

ORACLE
PRESS



CORE JAVA

Volume I: Fundamentals

THIRTEENTH EDITION

ORACLE

Cay S. Horstmann

FREE SAMPLE CHAPTER |



Core Java

Volume I—Fundamentals

Thirteenth Edition

Cay S. Horstmann



Hoboken, New Jersey

Cover image: Jon Chica/Shutterstock

Figure 1.1: Sourceforge

Figures 2.2, 3.2-3.5, 4.9, 5.4, 7.2, 10.5, 10.6, 11.1: Oracle Corporation

Figures 2.3-2.6, 12.2: Eclipse Foundation, Inc.

Figure 4.2: Violet UML Editor

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The views expressed in this book are those of the author and do not necessarily reflect the views of Oracle.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com. For questions about sales outside the U.S., please contact intlcs@pearson.com.

Please contact us with concerns about any potential bias at pearson.com/report-bias.html.

Visit us on the Web: informit.com/aw

Copyright © 2025 Pearson Education, Inc.

Portions copyright © 1996-2013 Oracle and/or its affiliates. All Rights Reserved.

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

Microsoft and/or its respective suppliers make no representations about the suitability of the information contained in the documents and related graphics published as part of the services for any purpose. All such documents and related graphics are provided "as is" without warranty of any kind. Microsoft and/or its respective suppliers hereby disclaim all warranties and conditions with regard to this information, including all warranties and conditions of merchantability, whether express, implied or statutory, fitness for a particular purpose, title and non-infringement. In no event shall Microsoft and/or its respective suppliers be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of information available from the services. The documents and related graphics contained herein could include technical inaccuracies or typographical errors. Changes are periodically added to the information herein. Microsoft and/or its respective suppliers may make improvements and/or changes in the product(s) and/or the program(s) described herein at any time. Partial screen shots may be viewed in full within the software version specified.

Microsoft® Windows®, and Microsoft Office® are registered trademarks of the Microsoft Corporation in the U.S.A. and other countries. This book is not sponsored or endorsed by or affiliated with the Microsoft Corporation.

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearson.com/permissions.

ISBN-13: 978-0-13-532837-8

ISBN-10: 0-13-532837-3

\$PrintCode

Table of Contents

Preface.....	xiii
To the Reader.....	xiii
A Tour of This Book	xiv
Conventions	xvi
Sample Code	xvii
Acknowledgments.....	xix
1. An Introduction to Java.....	1
1.1. Java as a Programming Platform	1
1.2. The Java “White Paper” Buzzwords	2
1.2.1. Simple	2
1.2.2. Object-Oriented.....	3
1.2.3. Distributed	3
1.2.4. Robust	3
1.2.5. Secure	4
1.2.6. Architecture-Neutral	4
1.2.7. Portable	5
1.2.8. Interpreted	6
1.2.9. High-Performance	6
1.2.10. Multithreaded.....	6
1.2.11. Dynamic.....	7
1.3. Java Applets and the Internet	7
1.4. A Short History of Java	8
1.5. Common Misconceptions about Java	12
2. The Java Programming Environment	15
2.1. Installing the Java Development Kit	15
2.1.1. Downloading the JDK	15
2.1.2. Setting Up the JDK.....	16
2.1.3. Installing Source Files and Documentation	18
2.2. Using the Command-Line Tools	19
2.3. Using an Integrated Development Environment	24
2.4. JShell.....	25
3. Fundamental Programming Structures in Java.....	31
3.1. A Simple Java Program	31
3.2. Comments	35
3.3. Data Types	36
3.3.1. Integer Types	36
3.3.2. Floating-Point Types.....	38
3.3.3. The char Type	39
3.3.4. Unicode and the char Type	41
3.3.5. The boolean Type.....	43
3.4. Variables and Constants.....	43
3.4.1. Declaring Variables	43
3.4.2. Initializing Variables.....	45
3.4.3. Constants	46
3.4.4. Enumerated Types	47

3.5. Operators	48
3.5.1. Arithmetic Operators	48
3.5.2. Mathematical Functions and Constants	49
3.5.3. Conversions between Numeric Types	51
3.5.4. Casts	52
3.5.5. Assignment	53
3.5.6. Increment and Decrement Operators	54
3.5.7. Relational and boolean Operators	54
3.5.8. The Conditional Operator	55
3.5.9. Switch Expressions	55
3.5.10. Bitwise Operators	57
3.5.11. Parentheses and Operator Hierarchy	58
3.6. Strings	59
3.6.1. Concatenation	60
3.6.2. Splitting Strings	61
3.6.3. Indexes and Substrings	62
3.6.4. Strings Are Immutable	63
3.6.5. Testing Strings for Equality	65
3.6.6. Empty and Null Strings	66
3.6.7. The String API	66
3.6.8. Reading the Online API Documentation	68
3.6.9. Building Strings	70
3.6.10. Text Blocks	73
3.7. Input and Output	76
3.7.1. Reading Input	76
3.7.2. Formatting Output	79
3.7.3. File Input and Output	83
3.8. Control Flow	85
3.8.1. Block Scope	86
3.8.2. Conditional Statements	86
3.8.3. Loops	90
3.8.4. Determinate Loops	95
3.8.5. Multiple Selections with switch	99
3.8.6. Statements That Break Control Flow	104
3.9. Big Numbers	107
3.10. Arrays	110
3.10.1. Declaring Arrays	111
3.10.2. Accessing Array Elements	112
3.10.3. The “for each” Loop	113
3.10.4. Array Copying	114
3.10.5. Command-Line Arguments	116
3.10.6. Array Sorting	117
3.10.7. Multidimensional Arrays	119
3.10.8. Ragged Arrays	122
4. Objects and Classes	127
4.1. Introduction to Object-Oriented Programming	127
4.1.1. Classes	128

4.1.2. Objects	129
4.1.3. Identifying Classes	130
4.1.4. Relationships between Classes	130
4.2. Using Predefined Classes	132
4.2.1. Objects and Object Variables	132
4.2.2. The LocalDate Class of the Java Library	136
4.2.3. Mutator and Accessor Methods	138
4.3. Defining Your Own Classes	142
4.3.1. An Employee Class	142
4.3.2. Use of Multiple Source Files	145
4.3.3. Dissecting the Employee Class	146
4.3.4. First Steps with Constructors	147
4.3.5. Declaring Local Variables with var	148
4.3.6. Working with null References	149
4.3.7. Implicit and Explicit Parameters	151
4.3.8. Benefits of Encapsulation	152
4.3.9. Class-Based Access Privileges	154
4.3.10. Private Methods	155
4.3.11. Final Instance Fields	156
4.4. Static Fields and Methods	157
4.4.1. Static Fields	157
4.4.2. Static Constants	158
4.4.3. Static Methods	159
4.4.4. Factory Methods	160
4.4.5. The main Method	161
4.5. Method Parameters	164
4.6. Object Construction	171
4.6.1. Overloading	171
4.6.2. Default Field Initialization	172
4.6.3. The Constructor with No Arguments	173
4.6.4. Explicit Field Initialization	174
4.6.5. Parameter Names	175
4.6.6. Calling Another Constructor	176
4.6.7. Initialization Blocks	176
4.6.8. Object Destruction and the finalize Method	182
4.7. Records	182
4.7.1. The Record Concept	183
4.7.2. Constructors: Canonical, Compact, and Custom	185
4.8. Packages	188
4.8.1. Package Names	188
4.8.2. Class Importation	189
4.8.3. Static Imports	191
4.8.4. Addition of a Class into a Package	192
4.8.5. Package Access	195
4.8.6. The Class Path	197
4.8.7. Setting the Class Path	199
4.9. JAR Files	200

4.9.1. Creating JAR files	200
4.9.2. The Manifest	201
4.9.3. Executable JAR Files	202
4.9.4. Multi-Release JAR Files	203
4.9.5. A Note about Command-Line Options	205
4.10. Documentation Comments	206
4.10.1. Comment Insertion	207
4.10.2. Class Comments	207
4.10.3. Method Comments	208
4.10.4. Field Comments	209
4.10.5. Package Comments	209
4.10.6. HTML Markup	210
4.10.7. Links	210
4.10.8. General Comments	212
4.10.9. Code Snippets	212
4.10.10. Comment Extraction	213
4.11. Class Design Hints	214
5. Inheritance	217
5.1. Classes, Superclasses, and Subclasses	217
5.1.1. Defining Subclasses	218
5.1.2. Overriding Methods	219
5.1.3. Subclass Constructors	221
5.1.4. Inheritance Hierarchies	225
5.1.5. Polymorphism	226
5.1.6. Understanding Method Calls	228
5.1.7. Preventing Inheritance: Final Classes and Methods	231
5.1.8. Casting	234
5.1.9. Pattern Matching for instanceof	236
5.1.10. Protected Access	239
5.2. Object: The Cosmic Superclass	240
5.2.1. Variables of Type Object	240
5.2.2. The equals Method	241
5.2.3. Equality Testing and Inheritance	242
5.2.4. The hashCode Method	246
5.2.5. The toString Method	250
5.3. Generic Array Lists	257
5.3.1. Declaring Array Lists	258
5.3.2. Accessing Array List Elements	260
5.3.3. Compatibility between Typed and Raw Array Lists	264
5.4. Object Wrappers and Autoboxing	265
5.5. Methods with a Variable Number of Arguments	270
5.6. Abstract Classes	271
5.7. Enumeration Classes	277
5.8. Sealed Classes	282
5.9. Pattern Matching	288
5.9.1. Null Handling	289
5.9.2. Guards	290

5.9.3. Exhaustiveness	290
5.9.4. Dominance.....	292
5.9.5. Patterns and Constants	293
5.9.6. Variable Scope and Falthrough	293
5.10. Reflection	296
5.10.1. The Class Class	297
5.10.2. A Primer on Declaring Exceptions	300
5.10.3. Resources	301
5.10.4. Using Reflection to Analyze the Capabilities of Classes	304
5.10.5. Using Reflection to Analyze Objects at Runtime.....	311
5.10.6. Using Reflection to Write Generic Array Code	316
5.10.7. Invoking Arbitrary Methods and Constructors	320
5.11. Design Hints for Inheritance	324
6. Interfaces, Lambda Expressions, and Inner Classes	327
6.1. Interfaces	327
6.1.1. The Interface Concept.....	327
6.1.2. Properties of Interfaces.....	335
6.1.3. Interfaces and Abstract Classes.....	337
6.1.4. Static and Private Methods	338
6.1.5. Default Methods	339
6.1.6. Resolving Default Method Conflicts.....	340
6.1.7. Interfaces and Callbacks	342
6.1.8. The Comparator Interface	345
6.1.9. Object Cloning.....	347
6.2. Lambda Expressions	354
6.2.1. Why Lambdas?	354
6.2.2. The Syntax of Lambda Expressions	355
6.2.3. Functional Interfaces	358
6.2.4. Function Types	359
6.2.5. Method References	361
6.2.6. Constructor References	365
6.2.7. Variable Scope.....	366
6.2.8. Processing Lambda Expressions	369
6.2.9. Creating Comparators.....	373
6.3. Inner Classes	375
6.3.1. Use of an Inner Class to Access Object State	376
6.3.2. Special Syntax Rules for Inner Classes.....	380
6.3.3. Are Inner Classes Useful? Actually Necessary? Secure?	381
6.3.4. Local Inner Classes	382
6.3.5. Accessing Variables from Outer Methods	383
6.3.6. Anonymous Inner Classes	385
6.3.7. Static Classes	389
6.4. Service Loaders	393
6.5. Proxies	395
6.5.1. When to Use Proxies	396
6.5.2. Creating Proxy Objects	396
6.5.3. Properties of Proxy Classes.....	400

7. Exceptions, Assertions, and Logging.....	403
7.1. Dealing with Errors.....	403
7.1.1. The Classification of Exceptions	405
7.1.2. Declaring Checked Exceptions.....	407
7.1.3. How to Throw an Exception.....	409
7.1.4. Creating Exception Classes.....	411
7.2. Catching Exceptions	412
7.2.1. Catching an Exception	412
7.2.2. Catching Multiple Exceptions	414
7.2.3. Rethrowing and Chaining Exceptions	417
7.2.4. The finally Clause.....	418
7.2.5. The try-with-Resources Statement.....	421
7.2.6. Analyzing Stack Trace Elements.....	423
7.3. Tips for Using Exceptions	427
7.4. Using Assertions	431
7.4.1. The Assertion Concept	431
7.4.2. Assertion Enabling and Disabling	432
7.4.3. Using Assertions for Parameter Checking	434
7.4.4. Using Assertions for Documenting Assumptions	435
7.5. Logging	436
7.5.1. Should You Use the Java Logging Framework?	436
7.5.2. Logging 101	437
7.5.3. The Platform Logging API.....	438
7.5.4. Logging Configuration	440
7.5.5. Log Handlers	441
7.5.6. Filters and Formatters	444
7.5.7. A Logging Recipe	445
7.6. Debugging Tips.....	452
8. Generic Programming	459
8.1. Why Generic Programming?	459
8.1.1. The Advantage of Type Parameters	459
8.1.2. Who Wants to Be a Generic Programmer?	461
8.2. Defining a Simple Generic Class.....	462
8.3. Generic Methods.....	464
8.4. Bounds for Type Variables	465
8.5. Generic Code and the Virtual Machine.....	468
8.5.1. Type Erasure	468
8.5.2. Translating Generic Expressions	469
8.5.3. Translating Generic Methods.....	470
8.5.4. Calling Legacy Code.....	472
8.5.5. Generic Record Patterns	474
8.6. Inheritance Rules for Generic Types	474
8.7. Wildcard Types	477
8.7.1. The Wildcard Concept.....	477
8.7.2. Supertype Bounds for Wildcards.....	478
8.7.3. Unbounded Wildcards	482
8.7.4. Wildcard Capture	482

8.8. Restrictions and Limitations	485
8.8.1. Type Parameters Cannot Be Instantiated with Primitive Types	485
8.8.2. Runtime Type Inquiry Only Works with Raw Types	485
8.8.3. You Cannot Create Arrays of Parameterized Types	486
8.8.4. Varargs Warnings	487
8.8.5. Generic Varargs Do Not Spread Primitive Arrays	488
8.8.6. You Cannot Instantiate Type Variables	489
8.8.7. You Cannot Construct a Generic Array	490
8.8.8. Type Variables Are Not Valid in Static Contexts of Generic Classes	492
8.8.9. You Cannot Throw or Catch Instances of a Generic Class	492
8.8.10. You Can Defeat Checked Exception Checking	493
8.8.11. Beware of Clashes after Erasure	495
8.8.12. Type Inference in Generic Record Patterns is Limited	496
8.9. Reflection and Generics	498
8.9.1. The Generic Class Class	498
8.9.2. Using Class<T> Parameters for Type Matching	499
8.9.3. Generic Type Information in the Virtual Machine	500
8.9.4. Type Literals	504
9. Collections	511
9.1. The Java Collections Framework	511
9.1.1. Separating Collection Interfaces and Implementation	511
9.1.2. The Collection Interface	514
9.1.3. Iterators	515
9.1.4. Generic Utility Methods	518
9.2. Interfaces in the Collections Framework	521
9.3. Concrete Collections	525
9.3.1. Linked Lists	526
9.3.2. Array Lists	537
9.3.3. Hash Sets	537
9.3.4. Tree Sets	542
9.3.5. Queues and Deques	545
9.3.6. Priority Queues	547
9.4. Maps	548
9.4.1. Basic Map Operations	549
9.4.2. Updating Map Entries	552
9.4.3. Map Views	554
9.4.4. Weak Hash Maps	557
9.4.5. Linked Hash Sets and Maps	557
9.4.6. Enumeration Sets and Maps	559
9.4.7. Identity Hash Maps	560
9.5. Copies and Views	562
9.5.1. Small Collections	562
9.5.2. Unmodifiable Copies and Views	565
9.5.3. Subranges	566
9.5.4. Sets From Boolean-Valued Maps	567

9.5.5. Reversed Views	568
9.5.6. Checked Views	568
9.5.7. Synchronized Views	569
9.5.8. A Note on Optional Operations	569
9.6. Algorithms.....	574
9.6.1. Why Generic Algorithms?.....	574
9.6.2. Sorting and Shuffling	576
9.6.3. Binary Search.....	578
9.6.4. Simple Algorithms	580
9.6.5. Bulk Operations.....	582
9.6.6. Converting between Collections and Arrays.....	583
9.6.7. Writing Your Own Algorithms	584
9.7. Legacy Collections	586
9.7.1. The Hashtable Class.....	587
9.7.2. Enumerations	587
9.7.3. Property Maps.....	588
9.7.4. System Properties	590
9.7.5. Stacks	593
9.7.6. Bit Sets	593
10. Concurrency	599
10.1. Running Threads.....	599
10.2. Thread States.....	605
10.2.1. New Threads	605
10.2.2. Runnable Threads	605
10.2.3. Blocked and Waiting Threads.....	606
10.2.4. Terminated Threads	608
10.3. Thread Properties	608
10.3.1. Virtual Threads.....	608
10.3.2. Thread Interruption	609
10.3.3. Daemon Threads	613
10.3.4. Thread Names and Ids	613
10.3.5. Handlers for Uncaught Exceptions	614
10.3.6. Thread Priorities	615
10.3.7. Thread Factories and Builders	616
10.4. Coordinating Tasks	618
10.4.1. Callables and Futures	618
10.4.2. Executors	621
10.4.3. Invoking a Group of Tasks.....	625
10.4.4. Thread-Local Variables.....	631
10.4.5. The Fork-Join Framework.....	632
10.5. Synchronization	635
10.5.1. An Example of a Race Condition.....	636
10.5.2. The Race Condition Explained	638
10.5.3. Lock Objects.....	640
10.5.4. Condition Objects.....	644
10.5.5. Deadlocks	649
10.5.6. The synchronized Keyword.....	652

10.5.7. Synchronized Blocks	657
10.5.8. The Monitor Concept	659
10.5.9. Volatile Fields	660
10.5.10. Final Fields	661
10.5.11. Atomics	662
10.5.12. On-Demand Initialization	664
10.5.13. Safe Publication	665
10.5.14. Sharing with Thread-Local Variables	666
10.6. Thread-Safe Collections	667
10.6.1. Blocking Queues	668
10.6.2. Efficient Maps, Sets, and Queues	674
10.6.3. Atomic Update of Map Entries	676
10.6.4. Bulk Operations on Concurrent Hash Maps	679
10.6.5. Concurrent Set Views	682
10.6.6. Copy on Write Arrays	682
10.6.7. Parallel Array Algorithms	682
10.6.8. Older Thread-Safe Collections	684
10.7. Asynchronous Computations	685
10.7.1. Completable Futures	685
10.7.2. Composing Completable Futures	687
10.7.3. Long-Running Tasks in User-Interface Callbacks	694
10.8. Processes	702
10.8.1. Building a Process	702
10.8.2. Running a Process	704
10.8.3. Process Handles	706
11. Annotations	711
11.1. Using Annotations	711
11.1.1. Annotation Elements	712
11.1.2. Multiple and Repeated Annotations	713
11.1.3. Annotating Declarations	713
11.1.4. Annotating Type Uses	714
11.1.5. Making Receivers Explicit	716
11.2. Defining Annotations	717
11.3. Annotations in the Java API	720
11.3.1. Annotations for Compilation	721
11.3.2. Meta-Annotations	723
11.4. Processing Annotations at Runtime	725
11.5. Source-Level Annotation Processing	729
11.5.1. Annotation Processors	729
11.5.2. The Language Model API	730
11.5.3. Using Annotations to Generate Source Code	731
11.6. Bytecode Engineering	736
11.6.1. Modifying Class Files	736
11.6.2. Modifying Bytecodes at Load Time	743
12. The Java Platform Module System	747
12.1. The Module Concept	747
12.2. Naming Modules	748

12.3. The Modular “Hello, World!” Program	749
12.4. Requiring Modules.....	751
12.5. Exporting Packages	753
12.6. Modular JARs	757
12.7. Modules and Reflective Access.....	759
12.8. Automatic Modules	762
12.9. The Unnamed Module.....	764
12.10. Command-Line Flags for Migration.....	765
12.11. Transitive and Static Requirements	766
12.12. Qualified Exporting and Opening	768
12.13. Service Loading	769
12.14. Tools for Working with Modules	772
Appendix.....	775
Index.....	781

Preface

To the Reader

In late 1995, the Java programming language burst onto the Internet scene and gained instant celebrity status. The promise of Java technology was that it would become the *universal glue* that connects users with information wherever it comes from—web servers, databases, information providers, or any other imaginable source. Indeed, Java is in a unique position to fulfill this promise. It is an extremely solidly engineered language that has gained wide acceptance. Its built-in security and safety features are reassuring both to programmers and to the users of Java programs. Java has built-in support for advanced programming tasks, such as network programming, database connectivity, and concurrency.

Since 1995, over twenty revisions of the Java Development Kit have been released. The Application Programming Interface (API) has grown from about a hundred to over 4,000 classes. The API now spans such diverse areas as concurrent programming, collections, user interface construction, database management, internationalization, security, and XML processing.

The book that you are reading right now is the first volume of the thirteenth edition of *Core Java*. Each edition closely followed a release of the Java Development Kit, and each time, I rewrote the book to take advantage of the newest Java features. This edition has been updated to reflect the features of Java 21.

As with the previous editions, *this book still targets serious programmers who want to put Java to work on real projects*. I think of you, the reader, as a programmer with a solid background in a programming language other than Java. I assume that you don't like books filled with toy examples (such as toasters, zoo animals, or "nervous text"). You won't find any of these in the book. My goal is to enable you to fully understand the Java language and library, not to give you an illusion of understanding.

In this book you will find lots of sample code demonstrating almost every language and library feature. The sample programs are purposefully simple to focus on the major points, but, for the most part, they aren't fake and they don't cut corners. They should make good starting points for your own code.

I assume you are willing, even eager, to learn about all the features that the Java language puts at your disposal. In this volume, you will find a detailed treatment of

- Object-oriented programming
- Reflection and proxies
- Interfaces and inner classes
- Exception handling
- Generic programming
- The collections framework
- Concurrency

- Annotations
- The Java platform module system

With the explosive growth of the Java class library, a one-volume treatment of all the features of Java that serious programmers need to know is simply not possible. Hence, the book is broken up into two volumes. This first volume concentrates on the fundamental concepts of the Java language. The second volume, *Core Java, Volume II: Advanced Features*, goes further into the most important libraries.

For twelve editions, user-interface programming was considered fundamental, but the time has come to recognize that it is no more, and to move it into the second volume. That volume includes detailed discussions of these topics:

- The Stream API
- File processing and regular expressions
- Databases
- XML processing
- Scripting and Compiling APIs
- Internationalization
- Network programming
- Graphical user interface design
- Graphics programming
- Native methods

When writing a book, errors and inaccuracies are inevitable. I'd very much like to know about them. But, of course, I'd prefer to learn about each of them only once. You will find a list of frequently asked questions and bug fixes at <https://horstmann.com/corejava>. Strategically placed at the end of the errata page (to encourage you to read through it first) is a form you can use to report bugs and suggest improvements. Please don't be disappointed if I don't answer every query or don't get back to you immediately. I do read all e-mail and appreciate your input to make future editions of this book clearer and more informative.

A Tour of This Book

Chapter 1 gives an overview of the capabilities of Java that set it apart from other programming languages. The chapter explains what the designers of the language set out to do and to what extent they succeeded. A short history of Java follows, detailing how Java came into being and how it has evolved.

In **Chapter 2**, you will see how to download and install the JDK and the program examples for this book. Then I'll guide you through compiling and running a console application and a graphical application. You will see how to use the plain JDK, a Java IDE, and the JShell tool.

Chapter 3 starts the discussion of the Java language. In this chapter, I cover the basics: variables, loops, and simple functions. If you are a C or C++ programmer, this is smooth sailing because the syntax for these language features is essentially the same as in C. If you come from a non-C background such as Visual Basic, you will want to read this chapter carefully.

Object-oriented programming (OOP) is now in the mainstream of programming practice, and Java is an object-oriented programming language. **Chapter 4** introduces encapsulation, the first of two fundamental building blocks of object orientation, and the Java language mechanism to implement it—that is, classes and methods. In addition to the rules of the Java language, you will also find advice on sound OOP design. Finally, I cover the marvelous javadoc tool that formats your code comments as a set of hyperlinked web pages. If you are familiar with C++, you can browse through this chapter quickly. Programmers coming from a non-object-oriented background should expect to spend some time mastering the OOP concepts before going further with Java.

Classes and encapsulation are only one part of the OOP story, and **Chapter 5** introduces the other—namely, *inheritance*. Inheritance lets you take an existing class and modify it according to your needs. This is a fundamental technique for programming in Java. The inheritance mechanism in Java is quite similar to that in C++. Once again, C++ programmers can focus on the differences between the languages.

Chapter 6 shows you how to use Java's notion of an *interface*. Interfaces let you go beyond the simple inheritance model of Chapter 5. Mastering interfaces allows you to have full access to the power of Java's completely object-oriented approach to programming. After covering interfaces, I move on to *lambda expressions*, a concise way for expressing a block of code that can be executed at a later point in time. I then explain a useful technical feature of Java called *inner classes*.

Chapter 7 discusses *exception handling*—Java's robust mechanism to deal with the fact that bad things can happen to good programs. Exceptions give you an efficient way of separating the normal processing code from the error handling. Of course, even after hardening your program by handling all exceptional conditions, it still might fail to work as expected. Then the chapter moves on to logging. In the final part of this chapter, I give you a number of useful debugging tips.

Chapter 8 gives an overview of generic programming. Generic programming makes your programs easier to read and safer. I show you how to use strong typing and remove unsightly and unsafe casts, and how to deal with the complexities that arise from the need to stay compatible with older versions of Java.

The topic of **Chapter 9** is the collections framework of the Java platform. Whenever you want to collect multiple objects and retrieve them later, you should use a collection that is best suited for your circumstances, instead of just tossing the elements into an array. This chapter shows you how to take advantage of the standard collections that are prebuilt for your use.

Chapter 10 covers concurrency, which enables you to program tasks to be done in parallel. This is an important and exciting application of Java technology in an era where processors have multiple cores that you want to keep busy.

In **Chapter 11**, you will learn about annotations, which allow you to add arbitrary information (sometimes called metadata) to a Java program. We show you how annotation processors can harvest these annotations at the source or class file level, and how annotations can be used to influence the behavior of classes at runtime. Annotations are only useful with tools, and we hope that our discussion will help you select useful annotation processing tools for your needs.

In **Chapter 12**, you will learn about the Java Platform Module System that facilitates an orderly evolution of the Java platform and core libraries. This module system provides encapsulation for packages and a mechanism for describing module requirements. You will learn the properties of modules so that you can decide whether to use them in your own applications. Even if you decide not to, you need to know the new rules so that you can interact with the Java platform and other modularized libraries.

The **Appendix** lists the reserved words of the Java language.

Conventions

As is common in many computer books, I use `monospace` type to represent computer code.



Note: Notes are tagged with "note" icons that look like this.



Tip: Tips are tagged with "tip" icons that look like this.



Caution: When there is danger ahead, I warn you with a "caution" icon.



Preview Note: Preview features that are slated to become a part of the language or API in the future are labeled with this icon.



C++ Note: There are many C++ notes that explain the differences between Java and C++. You can skip over them if you don't have a background in C++ or if you

consider your experience with that language a bad dream of which you'd rather not be reminded.

Java comes with a large programming library, or Application Programming Interface (API). When using an API call for the first time, I add a short summary description at the end of the section. These descriptions are a bit more informal but, hopefully, also a little more informative than those in the official online API documentation. The names of interfaces are in italics, just like in the official documentation. The number after a class, interface, or method name is the JDK version in which the feature was introduced, as shown in the following example:

Application Programming Interface 21

Programs whose source code is on the book's companion web site are presented as listings, for instance:

Listing 1.1 NotHelloWorld.java

```
1 void main()  
2 {  
3     System.out.println("We will not use 'Hello, World!'");  
4 }
```

Sample Code

The web site for this book at <https://horstmann.com/corejava> contains all sample code from the book. See Chapter 2 for more information on installing the Java Development Kit and the sample code.

Acknowledgments

Writing a book is always a monumental effort, and rewriting it doesn't seem to be much easier, especially with the continuous change in Java technology. Making a book a reality takes many dedicated people, and it is my great pleasure to acknowledge the contributions of the entire *Core Java* team.

A large number of individuals at Pearson provided valuable assistance but managed to stay behind the scenes. I'd like them all to know how much I appreciate their efforts. As always, my warm thanks go to my editor, Greg Doench, for steering the book through the writing and production process, and for allowing me to be blissfully unaware of the existence of all those folks behind the scenes. I am very grateful to Julie Nahil for production support, to Dmitry Kirsanov and Alina Kirsanova for copyediting the manuscript, and to Clovis L. Tondo for reviewing the final content. I wrote the book using HTML and CSS, and Prince (<https://princexml.com>) turned it into PDF—a workflow that I highly recommend. My thanks also to my coauthor of earlier editions, Gary Cornell, who has since moved on to other ventures.

Thanks to the many readers of earlier editions who reported errors and made lots of thoughtful suggestions for improvement. I am particularly grateful to the excellent reviewing team who went over the manuscript with an amazing eye for detail and saved me from many embarrassing errors.

Reviewers of this and earlier editions include Chuck Allison (Utah Valley University), Lance Andersen (Oracle), Gail Anderson (Anderson Software Group), Paul Anderson (Anderson Software Group), Alec Beaton (IBM), Cliff Berg, Andrew Binstock (Oracle), Joshua Bloch, David Brown, Corky Cartwright, Frank Cohen (PushToTest), Chris Crane (devXsolution), Dr. Nicholas J. De Lillo (Manhattan College), Rakesh Dhoopar (Oracle), Ahmad R. Elkomey, Robert Evans (Senior Staff, The Johns Hopkins University Applied Physics Lab), David Geary (Clarity Training), Jim Gish (Oracle), Brian Goetz (Oracle), Angela Gordon, Dan Gordon (Electric Cloud), Rob Gordon, John Gray (University of Hartford), Cameron Gregory (olabs.com), Andrzej Grzesik, Marty Hall (coreservlets.com, Inc.), Vincent Hardy (Adobe Systems), Dan Harkey (San Jose State University), Steve Haines, William Higgins (IBM), Marc Hoffmann (mtrail), Vladimir Ivanovic (PointBase), Jerry Jackson (CA Technologies), Heinz Kabutz (Java Specialists), Stepan V. Kalinin (I-Teco/Servionica LTD), Tim Kimmet (Walmart), John Kostaras, Jerzy Krolak, Chris Laffra, Charlie Lai (Apple), Angelika Langer, Jeff Langr (Langr Software Solutions), Doug Langston, Hang Lau (McGill University), Mark Lawrence, Doug Lea (SUNY Oswego), Gregory Longshore, Bob Lynch (Lynch Associates), Philip Milne (consultant), Mark Morrissey (The Oregon Graduate Institute), Mahesh Neelakanta (Florida Atlantic University), José Paumard (Oracle), Hao Pham, Paul Phillion, Blake Ragsdell, Ylber Ramadani (Ryerson University), Stuart Reges (University of Arizona), Simon Ritter (Azul Systems), Rich Rosen (Interactive Data Corporation), Peter Sanders (ESSI University, Nice, France), Dr. Paul Sanghera (San Jose State University and Brooks College), Paul Sevinc (Teamup AG), Devang Shah (Sun Microsystems), Yoshiki Shibata, Richard Slywczak (NASA/Glenn Research Center), Bradley A. Smith, Steven Stelting (Oracle), Christopher Taylor, Luke Taylor (Valtech), George Thiruvathukal, Kim Topley (StreamingEdge), Janet Traub, Paul Tyma (consultant), Christian Ullenboom, Peter van der

Linden, Joe Wang (Oracle), Sven Woltmann, Burt Walsh, Dan Xu (Oracle), and John Zavgren (Oracle).

