, but that is not a strict guarantee. Suppose you have the following two classes:

```
class NaturalNumber {

    private int i;

    public NaturalNumber(int i) { this.i = i; }
    // . . .
}

class EvenNumber extends NaturalNumber {
```

```
        public EvenNumber(int i) { super(i); }
        // . . .
    }
```

Consider the following code:

```
List<EvenNumber> le = new ArrayList<>();
List<? extends NaturalNumber> ln = le;
ln.add(new NaturalNumber(35));  // compile-time error
```

Because `List<EvenNumber>` is a subtype of `List<? extends NaturalNumber>`, you can assign `le` to `ln`. But you cannot use `ln` to add a natural number to a list of even numbers. The following operations on the list are possible:

- You can add `null`.
- You can invoke `clear`.
- You can get the iterator and invoke `remove`.
- You can capture the wildcard and write elements that you've read from the list.

You can see that the list defined by `List<? extends NaturalNumber>` is not read only in the strictest sense of the word, but you might think of it that way because you cannot store a new element or change an existing element in the list.

Type Erasure

Generics were introduced to the Java language to provide tighter type checks at compile time and to support generic programming. To implement generics, the Java compiler applies type erasure to achieve the following:

- Replace all type parameters in generic types with their bounds or `Object` if the type parameters are unbounded. The produced bytecode, therefore, contains only ordinary classes, interfaces, and methods.
- Insert type casts if necessary to preserve type safety.
- Generate bridge methods to preserve polymorphism in extended generic types.

Type erasure ensures that no new classes are created for parameterized types; consequently, generics incur no runtime overhead.

Erasure of Generic Types

During the type erasure process, the Java compiler erases all type parameters and replaces each with its first bound if the type parameter is bounded or `Object` if the type parameter is unbounded.

Consider the following generic class that represents a node in a singly linked list:

```
public class Node<T> {

    private T data;
    private Node<T> next;

    public Node(T data, Node<T> next) }
        this.data = data;
        this.next = next;
    }

    public T getData() { return data; }
    // . . .
}
```


Because the type parameter `T` is unbounded, the Java compiler replaces it with `Object`:

```
public class Node {

    private Object data;
    private Node next;

    public Node(Object data, Node next) {
        this.data = data;
        this.next = next;
    }

    public Object getData() { return data; }
    // . . .
}
```

In the following example, the generic `Node` class uses a bounded type parameter:

```
public class Node<T extends Comparable<T>> {

    private T data;
    private Node<T> next;

    public Node(T data, Node<T> next) {
        this.data = data;
        this.next = next;
    }

    public T getData() { return data; }
    // . . .
}
```

The Java compiler replaces the bounded type parameter `T` with the first bound class, `Comparable`:

```
public class Node {
```

```
    private Comparable data;
    private Node next;

    public Node(Comparable data, Node next) {
        this.data = data;
        this.next = next;
    }

    public Comparable getData() { return data; }
    // . . .
}
```

6

Erasure of Generic Methods

The Java compiler also erases type parameters in generic method arguments. Consider the following generic method:

```
// Counts the number of occurrences of elem in anArray.

public static <T> int count(T[] anArray, T elem) {
    int cnt = 0;
    for (T e : anArray)
        if (e.equals(elem))
            ++cnt
        return cnt;
}
```

Because `T` is unbounded, the Java compiler replaces it with `Object`:

```
public static int count(Object[] anArray, Object elem) {
    int cnt = 0;
    for (Object e : anArray)
        if (e.equals(elem))
            ++cnt;
        return cnt;
}
```

Suppose the following classes are defined:

```
class Shape { /* . . . */ }
class Circle extends Shape { /* . . . */ }
class Rectangle extends Shape { /* . . . */ }
```

You can write a generic method to draw different shapes:

```
public static <T extends Shape> void draw(T shape) { /* . . . */ }
```

The Java compiler replaces `T` with `Shape`:

```
public static void draw(Shape shape) { /* . . . */ }
```

Effects of Type Erasure and Bridge Methods

Sometimes type erasure causes a situation that you may not have anticipated. The following example shows how this can occur. The example shows how a compiler sometimes creates a synthetic method, called a bridge method, as part of the type erasure process.

Consider the following two classes:

```
public class Node<T> {

    private T data;

    public Node(T data) { this.data = data; }

    public void setData(T data) {
        System.out.println("Node.setData");
        this.data = data;
    }
}

public class MyNode extends Node<Integer> {
    public MyNode(Integer data) { super(data); }

    public void setData(Integer data) {
        System.out.println("MyNode.setData");
        super.setData(data);
    }
}
```


Now, consider the following code:

```
MyNode mn = new MyNode(5);
Node n = mn;            // A raw type - compiler throws an unchecked warning
n.setData("Hello");     // Causes a ClassCastException to be thrown.
Integer x = mn.data;
```

After type erasure, this code changes as follows:

```
MyNode mn = new MyNode(5);
Node n = (MyNode)mn;         // A raw type - compiler throws an unchecked warning
n.setData("Hello");
Integer x = (String)mn.data; // Causes a ClassCastException to be thrown.
```

Here is what happens as the code is executed:

- `n.setData("Hello");` causes the method `setData(Object)` to be executed on the object of class `MyNode`. (The `MyNode` class inherited `setData(Object)` from `Node`.)
- In the body of `setData(Object)`, the data field of the object referenced by `n` is assigned to a `String`.

- The data field of that same object, referenced via mn, can be accessed and is expected to be an integer (since mn is a MyNode, which is a Node<Integer>).
- Trying to assign a String to an Integer causes a ClassCastException from a cast inserted at the assignment by a Java compiler.

Bridge Methods

When compiling a class or interface that extends a parameterized class or implements a parameterized interface, the compiler may need to create a synthetic method, called a *bridge method*, as part of the type erasure process. You normally don't need to worry about bridge methods, but you might be puzzled if one appears in a stack trace.

6

After type erasure, the Node and MyNode classes are as follows:

```
public class Node {

    private Object data;

    public Node(Object data) { this.data = data; }

    public void setData(Object data) {
        System.out.println("Node.setData");
        this.data = data;
    }
}

public class MyNode extends Node {

    public MyNode(Integer data) { super(data); }

    public void setData(Integer data) {
        System.out.println(Integer data);
        super.setData(data);
    }
}
```

After type erasure, the method signatures do not match. The Node method becomes setData(Object) and the MyNode method becomes setData(Integer). Therefore, the MyNodesetData method does not override the NodesetData method.

To solve this problem and preserve the polymorphism of generic types after type erasure, a Java compiler generates a bridge method to ensure that subtyping works as expected. For the MyNode class, the compiler generates the following bridge method for setData:

```
class MyNode extends Node {
    // Bridge method generated by the compiler

    public void setData(Object data) {
        setData((Integer) data);
    }
```

```
    public void setData(Integer data) {
        System.out.println("MyNode.setData");
        super.setData(data);
    }

    // . . .
}
```

As you can see, the bridge method, which has the same method signature as the `Node` class's `setData` method after type erasure, delegates to the original `setData` method.

Nonreifiable Types

The "Type Erasure" section discusses the process where the compiler removes information related to type parameters and type arguments. Type erasure has consequences related to variable arguments (also known as *varargs*), methods where a vararg formal parameters contains nonreifiable type. See Chapter 4, "Passing Information to a Method or a Constructor," for more information about varargs methods.

Nonreifiable Types Defined

A *reifiable type* is a type whose type information is fully available at runtime. This includes primitives, nongeneric types, raw types, and invocations of unbound wildcards.

Nonreifiable types are types whose information has been removed at compile time by type erasure—invocations of generic types that are not defined as unbounded wildcards. A nonreifiable type does not have all its information available at runtime. Examples of nonreifiable types are `List<String>` and `List<Number>`; the Java Virtual Machine (Java VM) cannot tell the difference between these types at runtime. As shown in the section "Restrictions on Generics" later in this chapter, there are certain situations where nonreifiable types cannot be used (e.g., in an `instanceof` expression or as an element in an array).

Heap Pollution

Heap pollution occurs when a variable of a parameterized type refers to an object that is not of that parameterized type. This situation occurs if the program performed some operation that gives rise to an unchecked warning at compile time. An *unchecked warning* is generated if, either at compile time (within the limits of the compile-time type checking rules) or at runtime, the correctness of an operation involving a parameterized type (e.g., a cast or method call) cannot be verified. For example, heap pollution occurs when mixing raw types and parameterized types or when performing unchecked casts.

In normal situations, when all code is compiled at the same time, the compiler issues an unchecked warning to draw your attention to potential heap pollution. If you compile sections of your code separately, it is difficult to detect the potential risk of heap pollution. If you ensure that your code compiles without warnings, then no heap pollution can occur.

Potential Vulnerabilities of Varargs Methods with Nonreifiable Formal Parameters

Generic methods that include vararg input parameters can cause heap pollution. Consider the following `ArrayBuilder` class:

```
public class ArrayBuilder {

  public static <T> void addToList (List<T> listArg, T . . . elements)
{
    for (T x : elements) {
      listArg.add(x);
    }
  }

  public static void faultyMethod(List<String> . . . l) {
    Object[] objectArray = l;     // Valid
    objectArray[0] = Arrays.asList(42);
    String s = l[0].get(0);       // ClassCastException thrown here
  }

}
```

The following example, `HeapPollutionExample`, uses the `ArrayBuiler` class:

```
public class HeapPollutionExample {

  public static void main(String[] args) {

    List<String> stringListA = new ArrayList<String>();
    List<String> stringListB = new ArrayList<String>();

    ArrayBuilder.addToList(stringListA, "Seven", "Eight", "Nine");
    ArrayBuilder.addToList(stringListA, "Ten", "Eleven", "Twelve");
    List<List<String>> listOfStringLists =
      new ArrayList<List<String>>();
    ArrayBuilder.addToList(listOfStringLists,
      stringListA, stringListB);

    ArrayBuilder.faultyMethod(Arrays.asList("Hello!"),
                Arrays.asList("World!"));
  }
}
```

When this is compiled, the following warning is produced by the definition of the `ArrayBuilder.addToList` method:

```
warning: [varargs] Possible heap pollution from parameterized vararg type T
```

When the compiler encounters a varargs method, it translates the varargs formal parameter into an array. However, the Java programming language does not permit the creation of arrays of parameterized types. In the method `ArrayBuilder.addToList`, the compiler translates the varargs formal parameter `T ... elements` to the formal parameter `T[] elements`, an array. However, because of type erasure, the compiler converts the varargs formal parameter to `Object[] elements`. Consequently, there is a possibility of heap pollution.