Cay Horstmann
Düsseldorf, Germany
October 2023

Interfaces, Lambda Expressions, and Inner Classes

You have now learned about classes and inheritance, the key concepts of object-oriented programming in Java. This chapter shows you several advanced techniques that are commonly used. Despite their less obvious nature, you will need to master them to complete your Java tool chest.

The first technique, called *interfaces*, is a way of describing *what* classes should do, without specifying *how* they should do it. A class can *implement* one or more interfaces. You can then use objects of these implementing classes whenever conformance to the interface is required. After discussing interfaces, we move on to *lambda expressions*, a concise way to create blocks of code that can be executed at a later point in time. Using lambda expressions, you can express code that uses callbacks or variable behavior in an elegant and concise fashion.

We then discuss the mechanism of *inner classes*. Inner classes are technically somewhat complex—they are defined inside other classes, and their methods can access the fields of the surrounding class. Inner classes are useful when you design collections of cooperating classes.

This chapter concludes with a discussion of *proxies*, objects that implement arbitrary interfaces. A proxy is a very specialized construct that is useful for building system-level tools. You can safely skip that section on first reading.

6.1. Interfaces

In the following sections, you will learn what Java interfaces are and how to use them. You will also find out how interfaces have been made more powerful in recent versions of Java.

6.1.1. The Interface Concept

In the Java programming language, an interface is not a class but a set of *requirements* for the classes that want to conform to the interface.

Typically, the supplier of some service states: “If your class conforms to a particular interface, then I’ll perform the service.” Let’s look at a concrete example. The `sort` method

of the `Arrays` class promises to sort an array of objects, but under one condition: The objects must belong to classes that *implement* the `Comparable` interface.

Here is what the `Comparable` interface looks like:

```
public interface Comparable
{
    int compareTo(Object other);
}
```

In the interface, the `compareTo` method is *abstract*—it has no implementation. A class that implements the `Comparable` interface needs to have a `compareTo` method, and the method must have an `Object` parameter and return an integer. Otherwise, the class is also *abstract*—that is, you cannot construct any objects.



Note: As of Java 5, the `Comparable` interface has been enhanced to be a generic type.

```
public interface Comparable<T>
{
    int compareTo(T other); // parameter has type T
}
```

For example, a class that implements `Comparable<Employee>` must supply a method

```
int compareTo(Employee other)
```

You can still use the “raw” `Comparable` type without a type parameter. Then the `compareTo` method has a parameter of type `Object`, and you have to manually cast that parameter of the `compareTo` method to the desired type. I will do just that for a little while so that you don't have to worry about two new concepts at the same time.

All methods of an interface are automatically public. For that reason, it is not necessary to supply the keyword `public` when declaring a method in an interface.

Of course, there is an additional requirement that the interface syntax cannot express: When calling `x.compareTo(y)`, the `compareTo` method must *compare* the two objects and return an indication whether `x` or `y` is larger. The method is supposed to return a negative number if `x` is smaller than `y`, zero if they are equal, and a positive number otherwise.

This particular interface has a single method. Some interfaces have multiple methods. As you will see later, interfaces can also define constants. What is more important, however, is what interfaces *cannot* supply. Interfaces never have instance fields. Before Java 8, all methods in an interface were abstract. As you will see in Section 6.1.4 and Section 6.1.5, it is now possible to have other methods in interfaces. Of course, those methods cannot refer to instance fields—interfaces don't have any.

Now, suppose we want to use the `sort` method of the `Arrays` class to sort an array of `Employee` objects. Then the `Employee` class must *implement* the `Comparable` interface.

To make a class implement an interface, you carry out two steps:

1. You declare that your class intends to implement the given interface.
2. You supply definitions for all methods in the interface.

To declare that a class implements an interface, use the `implements` keyword:

```
class Employee implements Comparable
```

Of course, now the `Employee` class needs to supply the `compareTo` method. Let's suppose that we want to compare employees by their salary. Here is an implementation of the `compareTo` method:

```
public int compareTo(Object otherObject)
{
    Employee other = (Employee) otherObject;
    return Double.compare(salary, other.salary);
}
```

Here, we use the static `Double.compare` method that returns a negative if the first argument is less than the second argument, 0 if they are equal, and a positive value otherwise.



Caution: In the interface declaration, the `compareTo` method was not declared `public` because all methods in an *interface* are automatically `public`. However, when implementing the interface, you must declare the method as `public`. Otherwise, the compiler assumes that the method has package access—the default for a *class*. The compiler then complains that you're trying to supply a more restrictive access privilege.

We can do a little better by supplying a type parameter for the generic `Comparable` interface:

```
class Employee implements Comparable<Employee>
{
    public int compareTo(Employee other)
    {
        return Double.compare(salary, other.salary);
    }
    . . .
}
```

Note that the unsightly cast of the `Object` parameter has gone away.



Tip: The `compareTo` method of the `Comparable` interface returns an integer. If the objects are not equal, it does not matter what negative or positive value you return. This flexibility can be useful when you are comparing integer fields. For example, suppose each employee has a unique integer `id` and you want to sort by the employee ID number. Then you can simply return `id - other.id`. That value will be some negative value if the first ID number is less than the other, 0 if they are the same ID, and some positive value otherwise. However, there is one caveat: The range of the integers must be small enough so that the subtraction does not overflow. If you know that the IDs are not negative or that their absolute value is at most $(\text{Integer.MAX_VALUE} - 1) / 2$, you are safe. Otherwise, call the static `Integer.compare` method.

Of course, the subtraction trick doesn't work for floating-point numbers. The difference `salary - other.salary` can round to 0 if the salaries are close together but not identical. The call `Double.compare(x, y)` simply returns -1 if $x < y$ or 1 if $x > y$.



Note: The documentation of the `Comparable` interface suggests that the `compareTo` method should be compatible with the `equals` method. That is, `x.compareTo(y)` should be zero exactly when `x.equals(y)`. Most classes in the Java API that implement `Comparable` follow this advice.

A notable exception is `BigDecimal`. Consider `x = new BigDecimal("1.0")` and `y = new BigDecimal("1.00")`. Then `x.equals(y)` is false because the numbers differ in precision. But `x.compareTo(y)` is zero. Ideally, it shouldn't be, but there is no obvious way of deciding which one should come first.

Another exception is `StringBuilder`, which implements `Comparable` but does not override `equals`:

```
StringBuilder x = new StringBuilder("Hello");
StringBuilder y = new StringBuilder("Hello");
x.equals(y) // false
x.compareTo(y) // 0
```



Caution: There are minor differences between comparison operators with double operand and the corresponding methods of the `Double` class.

The first issue is negative zero, or `-0.0`. When compared with a relational operator such as `==` or `<`, it is indistinguishable from `0.0`:

```
-0.0 == 0.0 // true
-0.0 < 0.0 // false
```

However, wrapped into `Double` instances, they are different:

```
Double.valueOf(-0.0).equals(Double.valueOf(0.0)) // false
Double.valueOf(-0.0).compareTo(Double.valueOf(0.0)) // -1
```

The other issue is `Double.NaN`. Any comparison with a relational operator where an operand is `NaN` returns `false`:

```
Double.NaN == Double.NaN // false
Double.NaN < Double.NaN // false
```

However, wrapped into a `Double` value, it behaves differently:

```
Double.valueOf(Double.NaN).equals(Double.valueOf(Double.NaN)) // true
Double.valueOf(Double.NaN).compareTo(Double.valueOf(Double.NaN)) // 0
```

The static `Double.compare` method follows the logic of the wrapper class:

```
Double.compare(-0.0, 0.0) // -1
```

Remarkably, `Double.NaN` is deemed larger than `Double.POSITIVE_INFINITY`:

```
Double.compare(Double.POSITIVE_INFINITY, Double.NaN) // -1
```

Note that the `equals` method of a record with double components uses `Double.compare`.

Now you saw what a class must do to avail itself of the sorting service—it must implement a `compareTo` method. That’s eminently reasonable. There needs to be some way for the sort method to compare objects. But why can’t the `Employee` class simply provide a `compareTo` method without implementing the `Comparable` interface?

The reason for interfaces is that the Java programming language is *strongly typed*. When making a method call, the compiler needs to be able to check that the method actually exists. Somewhere in the sort method will be statements like this:

```
if (a[i].compareTo(a[j]) > 0)
{
    // rearrange a[i] and a[j]
    . . .
}
```

The compiler must know that `a[i]` actually has a `compareTo` method. If `a` is an array of `Comparable` objects, then the existence of the method is assured because every class that implements the `Comparable` interface must supply the method.



Note: You would expect that the sort method in the Arrays class is defined to accept a Comparable[] array so that the compiler can complain if anyone ever calls sort with an array whose element type doesn't implement the Comparable interface. Sadly, that is not the case. Instead, the sort method accepts an Object[] array and uses a clumsy cast:

```
// approach used in the standard library--not recommended
if (((Comparable) a[i]).compareTo(a[j]) > 0)
{
    // rearrange a[i] and a[j]
    . . .
}
```

If a[i] does not belong to a class that implements the Comparable interface, the virtual machine throws an exception.

Listing 6.1 presents the full code for sorting an array of instances of the class Employee (Listing 6.2).

Listing 6.1 interfaces/EmployeeSortTest.java

```
1 package interfaces;
2
3 import java.util.*;
4
5 /**
6  * This program demonstrates the use of the Comparable interface.
7  * @version 1.30 2004-02-27
8  * @author Cay Horstmann
9  */
10 public class EmployeeSortTest
11 {
12     public static void main(String[] args)
13     {
14         var staff = new Employee[3];
15
16         staff[0] = new Employee("Harry Hacker", 35000);
17         staff[1] = new Employee("Carl Cracker", 75000);
18         staff[2] = new Employee("Tony Tester", 38000);
19
20         Arrays.sort(staff);
21
22         // print out information about all Employee objects
23         for (Employee e : staff)
24             System.out.println("name=" + e.getName() + ",salary=" + e.getSalary());
25     }
26 }
```

Listing 6.2 interfaces/Employee.java

```
1 package interfaces;
2
3 public class Employee implements Comparable<Employee>
4 {
5     private String name;
6     private double salary;
7
8     public Employee(String name, double salary)
9     {
10         this.name = name;
11         this.salary = salary;
12     }
13
14     public String getName()
15     {
16         return name;
17     }
18
19     public double getSalary()
20     {
21         return salary;
22     }
23
24     public void raiseSalary(double byPercent)
25     {
26         double raise = salary * byPercent / 100;
27         salary += raise;
28     }
29
30     /**
31      * Compares employees by salary.
32      * @param other another Employee object
33      * @return a negative value if this employee has a lower salary than
34      * other, 0 if the salaries are the same, a positive value otherwise
35      */
36     public int compareTo(Employee other)
37     {
38         return Double.compare(salary, other.salary);
39     }
40 }
```

java.lang.Comparable<T> 1.0

- `int compareTo(T other)`
compares this object with other and returns a negative integer if this object is less than other, zero if they are equal, and a positive integer otherwise.

java.util.Arrays 1.2

- `static void sort(Object[] a)`
sorts the elements in the array `a`. All elements in the array must belong to classes that implement the `Comparable` interface, and they must all be comparable to each other.

java.lang.Integer 1.0

- `static int compare(int x, int y)` **7**
returns a negative integer if $x < y$, zero if x and y are equal, and a positive integer otherwise.

java.lang.Double 1.0

- `static int compare(double x, double y)` **1.4**
returns a negative integer if $x < y$, zero if x and y are equal, and a positive integer otherwise.



Note: According to the language standard: “The implementor must ensure $\text{sgn}(x.\text{compareTo}(y)) = -\text{sgn}(y.\text{compareTo}(x))$ for all x and y . (This implies that $x.\text{compareTo}(y)$ must throw an exception if $y.\text{compareTo}(x)$ throws an exception.)” Here, sgn is the *sign* of a number: $\text{sgn}(n)$ is -1 if n is negative, 0 if n equals 0 , and 1 if n is positive. In plain English, if you flip the arguments of `compareTo`, the sign (but not necessarily the actual value) of the result must also flip.

As with the `equals` method, problems can arise when inheritance comes into play.

Since `Manager` extends `Employee`, it implements `Comparable<Employee>` and not `Comparable<Manager>`. If `Manager` chooses to override `compareTo`, it must be prepared to compare managers to employees. It can’t simply cast an employee to a manager:

```
class Manager extends Employee
{
    public int compareTo(Employee other)
    {
        Manager otherManager = (Manager) other; // NO
        . . .
    }
    . . .
}
```

That violates the “antisymmetry” rule. If *x* is an *Employee* and *y* is a *Manager*, then the call *x.compareTo(y)* doesn’t throw an exception—it simply compares *x* and *y* as employees. But the reverse, *y.compareTo(x)*, throws a *ClassCastException*.

This is the same situation as with the *equals* method discussed in Chapter 5, and the remedy is the same. There are two distinct scenarios.

If subclasses have different notions of comparison, then you should outlaw comparison of objects that belong to different classes. Each *compareTo* method should start out with the test

```
if (getClass() != other.getClass()) throw new ClassCastException();
```

If there is a common algorithm for comparing subclass objects, simply provide a single *compareTo* method in the superclass and declare it as *final*.

For example, suppose you want managers to be better than regular employees, regardless of salary. What about other subclasses such as *Executive* and *Secretary*? If you need to establish a pecking order, supply a method such as *rank* in the *Employee* class. Have each subclass override *rank*, and implement a single *compareTo* method that takes the rank values into account.

6.1.2. Properties of Interfaces

Interfaces are not classes. In particular, you can never use the *new* operator to instantiate an interface:

```
x = new Comparable(. . .); // ERROR
```

However, even though you can’t construct interface objects, you can still declare interface variables.

```
Comparable x; // OK
```

An interface variable must refer to an object of a class that implements the interface:

```
x = new Employee(. . .); // OK provided Employee implements Comparable
```

Next, just as you use *instanceof* to check whether an object is of a specific class, you can use *instanceof* to check whether an object implements an interface:

```
if (anObject instanceof Comparable) { . . . }
```

Just as you can build hierarchies of classes, you can extend interfaces. This allows for multiple chains of interfaces that go from a greater degree of generality to a greater degree of specialization. For example, suppose you had an interface called *Moveable*.

```
public interface Moveable
{
    void move(double x, double y);
}
```

Then, you could imagine an interface called `Powered` that extends it:

```
public interface Powered extends Moveable
{
    double milesPerGallon();
}
```

Although you cannot put instance fields in an interface, you can supply constants in them. For example:

```
public interface Powered extends Moveable
{
    double milesPerGallon();
    double SPEED_LIMIT = 95; // a public static final constant
}
```

Just as methods in an interface are automatically public, fields are always public static final.



Note: It is legal to tag interface methods as public, and fields as public static final. Some programmers do that, either out of habit or for greater clarity. However, the Java Language Specification recommends that the redundant keywords not be supplied, and I follow that recommendation.

While each class can have only one superclass, classes can implement *multiple* interfaces. This gives you the maximum amount of flexibility in defining a class's behavior. For example, the Java programming language has an important interface built into it, called `Cloneable`. (This interface is discussed in detail in Section 6.1.9.) If your class implements `Cloneable`, the clone method in the `Object` class will make an exact copy of your class's objects. If you want both cloneability and comparability, simply implement both interfaces. Use commas to separate the interfaces that you want to implement:

```
class Employee implements Cloneable, Comparable
```



Note: Records and enumeration classes cannot extend other classes (since they implicitly extend the `Record` and `Enum` class). However, they can implement interfaces.



Note: Interfaces can be sealed. As with sealed classes, the direct subtypes (which can be classes or interfaces) must be declared in a `permits` clause or be located in the same source file.

6.1.3. Interfaces and Abstract Classes

If you read the section about abstract classes in Chapter 5, you may wonder why the designers of the Java programming language bothered with introducing the concept of interfaces. Why can't `Comparable` simply be an abstract class:

```
abstract class Comparable // why not?
{
    public abstract int compareTo(Object other);
}
```

The `Employee` class would then simply extend this abstract class and supply the `compareTo` method:

```
class Employee extends Comparable // why not?
{
    public int compareTo(Object other) { . . . }
}
```

There is, unfortunately, a major problem with using an abstract base class to express a generic property. A class can only extend a single class. Suppose the `Employee` class already extends a different class, say, `Person`. Then it can't extend a second class.

```
class Employee extends Person, Comparable // ERROR
```

But each class can implement as many interfaces as it likes:

```
class Employee extends Person implements Comparable // OK
```

Other programming languages, in particular C++, allow a class to have more than one superclass. This feature is called *multiple inheritance*. The designers of Java chose not to support multiple inheritance, because it makes the language either very complex (as in C++) or less efficient (as in Eiffel).

Instead, interfaces afford most of the benefits of multiple inheritance while avoiding the complexities and inefficiencies.



C++ Note: C++ has multiple inheritance and all the complications that come with it, such as virtual base classes, dominance rules, and transverse pointer casts. Few

C++ programmers use multiple inheritance, and some say it should never be used. Other programmers recommend using multiple inheritance only for the “mix-in” style of inheritance. In the mix-in style, a primary base class describes the parent object, and additional base classes (the so-called mix-ins) may supply auxiliary characteristics. That style is similar to a Java class with a single superclass and additional interfaces.



Tip: You have seen the `CharSequence` interface in Chapter 3. Both `String` and `StringBuilder` (as well as a few more esoteric string-like classes) implement this interface. The interface contains methods that are common to all classes that manage sequences of characters. A common interface encourages programmers to write methods that use the `CharSequence` interface. Those methods work with instances of `String`, `StringBuilder`, and the other string-like classes.

Sadly, the `CharSequence` interface is rather paltry. You can get the length, iterate over the code points or code units, extract subsequences, and lexicographically compare two sequences. Java 17 adds an `isEmpty` method.

If you process strings, and those operations suffice for your tasks, accept `CharSequence` instances instead of strings.

6.1.4. Static and Private Methods

As of Java 8, you are allowed to add static methods to interfaces. There was never a technical reason why this should be outlawed. It simply seemed to be against the spirit of interfaces as abstract specifications.

Previously, it had been common to place static methods in companion classes. In the standard library, you'll find pairs of interfaces and utility classes such as `Collection/Collections`.

As an example, you can construct a path to a file or directory from a URI, or from a sequence of strings, using static methods in the `Path` interface:

```
public interface Path
{
    public static Path of(URI uri) { . . . }
    public static Path of(String first, String... more) { . . . }
    . . .
}
```

In previous versions of Java, there was a separate `Paths` class to hold these methods. Nowadays, there is no longer a reason to provide a separate companion class for utility methods.

Methods in an interface can be private. A private method can be static or an instance method. Since private methods can only be used in the methods of the interface itself, their use is limited to being helper methods for the other methods of the interface.

6.1.5. Default Methods

You can supply a *default* implementation for any interface method. You must tag such a method with the default modifier.

```
public interface Comparable<T>
{
    default int compareTo(T other) { return 0; }
    // by default, all elements are the same
}
```

Of course, that is not very useful since every realistic implementation of `Comparable` would override this method. But there are other situations where default methods can be useful. For example, in Chapter 9 you will see an `Iterator` interface for visiting elements in a data structure. It declares a `remove` method as follows:

```
public interface Iterator<E>
{
    boolean hasNext();
    E next();
    default void remove() { throw new UnsupportedOperationException("remove"); }
    . . .
}
```

If you implement an iterator, you need to provide the `hasNext` and `next` methods. There are no defaults for these methods—they depend on the data structure that you are traversing. But if your iterator is read-only, you don't have to worry about the `remove` method.

A default method can call other methods. For example, a `Collection` interface can define a convenience method

```
public interface Collection
{
    int size(); // an abstract method
    default boolean isEmpty() { return size() == 0; }
    . . .
}
```

Then a programmer implementing `Collection` doesn't have to worry about implementing an `isEmpty` method.



Note: The Collection interface in the Java API does not actually do this. Instead, there is a class `AbstractCollection` that implements `Collection` and defines `isEmpty` in terms of `size`. Implementors of a collection are advised to extend `AbstractCollection`. That technique is obsolete. Just implement the methods in the interface.

An important use for default methods is *interface evolution*. Consider, for example, the `Collection` interface that has been a part of Java for many years. Suppose that a long time ago, you provided a class

```
public class Bag implements Collection
```

Later, in Java 8, a `stream` method was added to the interface.

Suppose the `stream` method was not a default method. Then the `Bag` class would no longer compile since it doesn't implement the new method. Adding a nondefault method to an interface is not *source-compatible*.

But suppose you don't recompile the class and simply use an old JAR file containing it. The class will still load, even with the missing method. Programs can still construct `Bag` instances, and nothing bad will happen. (Adding a method to an interface is *binary compatible*.) However, if a program calls the `stream` method on a `Bag` instance, an `AbstractMethodError` occurs.

Making the method a default method solves both problems. The `Bag` class will again compile. And if the class is loaded without being recompiled and the `stream` method is invoked on a `Bag` instance, the `Collection.stream` method is called.

6.1.6. Resolving Default Method Conflicts

What happens if the exact same method is defined as a default method in one interface and then again as a method of a superclass or another interface? Languages such as Scala and C++ have complex rules for resolving such ambiguities. Fortunately, the rules in Java are much simpler. Here they are:

1. Superclasses win. If a superclass provides a concrete method, default methods with the same name and parameter types are simply ignored.
2. Interfaces clash. If an interface provides a default method, and another interface contains a method with the same name and parameter types (default or not), then you must resolve the conflict by overriding that method.

Let's look at the second rule. Consider two interfaces with a `getName` method:

```
interface Person
{
    default String getName() { return ""; }
```

```

}

interface Named
{
    default String getName() { return getClass().getName() + "_" + hashCode(); }
}

```

What happens if you form a class that implements both of them?

```
class Student implements Person, Named { . . . }
```

The class inherits two inconsistent `getName` methods provided by the `Person` and `Named` interfaces. Instead of choosing one over the other, the Java compiler reports an error and leaves it up to the programmer to resolve the ambiguity. Simply provide a `getName` method in the `Student` class. In that method, you can choose one of the two conflicting methods, like this:

```

class Student implements Person, Named
{
    public String getName() { return Person.super.getName(); }
    . . .
}

```

Now assume that the `Named` interface does not provide a default implementation for `getName`:

```

interface Named
{
    String getName();
}

```

Can the `Student` class inherit the default method from the `Person` interface? This might be reasonable, but the Java designers decided in favor of uniformity. It doesn't matter how two interfaces conflict. If at least one interface provides an implementation, the compiler reports an error, and the programmer must resolve the ambiguity.

If neither interface provides a default for a shared method, then there is no conflict. An implementing class has two choices: implement the method, or leave it unimplemented. In the latter case, the class is itself abstract.

We just discussed name clashes between two interfaces. Now consider a class that extends a superclass and implements an interface, inheriting the same method from both. For example, suppose that `Person` is a class and `Student` is defined as

```
class Student extends Person implements Named { . . . }
```

In that case, only the superclass method matters, and any default method from the interface is simply ignored. In our example, `Student` inherits the `getName` method from `Person`,

and it doesn't make any difference whether the `Named` interface provides a default for `getName` or not. This is the "class wins" rule.

The "class wins" rule ensures compatibility with old versions of Java. If you add default methods to an interface, it has no effect on code that worked before there were default methods.



Caution: You can never make a default method that redefines one of the methods in the `Object` class. For example, you can't define a default method for `toString` or `equals`, even though that might be attractive for interfaces such as `List`. As a consequence of the "class wins" rule, such a method could never win against `Object.toString` or `Object.equals`.

6.1.7. Interfaces and Callbacks

A common pattern in programming is the *callback* pattern. In this pattern, you specify the action that should occur whenever a particular event happens. For example, you may want a particular action to occur when a button is clicked or a menu item is selected. However, as you have not yet seen how to implement user interfaces, we will consider a similar but simpler situation.

The `javax.swing` package contains a `Timer` class that is useful if you want to be notified whenever a time interval has elapsed. For example, if a part of your program contains a clock, you can ask to be notified every second so that you can update the clock face.

When you construct a timer, you set the time interval and tell it what it should do whenever the time interval has elapsed.

How do you tell the timer what it should do? In many programming languages, you supply the name of a function that the timer should call periodically. However, the classes in the Java standard library take an object-oriented approach. You pass an object of some class. The timer then calls one of the methods on that object. Passing an object is more flexible than passing a function because the object can carry additional information.

Of course, the timer needs to know what method to call. The timer requires that you specify an object of a class that implements the `ActionListener` interface of the `java.awt.event` package. Here is that interface:

```
public interface ActionListener
{
    void actionPerformed(ActionEvent event);
}
```

The timer calls the `actionPerformed` method when the time interval has expired.

Suppose you want to print a message “At the tone, the time is . . .”, followed by a beep, once every second. You would define a class that implements the `ActionListener` interface. You would then place whatever statements you want to have executed inside the `actionPerformed` method.

```
class TimePrinter implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        System.out.println("At the tone, the time is "
            + Instant.ofEpochMilli(event.getWhen()));
        Toolkit.getDefaultToolkit().beep();
    }
}
```

Note the `ActionEvent` parameter of the `actionPerformed` method. This parameter gives information about the event, such as the time when the event happened. The call `event.getWhen()` returns the event time, measured in milliseconds since the “epoch” (January 1, 1970). By passing it to the static `Instant.ofEpochMilli` method, we get a more readable description.

Next, construct an object of this class and pass it to the `Timer` constructor.

```
var listener = new TimePrinter();
Timer t = new Timer(1000, listener);
```

The first argument of the `Timer` constructor is the time interval that must elapse between notifications, measured in milliseconds. We want to be notified every second. The second argument is the listener object.

Finally, start the timer.

```
t.start();
```

Every second, a message like

```
At the tone, the time is 2017-12-16T05:01:49.550Z
```

is displayed, followed by a beep.



Caution: Be sure to import `javax.swing.Timer`. There is also a `java.util.Timer` class that is slightly different.

Listing 6.3 puts the timer and its action listener to work. After the timer is started, the program puts up a message dialog and waits for the user to click the OK button to stop.

While the program waits for the user, the current time is displayed every second. (If you omit the dialog, the program would terminate as soon as the main method exits.)

Listing 6.3 timer/TimerTest.java

```
1 package timer;
2
3 /**
4  * @version 1.02 2017-12-14
5  * @author Cay Horstmann
6  */
7
8 import java.awt.*;
9 import java.awt.event.*;
10 import java.time.*;
11 import javax.swing.*;
12
13 public class TimerTest
14 {
15     public static void main(String[] args)
16     {
17         var listener = new TimePrinter();
18
19         // construct a timer that calls the listener once every second
20         var timer = new Timer(1000, listener);
21         timer.start();
22
23         // keep program running until the user selects "OK"
24         JOptionPane.showMessageDialog(null, "Quit program?");
25         System.exit(0);
26     }
27 }
28
29 class TimePrinter implements ActionListener
30 {
31     public void actionPerformed(ActionEvent event)
32     {
33         System.out.println("At the tone, the time is " + Instant.ofEpochMilli(event.getWhen()));
34         Toolkit.getDefaultToolkit().beep();
35     }
36 }
```

javax.swing.JOptionPane 1.2

- `static void showMessageDialog(Component parent, Object message)`
displays a dialog box with a message prompt and an OK button. The dialog is centered over the parent component. If parent is null, the dialog is centered on the screen.

javax.swing.Timer 1.2

- `Timer(int interval, ActionListener listener)`
constructs a timer that notifies listener whenever interval milliseconds have elapsed.
- `void start()`
starts the timer. Once started, the timer calls `actionPerformed` on its listeners.
- `void stop()`
stops the timer. Once stopped, the timer no longer calls `actionPerformed` on its listeners.

java.awt.Toolkit 1.0

- `static Toolkit getDefaultToolkit()`
gets the default toolkit. A toolkit contains information about the GUI environment.
- `void beep()`
emits a beep sound.

6.1.8. The Comparator Interface

In Section 6.1.1, you have seen how you can sort an array of objects, provided they are instances of classes that implement the `Comparable` interface. For example, you can sort an array of strings since the `String` class implements `Comparable<String>`, and the `String.compareTo` method compares strings in dictionary order.

Now suppose we want to sort strings by increasing length, not in dictionary order. We can't have the `String` class implement the `compareTo` method in two ways—and at any rate, the `String` class isn't ours to modify.

To deal with this situation, there is a second version of the `Arrays.sort` method whose parameters are an array and a *comparator*—an instance of a class that implements the `Comparator` interface.

```
public interface Comparator<T>
{
    int compare(T first, T second);
}
```

To compare strings by length, define a class that implements `Comparator<String>`:

```
class LengthComparator implements Comparator<String>
{
    public int compare(String first, String second)
    {
```



```
        return first.length() - second.length();
    }
}
```

To actually do the comparison, you need to make an instance:

```
var comp = new LengthComparator();
if (comp.compare(words[i], words[j]) > 0) . . .
```

Contrast this call with `words[i].compareTo(words[j])`. The `compare` method is called on the comparator object, not the string itself.



Note: Even though the `LengthComparator` object has no state, you still need to make an instance of it. You need the instance to call the `compare` method—it is not a static method.

To sort an array, pass a `LengthComparator` object to the `Arrays.sort` method:

```
String[] friends = { "Peter", "Paul", "Mary" };
Arrays.sort(friends, new LengthComparator());
```

Now the array is either `["Paul", "Mary", "Peter"]` or `["Mary", "Paul", "Peter"]`.

You will see in Section 6.2 how to use a `Comparator` much more easily with a lambda expression.



Note: The `String` class provides a `Comparator` for case-insensitive comparison. Here is how you can use it:

```
Arrays.sort(friends, String.CASE_INSENSITIVE_ORDER);
```



Caution: Do not try to shuffle an array by sorting it with a comparator that randomly returns positive or negative integers.

There are three rules that a comparator needs to fulfill:

1. Reflexivity: When *x* and *y* are equal, the comparator yields 0.
2. Antisymmetry: When swapping the arguments of the comparator, the sign of the result is swapped.
3. Transitivity: When *x* comes before *y* and *y* comes before *z*, then *x* must come before *z*.

The algorithm that `Arrays.sort` uses (called “Timsort”) doesn't check these rules for all elements, but it can sometimes detect a rule violation at a trivial cost. Then it throws an exception with the message “Comparison method violates its general contract!”. With an array of 1,000 elements, the chance of this occurring with a random comparator is over 10%.

The `Collections.shuffle` method randomly shuffles a list. To shuffle an array, first turn it into a list and then shuffle that.

6.1.9. Object Cloning

In this section, we discuss the `Cloneable` interface that indicates that a class has provided a safe clone method. Since cloning is not all that common, and the details are quite technical, you may just want to glance at this material until you need it.

To understand what cloning means, recall what happens when you make a copy of a variable holding an object reference. The original and the copy are references to the same object (see Figure 6.1). This means a change to either variable also affects the other.

```
var original = new Employee("John Public", 50000);
Employee copy = original;
copy.raiseSalary(10); // oops--also changed original
```

If you would like copy to be a new object that begins its life being identical to original but whose state can diverge over time, use the clone method.

```
Employee copy = original.clone();
copy.raiseSalary(10); // OK--original unchanged
```

But it isn't quite so simple. The clone method is a protected method of `Object`, which means that your code cannot simply call it. Only the `Employee` class can clone `Employee` objects. There is a reason for this restriction. Think about the way in which the `Object` class can implement clone. It knows nothing about the object at all, so it can make only a field-by-field copy. If all instance fields in the object are numbers or other basic types, copying the fields is just fine. But if the object contains references to subobjects, then copying the field gives you another reference to the same subobject, so the original and the cloned objects still share some information.