The following statement assigns the varargs formal parameter `l` to the `Object` array `objectArgs`:

```
Object[] objectArray = l;
```


This statement can potentially introduce heap pollution. A value that does match the parameterized type of the varargs formal parameter `l` can be assigned to the variable `objectArray` and thus can be assigned to `l`. However, the compiler does not generate an unchecked warning at this statement. The compiler has already generated a warning when it translated the varargs formal parameter `List<String> ... l` to the formal parameter `List[] l`. This statement is valid; the variable `l` has the type `List[]`, which is a subtype of `Object[]`.

Consequently, the compiler does not issue a warning or error if you assign a `List` object of any type to any array component of the `objectArray` array as shown by this statement:

```
objectArray[0] = Arrays.asList(42);
```

The first array component of the `objectArray` array is assigned with a `List` object that contains one object of type `Integer`.

Suppose you invoke `ArrayBuilder.faultyArray` with the following statement:

```
ArrayBuilder.faultyMethod(Arrays.asList("Hello!"), Arrays.asList("World!"));
```

At runtime, the Java VM throws a `ClassCastException` at the following statement:

```
// ClassCastException thrown here
String s = l[0].get(0);
```

The object stored in the first array component of the variable `l` has the type `List<Integer>`, but this statement is expecting an object of type `List<String>`.

Prevent Warnings from Varargs Methods with Nonreifiable Formal Parameters

If you declare a varargs method that has parameters of a parameterized type and you ensure that the body of the method does not throw a `ClassCastException` or other similar exception due to improper handling of the varargs formal parameter, you can prevent the warning that the compiler generates for these kinds of varargs methods by adding the following annotation to static and nonconstructor method declarations:

```
@SafeVarargs
```

The `@SafeVarargs` annotation is a documented part of the method's contract; this annotation asserts that the implementation of the method will not improperly handle the varargs formal parameter.

6

It is also possible, though less desirable, to suppress such warnings by adding the following to the method declaration:

```
@SuppressWarnings({"unchecked", "varargs"})
```

However, this approach does not suppress warnings generated from the method's call site. If you are unfamiliar with the `@SuppressWarnings` syntax, see Chapter 4, "Annotations."

Restrictions on Generics

To use Java generics effectively, you must consider the following restrictions.

Cannot Instantiate Generic Types with Primitive Types

Consider the following parameterized type:

```
class Pair<K, V> {

    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    // . . .
}
```

When creating a `Pair` object, you cannot substitute a primitive type for the type parameter `K` or `V`:

```
Pair<int, char> p = new Pair<>(8, 'a');  // compile-time error
```

You can only substitute nonprimitive types for the type parameters `K` and `V`:

```
Pair<Integer, Character> p = new Pair<>(8, 'a');
```

Note that the Java compiler autoboxes `8` to `Integer.valueOf(8)` and 'a' to `Character('a')`:

```
Pair<Integer, Character> p = new Pair<>(Integer.valueOf(8), new Character('a'));
```

For more information on autoboxing, see Chapter 8.

Cannot Create Instances of Type Parameters

You cannot create an instance of a type parameter. For example, the following code causes a compile-time error:

```
public static <E> void append(List<E> list) {
    E elem = new E();  // compile-time error
    list.add(elem);
}
```

As a workaround, you can create an object of a type parameter through reflection:

```
public static <E> void append(List<E> list, Class<E> cls) throws Exception
{
    E elem = cls.newInstance();   // OK
    list.add(elem);
}
```

You can invoke the `append` method as follows:

```
List<String> ls = new ArrayList<>();
append(ls, String.class);
```

Cannot Declare Static Fields Whose Types Are Type Parameters

A class's static field is a class-level variable shared by all nonstatic objects of the class. Hence, static fields of type parameters are not allowed. Consider the following class:

```
public class MobileDevice<T> {
    private static T os;

    // . . .
}
```

If static fields of type parameters were allowed, then the following code would be confused:

```
MobileDevice<Smartphone> phone = new MobileDevice<>();
MobileDevice<Pager> pager = new MobileDevice<>();
MobileDevice<TabletPC> pc = new MobileDevice<>();
```

Because the static field `os` is shared by `phone`, `pager`, and `pc`, what is the actual type of `os`? It cannot be `Smartphone`, `Pager`, and `TabletPC` at the same time. You cannot, therefore, create static fields of type parameters.

Cannot Use Casts or instanceof with Parameterized Types

Because the Java compiler erases all type parameters in generic code, you cannot verify which parameterized type for a generic type is being used at runtime:

```
public static <E> void rtti(List<E> list) {
    if (list instanceof ArrayList<Integer>) {  // compile-time error
        // . . .
    }
}
```

The set of parameterized types passed to the `rtti` method is as follows:

```
S = { ArrayList<Integer>, ArrayList<String> LinkedList<Character>, . . . }
```

The runtime does not keep track of type parameters, so it cannot tell the difference between an `ArrayList<Integer>` and an `ArrayList<String>`. The most you can do is to use an unbounded wildcard to verify that the list is an `ArrayList`:

```
public static void rtti(List<?> list) {
    if (list instanceof ArrayList<?>) {  // OK; instanceof requires a
                                         // reifiable type
        // . . .
    }
}
```

Typically, you cannot cast to a parameterized type unless it is parameterized by unbounded wildcards. Here is an example:

```
List<Integer> li = new ArrayList<>();
List<Number>  ln = (List<Number>) li;  // compile-time error
```

However, in some cases, the compiler knows that a type parameter is always valid and allows the cast. Here's an example:

```
List<String> l1 = . . . ;
ArrayList<String> l2 = (ArrayList<String>)l1;  // OK
```

Cannot Create Arrays of Parameterized Types

You cannot create arrays of parameterized types. For example, the following code does not compile:

```
List<Integer>[] arrayOfLists = new List<Integer>[2];  // compile-time error
```

The following code illustrates what happens when different types are inserted into an array:


```
Object[] strings = new String[2];
strings[0] = "hi";   // OK
strings[1] = 100;    // An ArrayStoreException is thrown.
```

If you try the same thing with a generic list, there would be a problem:

```
Object[] stringLists = new List<String>[];  // compiler error, but pretend
                                            // it's allowed
stringLists[0] = new ArrayList<String>();   // OK
stringLists[1] = new ArrayList<Integer>();  // An ArrayStoreException
                                            // should be thrown,
                                            // but the runtime can't detect it.
```

If arrays of parameterized lists were allowed, the previous code would fail to throw the desired `ArrayStoreException`.

Cannot Create, Catch, or Throw Objects of Parameterized Types

A generic class cannot extend the `Throwable` class directly or indirectly. For example, the following classes will not compile:

```
// Extends Throwable indirectly
class MathException<T> extends Exception { /* . . . */ }    // compile-time error

// Extends Throwable directly
class QueueFullException<T> extends Throwable { /* . . . */ // compile-time error
```

A method cannot catch an instance of a type parameter:

```
public static <T extends Exception, J> void execute(List<J> jobs) {
    try {
        for (J job : jobs)
            // . . .
    } catch (T e) {   // compile-time error
        // . . .
```

```
    }
}
```

You can, however, use a type parameter in a `throws` clause:

```
class Parser<T extends Exception> {
    public void parse(File file) throws T {     // OK
        // . . .
    }
}
```

6

Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type

A class cannot have two overloaded methods that will have the same signature after type erasure:

```
public class Example {
    public void print(Set<String> strSet) { }
    public void print(Set<Integer> intSet) { }
}
```

The overloads would all share the same `classfile` representation, which will generate a compile-time error.

Questions and Exercises: Generics

Questions

1. Will the following class compile? If not, why?

```
public final class Algorithm {
    public static T max(T x, T y) {
        return x > y ? x : y;
    }
}
```

2. If the compiler erases all type parameters at compile time, why should you use generics?
3. What is the following class converted to after type erasure?

```
public class Pair<K, V> {


    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey(); { return key; }
```

```
        public V getValue(); { return value; }

        public void setKey(K key)     { this.key = key; }
        public void setValue(V value) { this.value = value; }

        private K key;
        private V value;
    }
```

4. What is the following method converted to after type erasure?

```
    public static <T extends Comparable<T>>
        int findFirstGreaterThan(T[] at, T elem) {
        // . . .
    }
```


5. Will the following method compile? If not, why?

```
    public static void print(List<? extends Number> list) {
        for (Number n : list)
            System.out.print(n + " ");
        System.out.println();
    }
```

6. Will the following class compile? If not, why?

```
    public class Singleton<T> {


        public static T getInstance() {
            if (instance == null)
                instance = new Singleton<T>();

            return instance;
        }

        private static T instance = null;
    }
```

7. Review the following classes:

```
    class Shape { /* . . . */ }
    class Circle extends Shape { /* . . . */ }
    class Rectangle extends Shape { /* . . . */ }

    class Node<T> { /* . . . */ }
```

Will the following code compile? If not, why?

```
    Node<Circle> nc = new Node<>();
    Node<Shape>  ns = nc;
```

8. Consider this class:

```
    class Node<T> implements Comparable<T> {
        public int compareTo(T obj) { /* . . . */ }
        // . . .
    }
```

Will the following code compile? If not, why?

```
Node<String> node = new Node<>();
Comparable<String> comp = node;
```

Exercises

1. Write a generic method to count the number of elements in a collection that have a specific property (e.g., odd integers, prime numbers, palindromes).
2. Write a generic method to exchange the positions of two different elements in an array.
3. Write a generic method to find the maximal element in the range `[begin, end]` of a list.