To visualize that, consider the `Employee` class that was introduced in Chapter 4. Figure 6.2 shows what happens when you use the clone method of the `Object` class to clone such an `Employee` object. As you can see, the default cloning operation is “shallow”—it doesn't clone objects that are referenced inside other objects. (The figure shows a shared `Date` object. For reasons that will become clear shortly, this example uses a version of the `Employee` class in which the hire day is represented as a `Date`.)

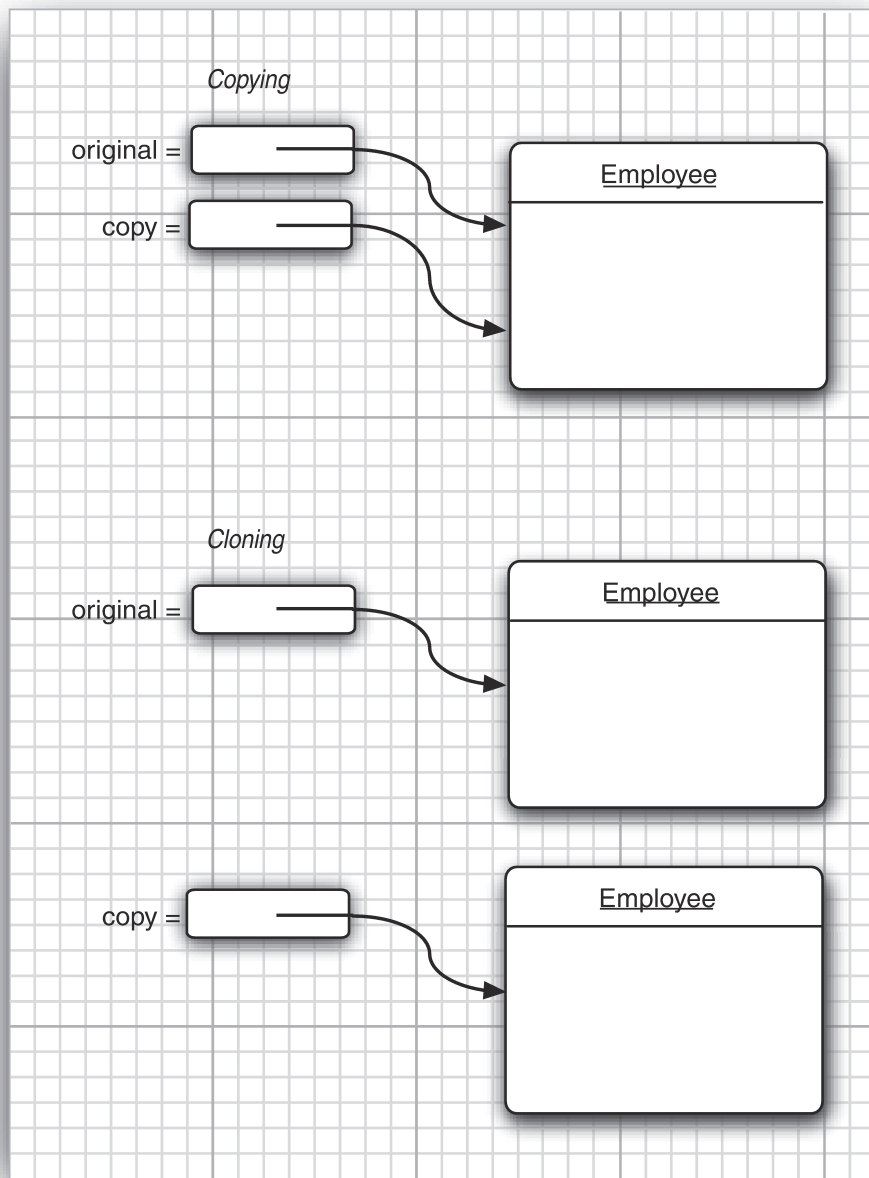


Figure 6.1: Copying and cloning

Does it matter if the copy is shallow? It depends. If the subobject shared between the original and the shallow clone is *immutable*, then the sharing is safe. This certainly happens if the subobject belongs to an immutable class, such as `String`. Alternatively, the subobject may simply remain constant throughout the lifetime of the object, with no mutators touching it and no methods yielding a reference to it.

Quite frequently, however, subobjects are mutable, and you must redefine the clone method to make a *deep copy* that clones the subobjects as well. In our example, the `hireDay` field is

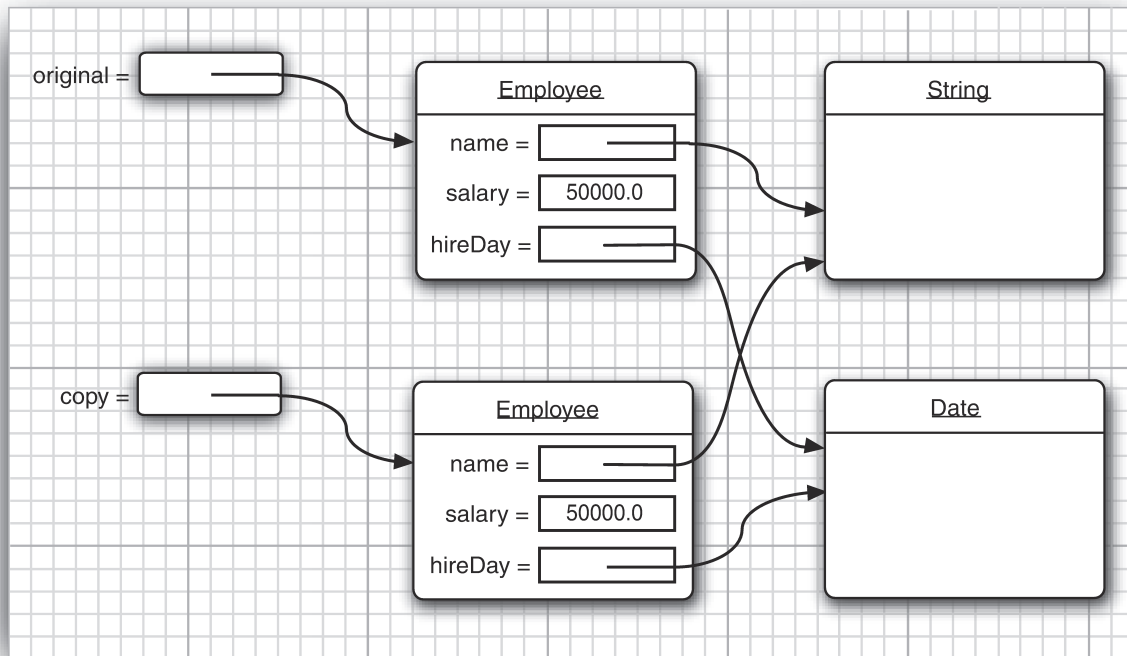


Figure 6.2: A shallow copy

a `Date`, which is mutable, so it too must be cloned. (For that reason, this example uses a field of type `Date`, not `LocalDate`, to demonstrate the cloning process. Had `hireDay` been an instance of the immutable `LocalDate` class, no further action would have been required.)

For every class, you need to decide whether

1. The default clone method is good enough;
2. The default clone method can be patched up by calling clone on the mutable subobjects; or
3. clone should not be attempted.

The third option is actually the default. To choose either the first or the second option, a class must

1. Implement the `Cloneable` interface; and
2. Redefine the clone method with the public access modifier.



Note: The clone method is declared protected in the `Object` class, so that your code can't simply call `anObject.clone()`. But aren't protected methods accessible from any subclass, and isn't every class a subclass of `Object`? Fortunately, the rules for

protected access are more subtle (see Chapter 5). A subclass can call a protected clone method only to clone *its own* objects. You must redefine clone to be public to allow objects to be cloned by any method.

In this case, the appearance of the Cloneable interface has nothing to do with the normal use of interfaces. In particular, it does *not* specify the clone method—that method is inherited from the Object class. The interface merely serves as a tag, indicating that the class designer understands the cloning process. Objects are so paranoid about cloning that they generate a checked exception if an object requests cloning but does not implement that interface.



Note: The Cloneable interface is one of a handful of *tagging interfaces* that Java provides. (Some programmers call them *marker interfaces*.) Recall that the usual purpose of an interface such as Comparable is to ensure that a class implements a particular method or set of methods. A tagging interface has no methods; its only purpose is to allow the use of instanceof in a type inquiry:

```
if (obj instanceof Cloneable) . . .
```

I recommend that you do not use tagging interfaces in your own programs.

Even if the default (shallow copy) implementation of clone is adequate, you still need to implement the Cloneable interface, redefine clone to be public, and call super.clone(). Here is an example:

```
class Employee implements Cloneable
{
    // public access, change return type
    public Employee clone() throws CloneNotSupportedException
    {
        return (Employee) super.clone();
    }
    . . .
}
```



Note: Note that in the Object class, the clone method has return type Object. In a subclass, you can specify the correct return type for your clone methods. This is an example of covariant return types (see Chapter 5).

The clone method that you just saw adds no functionality to the shallow copy provided by Object.clone. It merely makes the method public. To make a deep copy, you have to work harder and clone the mutable instance fields.

Here is an example of a clone method that creates a deep copy:

```
class Employee implements Cloneable
{
    . . .
    public Employee clone() throws CloneNotSupportedException
    {
        // call Object.clone()
        Employee cloned = (Employee) super.clone();

        // clone mutable fields
        cloned.hireDay = (Date) hireDay.clone();

        return cloned;
    }
}
```

The clone method of the Object class threatens to throw a CloneNotSupportedException—it does that whenever clone is invoked on an object whose class does not implement the Cloneable interface. Of course, the Employee and Date classes implement the Cloneable interface, so the exception won't be thrown. However, the compiler does not know that. Therefore, we declared the exception:

```
public Employee clone() throws CloneNotSupportedException
```



Note: Would it be better to catch the exception instead? (See Chapter 7 for details on catching exceptions.)

```
public Employee clone()
{
    try
    {
        Employee cloned = (Employee) super.clone();
        . . .
    }
    catch (CloneNotSupportedException e) { return null; }
    // this won't happen, since we are Cloneable
}
```

This is appropriate for final classes. Otherwise, it is better to leave the throws specifier in place. That gives subclasses the option of throwing a CloneNotSupportedException if they can't support cloning.

You have to be careful about cloning of subclasses. For example, once you have defined the clone method for the Employee class, anyone can use it to clone Manager objects. Can the

Employee clone method do the job? It depends on the fields of the Manager class. In our case, there is no problem because the bonus field has primitive type. But Manager might have acquired fields that require a deep copy or are not cloneable. There is no guarantee that the implementor of the subclass has fixed clone to do the right thing. For that reason, the clone method is declared as protected in the Object class. But you don't have that luxury if you want the users of your classes to invoke clone.

Should you implement clone in your own classes? If your clients need to make deep copies, then you probably should. Some authors feel that you should avoid clone altogether and instead implement another method for the same purpose. I agree that clone is rather awkward, but you'll run into the same issues if you shift the responsibility to another method. At any rate, cloning is less common than you may think. Less than five percent of the classes in the standard library implement clone.

The program in Listing 6.4 clones an instance of the class Employee (Listing 6.5), then invokes two mutators. The raiseSalary method changes the value of the salary field, whereas the setHireDay method changes the state of the hireDay field. Neither mutation affects the original object because clone has been defined to make a deep copy.



Note: All array types have a clone method that is public, not protected. You can use it to make a new array that contains copies of all elements. For example:

```
int[] luckyNumbers = { 2, 3, 5, 7, 11, 13 };
int[] cloned = luckyNumbers.clone();
cloned[5] = 12; // doesn't change luckyNumbers[5]
```

Listing 6.4 clone/CloneTest.java

```
1 package clone;
2
3 /**
4  * This program demonstrates cloning.
5  * @version 1.11 2018-03-16
6  * @author Cay Horstmann
7  */
8 public class CloneTest
9 {
10     public static void main(String[] args) throws CloneNotSupportedException
11     {
12         var original = new Employee("John Q. Public", 50000);
13         original.setHireDay(2000, 1, 1);
14         Employee copy = original.clone();
15         copy.raiseSalary(10);
16         copy.setHireDay(2002, 12, 31);
17         System.out.println("original=" + original);
```

```
18     System.out.println("copy=" + copy);
19 }
20 }
```

Listing 6.5 clone/Employee.java

```
1  package clone;
2
3  import java.time.*;
4  import java.util.*;
5
6  public class Employee implements Cloneable
7  {
8      private String name;
9      private double salary;
10     private Date hireDay;
11
12     public Employee(String name, double salary)
13     {
14         this.name = name;
15         this.salary = salary;
16         hireDay = new Date();
17     }
18
19     public Employee clone() throws CloneNotSupportedException
20     {
21         // call Object.clone()
22         Employee cloned = (Employee) super.clone();
23
24         // clone mutable fields
25         cloned.hireDay = (Date) hireDay.clone();
26
27         return cloned;
28     }
29
30     /**
31      * Set the hire day to a given date.
32      * @param year the year of the hire day
33      * @param month the month of the hire day
34      * @param day the day of the hire day
35      */
36     public void setHireDay(int year, int month, int day)
37     {
38         long epochMillis = LocalDate.of(year, month, day)
39             .atStartOfDay(ZoneId.systemDefault())
40             .toEpochSecond() * 1000;
41
42         // example of instance field mutation
43         hireDay.setTime(epochMillis);
44     }
45 }
```



```
45 |  
46 | public void raiseSalary(double byPercent)  
47 | {  
48 |     double raise = salary * byPercent / 100;  
49 |     salary += raise;  
50 | }  
51 |  
52 | public String toString()  
53 | {  
54 |     return "Employee[name=" + name + ",salary=" + salary + ",hireDay=" + hireDay + "];"  
55 | }  
56 | }
```

6.2. Lambda Expressions

In the following sections, you will learn how to use lambda expressions for defining blocks of code with a concise syntax, and how to write code that consumes lambda expressions.

6.2.1. Why Lambdas?

A lambda expression is a block of code that you can pass around so it can be executed later, once or multiple times. Before getting into the syntax (or even the curious name), let's step back and observe where we have used such code blocks in Java.

In Section 6.1.7, you saw how to do work in timed intervals. Put the work into the `actionPerformed` method of an `ActionListener`:

```
class Worker implements ActionListener  
{  
    public void actionPerformed(ActionEvent event)  
    {  
        // do some work  
    }  
}
```

Then, when you want to repeatedly execute this code, you construct an instance of the `Worker` class. You then submit the instance to a `Timer` object.

The key point is that the `actionPerformed` method contains code that you want to execute later.

Or consider sorting with a custom comparator. If you want to sort strings by length instead of the default dictionary order, you can pass a `Comparator` object to the `sort` method:

```
class LengthComparator implements Comparator<String>  
{  
    public int compare(String first, String second)
```

```
    {  
        return first.length() - second.length();  
    }  
}  
.  
.  
.  
Arrays.sort(strings, new LengthComparator());
```

The compare method isn't called right away. Instead, the sort method keeps calling the compare method, rearranging the elements if they are out of order, until the array is sorted. You give the sort method a snippet of code needed to compare elements, and that code is integrated into the rest of the sorting logic, which you'd probably not care to reimplement.

Both examples have something in common. A block of code was passed to someone—a timer, or a sort method. That code block was called at some later time.

In early versions of Java, giving someone a block of code was not easy. You couldn't just pass code blocks around. Java is an object-oriented language, so you had to construct an object belonging to a class that has a method with the desired code.

In other languages, it is possible to work with blocks of code directly. The Java designers have resisted adding this feature for a long time. After all, a great strength of Java is its simplicity and consistency. A language can become an unmaintainable mess if it includes every feature that yields marginally more concise code. However, in those other languages it isn't just easier to spawn a thread or to register a button click handler; large swaths of their APIs are simpler, more consistent, and more powerful. In Java, one could have written similar APIs taking objects of classes that implement a particular interface, but such APIs would be unpleasant to use.

For some time, the question was not whether to augment Java for functional programming, but how to do it. It took several years of experimentation before a design emerged that is a good fit for Java. In the next section, you will see how you can work with blocks of code in Java.

6.2.2. The Syntax of Lambda Expressions

Consider again the sorting example from the preceding section. We pass code that checks whether one string is shorter than another. We compute

```
first.length() - second.length()
```

What are first and second? They are both strings. Java is a strongly typed language, and we must specify that as well:

```
(String first, String second) ->  
    first.length() - second.length()
```

You have just seen your first *lambda expression*. Such an expression is simply a block of code, together with the specification of any variables that must be passed to the code.

Why the name? Many years ago, before there were any computers, the logician Alonzo Church wanted to formalize what it means for a mathematical function to be effectively computable. (Curiously, there are functions that are known to exist, but nobody knows how to compute their values.) He used the Greek letter lambda (λ) to mark parameters. Had he known about the Java API, he would have written

```
 $\lambda$ first. $\lambda$ second.first.length() - second.length()
```



Note: Why the letter λ ? Did Church run out of other letters of the alphabet? Actually, the venerable *Principia Mathematica* used the ^ accent to denote free variables, which inspired Church to use an uppercase lambda Λ for parameters. But in the end, he switched to the lowercase version. Ever since, an expression with parameter variables has been called a lambda expression.

What you have just seen is a simple form of lambda expressions in Java: parameters, the \rightarrow arrow, and an expression. If the code carries out a computation that doesn't fit in a single expression, write it exactly like you would have written a method: enclosed in {} and with explicit return statements. For example,

```
(String first, String second)  $\rightarrow$ 
{
    if (first.length() < second.length()) return -1;
    else if (first.length() > second.length()) return 1;
    else return 0;
}
```

If a lambda expression has no parameters, you still supply empty parentheses, just as with a parameterless method:

```
()  $\rightarrow$  { return 1 + (int)(Math.random() * 6); }
```

If the parameter types of a lambda expression can be inferred, you can omit them. For example,

```
Comparator<String> comp =
    (first, second) // same as (String first, String second)
         $\rightarrow$  first.length() - second.length();
```

Here, the compiler can deduce that *first* and *second* must be strings because the lambda expression is assigned to a string comparator. (We will have a closer look at this assignment in the next section.)

If a method has a single parameter with inferred type, you can even omit the parentheses:

```
ActionListener listener = event ->
    System.out.println("The time is "
        + Instant.ofEpochMilli(event.getWhen()));
// instead of (event) -> . . . or (ActionEvent event) -> . . .
```

You never specify the result type of a lambda expression. It is always inferred from context. For example, the expression

```
(String first, String second) -> first.length() - second.length()
```

can be used in a context where a result of type `int` is expected.

Finally, you can use `var` to denote an inferred type. This isn't common. The syntax was invented for attaching annotations (see Chapter 11):

```
(@NonNull var first, @NonNull var second) -> first.length() - second.length()
```



Note: It is illegal for a lambda expression to return a value in some branches but not in others. For example, `(int x) -> { if (x >= 0) return 1; }` is invalid.



Preview Note: If a parameter of a lambda expression is never used, you can denote it with an underscore:

```
ActionListener listener = _ ->
    System.out.println("The action occurred at " + Instant.now());
Comparator<String> comp = (_, _) -> 0;
```

This is a preview feature in Java 21.

The program in Listing 6.6 shows how to use lambda expressions for a comparator and an action listener.

Listing 6.6 lambda/LambdaTest.java

```
1 package lambda;
2
3 import java.util.*;
4
5 import javax.swing.*;
6 import javax.swing.Timer;
7
```

```
8  /**
9   * This program demonstrates the use of lambda expressions.
10  * @version 1.0 2015-05-12
11  * @author Cay Horstmann
12  */
13  public class LambdaTest
14  {
15      public static void main(String[] args)
16      {
17          var planets = new String[] { "Mercury", "Venus", "Earth", "Mars",
18              "Jupiter", "Saturn", "Uranus", "Neptune" };
19          System.out.println(Arrays.toString(planets));
20          System.out.println("Sorted in dictionary order:");
21          Arrays.sort(planets);
22          System.out.println(Arrays.toString(planets));
23          System.out.println("Sorted by length:");
24          Arrays.sort(planets, (first, second) -> first.length() - second.length());
25          System.out.println(Arrays.toString(planets));
26
27          var timer = new Timer(1000, event ->
28              System.out.println("The time is " + new Date()));
29          timer.start();
30
31          // keep program running until user selects "OK"
32          JOptionPane.showMessageDialog(null, "Quit program?");
33          System.exit(0);
34      }
35  }
```

6.2.3. Functional Interfaces

As we discussed, there are many existing interfaces in Java that encapsulate blocks of code, such as `ActionListener` or `Comparator`. Lambdas are compatible with these interfaces.

You can supply a lambda expression whenever an object of an interface with a single abstract method is expected. Such an interface is called a *functional interface*.



Note: You may wonder why a functional interface must have a single *abstract* method. Aren't all methods in an interface abstract? Actually, it has always been possible for an interface to redeclare methods from the `Object` class such as `toString` or `clone`, and these declarations do not make the methods abstract. (Some interfaces in the Java API redeclare `Object` methods in order to attach javadoc comments. Check out the `Comparator` API for an example.) More importantly, as you saw in Section 6.1.5, interfaces can declare nonabstract methods.

To demonstrate the conversion to a functional interface, consider the `Arrays.sort` method. Its second parameter requires an instance of `Comparator`, an interface with a single method. Simply supply a lambda:

```
Arrays.sort(words,
    (first, second) -> first.length() - second.length());
```

Behind the scenes, the `Arrays.sort` method receives an object of some class that implements `Comparator<String>`. Invoking the `compare` method on that object executes the body of the lambda expression. The management of these objects and classes is completely implementation-dependent, and it can be much more efficient than using traditional inner classes. It is best to think of a lambda expression as a function, not an object, and to accept that it can be passed to a functional interface.

This conversion to interfaces is what makes lambda expressions so compelling. The syntax is short and simple. Here is another example:

```
var timer = new Timer(1000, event ->
{
    System.out.println("At the tone, the time is "
        + Instant.ofEpochMilli(event.getWhen()));
    Toolkit.getDefaultToolkit().beep();
});
```

That's a lot easier to read than the alternative with a class that implements the `ActionListener` interface.

In fact, conversion to a functional interface is the *only* thing that you can do with a lambda expression in Java. In other programming languages that support function literals, you can declare function types such as `(String, String) -> int`, declare variables of those types, and use the variables to save function expressions. However, the Java designers decided to stick with the familiar concept of interfaces instead of adding function types to the language.



Note: You can't even assign a lambda expression to a variable of type `Object`—`Object` is not a functional interface.

6.2.4. Function Types

The Java API defines a number of very generic functional interfaces in the `java.util.function` package. One of the interfaces, `BiFunction<T, U, R>`, describes functions with parameter types `T` and `U` and return type `R`. You can save your string comparison lambda in a variable of that type:

```
BiFunction<String, String, Integer> comp =  
    (first, second) -> first.length() - second.length();
```

Note that this interface does not help you with sorting. There is no `Arrays.sort` method that wants a `BiFunction`. If you have used a functional programming language before, you may find this curious. But for Java programmers, it's pretty natural. An interface such as `Comparator` has a specific purpose, not just a method with given parameter and return types. When you want to do something with lambda expressions, you still want to keep the purpose of the expression in mind, and have a specific functional interface for it.

A particularly useful interface in the `java.util.function` package is `Predicate`:

```
public interface Predicate<T>  
{  
    boolean test(T t);  
    // additional default and static methods  
}
```

The `ArrayList` class has a `removeIf` method whose parameter is a `Predicate`. It is specifically designed to pass a lambda expression. For example, the following statement removes all null values from an array list:

```
list.removeIf(e -> e == null);
```

Another useful functional interface is `Supplier<T>`:

```
public interface Supplier<T>  
{  
    T get();  
}
```

A supplier has no parameters and yields a value of type `T` when the `get` method is called:

```
Supplier<Integer> die = () -> (int)(Math.random() * 6) + 1;  
int outcome = die.get();
```

Suppliers are used for *lazy evaluation*. For example, consider the call

```
LocalDate hireDay = Objects.requireNonNullElse(day,  
    LocalDate.of(1970, 1, 1));
```

This is not optimal. We expect that `day` is rarely null, so we only want to construct the default `LocalDate` when necessary. By using the supplier, we can defer the computation:

```
LocalDate hireDay = Objects.requireNonNullElseGet(day,  
    () -> LocalDate.of(1970, 1, 1));
```

The `requireNonNullElseGet` method only calls the supplier when the value is needed.

Functional interfaces that involve primitive types are a little cumbersome. Consider a function consuming an `int` and yielding an object of type `T`. You could use a `Function<Integer, T>`, but then the argument must be boxed in each call. Instead, there is a functional interface `IntFunction<T>`. Conversely, if a function has a return value of type `int`, the `ToIntFunction<T>` interface is more efficient than `Function<T, Integer>`. Finally, if both argument and return value are `int`, there is an `IntUnaryOperator` interface.

As the user of an API, you don't usually care about this subtlety. Consider the `Arrays.setAll` method. It sets all values of an array to the result of a function whose argument is the array index. Here, we set all elements to the square of the index:

```
var values = new int[100];
Arrays.setAll(values, i -> i * i); // [0, 1, 4, 9, 16, . . . , 9801]
```

There are overloaded versions of `setAll` for arrays of type `int[]`, `long[]`, `double[]`, and a generic array `T[]`. Here, the `int[]` overload has as second parameter an `IntUnaryOperator`. But as the user of the method, you don't care. You just supply the lambda expression, which you can do without worrying about the difference between primitive types and their wrapper classes.



Caution: It is nice that a lambda expression can match primitive and wrapper types in a functional interface. But it is an error if both matches could occur. Consider a utility class that provides these methods:

```
public static int[] fill(int n, IntUnaryOperator op)
public static Object[] fill(int n, IntFunction<Object> op)
```

A call `fill(n, i -> i * i)` will not compile since it is ambiguous.

You can catch such problems in your API by compiling with the `-Xlint` or `-Xlint:overloads` flag.

6.2.5. Method References

Sometimes, a lambda expression involves a single method. For example, suppose you simply want to print the event object whenever a timer event occurs. Of course, you could call

```
var timer = new Timer(1000, event -> System.out.println(event));
```

It would be nicer if you could just pass the `println` method to the `Timer` constructor. Here is how you do that:


```
var timer = new Timer(1000, System.out::println);
```

The expression `System.out::println` is a *method reference*. It directs the compiler to produce an instance of a functional interface, overriding the single abstract method of the interface to call the given method. In this example, an `ActionListener` is produced whose `actionPerformed(ActionEvent e)` method calls `System.out.println(e)`.



Note: Like a lambda expression, a method reference is not an object. It gives rise to an object when assigned to a variable whose type is a functional interface.



Note: There are ten overloaded `println` methods in the `PrintStream` class (of which `System.out` is an instance). The compiler needs to figure out which one to use, depending on context. In our example, the method reference `System.out::println` must be turned into an `ActionListener` instance with a method

```
void actionPerformed(ActionEvent e)
```

The `println(Object x)` method is selected from the ten overloaded `println` methods since `Object` is the best match for `ActionEvent`. When the `actionPerformed` method is called, the event object is printed.

Now suppose we assign the same method reference to a different functional interface:

```
Runnable task = System.out::println;
```

The `Runnable` functional interface has a single abstract method with no parameters

```
void run()
```

In this case, the `println()` method with no parameters is chosen. Calling `task.run()` prints a blank line to `System.out`.

As another example, suppose you want to sort strings regardless of letter case. You can pass this method expression:

```
Arrays.sort(strings, String::compareToIgnoreCase)
```

As you can see from these examples, the `::` operator separates the method name from the name of an object or class. There are three variants:

1. *object::instanceMethod*
2. *Class::instanceMethod*
3. *Class::staticMethod*

In the first variant, the method reference is equivalent to a lambda expression whose parameters are passed to the method. In the case of `System.out::println`, the object is `System.out`, and the method expression is equivalent to `x -> System.out.println(x)`.

In the second variant, the first parameter becomes the implicit parameter of the method. For example, `String::compareToIgnoreCase` is the same as `(x, y) -> x.compareToIgnoreCase(y)`.

In the third variant, all parameters are passed to the static method: `Math::pow` is equivalent to `(x, y) -> Math.pow(x, y)`.

Table 6.1 walks you through additional examples.

Note that a lambda expression can only be rewritten as a method reference if the body of the lambda expression calls a single method and doesn't do anything else. Consider the lambda expression

```
s -> s.length() == 0
```

There is a single method call. But there is also a comparison, so you can't use a method reference here.

Table 6.1: Method Reference Examples

Method Reference	Equivalent Lambda Expression	Notes
<code>separator::equals</code>	<code>x -> separator.equals(x)</code>	This is a method expression with an <i>object</i> and an instance method. The lambda parameter is passed as the explicit parameter of the method.
<code>String::strip</code>	<code>x -> x.strip()</code>	This is a method expression with a <i>class</i> and an instance method. The lambda parameter becomes the implicit parameter.
<code>String::concat</code>	<code>(x, y) -> x.concat(y)</code>	Again, we have an instance method, but this time, with an explicit parameter. As before, the <i>first</i> lambda parameter becomes the implicit parameter, and the remaining ones are passed to the method.
<code>Integer.valueOf</code>	<code>x -> Integer.valueOf(x)</code>	This is a method expression with a <i>static</i> method. The lambda parameter is passed to the static method.

Method Reference	Equivalent Lambda Expression	Notes
<code>Integer.sum</code>	<code>(x, y) -> Integer.sum(x, y)</code>	This is another static method, but this time with two parameters. Both lambda parameters are passed to the static method. The <code>Integer.sum</code> method was specifically created to be used as a method reference. As a lambda, you could just write <code>(x, y) -> x + y</code> .
<code>String::new</code>	<code>x -> new String(x)</code>	This is a constructor reference—see Section 6.2.6. The lambda parameters are passed to the constructor.
<code>String[]::new</code>	<code>n -> new String[n]</code>	This is an array constructor reference—see Section 6.2.6. The lambda parameter is the array length.



Note: When there are multiple overloaded methods with the same name, the compiler will try to find from the context which one you mean. For example, there are two versions of the `Math.max` method, one for integers and one for double values. Which one gets picked depends on the method parameters of the functional interface to which `Math::max` is converted. Just like lambda expressions, method references don't live in isolation. They are always turned into instances of functional interfaces.



Note: Sometimes, the API contains methods that are specifically intended to be used as method references. For example, the `Objects` class has a method `isNull` to test whether an object reference is null. At first glance, this doesn't seem useful because the test `obj == null` is easier to read than `Objects.isNull(obj)`. But you can pass the method reference to any method with a Predicate parameter. For example, to remove all null references from a list, you can call

```
list.removeIf(Objects::isNull);
// A bit easier to read than list.removeIf(e -> e == null);
```



Note: There is a tiny difference between a method reference with an object and its equivalent lambda expression. Consider a method reference such as `separator::equals`. If `separator` is null, forming `separator::equals` immediately throws a

`NullPointerException`. The lambda expression `x -> separator.equals(x)` only throws a `NullPointerException` if it is invoked.

You can capture the `this` parameter in a method reference. For example, `this::equals` is the same as `x -> this.equals(x)`. It is also valid to use `super`. The method expression

`super::instanceMethod`

uses `this` as the target and invokes the superclass version of the given method. Here is an artificial example that shows the mechanics:

```
class Greeter
{
    public void greet(ActionEvent event)
    {
        System.out.println("Hello, the time is "
            + Instant.ofEpochMilli(event.getWhen()));
    }
}

class RepeatedGreeter extends Greeter
{
    public void greet(ActionEvent event)
    {
        var timer = new Timer(1000, super::greet);
        timer.start();
    }
}
```

When the `RepeatedGreeter.greet` method starts, a `Timer` is constructed that executes the `super::greet` method on every timer tick.