Answers

You can find answers to these questions and exercises at `http://docs.oracle.com/javase/tutorial/java/generics/QandE/generics-answers.html`.

Index

Symbols

- (minus sign)
  operator, 56–59, 224–225
  in regular expressions, 488–490
-- operator, 56, 58, 63
_ (underscore)
  in constant names, 114
  in numeric literals, 48–49, 688
  in package names, 207
  in predefined character classes, 492
  in variable names, 43–44
, (comma)
  in numbers, 48, 222, 225
  in regular expressions, 498
; (semicolon)
  in class paths, 213, 536
  declaring abstract methods, 158
  in statements, 25, 66, 267
  listing enum types, 126
  terminating method signatures, 136, 138
: (colon), in class paths, 213
! operator, 58, 63
!/ separator, 569
!= operator, 56, 59–60
? (question mark)
  in regular expressions, 179–182, 323
?: operator, 56, 60
/ (forward slash)
  file name separator, 211, 310, 364, 530, 548
  operator, 56–57, 224–225
// in comments, 11, 23
/* in comments, 23
/** in comments, 23
/= operator, 56
. (dot)
  in class paths, 314–315, 539
  in JAR file commands, 546, 557
  in method invocations, 85, 103–106
  in numbers, 225, 234, 246, 299
  in regular expressions, 487, 494
  in variable names, 104
. . . (ellipsis), 95
^ (caret)
  operator, 56, 62–63
  in regular expressions, 489, 502–503, 504
^= operator, 56
~ operator, 56, 63
' (single quote), escape sequence for, 48, 235
" (double quote)
  escape sequence for, 49, 235
  in literals, 236
() (parentheses)
  in annotations, 130
  in declarations, 90, 104–105, 266
  in expressions, 66, 677
  in generics, 166
  in regular expressions, 499–501
[] (square brackets)
  in arrays, 51–52, 386
  in regular expressions, 323, 488, 519
{} (braces)
  in blocks, 67, 69, 116
  in declarations, 87–88, 90–91
  in methods, 42
  in regular expressions, 323, 497–498

@ (at)
  in annotations, 129–130
  in Javadoc, 132
$ (dollar sign)
  in DecimalFormat patterns, 225
  in variable names, 43
* (asterisk)
  in import statements, 208
  operator, 56–57, 224–225
  in regular expressions, 322, 354
*/ in comments, 24
*= operator, 56
\ (backslash)
  in escape sequences, 48, 235, 310, 493
  file name separator, 211, 364, 530
  in regular expressions, 323, 488, 502
& (ampersand) operator, 56, 62–63
&& operator, 56, 60, 63
&= operator, 56
(pound sign)
  in DecimalFormat patterns, 225
  in regular expressions, 487
% (percent sign)
  format specifier, 221–223, 301
  operator, 56–57, 224–225
%= operator, 56
+ (plus sign)
  operator, 56–58, 63, 224–225
  in regular expressions, 487
++ operator, 56, 58–59, 63, 66
+= operator, 56
< operator, 56, 59, 630
<< operator, 56, 62, 63
<<= operator, 56
<= operator, 56, 59, 63
<> (angle brackets), 165, 167, 170–171, 176
= operator, 56
-= operator, 56
== operator, 56, 59, 63, 677
> operator, 56, 59, 63
>= operator, 56, 59, 63
>> operator, 56, 62, 63
>>= operator, 56
>>> operator, 55–56, 62, 63
>>>= operator, 56
| (vertical bar)
  in exception handling, 264
  operator, 56, 62–63
  in regular expressions, 506
|= operator, 56
|| operator, 56, 60, 63

A

abs method, 226
abstract classes, 158–161
  example, 159–160
  as an implementation of a service, 529
  implementations, 439–440
  methods, 158–161
  numeric wrapper classes, 218
  versus interfaces, 159, 161
Abstract Window Toolkit (AWT), 209, 578, 599
access control list (ACL), 329
access modifiers
  classes and, 87–90, 111
  constants and, 138
  default, 110
  fields and, 146
  interfaces and, 138
  levels of, 110–111
  methods and, 90, 146, 152
  private keyword, 89, 110–111, 146
  protected, 110–111
  public, 89, 110–111, 138
AccessControlException, 534–535
accessor methods, 237, 242, 244, 330, 410, 415
acos method, 228
add method, 377, 381, 391, 395
addAll method
  in the Collection interface, 376, 379, 436
  generics and, 178
  in the List interface, 385, 387, 430
  in the Map interface, 400
  in the Set interface, 380, 383–384
addFirst method, 423
addLast method, 423
algorithms (collections), 371–374, 432–437
  in the Collections class, 435–436
  composition, 436
  finding extreme values with, 437
  generic, 164, 173, 176
  interoperability and, 443
  listing data, 394
  polymorphism, defined, 372
  routine data manipulation, 435–436
  searching data, 436
  shuffling data, 435
  signature block files and, 563
  sorting data, 433–435
  type inference, 176, 179
  work stealing, 475
ampersand. *See* &
Anagrams example, 404, 434
angle brackets. *See* <>

annotations, 129–134
    documentation, 130–131
    processing, 133
    used by the compiler, 132–133
APIs (Application Programming Interfaces)
    array-based versus collection-based, 430
    compatibility of, 440–442
    design of, 443–444
    interfaces as, 137
    JAR-related, 567–573
    Java core, 4, 32, 38
    logging, 277
`append` method, 197, 250
`appendReplacement` method, 511, 514–515
`appendTail` method, 511, 514–515
`Applet` class, 591, 595, 599, 606–607, 609
`applet` tag, 552, 600–602
    deploying with, 601–602
    JAR files and, 552
    JNLP and, 602, 653–655, 667
    manually coding, 602
`AppletContext` interface, 603, 607–608, 627, 631
applets, 591–634
    API of, 597, 603
    background color of, 653
    capabilities of, 603–617, 631
    classes and subclasses, 592–593
    common problems, 632–633
    communicating with other applets, 626–628
    core functionality versus deployment mechanism, 600
    debugging, 623
    defining and using applet parameters, 604–606
    deploying, 600–602, 650–655
    developing, 597–600
    directories of, 600–601, 604
    displaying customized loading progress indicator, 618–632
    displaying documents, 607–608
    displaying short status strings, 606–607
    draggable, 623–626
    event handling and, 614–616
    execution environment of, 596–597
    final cleanup, 594, 596
    finding and loading data files, 603–604
    GUIs in, 598–600
    JavaScript functions and, 597, 601–614
    leaving and returning to web pages, 596
    life cycle of, 594–496
    loading, 595–596
    milestones, 593–594
    packing in JAR files, 600–601
    parameters in, 604–606
    qualified names, 584, 622
    quitting the browser, 596
    reloading, 596
    security and, 522, 603
    server-side applications, 628–630
    signed, 632
    threads in, 596
    unsigned, 631–632
`appletviewer` application, 534
applications. *See* rich Internet applications (RIAs)
`archive` attribute, 633, 652
`args` variable, 43, 221, 527, 572
arguments
    arbitrary number of, 95–96
    command-line. *See* command-line arguments
    glob, 322–323
    number of, 93
    primitive data types, 96–97
    reference data types, 97
    versus parameters, 94
arithmetic operators, 56–58, 62–63
`ArithmeticDemo` example, 57, 64
`ArrayBlockingQueue` class, 426
`arraycopy` method, 53–54, 423, 535, 572
`ArrayCopyDemo` example, 53–54
`ArrayDemo` example, 49–50, 52
`ArrayIndexOutOfBoundsException`, 265, 269–273
    `Collection` interface and, 377–378
    compared to `Vector`, 423
    exception handling with, 261
    general-pupose `List` implementation, 385, 387, 392, 419, 423
    generics and, 175–176
    immutable multi-copy list, 430
    initial capacity and, 423
    linear time and positional access with, 423
    parameterized types error, 199
    speed of, 423, 437
    type parameters and, 175–176, 198–199
    wrappers and, 428
arrays, 49–54
    assigning values to, 52
    of characters, 236–237
    copying, 53–54
    creating, 52
    length of, 50, 53, 477
    `List` view of, 429–430
    looping through, 82
    multidimensional, 52–53
    of strings, 243, 379, 442, 526

asin method, 228
  as a convenience implementation, 429–430
  upward compatibility and, 441
  writing a custom implementation, 438
asList method, 180–181, 388–389
assert statement, 119, 684
assignments
  checking with assert, 119
  compound, 57, 64
  conditional operators and, 61
asterisk. *See* *
at. *See* @
atan method, 228–229
atomic file operations, 322, 327, 341, 459, 478
  access, 459
  actions, 459
  methods, 425
  synchronization, 428, 459, 478
  variables, 469, 478–479, 687, 690
ATOMIC_MOVE enum, 322, 327
AtomicCounter example, 479
AtomicInteger class, 218, 479
AtomicLong class, 218
Attributes class, 569–570
autoboxing, 167, 197, 230–232, 234, 683
AutoCloseable interface, 266, 269
autoflush, 296

B

backslash. *See* \
backspace, 48, 235
BadThreads example, 480
BasicMathDemo example, 225–226
BasicService interface, 589
Bicycle class, 33–93, 107, 111–115, 144–147, 150–153
BicycleDemo example, 35
BigDecimal class, 45, 218, 298, 307
BigInteger class, 218, 298
binary numbers, 47, 219–220
binarySearch method, 394, 436
bit shift operators, 62, 63, 677
BitDemo example, 62
bitwise operators, 62, 63, 506, 677
  precedence, 56
BlockDemo example, 67
BlockingQueue implementation, 396, 420, 426–427, 477
blocks, 67
boolean data type, 45
  default value of, 46
  unary operations on, 58
BorderLayout class, 577, 582–583, 598–599, 619–620
boxing. *See* autoboxing; unboxing
braces. *See* {}
branching statements, 41, 68, 78–82, 678
break statements, 71–72, 78–80
BreakDemo example, 78–79
BreakWithLabelDemo example, 79–80
browsers. *See* web browsers
BufferedInputStream class, 296, 306
BufferedOutputStream class, 296, 306, 338
BufferedReader class, 266–267, 294–298, 337–338
BufferedWriter class, 267–268, 296, 320–321, 337
buffers, 295–296
bugs. *See* errors
byte data type, 44–45
  data streams and, 305
  default value of, 46
  switch statement and, 70
byte streams, 290–293
  buffered, 296
  character streams and, 293–294
  classes, 299
  closing, 292
  I/O streams and, 303, 306
  standard streams and, 303
  using, 291–292
  when not to use, 292
bytecodes, 2, 5, 7, 18
  in the HelloWorld example, 12, 15, 18, 20, 27–28
  type erasure and, 188
byteValue method, 219, 232