6.2.6. Constructor References

Constructor references are just like method references, except that the name of the method is `new`. For example, `Person::new` is a reference to a `Person` constructor. Which constructor? It depends on the context. Suppose you have a list of strings. Then you can turn it into an array of `Person` objects, by calling the constructor on each of the strings, with the following invocation:

```
ArrayList<String> names = . . . ;
Stream<Person> stream = names.stream().map(Person::new);
List<Person> people = stream.toList();
```

We will discuss the details of the `stream`, `map`, and `toList` methods in Chapter 1 of Volume II. For now, what's important is that the `map` method calls the `Person(String)` constructor for each list element. If there are multiple `Person` constructors, the compiler picks the one with

a `String` parameter because it infers from the context that the constructor is called with a string.

You can form constructor references with array types. For example, `int[]::new` is a constructor reference with one parameter: the length of the array. It is equivalent to the lambda expression `n -> new int[n]`.

Array constructor references are useful to overcome a limitation of Java. As you will see in Chapter 8, it is not possible to construct an array of a generic type `T`. (The expression `new T[n]` is an error since it would be “erased” to `new Object[n]`). That is a problem for library authors. For example, suppose we want to have an array of `Person` objects. The `Stream` interface has a `toArray` method that returns an `Object` array:

```
Object[] people = stream.toArray();
```

But that is unsatisfactory. The user wants an array of references to `Person`, not references to `Object`. The stream library solves that problem with constructor references. Pass `Person[]::new` to the `toArray` method:

```
Person[] people = stream.toArray(Person[]::new);
```

The `toArray` method invokes this constructor to obtain an array of the correct type. Then it fills and returns the array.



Caution: Sometimes, it is surprising which overloaded variant is chosen when passing a method or constructor reference. Consider this code snippet:

```
var dates = new Date[100];  
Arrays.setAll(dates, Date::new);
```

At first glance, it looks as if all elements would be set to the current date, by calling the no-argument constructor `new Date()` each time. But actually, the second parameter of `setAll` is an `IntFunction`, which receives the index of the element. Therefore, an entirely different constructor is invoked, `new Date(i)`, where `i` ranges from 0 to 99. That constructor sets the date to a given number of milliseconds from the “epoch,” January 1, 1970.

6.2.7. Variable Scope

Often, you want to be able to access variables from an enclosing method or class in a lambda expression. Consider this example:

```
public static void repeatMessage(String text, int delay)  
{  
    ActionListener listener = event ->
```

```
    {  
        System.out.println(text);  
        Toolkit.getDefaultToolkit().beep();  
    };  
    new Timer(delay, listener).start();  
}
```

Consider a call

```
repeatMessage("Hello", 1000); // prints Hello every 1,000 milliseconds
```

Now look at the variable `text` inside the lambda expression. Note that this variable is *not* defined in the lambda expression. Instead, it is a parameter variable of the `repeatMessage` method.

If you think about it, something nonobvious is going on here. The code of the lambda expression may run long after the call to `repeatMessage` has returned and the parameter variables are gone. How does the `text` variable stay around?

To understand what is happening, we need to refine our understanding of a lambda expression. A lambda expression has three ingredients:

1. A block of code
2. Parameters
3. Values for the *free* variables—that is, the variables that are not parameters and not defined inside the code

In our example, the lambda expression has one free variable, `text`. The data structure representing the lambda expression must store the values for the free variables—in our case, the string "Hello". We say that such values have been *captured* by the lambda expression. (It's an implementation detail how that is done. For example, one can translate a lambda expression into an object with a single method, so that the values of the free variables are copied into instance variables of that object.)



Note: The technical term for a block of code together with the values of the free variables is a *closure*. If someone gloats that their language has closures, rest assured that Java has them as well. In Java, lambda expressions are closures.

As you have seen, a lambda expression can capture the value of a variable in the enclosing scope. In Java, to ensure that the captured value is well-defined, there is an important restriction. In a lambda expression, you can only reference variables whose value doesn't change. For example, the following is illegal:

```
public static void countDown(int start, int delay)
{
    ActionListener listener = event ->
    {
        start--; // ERROR: Can't mutate captured variable
        System.out.println(start);
    };
    new Timer(delay, listener).start();
}
```

There is a reason for this restriction. Mutating variables in a lambda expression is not safe when multiple actions are executed concurrently. This won't happen for the kinds of actions that we have seen so far, but in general, it is a serious problem. See Chapter 10 for more information on this important issue.

It is also illegal to refer, in a lambda expression, to a variable that is mutated outside. For example, the following is illegal:

```
public static void repeat(String text, int count)
{
    for (int i = 1; i <= count; i++)
    {
        ActionListener listener = event ->
        {
            System.out.println(i + ": " + text);
            // ERROR: Cannot refer to changing i
        };
        new Timer(1000, listener).start();
    }
}
```

The rule is that any captured variable in a lambda expression must be *effectively final*. An effectively final variable is a variable that is never assigned a new value after it has been initialized. In our case, `text` always refers to the same `String` object, and it is OK to capture it. However, the value of `i` is mutated, and therefore `i` cannot be captured.

The body of a lambda expression has *the same scope as a nested block*. The same rules for name conflicts and shadowing apply. It is illegal to declare a parameter or a local variable in the lambda that has the same name as a local variable.

```
Path first = Path.of("/usr/bin");
Comparator<String> comp =
    (first, second) -> first.length() - second.length();
// ERROR: Variable first already defined
```

Inside a method, you can't have two local variables with the same name, and therefore, you can't introduce such variables in a lambda expression either.

When you use the `this` keyword in a lambda expression, you refer to the `this` parameter of the method that creates the lambda. For example, consider

```
public class Application
{
    public void init()
    {
        ActionListener listener = event ->
        {
            System.out.println(this.toString());
            . . .
        }
        . . .
    }
}
```

The expression `this.toString()` calls the `toString` method of the `Application` object, *not* the `ActionListener` instance. There is nothing special about the use of `this` in a lambda expression. The scope of the lambda expression is nested inside the `init` method, and `this` has the same meaning anywhere in that method.

6.2.8. Processing Lambda Expressions

Up to now, you have seen how to produce lambda expressions and pass them to a method that expects a functional interface. Now let us see how to write methods that can consume lambda expressions.

The point of using lambdas is *deferred execution*. After all, if you wanted to execute some code right now, you'd do that, without wrapping it inside a lambda. There are many reasons for executing code later, such as:

- Running the code in a separate thread
- Running the code multiple times
- Running the code at the right point in an algorithm (for example, the comparison operation in sorting)
- Running the code when something happens (a button was clicked, data has arrived, and so on)
- Running the code only when necessary

Let's look at a simple example. Suppose you want to repeat an action `n` times. The action and the count are passed to a `repeat` method:

```
repeat(10, () -> System.out.println("Hello, World!"));
```

To accept the lambda, we need to pick (or, in rare cases, provide) a functional interface. Table 6.2 lists the most important functional interfaces that are provided in the Java API. In this case, we can use the `Runnable` interface:


```
public static void repeat(int n, Runnable action)
{
    for (int i = 0; i < n; i++) action.run();
}
```

Note that the body of the lambda expression is executed when `action.run()` is called.

Now let's make this example a bit more sophisticated. We want to tell the action in which iteration it occurs. For that, we need to pick a functional interface that has a method with an `int` parameter and a `void` return. The standard interface for processing `int` values is

```
public interface IntConsumer
{
    void accept(int value);
}
```

Here is the improved version of the `repeat` method:

```
public static void repeat(int n, IntConsumer action)
{
    for (int i = 0; i < n; i++) action.accept(i);
}
```

And here is how you call it:

```
repeat(10, i -> System.out.println("Countdown: " + (9 - i)));
```

Table 6.2: Common Functional Interfaces

Functional Interface	Parameter Types	Return Type	Abstract Method Name	Description	Other Methods
<code>Runnable</code>	none	<code>void</code>	<code>run</code>	Runs an action without parameters or return value	
<code>Supplier<T></code>	none	<code>T</code>	<code>get</code>	Supplies a value of type <code>T</code>	
<code>Consumer<T></code>	<code>T</code>	<code>void</code>	<code>accept</code>	Consumes a value of type <code>T</code>	<code>andThen</code>

Functional Interface	Parameter Types	Return Type	Abstract Method Name	Description	Other Methods
<code>BiConsumer<T, U></code>	<code>T, U</code>	<code>void</code>	<code>accept</code>	Consumes values of types <code>T</code> and <code>U</code>	<code>andThen</code>
<code>Function<T, R></code>	<code>T</code>	<code>R</code>	<code>apply</code>	A function with parameter of type <code>T</code>	<code>compose</code> , <code>andThen</code> , <code>identity</code>
<code>BiFunction<T, U, R></code>	<code>T, U</code>	<code>R</code>	<code>apply</code>	A function with parameters of types <code>T</code> and <code>U</code>	<code>andThen</code>
<code>UnaryOperator<T></code>	<code>T</code>	<code>T</code>	<code>apply</code>	A unary operator on the type <code>T</code>	<code>compose</code> , <code>andThen</code> , <code>identity</code>
<code>BinaryOperator<T></code>	<code>T, T</code>	<code>T</code>	<code>apply</code>	A binary operator on the type <code>T</code>	<code>andThen</code> , <code>maxBy</code> , <code>minBy</code>
<code>Predicate<T></code>	<code>T</code>	<code>boolean</code>	<code>test</code>	A boolean-valued function	<code>and</code> , <code>or</code> , <code>negate</code> , <code>isEqual</code> , <code>not</code>
<code>BiPredicate<T, U></code>	<code>T, U</code>	<code>boolean</code>	<code>test</code>	A boolean-valued function with two parameters	<code>and</code> , <code>or</code> , <code>negate</code>

Table 6.3 lists the 34 available specializations for primitive types `int`, `long`, and `double`. As you will see in Chapter 8, it is more efficient to use these specializations than the generic interfaces. For that reason, I used an `IntConsumer` instead of a `Consumer<Integer>` in the example of the preceding section.

Table 6.3: Functional Interfaces for Primitive Types
p, q is int, long, double; *P, Q* is Int, Long, Double

Functional Interface	Parameter Types	Return Type	Abstract Method Name
BooleanSupplier	none	boolean	getAsBoolean
PSupplier	none	<i>p</i>	getAs <i>P</i>
PConsumer	<i>p</i>	void	accept
ObjPConsumer<T>	T, <i>p</i>	void	accept
PFunction<T>	<i>p</i>	T	apply
PToQFunction	<i>p</i>	<i>q</i>	applyAsQ
ToPFunction<T>	T	<i>p</i>	applyAs <i>P</i>
ToPBiFunction<T, U>	T, U	<i>p</i>	applyAs <i>P</i>
PUnaryOperator	<i>p</i>	<i>p</i>	applyAs <i>P</i>
PBinaryOperator	<i>p, p</i>	<i>p</i>	applyAs <i>P</i>
PPredicate	<i>p</i>	boolean	test



Tip: Use the standard interfaces for function types whenever you can. For example, suppose you write a method to process files that match a certain criterion. There is a legacy interface `java.io.FileFilter`. But if you use the standard `Predicate<File>` interface, you can take advantage of methods for creating, adapting, and combining predicates. The only reason not to do so would be if you already have many useful methods producing `FileFilter` instances.



Note: Most of the standard functional interfaces have nonabstract methods for producing or combining functions. For example, `Predicate.isEqual(a)` is the same as `a::equals`, but it also works if `a` is null. There are default methods `and`, `or`, `negate` for combining predicates. For example, `Predicate.isEqual(a).or(Predicate.isEqual(b))` is the same as `x -> a.equals(x) || b.equals(x)`.



Note: If you design your own interface with a single abstract method, you can tag it with the `@FunctionalInterface` annotation. This has two advantages. The compiler

gives an error message if you accidentally add another abstract method. And the javadoc page includes a statement that your interface is a functional interface.

It is not required to use the annotation. Any interface with a single abstract method is, by definition, a functional interface. But using the `@FunctionalInterface` annotation is a good idea.



Note: Some programmers love chains of method calls, such as

```
String input = " 618970019642690137449562111 ";  
boolean isPrime = input.strip().transform(BigInteger::new).isProbablePrime(20);
```

The `transform` method of the `String` class (added in Java 12) applies a `Function` to the string and yields the result. You could have equally well written

```
boolean prime = new BigInteger(input.strip()).isProbablePrime(20);
```

But then your eyes jump inside-out and left-to-right to find out what happens first and what happens next: Calling `strip`, then constructing the `BigInteger`, and finally testing if it is a probable prime.

I am not sure that the eyes-jumping-inside-out-and-left-to-right is a huge problem. But if you prefer the orderly left-to-right sequence of chained method calls, then `transform` is your friend.

Sadly, it only works for strings. Why isn't there a `transform(java.util.function.Function)` method in the `Object` class?

The Java API designers weren't fast enough. They had one chance to do this right—in Java 8, when the `java.util.function.Function` interface was added to the API. Up to that point, nobody could have added a `transform(java.util.function.Function)` method to their own classes. But in Java 12, it was too late. Someone somewhere could have defined `transform(java.util.function.Function)` in their class, with a different meaning. Admittedly, it is unlikely that this ever happened, but there is no way to know.

That is how Java works. It takes its commitments seriously, and won't renege on them for convenience.

6.2.9. Creating Comparators

The `Comparator` interface has a number of convenient static methods for creating comparators. These methods are intended to be used with lambda expressions or method references.

The static `comparing` method takes a “key extractor” function that maps a type `T` to a comparable type (such as `String`). The function is applied to the objects to be compared, and the comparison is then made on the returned keys. For example, suppose you have an array of `Person` objects. Here is how you can sort them by name:

```
Arrays.sort(people, Comparator.comparing(Person::getName));
```

This is certainly much easier than implementing a `Comparator` by hand. Moreover, the code is clearer since it is obvious that we want to compare people by name.

You can chain comparators with the `thenComparing` method for breaking ties. For example,

```
Arrays.sort(people,
    Comparator.comparing(Person::getLastName)
        .thenComparing(Person::getFirstName));
```

If two people have the same last name, then the second comparator is used.

There are a few variations of these methods. You can specify a comparator to be used for the keys that the `comparing` and `thenComparing` methods extract. For example, here we sort people by the length of their names:

```
Arrays.sort(people, Comparator.comparing(Person::getName,
    (s, t) -> Integer.compare(s.length(), t.length())));
```

Moreover, both the `comparing` and `thenComparing` methods have variants that avoid boxing of `int`, `long`, or `double` values:

```
Arrays.sort(people, Comparator.comparing(Person::getName,
    Comparator.comparingInt(String::length)))
```

A shorter but perhaps less elegant way of producing the preceding operation would be:

```
Arrays.sort(people, Comparator.comparingInt(p -> p.getName().length()));
```

If your key function can return `null`, you will like the `nullsFirst` and `nullsLast` adapters. These static methods take an existing comparator and modify it so that it doesn’t throw an exception when encountering `null` values but ranks them as smaller or larger than regular values. For example, suppose `getMiddleName` returns a `null` when a person has no middle name. Then you can use `Comparator.comparing(Person::getMiddleName, Comparator.nullsFirst(. . .))`.

The `nullsFirst` method needs a comparator—in this case, one that compares two strings. The `naturalOrder` method makes a comparator for any class implementing `Comparable`. A `Comparator.<String>naturalOrder()` is what we need. (See Chapter 8 for an explanation of this syntax. Fortunately, the generic type can usually be inferred.) Here is the complete call for

sorting by potentially null middle names. I use a static import of `java.util.Comparator.*`, to make the expression more legible.

```
Arrays.sort(people, comparing(Person::getMiddleName, nullsFirst(naturalOrder())));
```

The static `reverseOrder` method gives the reverse of the natural order. To reverse any comparator, use the reversed instance method. For example, `naturalOrder().reversed()` is the same as `reverseOrder()`.

6.3. Inner Classes

An *inner class* is a class that is defined inside another class. Why would you want to do that? There are two reasons:

- Inner classes can be hidden from other classes in the same package.
- Inner class methods can access the data from the scope in which they are defined—including the data that would otherwise be private.

Inner classes used to be very important for concisely implementing callbacks, but nowadays lambda expressions do a much better job. Still, inner classes can be very useful for structuring your code. The following sections walk you through all the details.



C++ Note: C++ has *nested classes*. A nested class is contained inside the scope of the enclosing class. Here is a typical example: A linked list class defines a nested class to hold the nodes.

```
template<typename T>
class LinkedList
{
public:
    class Node // a nested class
    {
    public:
        . . .
    private:
        T data;
        Node* next;
    };
    . . .
private:
    Node* head;
    Node* tail;
};
```

Nested classes are similar to inner classes in Java. However, the Java inner classes have an additional feature that makes them richer and more useful than nested classes in C++. An object that comes from an inner class has an implicit reference to the outer class object that instantiated it. Through this pointer, it gains access to the total state of the outer object. For example, in Java, the `Iterator` class would not need an explicit pointer to the `LinkedList` into which it points.

In Java, nested classes that are declared as `static` do not have this added pointer. They are the Java analog to nested classes in C++.

6.3.1. Use of an Inner Class to Access Object State

The syntax for inner classes is rather complex. For that reason, I present a simple but somewhat artificial example to demonstrate the use of inner classes. Let's refactor the `TimerTest` example and extract a `TalkingClock` class. The constructor for a talking clock has two parameters: the interval between announcements and a flag to turn beeps on or off.

```
public class TalkingClock
{
    private int interval;
    private boolean beep;

    public TalkingClock(int interval, boolean beep) { . . . }
    public void start() { . . . }

    public class TimePrinter implements ActionListener
        // an inner class
    {
        . . .
    }
}
```

Note that the `TimePrinter` class is now located inside the `TalkingClock` class. This does *not* mean that every `TalkingClock` has a `TimePrinter` instance field. As you will see, the `TimePrinter` objects are constructed by methods of the `TalkingClock` class.

Here is the `TimePrinter` class in greater detail. Note that the `actionPerformed` method checks the `beep` flag before emitting a beep.

```
public class TimePrinter implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        System.out.println("At the tone, the time is "
            + Instant.ofEpochMilli(event.getWhen()));
    }
}
```

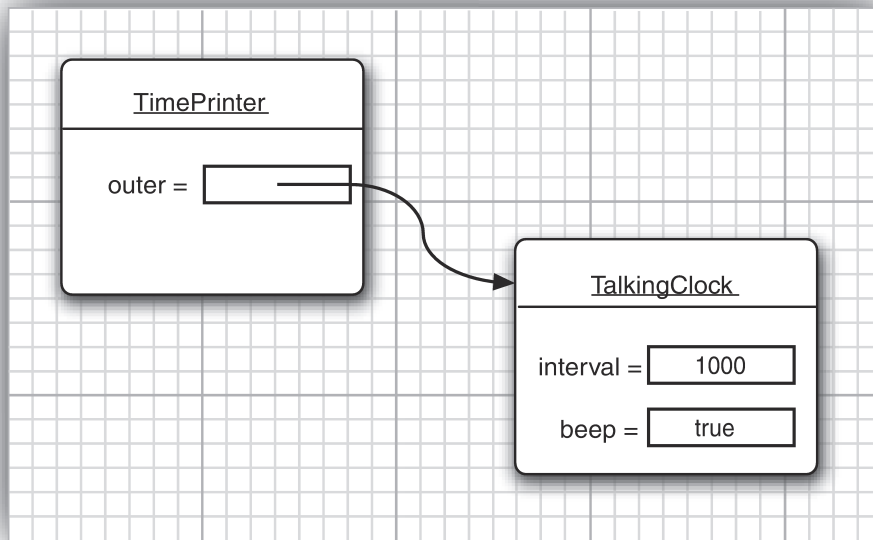


Figure 6.3: An inner class object has a reference to an outer class object.

```

        if (beep) Toolkit.getDefaultToolkit().beep();
    }
}

```

Something surprising is going on. The `TimePrinter` class has no instance field or variable named `beep`. Instead, `beep` refers to the field of the `TalkingClock` object that created this `TimePrinter`. As you can see, an inner class method gets to access both its own instance fields *and* those of the outer object creating it.

For this to work, an object of an inner class always gets an implicit reference to the object that created it (see Figure 6.3).

This reference is invisible in the definition of the inner class. However, to illuminate the concept, let us call the reference to the outer object *outer*. Then the `actionPerformed` method is equivalent to the following:

```

public void actionPerformed(ActionEvent event)
{
    System.out.println("At the tone, the time is "
        + Instant.ofEpochMilli(event.getWhen()));
    if (outer.beep) Toolkit.getDefaultToolkit().beep();
}

```

The outer class reference is set in the constructor. The compiler modifies all inner class constructors, adding a parameter for the outer class reference. The `TimePrinter` class

defines no constructors; therefore, the compiler synthesizes a no-argument constructor, generating code like this:

```
public TimePrinter(TalkingClock clock) // automatically generated code
{
    outer = clock;
}
```

Again, please note that *outer* is not a Java keyword. We just use it to illustrate the mechanism involved in an inner class.

When a `TimePrinter` object is constructed in the `start` method, the compiler passes the `this` reference to the current talking clock into the constructor:

```
var listener = new TimePrinter(this); // parameter automatically added
```

Listing 6.7 shows the complete program that tests the inner class. Have another look at the access control. Had the `TimePrinter` class been a regular class, it would have needed to access the `beep` flag through a public method of the `TalkingClock` class. Using an inner class is an improvement. There is no need to provide accessors that are of interest only to one other class.



Note: We could have declared the `TimePrinter` class as `private`. Then only `TalkingClock` methods would be able to construct `TimePrinter` objects. Only inner classes can be `private`. Regular classes always have either package or public access.

Listing 6.7 `innerClass/InnerClassTest.java`

```
1 package innerClass;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.time.*;
6
7 import javax.swing.*;
8
9 /**
10  * This program demonstrates the use of inner classes.
11  * @version 1.11 2017-12-14
12  * @author Cay Horstmann
13  */
14 public class InnerClassTest
15 {
16     public static void main(String[] args)
17     {
```

```
18     var clock = new TalkingClock(1000, true);
19     clock.start();
20
21     // keep program running until the user selects "OK"
22     JOptionPane.showMessageDialog(null, "Quit program?");
23     System.exit(0);
24 }
25 }
26
27 /**
28  * A clock that prints the time in regular intervals.
29  */
30 class TalkingClock
31 {
32     private int interval;
33     private boolean beep;
34
35     /**
36      * Constructs a talking clock.
37      * @param interval the interval between messages (in milliseconds)
38      * @param beep true if the clock should beep
39      */
40     public TalkingClock(int interval, boolean beep)
41     {
42         this.interval = interval;
43         this.beep = beep;
44     }
45
46     /**
47      * Starts the clock.
48      */
49     public void start()
50     {
51         var listener = new TimePrinter();
52         var timer = new Timer(interval, listener);
53         timer.start();
54     }
55
56     public class TimePrinter implements ActionListener
57     {
58         public void actionPerformed(ActionEvent event)
59         {
60             System.out.println("At the tone, the time is "
61                 + Instant.ofEpochMilli(event.getWhen()));
62             if (beep) Toolkit.getDefaultToolkit().beep();
63         }
64     }
65 }
```

6.3.2. Special Syntax Rules for Inner Classes

In the preceding section, we explained the outer class reference of an inner class by calling it *outer*. Actually, the proper syntax for the outer reference is a bit more complex. The expression

OuterClass.this

denotes the outer class reference. For example, you can write the `actionPerformed` method of the `TimePrinter` inner class as

```
public void actionPerformed(ActionEvent event)
{
    if (TalkingClock.this.beep) Toolkit.getDefaultToolkit().beep();
}
```

Conversely, you can write the inner object constructor more explicitly, using the syntax

outerObject.new InnerClass(construction arguments)

For example:

```
ActionListener listener = this.new TimePrinter();
```

Here, the outer class reference of the newly constructed `TimePrinter` object is set to the `this` reference of the method that creates the inner class object. This is the most common case. As always, the `this.` qualifier is redundant. However, it is also possible to set the outer class reference to another object by explicitly naming it. For example, since `TimePrinter` is a public inner class, you can construct a `TimePrinter` for any talking clock:

```
var jabberer = new TalkingClock(1000, true);
TalkingClock.TimePrinter listener = jabberer.new TimePrinter();
```

Note that you refer to an inner class as

OuterClass.InnerClass

when it occurs outside the scope of the outer class.



Note: As of Java 16, inner classes can have static members. Previously, static methods in inner classes were disallowed, and static fields declared in an inner class had to be `final` and initialized with a compile-time constant.

Static methods of an inner class can access static fields and methods from the inner class or enclosing classes.

6.3.3. Are Inner Classes Useful? Actually Necessary? Secure?

When inner classes were added to the Java language in Java 1.1, many programmers considered them a major new feature that was out of character with the Java philosophy of being simpler than C++. The inner class syntax is undeniably complex. (It gets more complex as we study anonymous inner classes later in this chapter.) It is not obvious how inner classes interact with other features of the language, such as access control and security.

Inner classes are translated into regular class files with \$ (dollar signs) separating the outer and inner class names. For example, the `TimePrinter` class inside the `TalkingClock` class is translated to a class file `TalkingClock$TimePrinter.class`. To see this at work, try the following experiment: run the `ReflectionTest` program of Chapter 5, and give it the class `TalkingClock$TimePrinter` to reflect upon. Alternatively, simply use the `javap` utility:

```
javap -private ClassName
```



Note: If you use UNIX, remember to escape the \$ character when you supply the class name on the command line. That is, run the `ReflectionTest` or `javap` program as

```
java --classpath ../../v1ch05 reflection.ReflectionTest \  
    innerClass.TalkingClock$TimePrinter
```

or

```
javap -private innerClass.TalkingClock$TimePrinter
```

You will get the following printout:

```
public class innerClass.TalkingClock$TimePrinter  
    implements java.awt.event.ActionListener  
{  
    final innerClass.TalkingClock this$0;  
    public innerClass.TalkingClock$TimePrinter(innerClass.TalkingClock);  
    public void actionPerformed(java.awt.event.ActionEvent);  
}
```

You can plainly see that the compiler has generated an additional instance field, `this$0`, for the reference to the outer class. (The name `this$0` is synthesized by the compiler—you cannot refer to it in your code.) You can also see the `TalkingClock` parameter for the constructor.



Note: Since Java 18, the `this$0` field is only provided when it is actually needed. It is dropped if no methods of the inner class access the outer class.

If the compiler can automatically do this transformation, couldn't you simply program the same mechanism by hand? Let's try it. We would make `TimePrinter` a regular class, outside the `TalkingClock` class. When constructing a `TimePrinter` object, we pass it the `this` reference of the object that is creating it.

```
class TalkingClock
{
    . . .
    public void start()
    {
        var listener = new TimePrinter(this);
        var timer = new Timer(interval, listener);
        timer.start();
    }
}

class TimePrinter implements ActionListener
{
    private TalkingClock outer;
    . . .
    public TimePrinter(TalkingClock clock)
    {
        outer = clock;
    }
}
```

Now let us look at the `actionPerformed` method. It needs to access `outer.beep`.

```
if (outer.beep) . . . // ERROR
```

Here we run into a problem. The inner class can access the private data of the outer class, but our external `TimePrinter` class cannot.

Thus, inner classes are genuinely more powerful than regular classes because they have more access privileges.

6.3.4. Local Inner Classes

If you look carefully at the code of the `TalkingClock` example, you will find that you need the name of the type `TimePrinter` only once: when you create an object of that type in the `start` method.

In a situation like this, you can define the class *locally in a single method*.

```
public void start()
{
    class TimePrinter implements ActionListener
    {
        public void actionPerformed(ActionEvent event)
        {
            System.out.println("At the tone, the time is "
                               + Instant.ofEpochMilli(event.getWhen()));
            if (beep) Toolkit.getDefaultToolkit().beep();
        }
    }

    var listener = new TimePrinter();
    var timer = new Timer(interval, listener);
    timer.start();
}
```

Local classes are never declared with an access specifier (that is, public or private). Their scope is always restricted to the block in which they are declared.

Local classes have one great advantage: They are completely hidden from the outside world—not even other code in the TalkingClock class can access them. No method except start has any knowledge of the TimePrinter class.

6.3.5. Accessing Variables from Outer Methods

Local classes have another advantage over other inner classes. Not only can they access the fields of their outer classes; they can even access local variables! However, those local variables must be *effectively final*. That means, they may never change once they have been assigned.

Here is a typical example. Let's move the interval and beep parameters from the TalkingClock constructor to the start method.

```
public void start(int interval, boolean beep)
{
    class TimePrinter implements ActionListener
    {
        public void actionPerformed(ActionEvent event)
        {
            System.out.println("At the tone, the time is "
                               + Instant.ofEpochMilli(event.getWhen()));
            if (beep) Toolkit.getDefaultToolkit().beep();
        }
    }
}
```

```
    var listener = new TimePrinter();
    var timer = new Timer(interval, listener);
    timer.start();
}
```

Note that the TalkingClock class no longer needs to store a beep instance field. It simply refers to the beep parameter variable of the start method.

Maybe this should not be so surprising. The line

```
if (beep) . . .
```

is, after all, ultimately inside the start method, so why shouldn't it have access to the value of the beep variable?

To see why there is a subtle issue here, let's consider the flow of control more closely.

1. The start method is called.
2. The object variable listener is initialized by a call to the constructor of the inner class TimePrinter.
3. The listener reference is passed to the Timer constructor, the timer is started, and the start method exits. At this point, the beep parameter variable of the start method no longer exists.
4. A second later, the actionPerformed method executes if (beep) . . .

For the code in the actionPerformed method to work, the TimePrinter class must have copied the beep field as a local variable of the start method, before the beep parameter value went away. That is indeed exactly what happens. In our example, the compiler synthesizes the name TalkingClock\$1TimePrinter for the local inner class. If you use the ReflectionTest program or the javap utility again to spy on the TalkingClock\$1TimePrinter class, you will get the following output:

```
class TalkingClock$1TimePrinter
{
    TalkingClock$1TimePrinter();

    public void actionPerformed(java.awt.event.ActionEvent);

    final boolean val$beep;
    final TalkingClock this$0;
}
```

When an object is created, the current value of the beep variable is stored in the val\$beep field. As of Java 11, this happens with “nest mate” access. Previously, the inner class constructor had an additional parameter to set the field. Either way, the inner class field persists even if the local variable goes out of scope.

6.3.6. Anonymous Inner Classes

When using local inner classes, you can often go a step further. If you want to make only a single object of this class, you don't even need to give the class a name. Such a class is called an *anonymous inner class*.

```
public void start(int interval, boolean beep)
{
    var listener = new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            System.out.println("At the tone, the time is "
                + Instant.ofEpochMilli(event.getWhen()));
            if (beep) Toolkit.getDefaultToolkit().beep();
        }
    };
    var timer = new Timer(interval, listener);
    timer.start();
}
```

This syntax is very cryptic indeed. What it means is this: Create a new object of a class that implements the `ActionListener` interface, where the required method `actionPerformed` is the one defined inside the braces `{ }`.

In general, the syntax is

```
new SuperType(construction arguments)
{
    inner class methods and data
}
```

Here, *SuperType* can be an interface, such as `ActionListener`; then, the inner class implements that interface. *SuperType* can also be a class; then, the inner class extends that class.