C

Calendar class, 307–308, 535
call stack
  exception handling, 258–259, 263, 270, 272, 283
  propagating errors up, 282–283
Callable objects, 473–474, 690
capturing groups, 499, 500–502
catch blocks, 263–264, 270, 277, 285, 321
Catch or Specify Requirement, 257–259, 680
  bypassing, 260
cd command, 16, 20–21, 27–28
ceil method, 226
char data type, 45
  character and string literals, 47–48, 236
  converting to strings, 236

in data streams, 305
default value of, 46
escape sequences in, 48, 235
generic methods and bounded type parameters, 173
getting by index, 242
translating individual tokens, 298
wrapper class. *See* Character class
Character class, 234–235, 253, 488
implementing Comparable, 406
restrictions on generics, 197
switch statement and, 70
useful methods in, 235
as a wrapper class, 234
character classes, 488–495
intersections of, 419–420
negation of, 489
predefined, 492–495
quantifiers and, 498–499
ranges of, 489–490
regular expressions and, 488, 493
simple, 488–489
subtractions of, 420
unions of, 490–491
character streams, 293–295
charAt method, 80–81, 237, 242–243, 252–254
CharSequence interface, 142–143, 243–245, 249, 337, 507, 509
ChessAlgorithm example, 158
Class class, 157
class files, 157
class library. *See* Java API
class paths, 212, 213, 529–530, 539, 557–558, 611, 627, 643, 664
class variables. *See* fields, static
ClassCastException, 191–192, 194–196, 406–407, 409, 429
classes, 34–35, 86–87, 117–118. *See also* inheritance; nested classes
abstract, 158–161
access modifiers and, 89–90
adapter, 444
base or parent. *See* superclasses
child, derived, or extended. *See* subclasses
constructors for, 92–93
declaring, 87–88
final, 158
hierarchy of, 144
instantiating, 100
interfaces implemented by, 88, 139
methods and, 90–92
naming, 90
numbers, 218–220
passing information, 94–97
static initialization blocks in, 116
variables (static fields), 42, 54, 111–114, 116–118
wrapper, 218–219, 230–232, 683
ClassNotFoundException, 307–308, 570, 572, 633
CLASSPATH system variable, 27–28, 213–214, 521, 535, 537, 539–540, 632
ClipboardService interface, 639
clone method, 155–156, 420
Cloneable interface, 155, 159
CloneNotSupportedException, 154–155
close method, 268–269, 297, 320
Closeable interface, 261, 266, 269, 320
cmd command, 16
code
case sensitivity in, 12, 16, 21, 43
error handling, 280–282
error-prone, 72, 105, 265, 285
platform-independent, 4, 29
readability of, 28, 92, 121, 211, 265, 279
codebase attribute, 652, 655, 657
CollationKey class, 406
Collection interface, 374–376, 386
array operations, 379–380
backward compatibility and, 442
bulk operations, 378–379, 480
implementations of, 429
views, 400–403, 416, 428
wrappers for, 432
collections, 371–444
concurrent, 477–478
elements of. *See* elements filtering
hierarchy of, 209
older APIs and, 169, 379, 440
ordered, 375–376, 384, 417, 424
read-only access to, 429
synchronized, 420, 427–428
traversing, 377–378
Collections class, 175, 394, 435, 437
backward compatibility and, 168–169, 174
methods in, 427, 431–433
polymorphic algorithms in, 294
colon. *See* :
comma. *See* ,

command-line arguments, 240–241, 526–527
  analogies to applet parameters, 604
  echoing, 527
  numeric, 526–527
  test harnesses and, 485
  URLs and, 571–572
comments, 23
  annotations and, 130, 132
  Pattern class methods, 504, 506
Comparable interface, 406–409
Comparator interface, 409–418, 426, 434, 436–437, 683
compare method, 409
compareTo method
  custom uses, 406–411
  legacy file I/O, 386
  for objects, 319
  for primitive data types, 219
  for strings, 248
compareToIgnoreCase method, 248
ComparisonDemo example, 59–60
comparisons
  between classes, 406
  of numbers, 56–60
  of object, 61–62
compatibility, 441–443
  backward, 442–443
  upward, 441–442
compile method, 485, 504
compilers
  error handling. *See* errors, compile-time
  information for, 129, 147
  verifying types of objects, 374
ComputeResult class, 254
concat method, 238
ConcatDemo example, 57–58
concurrency, 445–480
  collections, 477–478
  high-level objects, 469–480
  random numbers, 480
ConcurrentHashMap implementation, 425, 478
ConcurrentMap interface, 420, 425, 478
ConcurrentNavigableMap interface, 478
ConcurrentSkipListMap interface, 478
conditional operators, 60–62, 63, 677
ConditionalDemo1 example, 60
ConditionalDemo2 example, 60
constants, 114
  compile-time, 114
  data streams and, 305
  embedded flag expressions, 506
  empty, 431
  enum types and, 126–128
  importing, 210–211
  interfaces and, 136, 138
  naming, 44, 126
  numbers and, 225, 232
  for upper and lower bounds, 219
constructors, 85–87, 94–98
  calling, 108–109
  chaining, 154
  conversion, 376
  declaring, 87, 92–99
  default, 93
  for enum types, 127
  generic, 178–179
  inheritance and, 144, 153–154
  methods and, 93, 158
  no-argument, 93, 102, 109, 153, 154
  synchronization and, 456
containers. *See* collections
contains method, 244, 422
containsAll method, 378, 383, 401–402, 423
containsKey method, 398, 525
containsValue method, 398
continue statements, 80–81
ContinueDemo example, 80
ContinueWithLabelDemo example, 81
control flow statements, 68–83
  branching, 78–82
  decision-making, 68–72
controlling access. *See* access modifiers
converters, 221–223
cookies
  accessing, 644–646
  kinds of, 643
  rich Internet applications (RIAs) and, 643–646
copy method, 187, 326
CopyBytes example, 291–292, 294
CopyCharacters example, 293–294, 296
CopyLines example, 295
CopyOnWriteArrayList implementation, 423
CopyOnWriteArraySet implementation, 422–423
core collection interfaces, 371, 374–376, 417. *See also* by individual type
  compatibility of, 441
  hierarchy of, 209
  implementations of, 427–429
cos method, 225, 228
Countdown example, 396
Counter example, 453
CreateObjectDemo example, 98–99, 104–105
createTempFile method, 341, 368

currentTimeMillis method, 535
customized loading screens
  in applets, 653
  in Java Web Start applications, 581–585

D

data encapsulation, 31, 33
data types, 41–44, 46, 94, 292. *See also* by individual type
  advanced, 690
  reference, 94
  returned by expressions, 65, 90
  switch statement and, 70
DataInput interface, 305–307
DataInputStream class, 305–306
DataOutput interface, 305, 307
DataOutputStream class, 305, 306
DataStreams example, 305–307
dates, 143, 223
Deadlock example, 460
deadlocks, 420, 459–461, 470
Deal example, 392–393
decimal number system, 45, 47, 219–220
DecimalFormat class, 218, 224–225
declaration statements, 67, 266
declarations. *See* by individual type
decode method, 220
DelayQueue class, 426
delete method, 365
deleteOnExit method, 367
deployment, 649–674
  applets, 650–655
  best practices, 667–673
  Java Web Start applications, 655–658
  usage scenarios, 651–652
Deployment Toolkit, 579–580, 592, 600–602
@Deprecated annotation type, 132–133
@deprecated Javadoc tag, 132
destroy method, 594
diamond, 166–167, 177–179, 683, 689
dir command, 18, 27
directories
  changing, 16–18, 27
  checking, 324–325
  copying, 325–326
  creating, 19–21, 343–344
  current, 16, 19
  deleting, 325
  delimiters, 310
  error messages involving, 27
  filtering, 345–346
  home, 16, 19
  listing contents, 18, 344–345
  moving, 326–327
  packages, 211–212
  root, 313, 343
  temporary, 344
  verifying the existence of, 325
  watching for changes, 357–363
documentation, 130–131
  source code comments, 23
dollar sign. *See* $
dot. *See* .
double data type, 45
double quote. *See* "
doubleValue method, 219
do-while statements, 51, 68, 75–76, 78–80, 82, 678
DoWhileDemo example, 76
DownloadService interface, 589, 639
DownloadServiceListener interface, 582, 618–619, 639

E

E
  constant, 225
  in scientific notation, 47
  as type parameter naming convention, 166
Echo example, 526–527
element method, 395
elementAt method, 265, 385–387
elements (in collections), 375–376
  adding, 377, 379, 387
  checking, 378
  counting, 377, 381
  cursor position and, 390
  not duplicated, 381
  null, 379
  ordering, 375, 380–381
  removing, 377–379, 381, 386
  searching, 381
  swapping, 388
ellipsis. *See* . . .
else statement, 69, 71
emacs text editor, 19
emptyList method, 431
emptyMap method, 431
emptySet method, 431
EmptyStackException, 274
encapsulation, 212
end method, 511–512
endsWith method, 248, 319
EnhancedForDemo example, 78
ensureCapacity method, 250, 428

entrySet method, 397, 400–402, 440
enum keyword, 126
enum types, 70, 126–129, 422
constructors for, 127–128
naming, 126
enumerated types. *See* enum types
Enumeration collection, 203, 398, 525
compatibility and, 441–442
EnumMap implementation, 424–425
EnumSet implementation, 422
EnumTest example, 126–127
Env example, 528
environment, 521–527
properties of, 522–526
restricted, 586, 646
environment variables, 527–529, 535–540
CLASSPATH, 27, 28, 539
common problems with, 27
passing to new processes, 528
PATH, 535–539
platform dependency issues, 528–529
querying, 527–528
EnvMap example, 527–528
EOFException, 284, 306–307
equality operators. *See* comparisons
equals method, 156, 157, 408, 411, 425
equalsIgnoreCase method, 248
Error class, 275
error messages, 261–263
legacy file I/O code, 265
Microsoft Windows, 25, 27
unchecked, 169–170
UNIX systems, 25, 27
using to check assignments, 119
wildcard capture and, 184
errors
compiler, 27–28
compile-time, 46, 93, 147, 149, 154, 164, 172–173, 183, 197
grouping and differentiating types, 283–285
memory consistency, 453, 454–456, 459, 469, 687, 690
propagating in, 282–283
runtime, 27–28
semantic, 26–27
syntax, 26
escape sequences, 48, 235–236
in regular expressions, 505, 508
in Unicode, 48, 517
exception classes, 257, 273–278
creating, 277–279
grouping errors, 283–284
hierarchy, 278
PatternSyntaxException class, 515–517
exception handlers, 43, 258–266
associating with try blocks, 262–263
catching more than one exception type, 264
catching multiple exceptions, 268
constructing, 269–272
exceptions, 257–287
advantages of, 279–280
catching, 261–272
chained, 276–277
checked, 260–261
class hierarchy of, 278
creating exception classes, 277–279
external. *See* errors
in file operations, 320–321
kinds of, 259–260
logging, 257, 277, 623
specifying by method, 272–279
suppressed, 268–269
throwing, 273–279
unchecked, 279–280
exclamation sign. *See* !
Executor interface, 472–473, 687, 690
ExecutorService interface, 472–475
exit method, 535
exp method, 227–228
ExponentialDemo example, 227–228
exponents, 210
expression statements, 66–68
expressions, 65–66
ExtendedService interface, 589, 639
extends keyword, 36, 171, 179, 182, 187
extensions, 529, 539, 541, 600, 662

F

F or f in 32-bit float literals, 47
fields, 32, 152, 676, 678. *See also* variables
declaring, 116
default values of, 46
final, 210, 456
hiding, 152, 161
inherited, 146
initializing, 115–117, 676
members versus, 43
nonstatic, 42, 54
private, 146
qualified names, 96
referencing, 103–104
shadowing, 96
static, 42, 112–113, 196–198
static final. *See* constants

synchronization and, 453, 457
FIFO (first-in, first out), 375, 395
implementations of, 420, 426, 477–478
Queue interface and, 417
File class, 365
file descriptors, 157
file operations, 320–324
atomic, 322
catching exceptions, 320–321
method chaining, 322
releasing system resources, 320
varargs in, 321–322
file paths, 309–312
checking symbolic links, 325
comparing, 319–320
converting, 316–317
creating a path between two, 318–319
creating, 313
joining two, 317–318
relative versus absolute, 311
removing redundancies from, 314–316
retrieving information about, 314
symbolic links and, 311–312
FileInputStream class, 291, 294
Filename class, 245
FilenameDemo example, 246
FileNotFoundException, 260, 263–265, 283–284
FileOpenService interface, 639–641
FileOutputStream class, 291, 294
FileReader class, 260, 293–298
files
accessibility of, 325
basic attributes, 330
checking, 324–325
copying, 325–326
creating, 338, 340
deleting, 325
DOS attributes, 331
file stores, 364
finding, 354–357
I/O and, 309–368
moving, 326–327
POSIX file permissions, 331–333
random access, 309, 342–343, 367, 685
reading, 338–340
setting ownership, 333
temporary, 341–342
time stamps, 330–331
user-defined attributes, 333–334
verifying the existence of, 325
writing, 338–240
FileSaveService interface, 639–642
FileSystem class, 323, 335, 354, 359, 366
FileVisitor interface, 349–353
FileWriter class, 261–262, 265, 269–273, 293–295
final modifier, 114, 138
finalize method, 154, 157
finally block, 262, 264–266
find method, 510
FindDups example, 381–383, 418
FindDups2 example, 383
first method, 414–415
float data type, 45
default value of, 46
floatValue method, 240
floor method, 226–227
flush method, 296
for statement, 76–78, 82
enhanced, 78, 343, 364
nested, 79
skipping the current iteration, 80
terminating, 77
ForDemo example, 77–78
for-each construct, 127, 377–278, 400
fork/join framework, 475–477, 687, 691
form feed, 48, 235
Format example, 302
format method, 221–222, 225, 232, 239, 299–302
format specifiers, 221, 239, 299–302
format strings, 221, 299–300
Formatter class, 221
formatting
numeric print output, 220–224
stream objects, 296, 299–302
forward slash. *See* /
frequency method, 398–399
functions. *See* methods
Future object, 473

G

garbage collection, 5, 105–106, 118, 120
empty references and, 154
immutable objects and, 465–466
memory leaks and, 5
weak references and, 425
generic methods, 170–171
generic objects, 163–202
bounded type parameters and, 173
erasure of, 188–190
instantiating, 166, 177–178
invoking, 166
subtyping and, 175–176
type inference and, 176–179

generic types, 164–170, 183, 188, 192–193, 374, 683
get method, 157, 261–262, 313
getAbsolutePath method, 368
getApplet method, 627
getCanonicalPath method, 368
getCause method, 276
getChars method, 238
getClass method, 154, 157, 161, 171
getCodeBase method, 603–604, 629
getDescription method, 515–516
getEnv method, 527–528
getFields method, 157
getFirst method, 423
getHost method, 629
getImage method, 603–604
getIndex method, 515–516
getInterfaces method, 157
getLast method, 423
getMainAttributes method, 568–570
getMainClassName method, 568–570, 572
getMessage method, 321, 516
getName method, 315, 316
getParameter method, 606
getParent method, 314–317
getPattern method, 516
getProperty method, 525, 531, 635–637, 665
getResource method, 579
getSecurityManager method, 533–534
getSimpleName method, 157
getStackTrace method, 277
getSuperclass method, 157
getValue method, 570
globbing, 322–323, 344–346, 685, 692
  filtering a directory listing, 345–346
  finding files, 353–357
graphical user interfaces (GUIs), 4, 38, 98, 373, 576–578, 591–593, 597–600
groupCount method, 501
guarded blocks, 461–465

H

hard links, 309, 347–348
hashCode method, 154, 156–157, 380, 387, 396, 399, 407–408
HashMap implementation, 177–178, 397–400, 424–425
HashSet implementation, 380–384, 419, 421–422
Hashtable collection, 375, 417, 522, 525–526
  comparison to Map, 397–398
  compatibility and, 441–442
  concurrency through ConcurrentHashMap, 425
  synchronization and, 420
hasNext method, 377–378, 389–391
headSet method, 412–415
heap pollution, 193–195
HelloRunnable example, 447–448
HelloThread example, 448
HelloWorld, 6–24, 591–592
  applet, 591–592
  for Microsoft Windows, 14–18
  for Solaris and Linux, 18–21
  for the NetBeans IDE, 6–13
hexadecimal number system, 47, 159, 219–220, 301
HTML. *See also* web browsers; web pages
  generated code, 650, 655
  specification, 603
HTTP requests, 474, 643
HTTPS certificates, 587

I

I/O, 289–369
  atomic, 322
  binary, 305
  buffered, 295–296
  channel, 339–340, 342
  closing, 157, 292
  from the command line, 302–305
  command-line objects, 530
  exceptions, 283–284, 320–321
  interoperability, 365–366
  line oriented, 294–295
  memory mapped, 335
  method chaining, 322
  NIO.2, 309–366
  of objects, 307–309
  of primitive data type values, 305
  random access, 342
  scanning and formatting, 296–302
  streams, 289, 290, 530. *See* streams, I/O
icons, 662
IDE projects, 7–9, 14
IdentityHashMap implementation, 424–425
IfElseDemo example, 69
if-then statements, 68–69, 82, 677
if-then-else statements, 69–71, 82, 677
IllegalAccessException, 275
IllegalStateException, 395
immutable objects, 465–469
  defining, 467–469
immutable singleton set, 430–431

ImmutableRGB example, 468–469
implementations, 418–433
  abstract, 419, 437–440
  adapter, 438
  anonymous, 427
  concurrent, 419, 425, 426–427
  convenience, 419, 429–431
  custom, 437–444
  documenting, 375
  general purpose, 418, 419, 421–422, 423, 424, 426
  special purpose, 419, 422, 423–425
  wrapper, 419, 427–429
  writing, 437–440
implements keyword, 37, 88, 139
import statement, 208–211, 681
indexOf method, 244, 278, 385–387, 390, 392
indexOfSubList method, 394
information hiding, 34
inheritance, 31–32, 36–37, 143–163, 679, 682
  example, 144–146
  multiple, 127, 137–139, 141
init method, 593–594, 598–600, 615
initCause method, 276
initializer blocks, 117
inner classes, 85, 120, 122–125
  anonymous, 122, 124
  controlling access to, 121, 125
  example, 123–125
  instantiating, 122
  Java Language Programming Certification, 679
  local, 124
  modifiers, 124–125
InputStream class, 291, 326, 338
InputStreamReader class, 294, 303
insert method, 250
insertElementAt method, 386–387
instance members, 85, 117, 121
instance variables, 42, 54, 111–114, 117–118, 121, 124
  qualified names, 118
instanceof operator, 56, 61–63, 147, 198, 617, 681
InstanceofDemo example, 61
instances, 34, 42, 100, 112
  class members and, 111–115
  inner classes and, 122
  testing, 61–62
int data type, 44
  default value of, 46
  switch statement and, 70
Integer class, 172, 220
interfaces, 37–38, 135–143
  abstract classes and, 159, 161
  as APIs, 137
  body, 138
  collection. *See* core collection
  defining, 138
  implementing, 139–140
  multiple inheritance and, 137–138
  rewriting, 142
  as a type, 141
internationalization, 293
Internet domain names, 206–207, 211
Internet Explorer. *See* web browsers
interoperability, 440–444
  API design, 443–444
  compatibility, 441–442
  with legacy code, 365–366, 367–368
interprocess communication (IPC) resources, 446
interrupt mechanism, 450–451
interrupt status, 450–451
interrupted method, 450–451
InterruptedException, 449–452, 462–465, 471, 480
intValue method, 172, 219
invokeClass method, 570–572
IOError, 260, 349
IOException, 261–273
isAnnotation method, 157
isEmpty method, 377, 380–381, 396–398
isEnum method, 157
isInterface method, 157
isInterrupted method, 450
isLetter method, 235
isLowerCase method, 235
isUpperCase method, 235
isWhitespace method, 235, 297
Iterator class, 376–381, 385, 389, 391, 400–401, 412, 416, 426
iterator method, 123, 188, 319, 345, 376–381, 387, 389, 400, 412, 416, 426, 428, 439–440
Iterator object, 377–378
iterators, 376–381
  fail-fast, 420