An anonymous inner class cannot have constructors because the name of a constructor must be the same as the name of a class, and the class has no name. Instead, the construction arguments are given to the *superclass* constructor. In particular, whenever an inner class implements an interface, it cannot have any construction arguments. Nevertheless, you must supply a set of parentheses as in

```
new InterfaceType()
{
    methods and data
}
```


You have to look carefully to see the difference between the construction of a new object of a class and the construction of an object of an anonymous inner class extending that class.

```
var queen = new Person("Mary");  
    // a Person object  
var count = new Person("Dracula") { . . . };  
    // an object of an inner class extending Person
```

If the closing parenthesis of the construction argument list is followed by an opening brace, then an anonymous inner class is being defined.



Note: Even though an anonymous class cannot have constructors, you can provide an object initialization block:

```
var count = new Person("Dracula")  
{  
    { initialization }  
    . . .  
};
```

Listing 6.8 contains the complete source code for the talking clock program with an anonymous inner class. If you compare this program with Listing 6.7, you will see that in this case, the solution with the anonymous inner class is quite a bit shorter and, hopefully, with some practice, as easy to comprehend.

For many years, Java programmers routinely used anonymous inner classes for event listeners and other callbacks. Nowadays, you are better off using a lambda expression. For example, the start method from the beginning of this section can be written much more concisely with a lambda expression like this:

```
public void start(int interval, boolean beep)  
{  
    var timer = new Timer(interval, event ->  
    {  
        System.out.println( "At the tone, the time is "  
            + Instant.ofEpochMilli(event.getWhen()));  
        if (beep) Toolkit.getDefaultToolkit().beep();  
    });  
    timer.start();  
}
```



Note: If you store an anonymous class instance in a variable defined with var, the variable knows about added methods or fields:

```
var bob = new Object() { String name = "Bob"; }  
System.out.println(bob.name);
```

If you declare `bob` as having type `Object`, then `bob.name` does not compile.

The object constructed with `new Object() { String name = "Bob"; }` has type “Object with a `String` `name` field.” This is a *nondenotable* type—a type that you cannot express with Java syntax. Nevertheless, the compiler understands the type, and it can set it as the type for the `bob` variable.



Note: The following trick, called *double brace initialization*, takes advantage of the inner class syntax. Suppose you want to construct an array list and pass it to a method:

```
var friends = new ArrayList<String>();  
friends.add("Harry");  
friends.add("Tony");  
invite(friends);
```

If you don't need the array list again, it would be nice to make it anonymous. But then how can you add the elements? Here is how:

```
invite(new ArrayList<String>() {{ add("Harry"); add("Tony"); }});
```

Note the double braces. The outer braces make an anonymous subclass of `ArrayList`. The inner braces are an object initialization block (see Chapter 4).

In practice, this trick is rarely useful. More likely than not, the `invite` method is willing to accept any `List<String>`, and you can simply pass `List.of("Harry", "Tony")`.



Caution: It is often convenient to make an anonymous subclass that is almost, but not quite, like its superclass. But you need to be careful with the `equals` method. In Chapter 5, I recommended that your `equals` methods use a test

```
if (getClass() != other.getClass()) return false;
```

An anonymous subclass will fail this test.



Tip: When you produce logging or debugging messages, you often want to include the name of the current class, such as

```
System.err.println("Something awful happened in " + getClass());
```

But that fails in a static method. After all, the call to `getClass` calls `this.getClass()`, and a static method has no `this`. Use the following expression instead:

```
new Object(){}.getClass().getEnclosingClass() // gets class of static method
```

Here, `new Object(){}` makes an anonymous object of an anonymous subclass of `Object`, and `getEnclosingClass` gets its enclosing class—that is, the class containing the static method.

Listing 6.8 `anonymousInnerClass/AnonymousInnerClassTest.java`

```
1 package anonymousInnerClass;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.time.*;
6
7 import javax.swing.*;
8
9 /**
10  * This program demonstrates anonymous inner classes.
11  * @version 1.12 2017-12-14
12  * @author Cay Horstmann
13  */
14 public class AnonymousInnerClassTest
15 {
16     public static void main(String[] args)
17     {
18         var clock = new TalkingClock();
19         clock.start(1000, true);
20
21         // keep program running until the user selects "OK"
22         JOptionPane.showMessageDialog(null, "Quit program?");
23         System.exit(0);
24     }
25 }
26
27 /**
28  * A clock that prints the time in regular intervals.
29  */
30 class TalkingClock
31 {
32     /**
33      * Starts the clock.
34      * @param interval the interval between messages (in milliseconds)
35      * @param beep true if the clock should beep
36      */
37     public void start(int interval, boolean beep)
```

```

38 | {
39 |     var listener = new ActionListener()
40 |     {
41 |         public void actionPerformed(ActionEvent event)
42 |         {
43 |             System.out.println("At the tone, the time is "
44 |                 + Instant.ofEpochMilli(event.getWhen()));
45 |             if (beep) Toolkit.getDefaultToolkit().beep();
46 |         }
47 |     };
48 |     var timer = new Timer(interval, listener);
49 |     timer.start();
50 | }
51 | }

```

6.3.7. Static Classes

Occasionally, you may want to nest one class inside another, but you don't need the nested class to have a reference to the outer class object. You can suppress the generation of that reference by declaring the nested class static.

The Java Language Specification uses the term “nested class” for any class that is declared inside another class or interface, “static class” for a (necessarily nested) static class, and “inner class” for a nested class that is not static.

Here is a typical example of where you would want to do this. In an `ArrayAlg` class, we have a task that finds a range of elements of an array. Then you need to return the start and the end of the range. We can achieve that by defining a class `Range` that holds two values:

```

class Range
{
    private int from;
    private int to;

    public Range(int from) { . . . }
    public void extend() { . . . }
    . . .
}

```

Of course, `Range` is an exceedingly common name, and in a large project, it is quite possible that some other programmer had the same bright idea and defined another `Range` class in the same package. We can solve this potential name clash by making `Range` a public inner class inside `ArrayAlg`. Then the class will be known to the public as `ArrayAlg.Range`:

```

ArrayAlg.Range r = ArrayAlg.longestRun(numbers);

```

However, unlike the inner classes used in previous examples, we do not want to have a reference to any other object inside a `Range` object. That reference can be suppressed by declaring the nested class static:

```
class ArrayAlg
{
    public static class Range
    {
        . . .
    }
    . . .
}
```

A static class is exactly like an inner class, except that an object of a static class does not have a reference to the outer class object that generated it. In our example, we must use a static class because the nested class instance is constructed inside a static method:

```
public static Pair longestRun(double[] values)
{
    . . .
    Range current = new Range(. . .);
    . . .
    if (. . .) longest = current;
    . . .
    return longest;
}
```

Had the `Range` class not been declared as static, the compiler would have flagged the constructor call as an error. After all, there is no implicit object of type `ArrayAlg` available to initialize the inner class instance.

You should use a static class whenever a nested class does not need to access an outer class object.

Here, I purposefully made the `Range` class mutable. It might be better to make the `Range` class immutable, and to declare it as a record. A record is automatically static.



Note: Just like records, interfaces and enumerations that are declared inside a class or interface are automatically static.

In fact,



Note: Classes that are declared inside an interface are automatically static and public.



Note: Prior to Java 16, it was not possible to declare a static class inside an inner class. This restriction has now been removed.

Listing 6.9 contains the complete source code of the `ArrayAlg` class and the nested `Pair` class.

Listing 6.9 `staticInnerClass/StaticInnerClassTest.java`

```
1 package staticInnerClass;
2
3 /**
4  * This program demonstrates the use of static inner classes.
5  * @version 1.1 2023-12-19
6  * @author Cay Horstmann
7  */
8 public class StaticInnerClassTest
9 {
10     public static void main(String[] args)
11     {
12         double[] numbers = { 1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5, 6, 6, 6, 6 };
13         ArrayAlg.Range r = ArrayAlg.longestRun(numbers);
14         System.out.println("from = " + r.getFrom());
15         System.out.println("to = " + r.getTo());
16     }
17 }
18
19 class ArrayAlg
20 {
21     /**
22      * A range of index values.
23      */
24     public static class Range
25     {
26         private int from;
27         private int to;
28
29         /**
30          * Constructs a range of length 1.
31          * @param from the initial index value of this range
32          */
33         public Range(int from)
34         {
35             this.from = from;
```

```
36         this.to = from + 1;
37     }
38
39     /**
40     * Extends this range by one element.
41     */
42     public void extend()
43     {
44         this.to++;
45     }
46
47     /**
48     * Gets the starting index value of this range.
49     * @return the starting index
50     */
51     public int getFrom()
52     {
53         return from;
54     }
55
56     /**
57     * Gets the first index past the end of this range.
58     * @return the past-the-end index
59     */
60     public int getTo()
61     {
62         return to;
63     }
64
65     /**
66     * Returns the number of elements in this range.
67     * @return the number of elements
68     */
69     public int length()
70     {
71         return to - from;
72     }
73 }
74
75 /**
76 * A "run" is a sequence of repeating adjacent elements. For example, in the array
77 * 1 2 3 3 3 4 4, the runs are (trivially) 1 and 2, and 3 3 3 3 and 4 4.
78 * Returns the range of the longest run.
79 * @param values an array of length at least 1
80 * @return the range of the longest run
81 */
82 public static Range longestRun(double[] values)
83 {
84     Range longest = new Range(0);
85     Range current = new Range(0);
86     for (int i = 1; i < values.length; i++)
87     {
```

```
88         if (values[i] == values[i - 1]) current.extend();
89     else
90     {
91         if (longest.length() < current.length()) longest = current;
92         current = new Range(i);
93     }
94 }
95 if (longest.length() < current.length()) longest = current;
96 return longest;
97 }
98 }
```

6.4. Service Loaders

Sometimes, you develop an application with a service architecture. There are platforms that encourage this approach, such as OSGi (<https://osgi.org>), which are used in development environments, application servers, and other complex applications. Such platforms go well beyond the scope of this book, but the JDK also offers a simple mechanism for loading services, described here. This mechanism is well supported by the Java Platform Module System—see Chapter 12.

Often, when providing a service, a program wants to give the service designer some freedom of how to implement the service's features. It can also be desirable to have multiple implementations to choose from. The `ServiceLoader` class makes it easy to load services that conform to a common interface.

Define an interface (or, if you prefer, a superclass) with the methods that each instance of the service should provide. For example, suppose your service provides encryption.

```
package serviceLoader;

public interface Cipher
{
    byte[] encrypt(byte[] source, byte[] key);
    byte[] decrypt(byte[] source, byte[] key);
    int strength();
}
```

The service provider supplies one or more classes that implement this service, for example

```
package serviceLoader.impl;

public class CaesarCipher implements Cipher
{
    public byte[] encrypt(byte[] source, byte[] key)
    {
        var result = new byte[source.length];
```



```
        for (int i = 0; i < source.length; i++)
            result[i] = (byte)(source[i] + key[0]);
        return result;
    }

    public byte[] decrypt(byte[] source, byte[] key)
    {
        return encrypt(source, new byte[] { (byte) -key[0] });
    }

    public int strength() { return 1; }
}
```

The implementing classes can be in any package, not necessarily the same package as the service interface. Each of them must have a no-argument constructor.

Now add the names of the classes to a UTF-8 encoded text file in the META-INF/services directory whose name matches the fully qualified interface name. In our example, the file META-INF/services/serviceLoader.Cipher would contain the line

```
serviceLoader.impl.CaesarCipher
```

In this example, we provide a single implementing class. You could also provide multiple classes and later pick among them.

With this preparation done, the program initializes a service loader as follows:

```
public static ServiceLoader<Cipher> cipherLoader = ServiceLoader.load(Cipher.class);
```

This should be done just once in the program.

The iterator method of the service loader returns an iterator through all provided implementations of the service. (See Chapter 9 for more information about iterators.) It is easiest to use an enhanced for loop to traverse them. In the loop, pick an appropriate object to carry out the service.

```
public static Cipher getCipher(int minStrength)
{
    for (Cipher cipher : cipherLoader) // implicitly calls cipherLoader.iterator()
    {
        if (cipher.strength() >= minStrength) return cipher;
    }
    return null;
}
```

Alternatively, you can use streams (see Chapter 1 of Volume II) to locate the desired service. The stream method yields a stream of ServiceLoader.Provider instances. That

interface has methods `type` and `get` for getting the provider class and the provider instance. If you select a provider by type, then you just call `type` and no service instances are unnecessarily instantiated.

```
public static Optional<Cipher> getCipher2(int minStrength)
{
    return cipherLoader.stream()
        .filter(descr -> descr.type() == serviceLoader.impl.CaesarCipher.class)
        .findFirst()
        .map(ServiceLoader.Provider::get);
}
```

Finally, if you are willing to take any service instance, simply call `findFirst`:

```
Optional<Cipher> cipher = cipherLoader.findFirst();
```

The `Optional` class is explained in Chapter 1 of Volume II.

java.util.ServiceLoader<S> 1.6

- `static <S> ServiceLoader<S> load(Class<S> service)`
creates a service loader for loading the classes that implement the given service interface.
- `Iterator<S> iterator()`
yields an iterator that lazily loads the service classes. That is, a class is loaded whenever the iterator advances.
- `Stream<ServiceLoader.Provider<S>> stream() 9`
returns a stream of provider descriptors, so that a provider of a desired class can be loaded lazily.
- `Optional<S> findFirst() 9`
finds the first available service provider, if any.

java.util.ServiceLoader.Provider<S> 9

- `Class<? extends S> type()`
gets the type of this provider.
- `S get()`
gets an instance of this provider.

6.5. Proxies

In the final section of this chapter, we discuss *proxies*. You can use a proxy to create, at runtime, new classes that implement a given set of interfaces. Proxies are only necessary when you don't yet know at compile time which interfaces you need to implement. This is

not a common situation for application programmers, so feel free to skip this section if you are not interested in advanced wizardry. However, for certain systems programming applications, the flexibility that proxies offer can be very important.

6.5.1. When to Use Proxies

Suppose you want to construct an object of a class that implements one or more interfaces whose exact nature you may not know at compile time. This is a difficult problem. To construct an actual class, you can simply use the `newInstance` method or use reflection to find a constructor. But you can't instantiate an interface. You need to define a new class in a running program.

To overcome this problem, some programs generate code, place it into a file, invoke the compiler, and then load the resulting class file. Naturally, this is slow, and it also requires deployment of the compiler together with the program. The *proxy* mechanism is a better solution. The proxy class can create brand-new classes at runtime. Such a proxy class implements the interfaces that you specify. In particular, the proxy class has the following methods:

- All methods required by the specified interfaces; and
- All methods defined in the `Object` class (`toString`, `equals`, and so on).

However, you cannot define new code for these methods at runtime. Instead, you must supply an *invocation handler*. An invocation handler is an object of any class that implements the `InvocationHandler` interface. That interface has a single method:

```
Object invoke(Object proxy, Method method, Object[] args)
```

Whenever a method is called on the proxy object, the `invoke` method of the invocation handler gets called, with the `Method` object and arguments of the original call. The invocation handler must then figure out how to handle the call.

6.5.2. Creating Proxy Objects

To create a proxy object, use the `newProxyInstance` method of the `Proxy` class. The method has three parameters:

- A *class loader*. As part of the Java security model, different class loaders can be used for platform and application classes, classes that are downloaded from the Internet, and so on. We will discuss class loaders in Chapter 9 of Volume II. In this example, we specify the "system class loader" that loads platform and application classes.
- An array of `Class` objects, one for each interface to be implemented.
- An invocation handler.

There are two remaining questions. How do we define the handler? And what can we do with the resulting proxy object? The answers depend, of course, on the problem that we want to solve with the proxy mechanism. Proxies can be used for many purposes, such as

- Routing method calls to remote servers
- Associating user interface events with actions in a running program
- Tracing method calls for debugging purposes

In our example program, we use proxies and invocation handlers to trace method calls. We define a `TraceHandler` wrapper class that stores a wrapped object. Its `invoke` method simply prints the name and arguments of the method to be called and then calls the method with the wrapped object as the implicit argument.

```
class TraceHandler implements InvocationHandler
{
    private Object target;

    public TraceHandler(Object t)
    {
        target = t;
    }

    public Object invoke(Object proxy, Method m, Object[] args)
        throws Throwable
    {
        // print method name and arguments
        . . .
        // invoke actual method
        return m.invoke(target, args);
    }
}
```

Here is how you construct a proxy object that causes the tracing behavior whenever one of its methods is called:

```
Object value = . . . ;
// construct wrapper
var handler = new TraceHandler(value);
// construct proxy for one or more interfaces
var interfaces = new Class[] { Comparable.class };
Object proxy = Proxy.newProxyInstance(
    ClassLoader.getSystemClassLoader(),
    new Class[] { Comparable.class }, handler);
```

Now, whenever a method from one of the interfaces is called on proxy, the method name and arguments are printed out and the method is then invoked on value.

In the program shown in Listing 6.10, we use proxy objects to trace a binary search. We fill an array with proxies to the integers 1 . . . 1000. Then we invoke the `binarySearch` method of the `Arrays` class to search for a random integer in the array. Finally, we print the matching element.

```
var elements = new Object[1000];
// fill elements with proxies for the integers 1 . . . 1000
for (int i = 0; i < elements.length; i++)
{
    Integer value = i + 1;
    elements[i] = Proxy.newProxyInstance(. . .); // proxy for value;
}

// construct a random integer
Integer key = (int) (Math.random() * elements.length) + 1;

// search for the key
int result = Arrays.binarySearch(elements, key);

// print match if found
if (result >= 0) System.out.println(elements[result]);
```

The `Integer` class implements the `Comparable` interface. The proxy objects belong to a class that is defined at runtime. (It has a name such as `$Proxy0`.) That class also implements the `Comparable` interface. However, its `compareTo` method calls the `invoke` method of the proxy object's handler.



Note: As you saw earlier in this chapter, the `Integer` class actually implements `Comparable<Integer>`. However, at runtime, all generic types are erased and the proxy is constructed with the class object for the raw `Comparable` class.

The `binarySearch` method makes calls like this:

```
if (elements[i].compareTo(key) < 0) . . .
```

Since we filled the array with proxy objects, the `compareTo` calls the `invoke` method of the `TraceHandler` class. That method prints the method name and arguments and then invokes `compareTo` on the wrapped `Integer` object.

Finally, at the end of the sample program, we call

```
System.out.println(elements[result]);
```

The `println` method calls `toString` on the proxy object, and that call is also redirected to the invocation handler.

Here is the complete trace of a program run:

```
500.compareTo(288)
250.compareTo(288)
375.compareTo(288)
312.compareTo(288)
281.compareTo(288)
296.compareTo(288)
288.compareTo(288)
288.toString()
```

You can see how the binary search algorithm homes in on the key by cutting the search interval in half in every step. Note that the `toString` method is proxied even though it does not belong to the `Comparable` interface—as you will see in the next section, certain `Object` methods are always proxied.

Listing 6.10 proxy/ProxyTest.java

```
1 package proxy;
2
3 import java.lang.reflect.*;
4 import java.util.*;
5
6 /**
7  * This program demonstrates the use of proxies.
8  * @version 1.02 2021-06-16
9  * @author Cay Horstmann
10 */
11 public class ProxyTest
12 {
13     public static void main(String[] args)
14     {
15         var elements = new Object[1000];
16
17         // fill elements with proxies for the integers 1 . . . 1000
18         for (int i = 0; i < elements.length; i++)
19         {
20             Integer value = i + 1;
21             var handler = new TraceHandler(value);
22             Object proxy = Proxy.newProxyInstance(
23                 ClassLoader.getSystemClassLoader(),
24                 new Class[] { Comparable.class }, handler);
25             elements[i] = proxy;
26         }
27
28         // construct a random integer
29         Integer key = (int) (Math.random() * elements.length) + 1;
30
```

```
31     // search for the key
32     int result = Arrays.binarySearch(elements, key);
33
34     // print match if found
35     if (result >= 0) System.out.println(elements[result]);
36 }
37 }
38
39 /**
40  * An invocation handler that prints out the method name and parameters, then
41  * invokes the original method.
42  */
43 class TraceHandler implements InvocationHandler
44 {
45     private Object target;
46
47     /**
48      * Constructs a TraceHandler.
49      * @param t the implicit parameter of the method call
50      */
51     public TraceHandler(Object t)
52     {
53         target = t;
54     }
55
56     public Object invoke(Object proxy, Method m, Object[] args) throws Throwable
57     {
58         // print implicit argument
59         System.out.print(target);
60         // print method name
61         System.out.print("." + m.getName() + "(");
62         // print explicit arguments
63         if (args != null)
64         {
65             for (int i = 0; i < args.length; i++)
66             {
67                 System.out.print(args[i]);
68                 if (i < args.length - 1) System.out.print(", ");
69             }
70         }
71         System.out.println(")");
72
73         // invoke actual method
74         return m.invoke(target, args);
75     }
76 }
```

6.5.3. Properties of Proxy Classes

Now that you have seen proxy classes in action, let's go over some of their properties. Remember that proxy classes are created on the fly in a running program. However, once

they are created, they are regular classes, just like any other classes in the virtual machine.

All proxy classes extend the class `Proxy`. A proxy class has only one instance field—the invocation handler, which is defined in the `Proxy` superclass. Any additional data required to carry out the proxy objects' tasks must be stored in the invocation handler. For example, when we proxied `Comparable` objects in the program shown in Listing 6.10, the `TraceHandler` wrapped the actual objects.

All proxy classes override the `toString`, `equals`, and `hashCode` methods of the `Object` class. Like all proxy methods, these methods simply call `invoke` on the invocation handler. The other methods of the `Object` class (such as `clone` and `getClass`) are not redefined.

The names of proxy classes are not defined. The `Proxy` class in Oracle's virtual machine generates class names that begin with the string `$Proxy`.

There is only one proxy class for a particular class loader and ordered set of interfaces. That is, if you call the `newProxyInstance` method twice with the same class loader and interface array, you get two objects of the same class. You can also obtain that class with the `getProxyClass` method:

```
Class proxyClass = Proxy.getProxyClass(null, interfaces);
```

A proxy class is always public and final. If all interfaces that the proxy class implements are public, the proxy class does not belong to any particular package. Otherwise, all non-public interfaces must belong to the same package, and the proxy class will also belong to that package.

You can test whether a particular `Class` object represents a proxy class by calling the `isProxyClass` method of the `Proxy` class.



Note: Calling a default method of a proxy triggers the invocation handler. To actually invoke the method, use the static `invokeDefault` method of the `InvocationHandler` interface. For example, here is an invocation handler that calls the default methods and passes the abstract methods to another target:

```
InvocationHandler handler = (proxy, method, args) ->
{
    if (method.isDefault())
        return InvocationHandler.invokeDefault(proxy, method, args)
    else
        return method.invoke(target, args);
}
```

java.lang.reflect.InvocationHandler 1.3

- `Object invoke(Object proxy, Method method, Object[] args)`
define this method to contain the action that you want carried out whenever a method was invoked on the proxy object.
- `static Object invokeDefault(Object proxy, Method method, Object... args) 16`
invokes a default method of the proxy instance with the given arguments, bypassing the invocation handler.

java.lang.reflect.Proxy 1.3

- `static Class<?> getProxyClass(ClassLoader loader, Class<?>... interfaces)`
returns the proxy class that implements the given interfaces.
- `static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler handler)`
constructs a new instance of the proxy class that implements the given interfaces. All methods call the `invoke` method of the given handler object.
- `static boolean isProxyClass(Class<?> cl)`
returns true if `cl` is a proxy class.

This ends the final chapter on the object-oriented features of the Java programming language. Interfaces, lambda expressions, and inner classes are concepts that you will encounter frequently, whereas cloning, service loaders, and proxies are advanced techniques that are of interest mainly to library designers and tool builders, not application programmers. You are now ready to learn how to deal with exceptional situations in your programs in Chapter 7.

Index

Symbols

- ! operator 54, 59
- != operator 54, 59, 97
- """ . . . """ (triple quotes, for text blocks) 73
- ". . ." (single quotes, for strings) 35
- # (number sign)
 - in javadoc hyperlinks 211
 - printf flag 81
- \$ (dollar sign)
 - delimiter, for inner classes 381
 - in variable names 44
 - printf flag 81
- % (percent sign)
 - arithmetic operator 48, 59
 - conversion character 80
- & (ampersand)
 - bitwise operator 57, 59
 - in bounding types 466
 - in reference parameters (C++) 169
- && operator 54, 59
- > (right angle bracket)
 - in shell syntax 85, 454
 - relational operator 54, 59
- >& (shell syntax) 454
- >>, >>> operators 57, 59
- >= operator 54, 59
- < (left angle bracket)
 - in shell syntax 85
 - printf flag 81
 - relational operator 54, 59
- << operator 57, 59
- <. . .> (angle brackets) 258, 462
- <= operator 54, 59
- ' , " (single, double quote), escape sequences for 40
- ((left parenthesis) 81
 - printf flag 81
- (. . .) (parentheses)
 - empty, in method calls 35
 - for casts 52, 59, 234
 - for operator hierarchy 58
- * (asterisk)
 - arithmetic operator 48, 59
 - for annotation processors 729
 - in class path 198
 - in imports 189
- + (plus sign)
 - arithmetic operator 48, 52, 59
 - for objects and strings 60, 251, 252
 - printf flag 81
- ++ operator 54, 59
- , (comma)
 - operator (C++) 59
 - printf flag 81
- (minus sign)
 - arithmetic operator 48, 59
 - printf flag 81
- > operator
 - in lambda expressions 355
 - in switch expressions 103
- operator 54, 59
- . (period) 197, 198
- ... (ellipsis) 270
- .class extension 32
- .exe extension 203
- .java extension 32
- / (slash) 48, 59
- /* . . . */ comments 35
- /** . . . */ (Javadoc comment delimiters) 35, 206, 207
- // comments 35
- 0, 0b, 0B, 0x, 0X prefixes (in integers) 37
- 0, printf flag 81
- 2> (shell syntax) 454
- : (colon)
 - in assertions 432
 - in class path (UNIX) 197
 - inheritance token (C++) 218
- :: (C++ operator) 151, 160, 221, 361
- ; (semicolon)
 - in class path (Windows) 197
 - in statements 34, 43
- = operator 45, 53
- == operator 54, 59
 - for class objects 298
 - for enumerated types 278
 - for floating-point numbers 97
 - for identity hash maps 560
 - for strings 65
 - wrappers and 266
- ? (question mark)
 - for wildcard types 477
- ?: operator 55, 59
 - with pattern matching 237
- @ (at sign) 207, 208
 - in java command-line options 766
- [. . .] (brackets)
 - empty, in generics 464
 - for arrays 112, 115
- \ (backslash)
 - escape sequence for 40
 - in file names 83
 - in text blocks 74
- \b (backslash character literal) 40
- \f (form feed character literal) 40
- \n (newline character literal) 40, 74
- \r (carriage return character literal) 40
- \s (space character literal) 40, 75
- \t (tab character literal) 40
- \u (Unicode character literal) 40, 41
- ^ (caret) 57, 59, 356
- _ (underscore)
 - as a reserved word 779

- delimiter, in number literals 37
- in instance field names (C++) 176
- { . . . } (braces)
 - double, in inner classes 387
 - for blocks 33, 34, 86
 - for enumerated type 47
 - in annotation elements 713
 - in lambda expressions 356
- | operator 57, 59
- || operator 54, 59
- ~ operator 57, 59
- ☞ 42

A

- A, a conversion characters 80
- abstract keyword 271, 775
- Abstract classes 271
 - extending 273
 - interfaces and 328, 337
 - no instantiating for 273
 - object variables of 273
- Abstract methods 272
 - in functional interfaces 358
- AbstractCollection class 340, 519, 532
- AbstractProcessor class 729
- acceptEither method 690
- Access modifiers
 - checking 304
 - final 46, 156, 231, 335, 383, 661
 - private 146, 195, 378
 - protected 239, 324, 351
 - public 31, 32, 47, 144, 147, 195, 328, 329
 - public static final 336
 - redundant 336
 - static 33, 34, 157
 - static final 46
 - void 33, 34
- AccessibleObject class
 - canAccess, trySetAccessible methods 316
 - setAccessible method 312, 316
- Accessor methods 138, 152, 153, 478
- accumulate method
 - of LongAccumulator 664
- accumulateAndGet method 663
- ActionListener interface
 - actionPerformed method 342, 343, 354, 376, 377, 382, 385
 - implementing 359
- ActiveX 4
- add method
 - of ArrayList 258, 264
 - of BigDecimal 110
 - of BigInteger 110
 - of BlockingQueue 668, 669
 - of Collection 514, 519, 520, 522
 - of GregorianCalendar 139
 - of HashSet 539
 - of List 523, 535
 - of ListIterator 529, 530, 536
 - of LongAdder 663
 - of Queue 546
 - of Set 524
- addAll method
 - of ArrayList 461
 - of Collection 519, 521
 - of Collections 582
 - of List 535
- addExact method 51
- addFirst method
 - of LinkedList 536
 - of SequencedCollection 546
- Addition 48, 59
 - for different numeric types 52
 - for objects and strings 60, 251, 252
- addLast method
 - of LinkedList 536
 - of SequencedCollection 546
- addShutdownHook method 182
- addSuppressed method 423, 425
- Adobe Flash 7
- Agent code 743, 744
- Aggregation 130
- Algorithms 127
 - for binary search 578
 - for shuffling 577
 - for sorting 576
 - QuickSort 117, 576
 - simple, in the Java Collections Framework 580
 - writing 584
- Algorithms + Data Structures = Programs* (Wirth) 127
- Algorithms in C++* (Sedgewick) 576
- allOf method
 - of CompletableFuture 690
 - of EnumSet 561
- allProcesses method 706, 709
- Amazon 15
- and, andNot methods (BitSet) 594
- Andreessen, Mark 9
- Android 14, 695
- AnnotatedConstruct interface 730
- AnnotatedElement interface 726, 727
 - getAnnotation method 729
 - getAnnotations method 729
 - getAnnotationsByType method 729
 - getDeclaredAnnotations method 729
 - isAnnotationPresent method 728
- Annotation interface
 - extending 719
 - methods of 720
- Annotation interfaces 717
- Annotation processors 729
 - at bytecode level 736
- Annotations 473
 - accessing 719
 - applicability of 721
 - container 723, 726
 - declaration 713