J

JApplet class, 578, 591–595, 598–599, 603, 606–607, 624, 636
JAppletgetCodeBase method, 603
JAppletgetDocumentBase method, 603
JAR tool, 542, 544–550, 554
  setting entry points, 556–557
JarClassLoader class, 567–569, 571–572

JarRunner example, 567–572
JarURLConnection class, 567–569
Java 2D, 4, 651
Java Application Programming Interface (API), 3–5, 32, 38, 125, 300. *See also* APIs (Application Programming Interfaces)
  hierarchy of packages, 209
  legacy, 168–169
  raw types and, 168–169
  runtime exceptions and, 260, 275, 279
Java Archive (JAR) files, 541–573
  adding classes class path, 557–558
  applets packaged in, 552
  as applications, 553
  benefits of, 541–542
  creating, 542–546
  extracting contents of, 548–550
  manifest files, 553–560
  paths in, 548
  running JAR packaged software, 552–553
  sealing, 559–560
  signing, 560–566, 671–672
  uncompressed, 545
  updating, 550–551
  using, 542–553
  using JAR-related APIs, 561
  verifying, 560–564, 566
  viewing contents of, 546–548
Java Archive Tool. *See* JAR tool
Java Cache Viewer, 585–586
Java Collections Framework. *See* collections
Java Database Connectivity (JDBC) API, 4, 686, 689–690
Java HotSpot virtual machine, 2
Java Interactive Data Language (IDL) API, 4
java launcher tool, 2, 4, 27–28, 553
Java Naming and Directory Interface (JNDI) API, 4
Java Network Launching Protocol. *See* JNLP files
Java platform, 2–5, 521–540
  API specification. *See* Java Application Programming Interface (API)
  command-line arguments, 526–527
  configuration utilities, 521–529
  environment variables, 527–529
  language, 2–4
  properties, 522–526, 530–533
  supported encodings on. *See* Unicode encoding
  system utilities, 529–535
Java Plug-In software, 596–597, 626
Java Programming Language Certification, 675–692
  Java SE 7 Upgrade Exam, 688–692
  Programmer Level I Exam, 675–680
  Programmer Level II Exam, 680–688
Java Remote Invocation (RMI), 4
Java Remote Method Invocation over Internet Inter-ORB Protocol (Java RMI-IIOP), 4
Java SE Development Kit 7. *See* JDK 7
Java SE Runtime Environment. *See* JRE
Java Virtual Machine (Java VM), 2–3, 212–214
Java Web Start applications, 575–581
  changing the launch button of, 656
  common problems, 587–588
  deploying without codebase attribute, 657–658
  deploying, 579–581, 655–658
  developing, 576–579
  displaying customized loading progress indicator, 581–585
  Java Cache Viewer, 585–586
  retrieving resources, 579
  running, 585–586
  security and, 586–587
  separating core functionality from final deployment mechanism, 578–579
  setting up web servers for, 581
java.awt packages, 206, 209–210, 618–619, 638
java.io package, 221, 260–261, 266–267, 283, 289
java.lang.Character API, 235
java.nio.* packages, 289, 309, 312, 320, 337, 347, 354, 358, 366–367
java.util.concurrent.atomic package, 478–479
java.util.concurrent.locks package, 470
java.util.jar package, 567–569
java.util.regex package, 483–485, 507–509, 511–514, 516, 519
JavaBeans, 5
javac compiler, 2, 4, 7
  case sensitivity in, 12
javadoc tool, 23
JavaFX, 4
JavaScript
  applets and, 597, 600–614
  Deployment Toolkit scripts, 579, 600
  interpreter, 581, 592, 596–597
javax.jnlp package, 576, 643
javax.swing.JApplet class, 591–593, 598–599, 603
JButton, 577, 598, 625–626
JDialog, 593
JDK 7 (Java SE Development Kit 7), 4, 6

adding to platform list, 9
annotation processing in, 133
concurrent random numbers, 480
default manifest, 554, 556
directory structure, 535–539
generics and, 168
high-level concurrency objects and, 469
JAR tool in, 542
`ThreadLocalRandom`, 469, 480
`TransferQueue` implementation, 427
Unicode 6.0 support, 517
viewing applets in, 592, 614–615
`JFrame` class, 577
JNLP files, 659–667
API, 603, 631, 635, 638–641, 648, 659, 672
common errors, 588
commonly used elements and attributes, 661–666
embedding in `Applet` tag, 653–655
encoding, 660
rich Internet applications (RIAs) and, 638–643
structure of, 660–667
`join` method, 451
`JPanel` class, 577–578, 598–599
`JProgressBar` object, 582, 583, 619, 620
JRE (Java SE Runtime Environment), 530
checking client version of, 658–659
ensuring the presence of, 672–673

K

`K` type parameter naming convention, 166
`keys` method, 525
`keySet` method, 400–403
keywords, 88. *See also* by individual type

L

`last` method, 531, 570
`lastIndexOf` method, 244–246, 385–387
`lastIndexOfSubList` method, 394
`length` method, 237, 249
line feed, 48, 294, 301
line terminators, 294–295, 297, 301, 493, 505
link awareness, 324, 347
`LinkedBlockingQueue` class, 426
`LinkedHashMap` implementation, 397, 399, 419, 424
`LinkedHashSet` implementation, 380–381, 397, 419, 421–422
`LinkedList` implementation, 198, 385, 395–396, 419–420, 423, 426, 431
links
hard, 309, 347–348
symbolic, 311–312, 315–317, 324, 347–348
Linux. *See* UNIX/Linux
`List` interface, 384–394
algorithms, 394
collection operations, 386–387
comparison to `Vector`, 385–386
implementations, 422–424
iterators, 389–391
method, 441
positional acccess and search operations, 387–389
range view operations, 391–394
`listIterator` method, 385, 389–391, 439–440
`ListOfNumbers` example, 261–262, 269–271, 272, 286
`listRoots` method, 368
lists
cursor positions in, 390–391
iterating backward, 389
literals, 41, 46–49, 238, 688
character and string, 47–48
class, 48
floating point, 47
integer, 47
using underscore characters, 48–49
LiveConnect Specification, 608–609, 611
locales, 293, 298–299, 663
`lockInterruptibly` method, 470
locks, 469, 470–472
deadlocks, 460
intrinsic, 457–458
livelocks, 461
starvation, 461
synchronization and, 457–458
logarithms, 138, 225, 227–228
logical operators, 55–62
logical bitwise, 52
`Long` class, 219, 232–233, 240, 406, 452
`long` data type, 45
default value of, 46
`longValue` method, 219
`lookingAt` method, 510, 512–513
loops, 76–81, 678
infinite, 76–78
making more compact, 78
nested, 79, 81
test harness, 485, 496
`ls` command, 21–22, 28, 181, 197, 354

M

Mac OS, 2, 7
`main` method, 23–24

manifest files, 542–545, 553–560, 563, 573
  default, 554
  digest entries, 562–563
  fetching attributes, 569–570
  modifying, 554–555
  setting application entry point, 555–557
  setting package version information, 558–559
  signature block files, 563
  signature files, 562–563
Map interface, 397–405, 417, 419, 425, 431
  basic operations, 398–399
  bulk operations, 400
  comparison to Hastable, 397–398
  implementations of, 424–425
  viewing as a Collection, 400–401
Matcher class, 484, 509–515, 684
MatcherDemo example, 511–512
MatchesLooking example, 512–513
Math class, 210, 225–229, 233
MAX_VALUE constant, 219, 232, 351
members, 43
  controlling access to, 146
memory
  allocating sufficient, 52, 99–100
  consistency errors, 454–455
  error-handling, 280
  garbage collection, 105–106
  leaks, 5
  locations, 112
  reserving for a variable, 93
  saving in large arrays, 44–45
metadata, 327–335
method signatures, 55
  in interface declarations, 136–137
  in method declarations, 91
  overloaded methods and, 91
  type erasure and, 192–193
methods, 32–34. *See also* by individual type
  abstract, 158–161
  access modifiers and, 90, 146, 152
  accessor, 237, 242, 244, 330, 410, 415
  applet milestone, 593–594
  atomic, 425
  bridge, 191–193
  chaining, 322
  class, 113–114
  defining, 90–92
  final, 158
  generic, 170–171
  hiding, 148–149
  instance, 148
  naming, 91
  overloaded, 91–92
  overriding, 148–149
  package-private, 110
  qualified names, 118
  returning a class or interface, 107–108
  returning values from, 106–107
  synchronized, 455–457
  wildcards, 184–186
Microsoft Windows
  access control list (ACL), 329
  CLASSPATH in, 539
  common errors, 25–28
  environment variables on, 528–530
  file name separators on, 211
  HelloWorld, 14–18, 676
  log files, 623
  PATH in, 312–316, 318, 536–537
  path separators on, 213–214, 364
  root directories on, 310–311
  system file stores, 364
  Windows 7, 537
  Windows Vista, 537
  Windows XP, 536–537
MIME types, 239, 333–334, 368
  determining, 363
  JNLP and web servers, 581, 588–589, 662
MIN_VALUE constant, 219, 232, 411
modifiers. *See* access modifiers
modularity, 34
monitor locks. *See* locks, intrinsic
Mozilla Firefox add-ons, 650, 655
MultiDimArrayDemo example, 53
multimaps, 403–405
multisets, 438