- documented 721, 723
 - generating source code with 731
 - inherited 721, 723, 726
 - key/value pairs in 712, 719
 - meta 717, 724
 - modifiers and 715
 - multiple 713
 - processing
 - at runtime 725
 - source-level 729
 - repeatable 713, 721, 723, 724, 726
 - standard 720
 - type use 714
- Anonymous arrays 111
- Anonymous inner classes 385
- Antisymmetry rule 335
- anyOf method 690
- Apache
 - Commons CSV library 763
- append method
 - of `StringBuilder` 71, 73
- appendCodePoint method 73
- Applets 7, 13
 - changing warning string in 196
 - running in a browser 7
- Application Programming Interfaces (APIs), online documentation for 66, 68
- Applications
 - compiling/launching from the command line 19, 32
 - debugging 21, 403
 - executing
 - without a separate Java runtime 773
 - extensible 230
 - for different Java releases 203
 - localizing 133, 302
 - managing in JVM 456
 - monitoring 743, 744
 - responsive 694
 - terminating 34, 149
 - testing 431
- applyToEither method 690
- Arguments 35
 - of `ProcessHandle.Info` 710
 - string 35
 - variable number of 270
- Arguments. See Parameters
- Arithmetic operators 48
 - accuracy of 49
 - autoboxing with 266
 - combining with assignment 53
 - precedence of 59
- Array class 316
 - get, getXxx, set, setXxx methods 320
 - getLength method 317, 318, 320
 - newInstance method 317, 320
- Array lists 537
 - anonymous 387
 - capacity of 259
 - elements of
 - accessing 260
 - adding 258, 262
 - removing 262
 - traversing 262
 - generic 257
 - raw vs. typed 264
- Array variables 111
- ArrayBlockingQueue class 669, 673
- ArrayDeque class 545, 547
 - as a concrete collection type 525
- ArrayIndexOutOfBoundsException class 113, 406, 408
- ArrayList class 113, 257, 459, 526
 - add method 258, 264
 - addAll method 461
 - as a concrete collection type 525
 - declaring with var 258
 - ensureCapacity method 259, 260
 - get, set methods 260, 264
 - iterating over 516
 - remove method 262, 264
 - removeIf method 360
 - size, trimToSize methods 259, 260
 - synchronized 684
 - toArray method 491
- Arrays 112
 - annotating 715
 - anonymous 111
 - circular 514
 - cloning 352
 - converting to collections 583
 - copying 114
 - on write 682
 - creating 111
 - elements of
 - computing in parallel 683
 - numbering 113
 - remembering types of 228
 - removing from the middle 526, 528
 - traversing 113, 121
 - equality testing for 245, 246
 - generic methods for 316
 - hash codes of 249, 250
 - in command-line arguments 116
 - initializing 111, 113
 - length of 113
 - equal to 0 112
 - increasing 115
 - multidimensional 119, 124, 245, 252
 - not of generic types 366, 475, 486, 490
 - of integers 252
 - of subclass/superclass references 227
 - of wildcard types 487
 - out-of-bounds access in 406
 - parallel operations on 682
 - printing 121, 252
 - ragged 122
 - size of 259, 317
 - setting at runtime 257
 - sorting 117, 332, 682
- Arrays class
 - asList method 572

- binarySearch method 398, 580
- copyOf method 114, 119, 316
- copyOfRange method 119
- deepEquals method 245
- deepToString method 121, 252
- equals method 119, 245, 246
- fill method 119
- hashCode method 249, 250
- parallelXxx methods 682
- sort method 117, 119, 329, 332, 334, 355, 359
- toString method 114, 119
- ArrayStoreException class 228, 475, 486, 488
- arrayType method 317, 320
- ASCII 41
- asIterator method 587, 588
- asList method 572
- ASM library 736
- assert keyword 431, 775
- Assertions 431
 - checking 714
 - checking parameters with 434
 - defined 431
 - documenting assumptions with 435
 - enabling/disabling 432, 434
- Assignment 45, 53
- Asynchronous computations 685
- Asynchronous methods 618
- AsyncTask class (Android) 695
- atan, atan2 methods (Math) 49
- Atomic operations 662
 - client-side locking for 658
 - in concurrent hash maps 676
 - performance of 663
- AtomicType classes 662
- @author annotation 212, 214
- Autoboxing 265
- AutoCloseable interface 421
 - close method 421, 423
- await method 606, 649
 - of Condition 645
- awaitTermination method 625
- Azul 15

B

- B, b conversion characters 80
- Base classes. See Superclasses
- BASE64Encoder class 753
- Basic multilingual planes 41
- Batch files 199
- Beans class 201
- beep method
 - of Toolkit 345
- BiConsumer interface 370
- BiFunction interface 359, 370
- BIG-5 41
- BigDecimal class 107
 - add, compareTo, subtract, multiply, divide, mod methods 110

- BigInteger class 107, 110
 - add, compareTo, subtract, multiply, divide, mod, sqrt methods 110
 - valueOf method 107, 110
- Binary search 578
- BinaryOperator interface 370
- binarySearch method
 - of Arrays 398, 580
 - of Collections 578
- BiPredicate interface 370
- Bit masks 57
- Bit sets 593
- BitSet class 511, 593
 - methods of 594
- Bitwise operators 57, 59
- Blank lines, printing 35
- Blocking queues 668
- BlockingDeque interface, methods of 674
- BlockingQueue interface
 - add, element, peek, remove methods 668, 669
 - offer, poll, put, take methods 668, 669, 673
- Blocks 33, 34, 86
 - nested 86
 - synchronized 657
- boolean type 43, 775
 - converting from boolean 265
 - default initialization of 172
 - formatting output for 80
 - hashCode method 250
 - no casting to numeric types for 53
- boolean operators 54, 59
- Bounded collections 514
- Bounds checking 115
- break keyword 100, 107, 775
 - labeled/unlabeled 105
 - not allowed in switch expressions 104
- Bridge methods 471, 472, 496
- Buckets (of hash tables) 538
- Bulk operations 582
- byte type 36, 775
 - converting from byte 265
 - hashCode method 250
 - toUnsignedInt method 38
- Bytecode files 32
- Bytecodes
 - engineering 736
 - at load time 743, 744

C

- C
 - assert macro in 432
 - function pointers in 320
 - integer types in 5, 37
- C# 7
 - foreach loop in 85
 - polymorphism in 233
 - useful features of 10
- C++

- #include in 190
 - >> operator in 58
 - , (comma) operator in 59
 - :: operator in 151, 221
 - access privileges in 155
 - algorithms in 575
 - arrays in 115, 125
 - bitset template in 594
 - boolean values in 43
 - classes in 34, 375
 - copy constructors in 136
 - dynamic binding in 223
 - dynamic casts in 236
 - exceptions in 406, 409, 410, 414
 - fields in
 - instance 174, 176
 - static 160
 - for loop in 85, 95
 - function pointers in 320
 - inheritance in 218, 226, 337
 - integer types in 5, 37
 - iterators as parameters in 587
 - methods in
 - accessor 139
 - default 340
 - destructor 182
 - static 160
 - namespace directive in 190
 - new operator in 148
 - NULL pointer in 135
 - object pointers in 135
 - operator overloading in 108
 - passing parameters in 167, 169
 - performance of, compared to Java 595
 - polymorphism in 233
 - protected modifier in 239
 - pure virtual functions (= 0) in 274
 - references in 135
 - Standard Template Library in 511, 516
 - static member functions in 34
 - strings in 64, 65
 - superclasses in 222
 - syntax of 2
 - templates in 10, 463, 466, 469
 - this pointer in 176
 - type parameters in 465
 - using directive in 190
 - variables in 46
 - redefining in nested blocks 86
 - vector template in 260
 - virtual constructors in 299
 - void* pointer in 240
- C, c conversion characters 80
- CachedRowSetImpl class 753
- Calendar class 136
 - get/setTime methods 231
- Calendars
 - displaying 139, 141
 - vs. time measurement 137
- Call by reference 164
- Call by value 164
- Callable interface 625
 - call method 618, 620
 - wrapper for 619
- Callables 618
- Callbacks 342
- CamelCase 32
- canAccess method 316
- cancel method 618, 620, 623
 - of Future 697
- CancellationException class 697
- Cardinality 594
- Carriage return character 40
- case keyword 56, 99, 775
- cast method 499
 - of Class 499
- Casts 52, 234
 - annotating 715
 - bad 406
 - checking before attempting 235
- catch keyword 412, 775
 - annotating parameters of 713
- ceiling method
 - of NavigableSet 545
- char type 39, 775
- Character class
 - converting from char 265
 - hashCode method 250
 - isJavaIdentifierXxx methods 44
- Characters
 - escape sequences for 40
 - exotic 42
 - formatting output for 80
- charAt method 62, 67
- CharSequence interface 68, 338
- Checked exceptions 298, 300
 - applicability of 429
 - declaring 407
 - suppressing with generics 493
- Checked views 568
- checkedCollection methods (Collections) 572
- Checker Framework 714
- checkFromIndexSize, checkFromToIndex, checkIndex methods (Objects) 430
- Child classes. See Subclasses
- children method
 - of ProcessHandle 706, 709
- ChronoLocalDate 481
- Church, Alonzo 356
- Circular arrays 514
- Clark, Jim 9
- class keyword 31, 297, 775
 - arrayType method 317, 320
 - cast method 499
 - componentType method 320
 - forName method 297, 300
 - getClass method 297
 - getComponentType method 317, 320
 - getConstructor method 300, 499
 - getConstructors method 304, 309

- getDeclaredConstructor method 499
- getDeclaredConstructors method 304, 309
- getDeclaredMethods method 304, 309, 321
- getEnumConstants method 499
- getField, getDeclaredField methods 316
- getFields, getDeclaredFields methods 304, 309, 313, 316
- getGenericXxx methods 508
- getImage method 302
- getMethod method 321
- getMethods method 304, 309
- getName method 257, 297, 298
- getPackageName method 310
- getRecordComponents method 310
- getResource, getResourceAsStream methods 302, 303
- getResourceAsStream method 761
- getSuperclass method 257, 499
- getTypeParameters method 508
- isArray method 319
- isEnum, isInterface, isRecord methods 309
- newInstance method 299, 499
- Class constants 46
- Class declarations
 - annotations in 713, 723
- Class diagrams 131, 132
- Class files 192, 197
 - compiling 32
 - format of 736
 - locating 198, 199
 - modifying 736
 - names of 32, 144
 - transformers for 743
- Class literals
 - no annotations for 715
- Class loaders 396, 432
- Class path 197, 200
- Class wins rule 342
- Class<T> parameters 499
- ClassCastException class 235, 317, 335, 475, 491, 499, 569
- Classes 31, 128, 217
 - abstract 271, 328, 337
 - access privileges for 154
 - adding to packages 192
 - capabilities of 304
 - companion 338, 340
 - constructors for 146
 - defining 142
 - at runtime 396
 - deprecated 720, 721
 - designing 130, 214
 - documentation comments for 207, 211
 - encapsulation of 129, 152, 747
 - extending 129
 - final 231, 351
 - generic 257, 258, 462, 476, 490, 498, 501
 - immutable 156, 183, 325
 - implementing multiple interfaces 336, 337
 - importing 189
 - inner 375
 - instances of 128, 133
 - legacy 184
 - loading 455
 - multiple source files for 145
 - names of 21, 31, 188, 216
 - full package 189
 - nested 715
 - number of basic types in 215
 - objects of, at runtime 311
 - package scope of 195
 - parameters in 151
 - predefined 132
 - private methods in 155
 - protected 239
 - public 189, 207
 - relationships between 130
 - sealed 282
 - sharing, among programs 197
 - unit testing 161
 - wrapper 265
- ClassLoader class 436
- CLASSPATH 199
- clear method
 - of BitSet 594
 - of Collection 519, 521
- clearAssertionStatus method 436
- Client-side locking 657, 658
- clone method
 - of array types 352
 - of Object 154, 347, 358
- Cloneable interface 347
- CloneNotSupportedException class 350, 351
- close method 624
 - of AutoCloseable 421, 423
 - of Closeable 422
 - of Handler 450
- Closures 367
- Code errors 404
- Code generator tools 722
- Code planes 42
- Code points, code units 42, 62
- Collection interface 514, 522, 532
 - add method 514, 519, 520, 522
 - addAll method 519, 521
 - clear method 519, 521
 - contains, containsAll methods 519, 520, 532
 - equals method 519
 - generic 518, 521
 - implementing 340
 - isEmpty method 339, 519, 520
 - iterator method 514, 520
 - remove method 519, 521
 - removeAll method 519, 521
 - removeIf method 521, 582
 - retain method 519
 - retainAll method 521
 - size method 519, 520
 - stream method 340
 - toArray method 262, 519, 521, 584
- Collections class 511

- algorithms for 574
 - bounded 514
 - bulk operations in 582
 - concrete 525
 - concurrent modifications of 532
 - converting to arrays 583
 - debugging 532
 - elements of
 - inserting 522
 - maximum 574
 - removing 518
 - traversing 515, 516
 - interfaces for 511
 - legacy 586
 - mutable 564
 - ordered 523, 529
 - performance of 523, 539
 - searching in 578
 - sorted 542
 - thread-safe 569, 667
 - type parameters for 461
 - using for method parameters 585
- Collections class** 577
- `addAll` method 582
 - `binarySearch` method 578
 - `checkedCollection` methods 572
 - `copy` method 581
 - `disjoint` method 582
 - `emptyCollection` methods 572
 - `enumeration` method 587, 588
 - `fill` method 581
 - `frequency` method 582
 - `indexOfSubList` method 582
 - `lastIndexOfSubList` method 582
 - `list` method 588
 - `max, min` methods 581
 - `nCopies` method 564, 572
 - `replaceAll` method 582
 - `reverse` method 582
 - `rotate` method 582
 - `shuffle` method 577, 578
 - `sort` method 576
 - `swap` method 582
 - `synchronizedCollection` methods 569, 571, 685
 - `unmodifiableCollection` methods 565, 566, 571
- command method**
of `ProcessHandle.Info` 710
- Command line**
arguments in 116
compiling/launching from 19, 32
- commandLine method**
of `ProcessHandle.Info` 710
- Comments** 35
automatic documentation and 35, 206
blocks of 35
not nesting 36
to the end of line 35
- Commons CSV library** 763
- Companion classes** 338, 340
- Comparable interface** 328, 398, 466, 539, 576
- `compareTo` method 328, 333, 466, 480
- Comparator interface** 345, 354, 373, 552, 576
- chaining comparators in 374
 - `comparing` method 374
 - lambda expressions and 358
 - `naturalOrder` method 374
 - `nullFirst/Last` methods 374
 - of `SortedSet` 545
 - reversed, `reverseOrder` methods 375, 576, 578
 - `thenComparing` method 374
- compare method (integer types)** 334, 359
- compareAndSet method** 662
- compareTo method**
in subclasses 335
of `BigDecimal` 110
of `BigInteger` 110
of `Comparable` 328, 333, 466, 480
of `Enum` 282
of `String` 67
- Compilation errors** 24
- Compiler**
autoboxing in 267
bridge methods in 471
command-line options of 455
creating bytecode files in 32
deducting method types in 465
enforcing throws specifiers in 413
error messages in 24, 408
just-in-time 5, 6, 13, 152, 233, 595
launching 19
optimizing method calls in 6, 233
overloading resolution in 228
shared strings in 64, 65
translating typed array lists in 265
type parameters in 460
warnings in 102, 265
whitespace in 33
- CompletableFuture class** 687
- `acceptEither` method 690
 - `allOf, anyOf` methods 690
 - `applyToEither` method 690
 - exceptionally, `exceptionallyCompose` methods 689, 690
 - `handle` method 689
 - `orTimeout` method 689
 - `runAfterXxx` methods 690
 - `thenAccept, thenAcceptBoth, thenCombine, thenRun` methods 689
 - `thenApply, thenApplyAsync, thenCompose` methods 688, 689
 - `whenComplete` method 689
- CompletionStage interface** 691
- Components (of records)** 183
- componentType method** 320
- Computations**
asynchronous 685
performance of 49, 50
truncated 49
- compute, computeIfXxx methods**
of `ConcurrentHashMap` 677

- of Map 554
- Concrete collections 525
- Concrete methods 273
- Concurrent hash maps
 - atomic updates in 676
 - bulk operations on 679
 - efficiency of 675
 - size of 674
 - vs. synchronization wrappers 684
- Concurrent modification detection 532
- Concurrent programming 6, 599, 705
 - records in 185
 - synchronization in 635
- Concurrent sets 682
- ConcurrentHashMap class 674
 - atomic updates in 676
 - compute, computeIfXxx methods 677, 678
 - forEach method 679
 - forEach, forEachXxx methods 681
 - get method 676
 - keySet, newKeySet methods 682
 - mappingCount method 674
 - merge method 677, 678
 - organizing buckets as trees in 675
 - put, putIfAbsent methods 676
 - reduce, reduceXxx methods 679, 681
 - replace method 676
 - search, searchXxx methods 679, 681
- ConcurrentLinkedQueue class 674
- ConcurrentModificationException class 532, 675, 684
- ConcurrentSkipListMap/Set classes 674
- Condition interface 652
 - await method 606, 649
 - signal method 649
 - signal, signalAll methods 650
 - signalAll method 649
 - vs. synchronization methods 654
- Condition objects 644
- Condition variables 644
- Conditional operator 55
 - with pattern matching 237
- Conditional statements 86
- Configuration files
 - editing 440
- Console class
 - of System 79
 - printing output to 31, 79
 - reading input from 76
- Console class 78
 - readLine/Password methods 79
- ConsoleHandler class 441, 444, 451
- const keyword 47, 775
- Constants 46
 - documentation comments for 209
 - names of 46
 - public 47, 158
 - static 158
- Constructor class 304
 - getDeclaringClass method 310
 - getModifiers, getName methods 304, 310
 - getXxxTypes methods 310
 - newInstance method 300
- Constructor expressions 489
- Constructor references 365
 - annotating 715
- Constructors 146, 148, 171
 - annotating 713, 715
 - calling another constructor in 176
 - canonical, compact, custom 185
 - defined 133
 - documentation comments for 207
 - field initialization in 172, 174
 - final 304
 - initialization blocks in 176
 - names of 133, 147
 - no-argument 173, 221, 394
 - overloading 171
 - parameter names in 175
 - private 304
 - protected 207
 - public 207, 304
 - with super keyword 221
- Consumer interface 370
- Consumer threads 668
- contains method
 - of Collection 519, 520, 532
 - of HashSet 539
- containsAll method 519, 520, 532
- containsKey/Value methods (Map) 551
- continue keyword 106, 107, 775
 - not allowed in switch expressions 104
- Control flow 85
 - block scope 86
 - breaking 104
 - conditional statements 86
 - loops 90
 - determinate 95
 - "for each" 113
 - multiple selections 99
- Conversion characters 80
- Cooperative scheduling 606
- Coordinated Universal Time (UTC) 136
- Copies 562
 - unmodifiable 565, 570
- copy method
 - of Collections 581
- copyOf method
 - of Arrays 114, 119, 316
 - of EnumSet 561
 - of List, Map, Set 565, 570
 - of Map.Entry 557
- copyOfRange method 119
- CopyOnWriteArrayList class 682, 684
- CopyOnWriteArraySet class 682
- CORBA 747
- Cornell, Gary 1
- Corruption of data 636, 640
- cos method
 - of Math 49
- Count of Monte Cristo, The* (Dumas) 696, 698

Covariant return types 472
 CSV files 763
 Ctrl+\, for thread dump 650
 Ctrl+C, for program termination 637, 647
 current method
 of `ProcessHandle` 706, 709
 of `ThreadLocalRandom` 667
`currentThread` method 609

D

d conversion character 80
 D, d suffixes (for double numbers) 38
 Daemon threads 613
 Data types 36
 boolean type 43
 casting between 52
 char type 39
 conversions between 51, 234
 floating-point 38
 integer 36
 Databases 711
`DataFlavor` class 748
 Date class 136
 `getDay/Month/Year` methods (deprecated) 138
 `toString` method 133
 Date and time
 formatting output for 80
 hash codes for 248
 no built-in types for 133
 Deadlocks 646, 649
 Debugging 7, 452
 collections 532
 debuggers for 452
 generic types 568
 GUI programs 412
 including class names in 387
 messages for 411
 reflection for 312
 trapping program errors in a file for 454
 when running applications in terminal window 21
 Decrement operators 54
`decrementExact` method 51
 Deep copies 348
`deepEquals` method 245
`deepToString` method 121, 252
 default keyword 100, 339, 775
 sealed classes and 284
 Default for annotation element 719
 Default methods 339
 conflicts in 340
 Deferred execution 369
 delete method
 of `StringBuilder` 73
 Dependence 130
 @Deprecated annotation 720, 721
 Deprecated methods 138
 Deque interface 545
 methods of 547
 Deques 545
 Derived classes. See Subclasses
 descendants method
 of `ProcessHandle` 706, 709
 destroy, destroyForcibly methods (`Process`) 705, 709
 Determinate loops 95
 Development environments
 choosing 19
 in terminal window 21
 integrated 24
 Device errors 404
 Dialogs
 centering 344
 Diamond syntax 258
 with anonymous subclasses 460
 Digital signatures 4
 Directories
 starting, for a launched program 84
 working, for a process 702
 directory method 708
 of `ProcessBuilder` 702
 disjoint method
 of `Collections` 582
 divide method
 of `BigDecimal` 110
 of `BigInteger` 110
 Division 48
 do/while loop 92, 93, 775
 Documentation comments 35, 206
 extracting 213
 for fields 209
 for methods 208
 for packages 209
 general 212
 HTML markup in 210
 hyperlinks in 211
 inserting 207
 links to other files in 212
 overview 214
 @Documented annotation 721, 723
`doInBackground` method 696, 697, 701
 double type 38, 775
 arithmetic computations with 48
 compare method 334
 converting from double 265
 converting to other numeric types 51
 hashCode method 250
 POSITIVE_INFINITY, NEGATIVE_INFINITY, NaN
 constants 39
 Double brace initialization 387
 Double-precision numbers 38
`DoubleAccumulator`, `DoubleAdder` classes 664
 Doubly linked lists 526
 Dynamic binding 223, 228
 Dynamic languages 7

E

E

- constant (Math) 50
- E, e conversion characters 39, 80
- Eclipse 19, 24, 452
 - Adoptium 15
 - configuring projects in 26
 - editing source files in 27
 - error messages in 24, 25
 - IDE 756
 - imports in 190
 - Yasson framework 761
- Effectively final variables 422
- Eiffel programming language 337
- element method 730, 731
 - of BlockingQueue 668, 669
 - of Queue 546
- Elements 712
- else keyword 87, 88, 775
- else if 88, 90
- Emoji characters 42
- emptyCollection methods (Collections) 572
- EmptyStackException class 428, 430
- Encapsulation 129, 747
 - benefits of 152
 - compile-time 765
 - protected instance fields and 324
- endsWith method 67
- ensureCapacity method 259, 260
- Enterprise Edition 10
- entry method
 - of Map 564, 571
- EntryLogger 743
- EntryLoggingAgent.mf 743
- entrySet method 554, 556
- enum keyword 47, 277, 775
 - compareTo, ordinal methods 282
 - toString, valueOf methods 279, 281
- Enumerated types 47
 - equality testing for 278
 - in switch statement 56
- Enumeration interface 511, 587
 - asIterator method 587, 588
 - hasMoreElements, nextElement methods 516, 587
 - of Collections 588
- Enumeration maps/sets 559
- Enumerations 277
 - always final 234
 - annotating 713
 - declared inside a class 390
 - implementing interfaces 336
 - legacy 587
- EnumMap class 559, 562
 - as a concrete collection type 525
- EnumSet class 559
 - allOf, copyOf, noneOf, of, range methods 561
 - as a concrete collection type 525
- environment method
 - of ProcessBuilder 708
- Environment variables, modifying 704
- EOFException class 409, 410
- Epoch 136
- equals method 342
 - hashCode method and 248, 249
 - implementing 244
 - inheritance and 242
 - of Annotation 720
 - of Arrays 119, 245, 246
 - of Collection 519
 - of Object 241, 246, 256, 566
 - of proxy classes 401
 - of records 184, 242
 - of Set 524
 - of String 65, 67
 - redefining 248, 249
 - wrappers and 267
- equalsIgnoreCase method 65, 67
- Error class 405
- Errors
 - checking, in mutator methods 153
 - code 404
 - compilation 24
 - device 404
 - internal 405, 408, 434
 - messages for 415
 - NoClassDefFoundError 21
 - physical limitations 404
 - user input 404
- Escape sequences 40
- Exception class 406, 426
- Exception handlers 300, 404
- Exception specification 407
- exceptionally, exceptionallyCompose methods (CompletableFuture) 689, 690
- Exceptions 405
 - annotating 715
 - ArrayIndexOutOfBoundsException 113, 406, 408
 - ArrayStoreException 228, 475, 486, 488
 - CancellationException 697
 - catching 149, 300, 351, 408, 412
 - changing type of 417
 - checked 298, 300, 406, 409, 427, 429
 - ClassCastException 235, 317, 335, 475, 491, 499, 569
 - CloneNotSupportedException 350, 351
 - ConcurrentModificationException 532, 675, 684
 - creating classes for 410, 411
 - documentation comments for 208
 - EmptyStackException 428, 430
 - EOFException 409, 410
 - FileNotFoundException 407, 409
 - finally clause in 418
 - generics in 493
 - hierarchy of 405, 429
 - IllegalAccessError 312, 316
 - IllegalStateException 518, 521, 536, 546, 547, 668
 - InaccessibleObjectException 312
 - InterruptedException 601, 609, 618
 - InvocationTargetException 299
 - IOException 84, 407, 409, 413, 422
 - logging 446
 - micromanaging 428
 - NoSuchElementException 515, 521, 536, 546, 547

- NullPointerException 149, 150, 164, 267, 364, 406, 430
 - NumberFormatException 429
 - out-of-bounds 430
 - propagating 413, 430
 - rethrowing and chaining 417, 454
 - RuntimeException 406, 429
 - ServletException 417
 - sqlclching 429
 - stack trace for 423
 - “throw early, catch late” 431
 - throwing 300, 409
 - TimeoutException 618
 - tips for using 427
 - type variables in 492
 - uncaught 454, 608, 614
 - unchecked 300, 406, 408, 429
 - unexpected 446
 - UnsupportedOperationException 556, 566, 569, 571
 - variables for, implicitly final 415
 - vs. simple tests 427
 - wrapping 417
 - exec method
 - of Runtime 702
 - Executable class 321
 - Executable JAR files 202
 - ExecutableElement interface 730
 - execute method 697
 - of SwingWorker 702
 - ExecutorCompletionService class 625
 - poll, submit, take methods 630
 - Executors class 621
 - groups of tasks, controlling 625
 - Executors class, newXxx methods 621, 624
 - ExecutorService interface 624
 - awaitTermination method 625
 - close method 624
 - invokeAny/All methods 625, 630
 - shutdown method 623, 624
 - shutdownNow method 623
 - submit method 622, 624
 - exit method
 - of System 34
 - Exit codes 34
 - exitValue method 705, 709
 - exp method
 - of Math 50
 - Explicit parameters 151
 - Exploratory programming 6
 - exports keyword 753, 755, 756, 768, 776
 - Expressions 53
 - extends keyword 217, 466, 776
- ## F
- F, f conversion characters 80
 - F, f suffixes (for floatnumbers) 38
 - Factory methods 160
 - Fair locks 644
 - Fallthrough behavior 100
 - false literal 776
 - fdlibm library 50
 - Field class 304
 - get method 311, 316
 - getDeclaringClass method 310
 - getModifiers, getName methods 304, 310
 - getType method 304
 - set method 316
 - Fields
 - adding, in subclasses 221
 - annotating 713
 - default initialization of 172
 - documentation comments for 207, 209
 - final 158, 231
 - instance 129, 147, 152, 156, 174, 214
 - private 214, 219, 220
 - protected 207, 239, 324
 - public 207, 209
 - public static final 336
 - static 157, 178, 191, 492
 - volatile 660
 - with the null value 150
 - File handlers 442, 444
 - FileHandler class 442, 444, 451
 - FileNotFoundException class 407, 409
 - Files class
 - locating 84
 - names of 21, 83
 - reading 83
 - all words from 422
 - in a separate thread 696
 - writing 83
 - fill method
 - of Arrays 119
 - of Collections 581
 - Filter class 444
 - isLoggable method 452
 - final keyword 46, 231, 776
 - checking 304
 - for fields in interfaces 336
 - for instance fields 156
 - for methods in superclass 335
 - for shared fields 661
 - inner classes and 383
 - finalize method
 - of Object 182
 - finally keyword 418, 776
 - return statements in 420
 - unlock operation in 641
 - without catch 420
 - Financial calculations 39
 - findFirst method 395
 - first method
 - of SortedSet 545
 - First Person, Inc. 9
 - firstKey method 552
 - Flags, for formatted output 81
 - float type 38, 776
 - converting from float 265

- converting to other numeric types 51
- hashCode method 250
- POSITIVE_INFINITY, NEGATIVE_INFINITY, NaN constants 39
- Floating-point numbers 38
 - arithmetic computations with 48
 - converting from/to integers 234
 - equality of 97
 - formatting output for 80
 - rounding 39, 52
- floor method
 - of NavigableSet 545
- floorMod method 48
- flush method
 - of Handler 450
- for keyword 95, 776
 - comma-separated expressions in 59
 - defining variables inside 97
 - for collections 515
- “for each” loop 114
 - for array lists 262
 - for collections 515, 684
 - for multidimensional arrays 121
- forEach method
 - of ConcurrentHashMap 679, 681
 - of Map 551
 - of StackWalker 426
- forEachRemaining method 515, 521
- Fork-join framework 632
- Form feed character 40
- Format specifiers (printf) 80, 82
- format, formatted, formatTo methods (String) 82
- formatMessage method 452
- Formattable interface 81
- Formatter class 444
- Formatter class, methods of 452
- forName method 297, 300
- frequency method
 - of Collections 582
- Function interface 370, 373
- Functional interfaces 358, 721, 722
 - abstract methods in 358
 - annotating 372
 - conversion to 359
 - generic 359
 - using supertype bounds in 481
- @FunctionalInterface annotation 372, 721, 722, 723
- Functions. See Methods
- Future interface 625
 - cancel, get methods 618, 620, 623, 697
 - isCancelled, isDone methods 618, 620, 623
- Futures 618
 - combining 690
 - completable 687
- FutureTask class 618

G

- G, g conversion characters 80

- Garbage collection 64, 136
 - hash maps and 557
- GB18030 41
- General Public License (GPL) 13
- @Generated annotation 721, 722
- Generic programming 459
 - arrays and 366, 490
 - classes in 257, 258, 462
 - extending/implementing other generic classes 476
 - no throwing or catching instances of 492
 - collection interfaces in 583
 - converting to raw types 475
 - debugging 568
 - expressions in 469
 - in JVM 468, 500
 - inheritance rules for 474, 476
 - legacy code and 472
 - methods in 464, 470, 518
 - reflection and 498
 - required skill levels for 461
 - static fields or methods and 492
 - type erasure in 468, 485, 490
 - clashes after 495
 - type matching in 499
 - vs. inheritance 459
 - wildcard types in 477
- Generic types
 - annotating 715
- GenericArrayType interface 500, 501
 - getGenericComponentType method 509
- get method
 - of Array 320
 - of ArrayList 260, 264
 - of BitSet 594
 - of ConcurrentHashMap 676
 - of Field 311, 316
 - of Future 618, 620, 623, 697
 - of LinkedList 533
 - of List 523, 536
 - of LongAccumulator 664
 - of Map 522, 549, 551
 - of Paths 338
 - of ServiceLoader.Provider 394, 395
 - of ThreadLocal 632
 - of Vector 658
- getAccessor method 310
- getActualTypeArguments method (ParameterizedType) 509
- getAndType methods (AtomicType) 663
- getAnnotation method 729
- getAnnotation, getAnnotationsByType methods
 - of AnnotatedConstruct 730
 - of AnnotatedElement 726, 727
- getAnnotations method 729
- getAnnotationsByType method 729
- getBoolean method
 - of Array 320
- getBounds method (TypeVariable) 508
- getByte method 320
- getCause method 425