N

N type parameter naming convention, 166
NameSort example, 408–409
  of classes, 90
  of JAR files, 670
  of methods, 90, 91
  of packages, 206–207
  of type parameter, 165–166
  of variables, 43–44, 54–55, 90
nanoTime method, 535
nCopies method, 430
NegativeArraySizeException, 275
nested classes, 120–129
  controlling access and, 146
  importing, 209
  inheritance and, 144
  inner. *See* inner classes

nonstatic, 120–121
static, 121–122
NetBeans IDE, 1, 4–10, 12–14, 357, 676
HelloWorld application, 6–13
new keyword, 46, 99, 166, 236
newCachedThreadPool method, 474
newFixedThreadPool method, 474
newline. *See* line terminators
newSingleThreadExecutor method, 474
next method, 378, 389
nextIndex method, 389–390
NIO.2, 309–366
NoSuchElementException, 395
NoSuchMethodError, 28
Notepad demo, 581, 585, 656–658
Notepad text editor, 15–16
notifyAll method, 154–155, 462–464
null value, 48, 292, 360
NullPointerException, 75, 260, 275–276, 279, 391, 407–408
Number class, 217, 219–220, 232, 527
number systems, 47
converting between, 219–220
decimal, 45, 47, 219–220
hexidecimal, 47, 159, 219–220, 301
octal, 219–220
NumberFormatException, 452, 527
numbers, 217–234
converting between strings and, 240–242
currency values and, 45, 307
formatting, 220–224, 301
random, 230

O

Object class, 135, 154–161, 181, 187, 681
object ordering, 405–412
object references, 56, 85, 106, 156, 308, 679
ObjectInput interface, 307
ObjectInputStream class, 307
object-oriented programming, 31–39, 150
ObjectOutput interface, 307
ObjectOutputStream class, 307
objects, 32–34, 98–106, 117–118
as a superclass, 154–158, 677
calling methods, 104–105
casting, 146–147
creating, 99–103
declaring variables to refer to, 99–100
hash codes of, 154, 157, 408
immutable, 465–469
initializing, 100–103
lock, 470–472
referencing fields, 103–104
ObjectStreams example, 307
octal number system, 219–220
offer method, 395
operators, 55–65. *See also* by individual type
assignment, 56, 62
precedence of, 55–56, 66, 677
prefix/postfix, 56, 58–59, 100
OutputStream class, 291, 293–295, 326, 338
OutputStreamWriter class, 294, 642
@Override annotation class, 132, 148

P

package members, 207
package importing, 208
package referring to, 207–208
package using, 207–211
package statements, 205–206, 208, 214, 604
packages, 4, 31, 32, 38, 203–215, 676
apparent hierarchies of, 209
creating, 205–206
importing, 208–209
name ambiguities, 209–210
naming, 206–207
qualified names, 206–211
using package members, 207–211
pages. *See* web pages
palindromes, 202, 237–238, 242
Panel class, 577
parameterized types, 166, 166
assigning raw types, 168–169
backward compatibility and, 168
bounded, 171
casting, 198
generic, 196
heap pollution and, 193, 195
primitive, 196
restrictions, 196, 198–199
type erasure and, 188
type inference and, 177–178
varargs methods and, 195–196
parameters, 43
naming, 96
types, 94
parentheses. *See* ()
parseXXX methods, 241, 527
PassPrimitiveByValue class, 96–97
Password example, 303–305
passwords, 305, 530, 564, 566
Path class, 312–320
PATH variable, 26, 28, 535–539

Pattern class, 484, 485, 488, 504, 507. *See also* regular expressions
PatternSyntaxException, 483–485, 518
  methods of, 515–517
peek method, 395
percent sign. *See* %
Perl, 483–484, 505, 517
PI constant, 114, 210–211, 222, 225, 301–302
Planet class, 127–129
poll method, 360, 375, 394–395, 426, 440
polymorphism, 150–152
  in collections, 372, 378, 388, 391–392, 394, 432
  in generic types, 188, 192
pound sign. *See* #
pow method, 94, 227, 228
Preferences API, 529
PrePostDemo example, 59, 64–65
primitive data types, 44–49, 55–56. *See also by individual type*; numbers
print method, 299–300
printf method, 95–96, 239
printIn method, 299–300
PriorityBlockingQueue class, 426
PriorityQueue implementation, 419–420, 426
private keyword, 89, 110–111, 146
problems. *See* errors
ProcessBuilder onject, 446, 528
processes, 446
  lightweight. *See* threads
Producer example, 464–465
ProducerConsumerExample example, 465
programs. *See* applications
properties
  managing, 523
  saving, 525–526
  setting, 525–526
  system, 531–533, 637–638
PropertiesTest example, 532
propertyNames method, 525
protected modifier, 110–111
public modifier, 89, 110–111, 138
put method, 440
putAll method, 397, 400–401
pwd command, 21, 28

Q

qualified names
  in applets, 584, 622
  for fields, 96
  for instance variables, 118
  for methods, 118
  for packages, 206–211
quantifiers, 484, 495–500
question mark. *See* ?
Queue implementations, 425–427
Queue interface, 394–396, 417, 423, 431
queues, 394–396
  bounded, 395
  priority, 375
QuoteClientApplet applet, 629–630
quoteReplacement method, 511
QuoteServer applet, 629–630

R

radians, 228–229
random access files, 339, 342–343, 367, 685
random method, 230, 233
random numbers, 230, 233, 469, 480
RandomAccessFile class, 286, 367
raw types, 168–170, 178, 193, 200, 683
readDouble method, 305–306
readInt method, 305
readObject method, 307–309
readPassword method, 303–304
readUTF method, 305–306
Receiver applet, 627–628
RegexTestHarness example, 485–486, 505–506
RegexTestHarness2 example, 516–517
regionMatches method, 247–248
RegionMatchesDemo example, 247
regular expressions, 354, 483–519, 684
  backreferences in, 502
  boundary matchers in, 502–504
  capturing groups in, 498–499, 500–502
  character classes in, 488–492, 492–495, 498–499
  greedy quantifiers, 499
  intersections, 491–192
  Matcher class, 509–515
  metacharacters in, 487–488
  negation, 489
  numbering groups, 501
  Pattern class, 504–509
  PatternSyntaxException class, 515–517
  possessive quantifiers, 500
  quantifiers in, 495–500
  ranges, 489–490
  reluctant quantifiers, 499
  string literals, 486–488
  subtraction, 492
  test harness, 485–486
  Unicode support, 517–518
  unions, 490–491
  zero-length matches in, 495–498

Relatable interface, 139–141
relational operators. *See* comparisons
remove method, 377, 378, 381, 417, 430
removeAll method, 379, 383–384, 401–403
removeDups method, 381
removeEldestEntry method, 424
removeElementAt method, 387
removeFirst method, 423
renameTo method, 367
replace method, 425
REPLACE_EXISTING enum, 325–327
replaceAll method, 245, 394, 511, 513–515
ReplaceDemo example, 513–514
ReplaceDemo2 example, 514
replaceFirst method, 245, 511, 513–515
reserved words. *See* keywords
retainAll method, 377–379, 380, 383–384, 401–402
return statements, 82, 106, 411
return types, 90–92
  constructors, 101
  covariant, 108, 148
  generic, 170
  implementations using, 137
  inferring, 179
  instance methods, 148
  interface names as, 117
  JavaScript code and, 613
  methods, 90, 106–108
  wildcards and, 179, 187
reverse method, 251–253
rich Internet applications (RIAs), 635–648
  cookies and, 643–646
  customizing the loading experience in, 646
  security in, 646–647
  setting secure properties, 635–637
  setting trusted arguments, 635–637
  system properties, 637–638
  using the JNLP API, 638–643
rint method, 226–227
Root example, 299–300
Root2 example, 300–301
round method. 226
run method, 448–450
Runnable interface, 447–448
runtime, 2, 24
  checks at, 147, 169
  errors, 27–28
  examining annotations at, 129, 133
RuntimeException, 257, 260, 275–276, 279–280, 680

S

S type parameter, 166
Safelock example, 470–472
sandboxes, 586–589, 631–632, 646–647
Scanner class, 297–298, 404
scanning, 296–297
ScanSum example, 298–299
ScanXan example, 297–298
ScheduledExecutorService interface, 472–475
ScheduledThreadPoolExecutor class, 475
security
  applets and, 522, 603
  certificate, 631
  digitally signed files, 541, 560–562
  JAR files, 541, 560–562, 668, 671
  Java Web Start applications and, 586–587
  keystores and, 564
  legacy file I/O code and, 365
  managers, 522, 531, 533–535
  password entry and, 303–304
  public and private keys, 561, 564
  related documentation, 564–565, 566
  rich Internet applications (RIAs) and, 646–647
  sandboxes, 631–632
  TOCTTOU, 324
  violations, 534–535
  web browsers and, 603
SecurityException, 533–535, 567
SecurityManager class, 533–534, 632
semicolon. *See* ;
Sender applet, 627–628
sequences. *See* collections, ordered
Serializable interface, 176, 307, 420
servers, 589, 603. *See* web servers
ServiceLoader class, 529
Set implementations, 421–422
Set interface, 380–384, 387, 417, 431
  array operations of, 384
  basic operations of, 381–282
  bulk operations of, 383–384
set method, 386, 391, 478
setDefaultHostnameVerifier method, 587
setDefaultSSLSocketFactory method, 587
setElementAt method, 385–387
setLastModified method, 367
setLayout method, 582
setLength method, 250
setProperties method, 531–533
setProperty method, 526
setSize method, 386
Short class, 70, 219, 232, 240, 406