- getChar method 320
- getClass method
 - always returning raw types 486
 - of Class 297
 - of Object 256
- getClassName method
 - of StackFrame 426
 - of StackTraceElement 427
- getComponentType method 317, 320
- getConstructor method 300, 499
- getConstructors method 304, 309
- getDay method 138
- getDayXxx methods (LocalDate) 137, 142
- getDeclaredAnnotations method 729
- getDeclaredAnnotationXxx methods (AnnotatedElement) 726, 727
- getDeclaredConstructor method 499
- getDeclaredConstructors method 304, 309
- getDeclaredField method 316
- getDeclaredFields method 304, 309, 313, 316
- getDeclaredMethods method 304, 309, 321
- getDeclaringClass method
 - of java.lang.reflect 310
 - of StackFrame 426
- getDefaultToolkit method 345
- getDefaultUncaughtExceptionHandler method 615
- getDouble method
 - of Array 320
- getElementsAnnotatedWith method 730
- getEnclosedElements method 731
- getEnumConstants method 499
- getErrorStream method 703, 704, 708
- getExceptionTypes method 310
- getField method 316
- getFields method 304, 309, 316
- getFileName method
 - of StackFrame 426
 - of StackTraceElement 427
- getFilter method
 - of Handler 450
- getFirst method 546
- getFloat method
 - of Array 320
- getFormatter method 450
- getGenericComponentType method (GenericArrayType) 509
- getGenericXxx methods (Class) 508
- getGenericXxx methods (Method) 508
- getGlobal method 453
- getHead method 445, 452
- getImage method
 - of Class 302
- getInputStream method 703, 708
- getInstance method 423, 426
- getInstant method 451
- getInt method
 - of Array 320
- getKey method 556
- getLast method 546
- getLength method 317, 318, 320
- getLevel method
 - of Handler 450
 - of LogRecord 451
- getLineNumber method
 - of StackFrame 426
 - of StackTraceElement 427
- getLogger method 437, 439
- getLoggerName method 451
- getLong method
 - of Array 320
- getLongThreadID method 452
- getLowerBounds method (WildcardType) 508
- getMessage method
 - of LogRecord 451
 - of Throwable 411
- getMethod method 321
- getMethodName method
 - of StackFrame 427
 - of StackTraceElement 427
- getMethods method 304, 309
- getMillis method 451
- getModifiers method
 - of java.lang.reflect 310
 - of java.lang.reflect.Member 304
- getMonth method 138
- getMonthXxx methods (LocalDate) 137, 142
- getName method
 - of Class 257, 297, 298
 - of java.lang.reflect 310
 - of java.lang.reflect.Member 304
 - of RecordComponent 310
 - of System.Logger 450
 - of TypeVariable 508
- getOrDefault method 551
- getOutputStream method 703, 708
- getOwnerType method (ParameterizedType) 509
- getPackageName method 310
- getParameters method 451
- getParameterTypes method 310
- getProperties method 590, 591
- getProperty method
 - of Properties 589, 590
 - of System 84
- getProxyClass method 401, 402
- getQualifiedName method 731
- getRawType method (ParameterizedType) 509
- getRecordComponents method 310
- getResource, getResourceAsStream methods (Class) 302, 303
- getResourceAsStream method (Class, Module) 761
- getResourceBundle, getResourceBundleName methods (LogRecord) 451
- getReturnType method 310
- getSequenceNumber method 452
- getShort method 320
- getSimpleName method
 - of Element 731
- getSourceXxxName methods (LogRecord) 451
- getStackTrace method 424, 425
- getState method
 - of SwingWorker 702

- of Thread 608
- getSuperclass method 257, 499
- getSuppressed method 423, 425
- getTail method 445, 452
- Getters/setters, generated automatically 734
- getThrown method 451
- getTime method 231
- getType method
 - of Field 304
 - of RecordComponent 310
- getTypeParameters method
 - of Class 508
 - of Method 508
- getUncaughtExceptionHandler method 615
- getUpperBounds method (WildcardType) 508
- getValue method
 - of Map.Entry 556
- getYear method
 - of Date (deprecated) 138
 - of LocalDate 137, 142
- GMT (Greenwich Mean Time) 136
- Goetz, Brian 599, 660
- Gosling, James 8, 9
- goto keyword 85, 104, 776
- Graphical User Interface
 - debugging 412
 - long-running tasks in 694
- Green project 8, 9
- GregorianCalendar class 139
 - add method 139
 - constructors for 137, 171
- GUI. See Graphical User Interface

H

- H, h conversion characters 80
- handle method
 - of CompletableFuture 689
- Handler class
 - close method 450
 - flush method 450
 - get/setFilter methods 450
 - get/setFormatter methods 450
 - get/setLevel methods 450
 - publish method 450
- Hansen, Per Brinch 659, 660
- hash method
 - of Objects 248, 250
- Hash codes 246, 537
 - default 247
 - formatting output for 80
- Hash collisions 248, 538
- Hash maps 549
 - concurrent 674
 - identity 560
 - linked 557
 - setting 549
 - vs. tree maps 549
 - weak 557
- Hash sets 537
 - linked 557
- Hash tables 537, 538
 - legacy 587
 - load factor of 539
 - rehashing 539
- hashCode method 246, 720
 - equals method and 248, 249
 - null-safe 248
 - of Arrays 249, 250
 - of Boolean, Byte, Character, Double, Float, Integer, Long, Short 250
 - of LocalDate 248
 - of Object 250, 542
 - of Objects 248, 250
 - of proxy classes 401
 - of records 184, 249
 - of Set 524
 - of String 537
- HashMap class 549, 551
 - as a concrete collection type 525
- HashSet class 539, 541
 - add, contains methods 539
 - as a concrete collection type 525
 - iterating over 516
- Hashtable class 511, 586, 587, 684
 - as a concrete collection type 525
 - synchronized methods 587
- hasMoreElements method 516, 587
- hasNext method
 - of Iterator 515, 516, 521
 - of Scanner 78
- hasNextXxx methods (Scanner) 79
- hasPrevious method 530, 536
- "Has-a" relationship 130
- headMap method
 - of NavigableMap 573
 - of SortedMap 567, 573
- headSet method
 - of NavigableSet 567, 573
 - of SortedSet 567, 573
- Heap 547
- Helper methods 155, 339, 483
- Hexadecimal numbers
 - formatting output for 80
 - prefix for 37
- HexFormat class 80
- higher method
 - of NavigableSet 545
- Hoare, Tony 659
- Hold count 643
- HotJava browser 9
- Hotspot just-in-time compiler 16, 595
- HTML 10, 12
 - generating documentation in 733
 - in javadoc comments 210

- I**
- Identifiers 775
- Identity hash maps 560
- identityHashCode method 560, 562
- IdentityHashMap class 560
 - as a concrete collection type 525
- IEEE 754 specification 39, 50
- if keyword 86, 776
- IllegalAccessException class 312, 316
- IllegalStateException class 518, 521, 536, 546, 547, 668
- Immutable classes 156, 325
- Implementations 511
- implements keyword 329, 776
- Implicit parameters 151
 - none, in static methods 159
 - state of 453
- import keyword 189, 776
 - no annotations for 715
- InaccessibleObjectException class 312
- increment method
 - of LongAdder 663
- Increment operators 54
- Incremental linking 6
- incrementAndGet method 662
- incrementExact method 51
- Indentation, in text blocks 75
- Index class 112, 212
- indexOf method
 - of List 536
 - of String 67
- indexOfSubList method 582
- Inferred types 357
- info method
 - of ProcessHandle 709
- Information hiding. See Encapsulation
- Inheritance 130, 217
 - design hints for 324
 - equality testing and 242
 - hierarchies of 225
 - multiple 226, 337
 - preventing 231
 - private fields and 219
 - vs. type parameters 459, 474
- @Inherited annotation 721, 723
- inheritIO method 708
- initCause method 425
- Initialization blocks 176
 - static 178
- Inlining 6, 233
- Inner classes 375
 - accessing object state with 376
 - anonymous 385
 - applicability of 381
 - defined 375
 - local 382
 - private 378
 - static 376, 389
 - syntax of 380
 - translated into regular classes 381
 - vs. lambda expressions 359
- Input, reading 76
- insert method
 - of StringBuilder 73
- Instance fields 129
 - final 156
 - initializing 176, 215
 - explicit 174
 - names of 184
 - not present in interfaces 328, 336
 - private 147, 214
 - protected 324
 - public 147
 - shadowing 148, 175
 - values of 152, 153
 - volatile 660
 - vs. local variables 148, 151, 173
- instanceof keyword 59, 235, 245, 335, 776
 - annotating 715
 - pattern matching for 236
- Instances 128
 - creating on the fly 299
- Instrumentation API 743
- int type 36, 776
 - converting to other numeric types 51
 - fixed size for 5
 - platform-independent 37
- Integer class
 - compare method 334, 359
 - converting from int 265
 - hashCode method 250
 - intValue method 269
 - parseInt method 268, 269
 - toString method 269
 - valueOf method 269
- Integer types 36
 - arithmetic computations with 48
 - arrays of 252
 - computations of 51
 - converting from/to floating-point 234
 - formatting output for 80
 - no unsigned types in Java 38
- Integrated Development Environment (IDE) 24
- IntelliJ IDEA 24
- interface keyword 328, 717, 719, 776
- Interface types 514
- Interface variables 335
- Interfaces 327
 - abstract classes and 337
 - annotating 713, 715
 - binary- vs. source-compatible 340
 - callbacks and 342
 - constants in 336
 - declared inside a class 390
 - documentation comments for 207
 - evolution of 340
 - extending 335
 - for custom algorithms 584

- functional 358, 721, 722
- implementing 329, 335, 338
- methods in
 - clashes between 340
 - nonabstract 358
 - private 339
 - static 338
- no instance fields in 328, 336
- properties of 335
- public 207
- sealed 337
- tagging 350, 469, 523
- vs. implementations 511
- Internal errors 405, 408, 434
- Internationalization. See Localization
- Internet Explorer 7
- Interpreted languages 13
- Interpreter 6
- interrupt method
 - of Thread 609
- interrupted method
 - of Thread 611, 613
- InterruptedException class 601, 609, 618
- Intrinsic locks 652, 659, 661
- Introduction to Algorithms* (Cormen et al.) 542
- intValue method 269
- Invocation handlers 396
- InvocationHandler interface 396, 401, 402
- InvocationTargetException class 299
- invoke method
 - of InvocationHandler 396, 401, 402
 - of Method 320
- invokeAny/All methods (ExecutorService) 625, 630
- invokeDefault method 402
- IOException class 84, 407, 409, 413, 422
- isAbstract method 311
- isAlive method 705, 709
- isAnnotationPresent method 728
- isArray method 319
- isBlank method 67
- isCancelled, isDone methods (Future) 618, 620, 623
- isEmpty method
 - of Collection 339, 519, 520
 - of String 67
- isEnum method 309
- isFinal method 304, 311
- isInterface method
 - of Class 309
 - of Modifier 311
- isInterrupted method 609
- isJavaIdentifierXXX methods (Character) 44
- isLoggable method
 - of Filter 444
 - of System.Logger 450
- isLoggable method (Filter) 452
- isNaN method 39
- isNative method 311
- isNativeMethod method
 - of StackFrame 427
 - of StackTraceElement 427

- ISO 8601 format 722
- ISO 8859-1 41, 589
- isPrivate, isProtected, isPublic methods (Modifier) 304, 311
- isProxyClass method 401, 402
- isRecord method 309
- isStatic, isStrict, isSynchronized methods (Modifier) 311
- isVolatile method 311
- "Is-a" relationship 130, 226, 324
- Iterable interface 113
- iterator method 515
 - "for each" loop 515
 - forEachRemaining method 515, 521
 - generic 518
 - hasNext method 515, 516, 521
 - next method 515, 518, 521
 - of Collection 514, 520
 - of ServiceLoader 395
 - remove method 515, 517, 518, 521
- Iterators 515
 - being between elements 516
 - weakly consistent 675
- IzPack 203

J

- J#, J++ programming languages 7
- jar 200, 757
 - command-line options of 201, 202, 205
- Jar Bundler 203
- JAR files 197, 200
 - analyzing dependencies of 772, 773
 - creating 200
 - executable 202
 - file resources in 761
 - in jre/lib/ext directory 200
 - manifest of 201, 762
 - META-INF/services directory 770
 - modular 757, 763
 - multi-release 203
 - resources and 301
 - scanning for deprecated elements 721
- Java 19
 - add-exports option 765
 - add-opens option 765
 - illegal-access option 765
 - module, --module-path options 750
 - javaagent option 743
 - architecture-neutral object file format of 5
 - as a programming platform 1
 - available under GPL 13
 - backward compatibility of 203, 237, 373, 459
 - basic syntax of 31, 142
 - case-sensitiveness of 21, 31, 43, 587
 - command-line options of 205, 432
 - design of 2
 - documentation for 18
 - dynamic 7

- history of 8
- interpreter in 6
- libraries in 3, 10, 12
 - installing 18
- misconceptions about 12
- networking capabilities of 3
- no multiple inheritance in 337
- no operator overloading in 108
- no unsigned types in 38
- reliability of 4
- security of 4, 14
- simplicity of 2, 355
- strongly typed 36, 331
- versions of 10, 11
- vs. C++ 2, 595
- Java bug parade 32
- Java Collections Framework 511
 - algorithms in 574
 - converting to/from arrays in 583
 - copies and views in 562
 - interfaces in 521
 - vs. implementations 511
 - legacy classes in 586
 - operations in
 - bulk 582
 - optional 569
 - vs. traditional collections libraries 516
- Java Concurrency in Practice* (Goetz) 599
- Java Development Kit (JDK) 5, 15
 - documentation in 68
 - downloading 15
 - installation of 15
 - default 200
 - obsolete features in 747
 - setting up 16
- Java Language Specification 32
- Java Memory Model and Thread Specification 660
- Java Persistence Architecture 711
- Java Platform Module System 747, 774
 - migration to 762, 766
- Java Runtime Environment (JRE) 16
- Java Virtual Machine (JVM) 5
 - generics in 468, 500
 - launching 19
 - managing applications in 456
 - method tables in 229
 - thread priority levels in 616
 - truncating computations in 49
 - watching class loading in 455
- Java Virtual Machine Specification 32, 736
- java.awt package 748
- java.awt.Toolkit class 345
- java.desktop module 766, 767
- java.io.Console class 79
- java.io.PrintWriter class 85
- java.lang.annotation package 720
- java.lang.annotation.Annotation interface 720
- java.lang.Boolean class 250
- java.lang.Byte class 250
- java.lang.Character class 250
- java.lang.Class class 257, 300, 303, 309, 310, 316, 319, 508
- java.lang.ClassLoader class 436
- java.lang.Comparable interface 333
- java.lang.Double class 250, 334
- java.lang.Enum class 281
- java.lang.Exception class 426
- java.lang.Float class 250
- java.lang.Integer class 250, 269, 334
- java.lang.Long class 250
- java.lang.Object class 129, 250, 256, 542
- java.lang.ref.Cleaner class 182
- java.lang.reflect package 304, 316
- java.lang.reflect.AccessibleObject class 316
- java.lang.reflect.AnnotatedElement interface 728, 729
- java.lang.reflect.Array class 320
- java.lang.reflect.Constructor class 300, 310
- java.lang.reflect.Field class 310, 316
- java.lang.reflect.GenericArrayType interface 509
- java.lang.reflect.InvocationHandler interface 402
- java.lang.reflect.Method class 310, 324, 508
- java.lang.reflect.Modifier class 311
- java.lang.reflect.ParameterizedType interface 509
- java.lang.reflect.Proxy class 402
- java.lang.reflect.RecordComponent class 310
- java.lang.reflect.TypeVariable interface 508
- java.lang.reflect.WildcardType interface 508
- java.lang.RuntimeException class 426
- java.lang.Short class 250
- java.lang.StackTraceElement class 427
- java.lang.StackWalker class 426
- java.lang.StackWalker.StackFrame interface 426, 427
- java.lang.String class 67, 68
- java.lang.StringBuilder class 73
- java.lang.System class 79, 562, 591
- java.lang.Throwable class 300, 411, 425
- java.logging module 767
- java.math.BigDecimal class 110
- java.math.BigInteger class 110
- java.nio.file.Path interface 85
- java.se module 767
- java.text.NumberFormat class 270
- java.time.LocalDate class 142
- java.util.ArrayDeque class 547
- java.util.ArrayList class 260, 264
- java.util.Arrays class 119, 246, 250, 334, 572
- java.util.BitSet class 594
- java.util.Collection interface 520, 521, 582
- java.util.Collections class 571, 572, 577, 578, 579, 581, 582, 588, 685
- java.util.Comparator interface 578
- java.util.concurrent package 640
 - efficient collections in 674
- java.util.concurrent.ArrayBlockingQueue class 673
- java.util.concurrent.atomic package 662
- java.util.concurrent.BlockingDeque interface 674
- java.util.concurrent.BlockingQueue interface 673
- java.util.concurrent.Callable interface 620
- java.util.concurrent.ExecutorCompletionService class 630

- java.util.concurrent.Executors class 624
 - java.util.concurrent.ExecutorService interface 624, 630
 - java.util.concurrent.Future interface 620
 - java.util.concurrent.FutureTask class 620
 - java.util.concurrent.LinkedBlockingDeque class 673
 - java.util.concurrent.LinkedBlockingQueue class 673
 - java.util.concurrent.locks.Condition interface 649
 - java.util.concurrent.locks.Lock interface 643, 649
 - java.util.concurrent.locks.ReentrantLock class 643
 - java.util.concurrent.PriorityBlockingQueue class 673
 - java.util.concurrent.ThreadLocalRandom class 667
 - java.util.concurrent.TransferQueue interface 674
 - java.util.Deque interface 547
 - java.util.Enumeration interface 587
 - java.util.EnumMap class 562
 - java.util.EnumSet class 561
 - java.util.function package 359
 - java.util.HashMap class 551
 - java.util.HashSet class 541
 - java.util.IdentityHashMap class 562
 - java.util.Iterator interface 521
 - java.util.LinkedHashMap class 561
 - java.util.LinkedHashSet class 561
 - java.util.LinkedList class 536
 - java.util.List interface 535, 570, 572, 578, 582
 - java.util.ListIterator interface 536
 - java.util.logging package 436, 440
 - java.util.logging.ConsoleHandler class 451
 - java.util.logging.FileHandler class 451
 - java.util.logging.Filter interface 452
 - java.util.logging.Formatter class 452
 - java.util.logging.Handler class 450
 - java.util.logging.LogRecord class 451
 - java.util.Map interface 551, 554, 556, 571
 - java.util.NavigableMap interface 573
 - java.util.NavigableSet interface 545, 573
 - java.util.Objects class 164, 246
 - java.util.PriorityQueue class 548
 - java.util.Properties class 590
 - java.util.Queue interface 546
 - java.util.random.RandomGenerator interface 181
 - java.util.Scanner class 78, 79, 85
 - java.util.SequencedCollection interface 546
 - java.util.Set interface 570
 - java.util.SortedMap interface 552, 573
 - java.util.SortedSet interface 545, 573
 - java.util.Stack class 593
 - java.util.Timer class 343
 - java.util.TreeMap class 552
 - java.util.TreeSet class 545
 - java.util.WeakHashMap class 560
 - javac 19
 - processor option 729
 - XprintRounds option 733
 - current directory in 198
 - javadoc 206
 - command-line options of 214
 - comments in 207, 209
 - extracting 213
 - overview 214
 - redeclaring Object methods for 358
 - HTML markup in 210
 - including annotations in 723
 - links in 211, 212
 - online documentation of 214
 - JavaFX 695
 - javafx.css.CssParser class 203
 - javan.log files 442
 - javap 204, 381
 - JavaScript 14
 - javax.annotation package 720
 - javax.swing.JOptionPane class 344
 - javax.swing.SwingWorker class 701
 - javax.swing.Timer class 343, 345
 - JAXB 759
 - JCommander 711
 - jconsole 441, 456, 649, 650
 - jdepscan 721
 - jdeps 772, 773
 - JEP 264 (platform logging API) 437
 - jimage 774
 - jlink 773
 - jmod 774
 - JMOD files 774
 - Jmol applet 7
 - join method
 - of String 68
 - of Thread 606, 608
 - JOptionPane class
 - showMessageDialog method 344
 - JShell 6, 25
 - JShell, loading modules into 759
 - JSlider class
 - setLabelTable method 472
 - JSON 282
 - JSON-B 759, 761
 - JUnit 711, 712
 - JUnit framework 453
 - Just-in-time compiler 5, 6, 13, 152, 233, 595
 - JVM
 - specification for 736
- ## K
- Key/value pairs
 - in annotations 712, 719
 - keySet method
 - of ConcurrentHashMap 682
 - of Map 554, 556
 - Keywords 775
 - hyphenated 285
 - not used 47
 - redundant 336
 - reserved 31, 44
 - restricted 775
 - Knuth, Donald 104
 - KOI-8 41

L

- L, l suffixes (for long integers) 37
- Lambda expressions 354
 - accessing variables in 366
 - annotating targets for 722
 - atomic updates with 663
 - capturing values by 367
 - for loggers 439
 - functional interfaces and 358
 - method references and 361
 - not for variables of type `Object` 359
 - parameter types of 356
 - processing 369
 - result type of 357
 - scope of 368
 - syntax of 355
 - this keyword in 369
 - vs. inner classes 359
 - vs. method references 364
- Langer, Angelika 509
- Language model API 730
- last method
 - of `SortedSet` 545
- lastIndexOf method
 - of `List` 536
 - of `String` 67
- lastIndexOfSubList method 582
- lastKey method 552
- Launch4J 203
- Legacy classes 184
 - generics and 472
- Legacy collections 586
 - bit sets 593
 - enumerations 587
 - hash tables 587
 - property maps 588
 - stacks 593
- length method
 - of arrays 113
 - of `BitSet` 594
 - of `String` 62, 66, 67
 - of `StringBuilder` 73
- Line feed character
 - escape sequence for 40
 - in output 35, 74
 - in text blocks 73
- @link annotation 211
- Linked hash maps/sets 557
- Linked lists 526
 - concurrent modifications of 532
 - doubly linked 526
 - printing 534
 - random access in 533, 574
 - removing elements from 526
- `LinkedBlockingDeque` class 673
- `LinkedBlockingQueue` class 669, 673
- `LinkedHashMap` class 557, 561
 - access vs. insertion order in 558
 - as a concrete collection type 525
 - `removeEldestEntry` method 559, 561
- `LinkedHashSet` class 557, 561
 - as a concrete collection type 525
- `LinkedList` class 526, 532, 545
 - `addFirst/Last` methods 536
 - as a concrete collection type 525
 - `get` method 533
 - `getFirst/Last` methods 536
 - `listIterator` method 530
 - `next/previousIndex` methods 533
 - `removeAll` method 534
 - `removeFirst/Last` methods 537
- Linux
 - IDEs for 24
 - JDK in 15
 - no thread priorities in OpenJDK VM for 616
 - paths in 197, 199
 - troubleshooting Java programs in 21
- `List` class 523
 - `add` method 523, 535
 - `addAll` method 535
 - `copyOf` method 565, 570
 - `get` method 523, 536
 - `indexOf`, `lastIndexOf` methods 536
 - `listIterator` method 535
 - of `Collections` 588
 - of method 562, 570, 584
 - `remove` method 523, 535
 - `replaceAll` method 582
 - `set` method 523, 536
 - `sort` method 578
 - `subList` method 566, 572
- `ListIterator` interface 532
 - `add` method 529, 530, 536
 - `hasPrevious` method 530, 536
 - `next/previousIndex` methods 536
 - of `LinkedList` 530
 - of `List` 535
 - `previous` method 530, 536
 - `remove` method 530
 - `set` method 531, 536
- Lists 523
 - modifiable/resizable 577
 - unmodifiable 570
 - with given elements 562
- load method
 - of `Properties` 589, 590
 - of `ServiceLoader` 395
- Load time 743
- Local inner classes 382
 - accessing variables from outer methods in 383
- Local variables
 - annotating 473, 713, 714
 - vs. instance fields 148, 151, 173
- `LocalDate` class 136
 - `getXxx` methods 137, 142
 - `hashCode` method 248
 - `minusDays` method 142
 - `now`, of methods 137, 142

- plusDays method 137, 142
 - processing arrays of 481
- Locales 82
- Localization 133, 302
- Lock interface 652
 - await method 645
 - lock method 643
 - newCondition method 645, 649
 - signal method 646
 - signalAll method 645
 - tryLock method 606
 - unlock method 641, 643
 - vs. synchronization methods 654
- Locks 640
 - client-side 658
 - condition objects for 644
 - deadlocks 646, 649
 - fair 644
 - hold count for 643
 - in synchronized blocks 657
 - intrinsic 652, 659, 661
 - not with try-with-resources statement 641
 - not wrapper objects for 267
 - reentrant 643
- Log file pattern variables 444
- Log handlers 441
 - filtering/formatting 444
- Log messages, adding to classes 736
- log, log10 methods (Math) 50
- Log4j 436
- Logback 436
- @LogEntry annotation 736
- Logger class
 - getGlobal method 453
- Logger interface (System) 437
 - getName method 450
 - isLoggable method 450
 - log method 437, 450
- Loggers
 - filtering/formatting 444
 - hierarchy of 441
 - naming 437
- Logging 436
 - configuring 440, 441
 - including class names in 387
 - levels of 438, 441
 - messages for 252
 - recipe for 445
- Logging proxy 453
- Logical “and”, “or” 54
- Logical conditions 43
- LogRecord class, methods of 451
- long type 36, 776
 - converting from long 265
 - hashCode method 250
 - platform-independent 37
- Long Term Support (LTS) 16
- LongAccumulator class, methods of 664
- LongAdder class 663, 677
 - add, increment, sum methods 663
- Loops
 - break statements in 104
 - continue statements in 106, 107
 - determinate (for) 95
 - “for each” 113
 - while 90
- lower method
 - of NavigableSet 545
- M**
- Mac OS X
 - executing JARs in 203
 - IDEs for 24
 - JDK in 15
- main ,method
 - declared public 32
- main method 161
 - body of 33
 - declared static void 33, 34
 - not defined 142, 179
 - separate for each class 453
 - String[] args parameter of 116
 - tagged with throws 84
- make tool 146
- MANIFEST.MF 201
 - editing 202
 - newline characters in 203
- Map interface 522
 - compute, computeIfXxx methods 554
 - containsKey/Value methods 551
 - copyOf method 565, 571
 - entry method 564, 571
 - entrySet method 554, 556
 - forEach method 551
 - get method 522, 549, 551
 - getOrDefault method 551
 - keySet method 554, 556
 - merge method 554
 - of method 563, 564, 571
 - ofEntries method 564, 571
 - put method 522, 549, 551
 - putAll method 551
 - putIfAbsent method 554
 - remove method 550
 - replaceAll method 554
 - values method 554, 556
- Map.Entry interface 554, 556
 - copyOf, getKey, get/setValue methods 556
- mappingCount method 674
- Maps 548
 - adding/retrieving objects to/from 549
 - concurrent 674
 - garbage collecting 557
 - hash vs. tree 549
 - implementations for 549
 - keys for 550
 - enumerating 555
 - subranges of 567

- unmodifiable 571
- with given key/value pairs 562
- Marker interfaces 350
- Math class 28, 49
 - E, PI static constants 50, 158
 - floorMod method 48
 - log, log10 methods 50
 - pow method 49, 159
 - round method 52
 - sqrt method 49, 321, 322
 - trigonometric functions 49
 - xxxExact methods 51
- max method
 - of Collections 581
- Maximum value, computing 463
- merge method
 - of ConcurrentHashMap 677, 678
 - of Map 554
- Merge sort algorithm 576
- Meta-annotations 717, 724
- META-INF 202
- META-INF/versions directory 203
- Method class 304
 - getDeclaringClass method 310
 - getGenericXxx methods 508
 - getModifiers, getName methods 304, 310
 - getReturnType method 310
 - getTypeParameters method 508
 - getXxxTypes methods 310
 - invoke method 320
 - toString method 304
- Method parameters. See Parameters
- Method pointers 320, 321, 322
- Method references 361
 - annotating 715
 - this, super parameters in 365
 - vs. lambda expressions 364
- Method tables 229
- MethodHandles class 762
- Methods 129
 - abstract 272
 - in functional interfaces 358
 - accessor 138, 152, 153, 478
 - adding logging messages to 736
 - adding, in subclasses 221
 - annotating 713
 - applying to objects 133
 - asynchronous 618
 - body of 33, 34
 - bridge 471, 472, 496
 - calling by reference vs. by value 164
 - casting 234
 - chaining calls of 373
 - concrete 273
 - conflicts in 340
 - consistent 242
 - default 339
 - deprecated 138, 720, 721
 - destructor 182
 - documentation comments for 207, 211
 - dynamic binding for 223, 228
 - error checking in 153
 - exception specification in 407
 - factory 160
 - final 229, 233, 304, 335
 - generic 464, 470, 518
 - getters/setters, generated automatically 734
 - helper 155, 483
 - inlining 6, 233
 - invoking 35, 320
 - mutator 138, 153, 478
 - names of 184, 216
 - overloading 172
 - overriding 219, 246, 325, 720, 721
 - exceptions and 409
 - return type and 470
 - package scope of 195
 - passing objects to 133
 - private 155, 229, 304, 339
 - protected 207, 239, 324, 351
 - public 207, 304, 329
 - reflexive 242
 - return type of 172, 229
 - signature of 172, 229
 - static 159, 191, 229, 492, 654
 - adding to interfaces 338
 - symmetric 242
 - tracing 397
 - transitive 242
 - used for serialization 720, 722
 - utility 338
 - varargs 270, 487
 - visibility of, in subclasses 230
- Micro Edition 10, 16
- Microsoft
 - ActiveX 4
 - C# 7, 10, 233
 - Internet Explorer 7
 - J#, J++ 7
 - JDK in 15
 - .NET platform 5
 - Visual Basic 2, 133
 - Visual Studio 19
- Microsoft Windows. See Windows operating system
- min method
 - of Collections 581
- Minimum value, computing 463
- minusDays method 142
- mod method
 - of BigDecimal 110
 - of BigInteger 110
- Modifier class
 - isXxx methods 304, 311
 - toString method 311
- module keyword 750, 777
- Module class
 - getResourceAsStream method 761
- Module path 200
- module-info.class 758, 762
- module-info.java 749, 763

- Modules 10, 197, 747, 774
 - accessing 759, 765, 766
 - automatic 762, 765
 - declaration of 750, 751
 - explicit 764
 - exporting packages 753
 - loading into JShell 759
 - migration to 762, 766
 - naming 748, 762
 - not passing access rights 753
 - open 761
 - opening packages in 760
 - packages with the same names in 756
 - qualified exports of 768
 - requiring 751
 - service implementations and 770
 - tools for 772
 - unnamed 312, 764
 - versioning 748, 751
- Modulus 48
- Monitor concept 659
- Mosaic 9
- Multi-release JARs 203
- Multidimensional arrays 119, 124
 - printing 252
 - ragged 122
- Multiple inheritance 337
 - not supported in Java 226
- Multiple selections 99
- Multiplication 48
- multiply method
 - of BigDecimal 110
 - of BigInteger 110
- multiplyExact method 51
- Multitasking 599
- Multithreading 6, 599
 - deadlocks in 646, 649
 - deferred execution in 369
 - performance and 643, 663, 669
 - preemptive vs. cooperative scheduling for 605
 - synchronization in 635
 - using pools for 621
- Mutator methods 138, 478
 - error checking in 153

N

- n conversion character 80
- NaN 39
- native keyword 776
- naturalOrder method 374
- Naughton, Patrick 8, 9
- NavigableMap interface 524
 - headMap, subMap, tailMap methods 573
- NavigableSet interface 524, 543
 - ceiling, floor methods 545
 - headSet, subSet, tailSet methods 567, 573
 - higher, lower methods 545
 - pollFirst/Last methods 545
- nCopies method 564, 572
- negateExact method 51
- Negation operator 54
- Negative infinity 39
- Nested classes
 - annotating 715
- .NET platform 5
- NetBeans 19, 24, 452
- Netscape 9
 - LiveScript/JavaScript 14
 - Navigator browser 7
- Networking 3
- new keyword 59, 133, 147, 776
 - in constructor references 365
 - not for interfaces 335
 - return value of 135
 - with arrays 111
 - with generic classes 258
 - with threads 605
- newCachedThreadPool method 621, 624
- newCondition method 645, 649
- newFixedThreadPool method 621, 624
- newInstance method
 - of Array 317, 320
 - of Class 299, 499
 - of Constructor 300
- newKeySet method 682
- Newline. See Line feed character
- newProxyInstance method 396, 401, 402
- newScheduledThreadPool method 621
- newSingleThreadXxx methods (Executors) 621, 624
- next method
 - of Iterator 515, 518, 521
 - of Scanner 78
- nextDouble method 76, 78
- nextElement method 516, 587
- nextIndex method
 - of LinkedList 533
 - of ListIterator 536
- nextInt method
 - of RandomGenerator 179, 181
 - of Scanner 76, 78
- nextLine method 76, 78
- No-argument constructors 173, 221, 394
- NoClassDefFoundError class 21
- non-sealed keyword 285, 776
- noneOf method 561
- @NonNull annotation 714
- NoSuchElementException class 515, 521, 536, 546, 547
- Notepad 21
- notify, notifyAll methods (Objects) 656
- notify, notifyAll methods (of Object) 653
- now method
 - of LocalDate 137, 142
- null literal 135, 776
 - as a reference 149
 - equality testing to 242
- nullFirst/Last methods (Comparator) 374
- NullPointerException class 57, 149, 150, 164, 267, 364, 406, 430