short data type, 45
  default value of, 46
  switch statement and, 70
shortValue method, 219, 232
ShowDocument applet, 608
showDocument method, 607–608, 631
showStatus method, 607
Shuffle example, 388–389, 393
signature block files, 563
signature files, 562–563
Simple applet, 594
SimpleThreads example, 448, 451–453
sin method, 228–229
single quote. *See* '
singleton method, 379, 419, 430–431, 682, 689
size method, 439–440
slash. *See* /
sleep method, 448–449
SleepMessages example, 449–450
Smalltalk's collection hierarchy, 373
Socket class, 38
sockets, 38, 446
software. *See* applications
Solaris, 676. *See also* UNIX/Linux
  HelloWorld, 18–22
  paths in, 310–317
  updating PATH variable, 538–539
sort method, 396, 434
SortedMap interface, 376, 415–417, 443–444, 683
  Sorted comparison to SortedSet, 416
  Sorted map operations, 416
  Sorted standard conversion constructor in, 416
SortedSet interface, 374, 376, 412–419, 421, 427, 429, 683
  Sortedcomparator accessor, 415
  Sorted comparison to SortedMap, 416
  Sorted endpoint operations in, 414–415
  Sorted range-view operations, 413–414
  Sorted set operations, 412
  Sorted standard conversion constructors, 413
source files, 21–22
  source compiling, 12, 15–18, 20–23
  source creating, 14–15, 19–20
  source managing, 211–215
  source skeleton, 10
split method, 477, 507, 509
SplitDemo example, 507
SplitDemo2 example, 508
sqrt method, 95, 227–228, 299–300
square brackets. *See* [ ]
square root, 210, 227, 300
SSLSocketFactory class, 587
stack trace, 192, 284
  accessing information, 276–277
  printing, 260
StackOfInts class, 207
standard error, 302, 623
standard input, 302–303, 548
standard output, 606, 623
start method, 512, 594, 616, 634
startsWith method, 248, 319
statements, 66–67. *See also* by individual type
  synchronized, 455, 457–458
static import statement, 210–211
static initialization blocks, 116
static modifier, 42, 54, 112–114, 138
stop method, 594
streams, I/O, 289–309. *See also* by individual type
  buffered, 295–296
  byte, 291–292
  character, 293–295
  closing, 292
  creating a file using, 338
  data, 305–307
  flushing, 296
  object, 307–309
  reading a file using, 338
  unbuffered, 295
  writing a file using, 338
string builders, 249–254
String class, 45, 46, 158, 217, 236, 238, 239, 242, 244, 247, 249, 250, 253, 515
StringBuilder class, 217, 247–253, 677
StringDemo example, 237, 250
StringIndexOutOfBoundsException, 246
stringPropertyNames method, 525
strings, 236–254
  capacity of, 249–250
  comparing portions of, 247, 248
  concatenating, 238–239
  converting between numbers and, 240–242
  creating format strings, 239
  creating, 236–255
  getting by index, 242
  length, 237–238, 249–250
  manipulating characters in, 242–247
  replacing characters in, 244
  searching within, 244
StringSwitchDemo example, 73–75
subclasses, 36–37, 86–88, 111, 146, 681
  abstract methods and, 158–160
  access levels and, 110
  capabilities of, 146

constructors, 153–154
creating, 36–37
final methods and, 116, 158–159
inheritance and, 36–37, 135, 144
overloading methods in, 149
polymorphism in, 150
returning, 107–108
subList method, 385–386, 391–394, 413
submit method, 473
subSequence method, 243
subSet method, 412–414
substring method, 242, 246
subtyping, 175, 183, 192
super keyword, 152–154, 182, 187
superclasses, 143, 677
accessing, 152–153
choosing, 278
constructors for, 93, 153–154
declaring, 87–88
information hiding, 147–149, 152
inheritance and, 36–37, 143–144, 161
Object class, 154–158
overriding methods of, 147
private members in, 146
@SuppressWarnings annotation type, 130, 132–133, 170, 196, 362
swap method, 387–388, 394, 436
Swing, 4, 156. *See* graphical user interfaces (GUIs)
switch block, 71–72
switch statements, 70–75
falling through, 71–72
using strings in, 73–75
SwitchDemo example, 70
SwitchDemo2 example, 73
SwitchDemoFallThrough example, 72
symbolic links, 311–312, 315–317, 324, 347–348
synchronization, 453–459
atomic access, 459
intrinsic locks and, 457–458
reentrant, 458
synchronized class example, 466–467
synchronized keyword, 455–456
synchronizedCollection method, 427–428
SynchronizedCounter example, 455–456, 479
synchronizedList method, 423, 427–428
synchronizedMap method, 427–428
SynchronizedRGB example, 466–468
synchronizedSet method, 427
synchronizedSortedMap method, 427
synchronizedSortedSet method, 427
SynchronousQueue class, 426
System class, 25, 53, 523, 530–531
System.console, 303, 486, 516
System.err, 299, 302–303
System.in, 302–303
System.out, 299

T

T type parameter, 166
tab, 48, 235
tailSet method, 412–415
tan method, 228–229
ternary operators, 56, 60, 63
test harness, 485–486
TestFormat example, 222–223
this keyword, 85, 108–109, 114
using with constructors, 109
using with fields, 108–109
Thread class, 448, 686
thread pools, 358, 469, 473–475, 687, 690
ThreadLocalRandom, 469, 480
ThreadPoolExecutor class, 475
threads, 446–447
in applets, 596
defining, 447–448
guarded blocks and, 461–465
interference, 453
interrupts, 449–451
joins, 451
locks and, 460–461
multi-threaded applications, 218
pausing, 448–449
starting, 447–448
synchronization of, 453–459
thread objects, 447–453
thread pools, 474
thread safe, 253
worker, 473–474
throw statement, 273–274, 285
Throwable class, 199, 263, 269, 273–276, 284–285
throws keyword, 273
TicTacToe example, 544–552
time, 143, 223
toArray method, 379–380, 412, 416, 426, 439, 442
TOCTTOU, 324
toDegrees method, 228–229
tokens, 296–298
toLowerCase method, 74–75, 235, 243, 253
toRadians method, 228–229
toString() method, 157–158
ToStringDemo example, 241–242

toUpperCase method, 235, 243, 253
TransferQueue implementation, 427
TreeMap implementation, 397, 399, 416, 419, 424, 478
TreeSet implementation, 380, 382, 397, 411, 413, 416, 418–419, 421–422
TrigonometricDemo example, 228–229
trigonometry, 218, 228–229, 233
trim method, 243, 418
troubleshooting. *See* errors
try blocks, 262–271, 285, 321, 680, 684
tryLock method, 470–471
try-with-resources statement, 261, 266–269, 320, 345, 684, 688
type erasure, 188–196, 200–201
    bridge methods and, 191–193
    effects of, 191–193
type inference, 167, 171, 176–179, 185
type parameters, 97, 164–168, 170–175, 177–179, 181, 188–190, 193, 196–200
    bounded, 171–173
    multiple bounds, 172
    multiple, 167–168
type variables, 165, 172
types. *See also* by individual type
    nonreifiable, 193–196
    parameterized. *See* parameterized types
    raw, 168–170

U

U type parameter, 166
unary operators, 56, 58–59, 63, 231, 677
UnaryDemo example, 58
unboxing, 217, 230–232, 234, 683
underscore. *See* _
Unicode encoding
    character properties, 518
    regular expressions and, 517–518
    UTF-8, 306, 543, 555, 667
    UTF-16, 47
Uniform Resource Identifiers (URIs), 313, 368
UNIX/Linux. *See also* Solaris
    common error messages on, 25, 28
    HelloWorld, 18–22
unset command, 28, 214
UnsupportedOperationException, 334, 375, 429–430
URLClassLoader implementation, 567–568, 570
useDelimiter method, 298
UTF. *See* Unicode encoding

V

V type parameter, 166
valueOf method, 177, 197, 220, 232–233, 240–241, 253, 414
ValueOfDemo example, 233, 240–241
values method, 127
varargs, 95
    potential vulnerabilities of, 194–195
    preventing warnings from, 195–196
variables, 42–55. *See also* by individual type; fields
    atomic, 478–479
    class. *See* fields, static
    environment, 527–529, 535–540
    instance. *See* instance variables
    local, 42
    naming conventions, 43–44, 54–55, 90
    refering to objects, 99–100
Vector collection, 375
    compared to ArrayList, 423
    compared to List, 385–386
    compatibility and, 420
vertical bar. *See* |
vi text editor, 19
volatile keyword, 459, 478

W

wait method, 154
warning messages, 169
WarningDemo example, 169
watch keys, 360–361
WatchService API, 358–363
WeakHashMap implementation, 424–425
web browsers
    displaying documents in, 607–608
    frames in, 607–608
    security in, 603
web pages
    HTML frames, 607–608
    invoking applets, 552, 601–603, 629–632
    Java applications, 580–581. *See* Java Web Start applications
web servers
    applets and, 401
    placing applications on, 575, 581
    JNLP errors in, 588
    testing, 669
    setting up, 581, 650
while statement, 75–76, 82
WhileDemo example, 75–76

white space
 allowing, 504
 character construct for, 493
 clearning up, 668
 determining presence of, 235
 disallowed 43
 leading, 503
 tokens and, 255, 298
 trailing, 243
wildcards, 179–188, 661
 capture and, 184–186
 guidelines for using, 187–188
 helper methods and, 184–186
 lower-bounded, 182
 subtyping and, 182–183
 unbounded, 180–181
 upper-bounded, 179–180
Windows. *See* Microsoft Windows
wrappers, 218, 230–234, 253, 294, 306, 371, 420, 432, 683
 `checked` interface, 429
 implentations of, 419, 427–429
 synchronization, 428–429
 unmodifiable, 429–429
`write` method, 299, 335–337
`writeDouble` method, 305–306
`writeInt` method, 305–306
`writeObject` method, 308–309
`writer` method, 303
`writeUTF` method, 305–306

Z

ZIP archives, 542