- Number class 265
- NumberFormat class
 - factory methods 160
 - parse method 270
- NumberFormatException class 429
- Numbers
 - floating-point 38, 48, 52, 80, 97, 234
 - generated random 667
 - hexadecimal 37, 80
 - octal 37, 80
 - prime 594
 - rounding 39, 52, 110
 - unsigned 38
- Numeric types
 - casting 52
 - comparing 54, 374
 - converting
 - to other numeric types 51, 234
 - to strings 268
 - default initialization of 172
 - fixed sizes for 5
 - precision of 79, 107
 - printing 79

O

- o conversion character 80
- Oak 9, 406
- Object class 129, 240, 656
 - clone method 154, 347, 358
 - equals method 241, 246, 256, 342, 566
 - getClass method 256
 - hashCode method 247, 250, 542
 - no redefining for methods of 342
 - notify, notifyAll methods 653, 656
 - toString method 250, 342, 358
 - wait method 606, 653, 657
- Object references
 - as method parameters 165
 - converting 234
 - default initialization of 172
 - modifying 165
- Object traversal algorithms 560
- Object variables 273
 - in predefined classes 132
 - initializing 134
 - setting to null 135
 - vs. C++ object pointers 135
 - vs. objects 134
- Object-oriented programming (OOP) 3, 127, 217
 - passing objects in 342
 - time measurement in 137
 - vs. procedural 127
- Object-relational mappers 759
- Objects 127, 130
 - analyzing at runtime 311
 - applying methods to 133
 - behavior of 129
 - cloning 347
 - comparing 335
 - concatenating with strings 251, 252
 - constructing 128, 171
 - default hash codes of 247
 - destruction of 182
 - equality testing for 241, 246, 298
 - finalize method of 182
 - identity of 129
 - implementing an interface 335
 - in predefined classes 132
 - initializing 133
 - intrinsic locks of 652
 - passing to methods 133
 - references to 134
 - runtime type identification of 297
 - serializing 560
 - sorting 329
 - state of 129, 376
 - vs. object variables 134
- Objects class
 - checkXxx methods 430
 - hash, hashCode methods 248, 250
 - requireNonNull method 150, 164, 430
 - requireNonNullElse method 150, 164
- Octal numbers
 - formatting output for 80
 - prefix for 37
- of method
 - of EnumSet 561
 - of List, Map, Set 562, 570, 584
 - of LocalDate 137, 142
 - of Path 83, 84, 85, 338
 - of ProcessHandle 706, 709
 - of RandomGenerator 181
- ofEntries method 571
 - of Map 564
- offer method
 - of BlockingQueue 668, 669, 673
 - of Queue 546
- offerFirst/Last methods
 - of BlockingDeque 674
 - of Deque 547
- On-demand initialization 664
- onExit method 709
- Online documentation 66, 68, 206, 214
- open keyword 761, 777
- OpenJ9 just-in-time compiler 16
- OpenJDK 15, 16
- opens keyword 760, 769, 777
- Operators
 - arithmetic 48
 - bitwise 57, 59
 - boolean 54
 - hierarchy of 58
 - increment/decrement 54
 - no overloading for 108
 - relational 54
- Optional operations 569
- or method (BitSet) 594
- Oracle 10

- Ordered collections 523, 529
- ordinal method
 - of Enum 282
- org.omg.corba package 747
- orTimeout method 689
- OSGi platform 393
- Out-of-bounds exceptions 430
- Output
 - formatting 79
 - statements in 60
- Overloading resolution 172, 228
- @Override annotation 246, 720, 721
- overview.html file 214

P

- p (hexadecimal floating-point literals) 39
- package keyword 189, 192, 777
- package-info.java 209, 714
- package.html file 209
- Packages 188, 747
 - accessing 195
 - adding classes into 192
 - annotating 713, 714
 - documentation comments for 207, 209
 - exporting 753
 - hidden 756
 - importing 189
 - names of 188, 297
 - opening 760
 - split 758
 - unnamed 192, 195, 214, 433
- Parallelism threshold 680
- parallelXxx methods (Arrays) 682
- Parameter variables
 - annotating 713
- Parameterized types. See Type parameters
- ParameterizedType interface 500, 501
 - getXxx methods 509
- Parameters 164
 - checking, with assertions 434
 - documentation comments for 208
 - explicit 151
 - implicit 151, 159, 453
 - modifying 165, 168
 - names of 175
 - using collection interfaces in 585
 - variable number of
 - passing generic types to 487
- Parent classes. See Superclasses
- parse method
 - of NumberFormat 270
- parseInt method 268, 269
- Pascal 8
 - compiled code in 5
 - passing parameters in 167
- Passwords
 - reading from console 78, 79
- Path interface, of method 83, 84, 85, 338
- Paths class, get method 338
- Pattern matching 236
- Payne, Jonathan 9
- peek method
 - of BlockingQueue 668, 669
 - of Queue 546
 - of Stack 593
- peekFirst/Last methods (Deque) 547
- Performance 6
 - computations and 49, 50
 - JAR files and 197
 - measuring 594, 597
 - multithreading and 643, 663, 669
 - of collections 523, 539, 675
 - of Java vs. C++ 595
 - of simple tests vs. catching exceptions 428
- permits keyword 283, 337, 777
- @Persistent annotation 723
- Physical limitations 404
- PI
 - constant (Math) 50, 158
- Picocli 711
- pid method
 - of ProcessHandle 709
- Platform logging API 437, 440
- plusDays method 137, 142
- Point class 182, 184
- poll method
 - of BlockingQueue 668, 669, 673
 - of ExecutorCompletionService 630
 - of Queue 546
- pollFirst/Last methods
 - of Deque 547, 674
 - of NavigableSet 545
- Polymorphism 223, 226, 284, 326
- pop method
 - of Stack 593
- Portability 5, 12, 48
- Positive infinity 39
- pow method
 - of Math 49, 159
- Precision, of numbers 79
- Preconditions 435
- Predefined classes 132
 - mutator and accessor methods in 138
 - objects, object variables in 132
- Predicate interface 360, 370
- Preemptive scheduling 605
- remain method (Instrumentation API) 743
- previous method
 - of ListIterator 530, 536
- previousIndex method
 - of LinkedList 533
 - of ListIterator 536
- Prime numbers 594
- Primitive types 36
 - as method parameters 165
 - comparing 374
 - converting to objects 265
 - final fields of 156

- not for type parameters 485
 - transforming hash map values to 681
 - values of, not object 240
 - Princeton University 4
 - print method
 - of System.out 35, 79
 - printf method
 - arguments of 270
 - conversion characters for 80
 - flags for 81
 - of System.out 79, 82
 - println method
 - of System.out 35, 76
 - printStackTrace method 300, 423, 454
 - PrintStream class
 - print method 436
 - PrintWriter class 83, 85
 - Priority queues 547
 - PriorityBlockingQueue class 669, 673
 - PriorityQueue class 548
 - as a concrete collection type 525
 - private keyword 146, 195, 378, 777
 - checking 304
 - for fields, in superclasses 220
 - for methods 155
 - Procedures 127
 - process method 702, 708, 709
 - destroy, destroyForcibly methods 705, 709
 - exitValue method 705, 709
 - getXxxStream methods 703, 704, 708
 - isActive method 705, 709
 - of SwingWorker 696, 698, 702
 - onExit method 709
 - supportsNormalTermination method 709
 - toHandle method 706, 709
 - waitFor method 705, 709
 - ProcessBuilder class 702, 708
 - directory method 702, 708
 - environment method 708
 - inheritIO method 708
 - redirectXxx methods 703, 708
 - start method 704, 708
 - startPipeline method 704, 708
 - Processes 702
 - building 702
 - killing 705
 - running 704
 - vs. threads 599
 - ProcessHandle interface 709
 - allProcesses method 706, 709
 - children, descendants methods 706, 709
 - current method 706, 709
 - info method 709
 - of method 706, 709
 - pid method 709
 - ProcessHandle.Info interface 710
 - Processor interface 729
 - Producer threads 668
 - Programs. See Applications
 - Properties class 586
 - getProperty method 589, 590
 - load method 589, 590
 - setProperty method 590
 - store method 589, 590
 - stringPropertyNames method 590
 - @Property annotation 734
 - Property files
 - generating 733
 - Property maps 588
 - reading/writing 589
 - protected keyword 239, 324, 351, 777
 - provides keyword 771, 777
 - Proxies 395
 - properties of 400
 - purposes of 397
 - Proxy class 400
 - get/isProxyClass methods 401, 402
 - newProxyInstance method 396, 401, 402
 - public keyword 31, 47, 144, 147, 195, 329, 777
 - checking 304
 - for fields in interfaces 336
 - for main method 32
 - for only one class in source file 144
 - not specified for interfaces 328
 - publish method 702
 - of Handler 450
 - of SwingWorker 696
 - Pure virtual functions (C++) 274
 - push method
 - of Stack 593
 - put method
 - of BlockingQueue 668, 669, 673
 - of ConcurrentHashMap 676
 - of Map 522, 549, 551
 - putAll method 551
 - putFirst/Last methods (BlockingDeque) 674
 - putIfAbsent method
 - of ConcurrentHashMap 677
 - of Map 554
- ## Q
- Qualified exports 768
 - Queue class 545
 - implementing 512
 - methods of 546
 - Queues 511, 545
 - blocking 668
 - concurrent 674
 - QuickSort algorithm 117, 576
- ## R
- Race conditions 636, 640
 - and atomic operations 662
 - Ragged arrays 122
 - Random class
 - thread-safe 667
 - RandomAccess interface 523, 577, 579

- RandomGenerator interface
 - nextInt method 179, 181
 - of method 181
- range method
 - of EnumSet 561
- Raw types 468
 - converting type parameters to 475
 - type inquiring at runtime 485
- readLine/Password methods (Console) 79
- Receiver parameter 716
- record keyword 777
- RecordComponent class, getXxx methods 310
- Records 182, 219
 - adding methods to 184
 - always final 234
 - declared inside a class 390
 - equals method of 242
 - hashCode method of 249
 - implementing interfaces 336
 - instance fields of 183, 184
 - toString method of 253
- Rectangle class 543
- Rectangles
 - comparing 543
- Recursive computations 633
- RecursiveAction, RecursiveTask classes 633
- Red Hat 15
- Red-black trees 542
- redirectXxx methods (ProcessBuilder) 703, 708
- reduce, reduceXxx methods (ConcurrentHashMap) 679, 681
- Reentrant locks 643
- ReentrantLock class 640
- Reflection 217, 296
 - accessing
 - private members 759, 766
 - accessing nonpublic features with 312
 - analyzing
 - classes 304
 - objects, at runtime 311
 - generics and 316, 498
 - overusing 326
 - processing annotations with 725
- Reinhold, Mark 10
- Relational operators 54, 59
- Relative resource names 302
- remove method
 - of ArrayList 262, 264
 - of BlockingQueue 668, 669
 - of Collection 519, 521
 - of Iterator 515, 517, 518, 521
 - of List 523, 535
 - of ListIterator 530
 - of Map 550
 - of Queue 546
 - of ThreadLocal 632
- removeAll method
 - of Collection 519, 521
 - of LinkedList 534
- removeEldestEntry method 559, 561
- removeFirst/Last methods
 - of LinkedList 537
 - of SequencedCollection 546
- removeIf method
 - of ArrayList 360
 - of Collection 521, 582
- repeat method
 - of String 61, 68
- @Repeatable annotation 721, 723, 724
- @RepeatedTest annotation 713
- REPL 25
- replace method
 - of ConcurrentHashMap 676
 - of String 67
- replaceAll method
 - of Collections 582
 - of List 582
 - of Map 554
- requireNonNull method 150, 164, 430
- requireNonNullElse method 150, 164
- requires keyword 751, 753, 755, 756, 762, 766
- Reserved words. See Keywords
- Resources 301
 - exhaustion of 405
 - in JAR files 761
 - localizing 302
 - names of 302
- Restricted views 569
- resume method
 - of Thread (deprecated) 608
- retain method
 - of Collection 519
- retainAll method 521
- @Retention annotation 717, 721
- return keyword 208, 777
 - in finally blocks 420
 - in lambda expressions 356
 - not allowed in switch expressions 104
- Return types 229
 - covariant 472
 - documentation comments for 208
 - for overridden methods 470
- Return values 135
- reverse method
 - of Collections 582
- reversed, reverseOrder methods (Comparator) 375, 576, 578
- rotate method
 - of Collections 582
- round method
 - of Math 52
- RoundEnvironment interface 730
- RoundingMode enumeration 110
- rt.jar file
 - no longer present 774
- run method
 - of Thread 601, 604
- runAfterXxx methods (CompletableFuture) 690
- runFinalizersOnExit method 182
- Runnable interface 370, 599, 605
 - lambda expressions and 358

- run method 369, 605
- Runtime class
 - adding shutdown hooks at 182
 - analyzing objects at 311
 - creating classes at 396
 - exec method 702
 - setting the size of an array at 257
 - type identification at 235, 297, 485
- Runtime image file 774
- RuntimeException class 406, 426, 429

S

- S, s conversion characters 80, 81
- @SafeVarargs annotation 487, 721, 722
- Scala programming language 340
- Scanner class 76, 83, 85
 - hasNext method 78
 - hasNextXxx methods 79
 - next method 78
 - nextXxx methods 76, 78
- sealed keyword 283, 337, 777
- search, searchXxx methods (ConcurrentHashMap) 679, 681
- Security class 4, 14
- @see annotation 210, 211
- SequencedCollection interface
 - methods of 546
- @Serial annotation 720, 722
- Serialization 560
- Service loaders 393, 769
- ServiceLoader class 393, 395, 769
 - iterator, load methods 395
 - stream method 394, 395
- ServiceLoader.Provider interface 395
- ServiceLoader.Provider interface, methods of 394, 395
- Services 393
- ServletException 417
- Servlets 417
- set method
 - add, equals, hashCode, methods of 524
 - copyOf method 565, 570
 - of Array 320
 - of ArrayList 260, 264
 - of BitSet 594
 - of Field 316
 - of List 523, 536
 - of ListIterator 531, 536
 - of method 562, 570
 - of ThreadLocal 632
 - of Vector 658
- setAccessible method 312, 316
- setBoolean method 320
- setByte, setChar methods (Array) 320
- setClassAssertionStatus method 436
- setDaemon method 613
- setDefaultAssertionStatus method 436
- setDefaultUncaughtExceptionHandler method 454, 614, 615
- setDouble method 320
- setFilter method
 - of Handler 450
- setFloat method 320
- setFormatter method 450
- setInt method 320
- setLabelTable method 472
- setLevel method
 - of Handler 450
- setLong method 320
- setOut method 159
- setPackageAssertionStatus method 436
- setPriority method 616
- setProperty method 440
 - of Properties 590
- Sets 539
 - concurrent 674
 - intersecting 583
 - mutating elements of 540
 - subranges of 567
 - thread-safe 682
 - unmodifiable 570
 - with given elements 562
- setShort method 320
- setTime method 231
- setUncaughtExceptionHandler method 615
- setValue method 556
- Shallow copies 347, 350
- Shell
 - redirection syntax of 85
 - scripts for, generating 733
 - scripts in 199
- Shift operators 57
- short type 36, 777
 - converting from short 265
 - hashCode method 250
- showMessageDialog method 344
- shuffle method
 - of Collections 577, 578
- Shuffling 577
- shutdown method
 - of ExecutorService 623, 624
- Shutdown hooks 182
- shutdownNow method 623
- Sieve of Eratosthenes benchmark 594, 597
- signal method
 - of Condition 646, 649, 650
- signalAll method 645, 649, 650
- Signatures (of methods) 172, 229
- Signatures. See Digital signatures
- sin method
 - of Math 49
- size method
 - of ArrayList 259, 260
 - of BitSet 594
 - of Collection 519, 520
 - of concurrent collections 674
- sleep method
 - of Thread 601, 604, 610
- SLF4J 437
- Smart cards 3

- SOAP 748
- SocketHandler class 442
- sort method
 - of Arrays 117, 119, 329, 332, 334, 355, 359
 - of Collections 576
 - of List 578
- SortedMap interface 524
 - comparator, first/lastKey methods 552
 - headMap, subMap, tailMap methods 567, 573
- SortedSet interface 524
 - comparator, first, last methods 545
 - headSet, subSet, tailSet methods 567, 573
- Sorting
 - algorithms for 117, 576
 - arrays 117, 332
 - assertions for 434
 - order of 576
 - people, by name 374
 - strings by length 345, 354, 356
- Source code, generating 721, 722, 731
- Source files 199
 - editing in Eclipse 27
 - installing 18
- Space. See Whitespace
- Special characters 40
- Split packages 758
- sqrt method
 - of BigInteger 110
 - of Math 49, 321, 322
- src.zip file 18
- Stack class 511, 586, 593
 - methods of 593
- Stack trace 423, 649
 - no displaying to users 431
- StackFrame
 - getXxx methods 426
 - isNativeMethod method 427
 - toString method 424, 427
- Stacks 593
- StackTraceElement class, methods of 427
- StackWalker class 423
 - forEach method 426
 - getInstance method 423, 426
 - walk method 423, 426
- Standard Edition 10, 16
- Standard Java library
 - companion classes in 338
 - online API documentation for 66, 68, 206, 214
- Standard Template Library (STL) 511, 516
- start method
 - of ProcessBuilder 704, 708
 - of Thread 601, 604, 605
 - of Timer 345
- Starting directory, for a launched program 84
- startInstant method
 - of ProcessHandle.Info 710
- startPipeline method 704, 708
- startsWith method 67
- Statements 34
 - conditional 86
 - in output 60
- static keyword 46, 157, 768, 777
 - for fields in interfaces 336
 - for main method 33, 34
- Static binding 229
- Static constants 158
 - documentation comments for 209
- Static fields 157
 - accessing, in static methods 159
 - importing 191
 - initializing 178
 - no type variables in 492
- Static imports 191
- Static methods 159
 - accessing static fields in 159
 - adding to interfaces 338
 - importing 191
 - no type variables in 492
- Static nested classes 376, 389
- Static variables 158
- stop method
 - of Thread (deprecated) 608, 609
 - of Timer 345
- store method
 - of Properties 588, 590
- stream method
 - of BitSet 594
 - of Collection 340
 - of ServiceLoader 394, 395
- Stream interface, toArray method 366
- strictfp keyword 777
- StrictMath class 49, 50
- String class 59
 - charAt method 62, 67
 - compareTo method 67
 - endsWith method 67
 - equals, equalsIgnoreCase methods 65, 67
 - format, formatted, formatTo methods 82
 - hashCode method 246, 537
 - immutability of 63, 156, 231
 - implementing CharSequence 338
 - indexOf method 67, 172
 - isBlank, isEmpty methods 67
 - join method 68
 - lastIndexOf method 67
 - length method 62, 66, 67
 - repeat method 61, 68
 - replace method 67
 - startsWith method 67
 - strip method 68
 - stripLeading/Trailing methods 68
 - substring method 63, 68, 566
 - toLowerCase, toUpperCase methods 68
 - transform method 373
 - trim method 68
- StringBuffer class 73
- StringBuilder class 70
 - append method 71, 73
 - appendCodePoint method 73
 - delete method 73

- implementing `CharSequence` 338
- insert method 73
- length method 73
- toString method 72, 73
- stringPropertyName method 590
- Strings 59
 - building 70
 - code points/code units of 62
 - comparing 345
 - concatenating 60
 - with objects 251, 252
 - converting to numbers 268
 - empty 66, 67
 - equality of 65
 - formatting output for 79
 - immutability of 63
 - length of 63, 66
 - null 66
 - shared, in compiler 64, 65
 - sorting by length 345, 354, 356
 - spanning multiple lines 73
 - substrings of 63
 - using ". . ." for 35
- strip method
 - of `String` 68
- stripLeading/Trailing methods (`String`) 68
- Strongly typed languages 36, 331
- Subclasses 217
 - adding fields/methods to 221
 - anonymous 387, 460
 - cloning 351
 - comparing objects from 335
 - constructors for 221
 - defining 218
 - forbidding 282
 - inheriting annotations 721
 - method visibility in 230
 - no access to private fields of superclass 239
 - non-sealed 286
 - overriding superclass methods in 221
- subList method (`List`) 566, 572
- subMap method
 - of `NavigableMap` 573
 - of `SortedMap` 567, 573
- submit method
 - of `ExecutorCompletionService` 630
 - of `ExecutorService` 622, 624
- Subranges 566
- subSet method
 - of `NavigableSet` 567, 573
 - of `SortedSet` 567, 573
- Substitution principle 226
- substring method
 - of `String` 63, 68, 566
- subtract method
 - of `BigDecimal` 110
 - of `BigInteger` 110
- subtractExact method 51
- Subtraction 48
- sum method
 - of `LongAdder` 663
- Sun Microsystems 1, 4, 10, 13
 - HotJava browser 9
- super keyword 220, 479, 778
 - in method references 365
 - vs. `this` 221
- Superclass wins rule 340
- Superclasses 217
 - accessing private fields of 220
 - annotating 715
 - common fields and methods in 273, 324
 - overriding methods of 246
 - throws specifiers in 409, 414
- Supertype bounds 478
- Supplier interface 370
- supportsNormalTermination method 709
- @SuppressWarnings annotation 102, 265, 473, 487, 493, 721, 722, 723
- Surrogates area (Unicode) 42
- suspend method
 - of `Thread` (deprecated) 608
- swap method
 - of `Collections` 582
- Swing 695
- SwingWorker class 695
 - doInBackground method 696, 697, 701
 - execute method 697, 702
 - getState method 702
 - process, publish methods 696, 698, 702
- switch keyword 55, 99, 778
 - enumerated constants in 56
 - throwing exceptions in 103
 - value of 56
 - with fallthrough 102, 103
 - with pattern matching 284
- Synchronization 635
 - condition objects for 644
 - final fields and 661
 - in `Vector` 537
 - lock objects for 640
 - monitor concept for 659
 - race conditions in 636, 640, 662
 - volatile fields and 660
- Synchronization wrappers 684
- synchronized keyword 640, 652, 659, 778
- Synchronized blocks 657
- Synchronized views 569
- synchronizedCollection methods (`Collections`) 569, 571, 685
- System class
 - console method 79
 - exit method 34
 - getLogger method 437, 439
 - getProperties method 590, 591
 - getProperty method 84, 590, 591
 - identityHashCode method 560, 562
 - runFinalizersOnExit method 182
 - setOut method 159
 - setProperty method 440
- System.err 441, 454

- System.in 76
- System.Logger interface 437
 - getName method 450
 - isLoggable method 450
 - log method 437, 450
- System.Logger.Level enumeration 438
- System.out 35, 158, 436, 454
 - print method 79
 - printf method 79, 82, 270
 - println method 76
- T**
- T, t conversion characters 80
- Tab completion 28
- Tabs, in text blocks 75
- Tagging interfaces 350, 469, 523
- tailMap method
 - of NavigableMap 573
 - of SortedMap 567, 573
- tailSet method
 - of NavigableSet 567, 573
 - of SortedSet 567, 573
- take method
 - of BlockingQueue 668, 669, 673
 - of ExecutorCompletionService 630
- takeFirst/Last methods (BlockingDeque) 674
- tan method
 - of Math 49
- tar command 200
- @Target annotation 717, 718, 721
- Tasks
 - asynchronously running 618
 - controlling groups of 625
 - decoupling from mechanism of running 601
 - long-running 694
 - multiple 599
 - work stealing for 634
- Template code bloat 469
- Terminal window 21
- @Test annotation 712, 717
- Text blocks 73
- thenAccept, thenAcceptBoth, thenCombine methods (CompletableFuture) 689, 690
- thenApply, thenApplyAsync methods (CompletableFuture) 688, 689
- thenComparing method 374
- thenCompose method 689
- thenRun method 689
- this keyword 151, 175, 778
 - annotating 716
 - in first statement of constructor 176
 - in inner classes 380
 - in lambda expressions 369
 - in method references 365
 - vs. super 221
- Thread class 604, 606, 608, 613, 615, 616
 - currentThread method 609
 - extending 601
 - get/setUncaughtExceptionHandler methods 615
 - getDefaultUncaughtExceptionHandler method 615
 - getState method 608
 - interrupt, isInterrupted methods 609
 - interrupted method 611, 613
 - join method 606, 608
 - methods with timeout 606
 - resume method 608
 - run method 601, 604
 - setDaemon method 613
 - setDefaultUncaughtExceptionHandler method 454, 614, 615
 - setPriority method 616
 - sleep method 601, 604, 610
 - start method 601, 604, 605, 606
 - stop method (deprecated) 608, 609
 - suspend method (deprecated) 608
 - yield method 606
- Thread dump 650
- Thread groups 615
- Thread pools 621
- Thread-safe collections 667
 - callables and futures 618
 - concurrent 674
 - copy on write arrays 682
 - synchronization wrappers 684
- Thread.UncaughtExceptionHandler interface 614, 615
 - uncaughtException method 615
- ThreadGroup class 615
 - uncaughtException method 615
- ThreadLocal class 632
- ThreadLocal class, methods of 632
- ThreadLocalRandom class, current method 667
- ThreadPoolExecutor class 621
- Threads
 - accessing collections from 569, 667
 - blocked 606, 610
 - condition objects for 644
 - daemon 613
 - executing code in 369
 - idle 632
 - interrupting 609
 - listing all 650
 - locking 657
 - new 605
 - preemptive vs. cooperative scheduling for 605
 - priorities of 615
 - producer/customer 668
 - runnable 605
 - states of 605
 - synchronizing 635
 - terminated 601, 608, 609
 - thread-local variables in 666
 - timed waiting 606
 - unblocking 646
 - uncaught exceptions in 614
 - vs. processes 599
 - waiting 606, 645
 - work stealing for 634
 - worker 694

- throw keyword 409, 778
 - Throwable class 405, 429
 - add/getSuppressed methods 423, 425
 - get/initCause methods 425
 - getMessage method 411
 - getStackTrace method 424, 425
 - printStackTrace method 300, 423, 454
 - toString method 411
 - throws keyword 208, 301, 407, 778
 - for main method 84
 - Time measurement vs. calendars 137
 - Timed waiting threads 606
 - TimeoutException class 618, 690
 - Timer class 342, 354
 - start, stop methods 345
 - to keyword 778
 - toArray method
 - of ArrayList 491
 - of Collection 262, 519, 521, 584
 - of Stream 366
 - toHandle method 706, 709
 - toLowerCase method 68
 - Toolkit class
 - beep method 345
 - getDefaultToolkit method 345
 - toString method
 - adding to all classes 253
 - Formattable and 81
 - of Annotation 720
 - of Arrays 114, 119
 - of Date 133
 - of Enum 279, 281
 - of Integer 269
 - of Modifier 304, 311
 - of Object 250, 342
 - of proxy classes 401
 - of records 184, 253
 - of StackFrame 424, 427
 - of StackTraceElement 427
 - of StringBuilder 72, 73
 - of Throwable 411
 - redeclaring 358
 - working with any class 313
 - Total ordering 543
 - totalCpuDuration method
 - of ProcessHandle.Info 710
 - toUnsignedInt method 38
 - toUpperCase method 68
 - TraceHandler 397
 - TransferQueue interface 670
 - transfer, tryTransfer methods 674
 - transform method 373
 - of String 373
 - transient keyword 778
 - transitive keyword 767, 778
 - Tree maps 549
 - Tree sets 542
 - red-black 542
 - total ordering of 543
 - vs. priority queues 547
 - TreeMap class 524, 549, 552
 - as a concrete collection type 525
 - vs. HashMap 549
 - TreeSet class 524, 542
 - as a concrete collection type 525
 - Trigonometric functions 49
 - trim method
 - of String 68
 - trimToSize method 260
 - Troubleshooting. See Debugging
 - true literal 778
 - Truncated computations 49
 - try keyword 778
 - try-with-resources statement 421
 - effectively final variables in 422
 - no locks with 641
 - try/catch 412, 418
 - generics and 492
 - wrapping entire task in try block 428
 - try/finally 418
 - tryLock method 606
 - trySetAccessible method 316
 - Two-dimensional arrays 119, 124
 - type method 500, 501
 - of ServiceLoader.Provider 394, 395
 - Type bounds
 - annotating 715
 - Type erasure 468, 485
 - clashes after 495
 - Type parameters 257
 - annotating 713
 - converting to raw types 475
 - not for arrays 475, 486
 - not instantiated with primitive types 485
 - vs. inheritance 459
 - Type variables
 - bounds for 465
 - common names of 462
 - in exceptions 492
 - in static fields or methods 492
 - matching in generic methods 499
 - no instantiating for 489
 - replacing with bound types 468
 - TypeElement interface 731
 - Types. See Data types
 - TypeVariable interface 500, 501
 - getBounds, getName methods 508
- ## U
- UCSD Pascal system 5
 - UML (Unified Modeling Language) notation 131
 - UnaryOperator interface 370
 - uncaughtException method 615
 - Unchecked exceptions 300, 406, 408
 - applicability of 429
 - Unequality operator 54
 - Unicode 5, 39, 43, 59
 - Unit testing 161

- Unit tests 711
- University of Illinois 9
- UNIX 197, 199
- unlock method
 - of Lock 641, 643
- Unmodifiable copies 565, 570
- Unmodifiable views 565
- unmodifiableCollection methods (Collections) 565, 566, 571
- Unnamed modules 312
- Unnamed packages 192, 195, 214, 433
- UnsupportedOperationException class 556, 564, 566, 569, 571
- updateAndGet method 663
- user method
 - of ProcessHandle.Info 710
- User input 404
- User Interface. See Graphical User Interface
- User-defined types 267
- uses keyword 770, 771, 778
- “Uses-a” relationship 130
- UTC (Coordinated Universal Time) 136
- UTF-8 83
- Utility classes/methods 338, 340

V

- valueOf method
 - of BigInteger 107, 110
 - of Enum 279, 281
 - of Integer 269
- values method
 - of Map 554, 556
- var keyword 148, 357, 386, 778
 - diamond syntax and 258
- Varargs methods 270
 - passing generic types to 487
- Varargs parameters
 - safety of 721, 722
- VarHandle class 313, 761
- Variable handles 313, 761
- VariableElement interface 730
- Variables 43
 - accessing
 - from outer methods 383
 - in lambda expressions 366
 - annotating 473
 - copying 347
 - declarations of 43, 236
 - deprecated 720, 721
 - effectively final 368, 422
 - initializing 45, 215
 - local 148, 238, 473
 - mutating in lambda expressions 368
 - names of 43
 - package scope of 195
 - printing/logging values of 452
 - static 158
 - thread-local 666
- Vector class 511, 586, 587, 658, 659, 684
 - for dynamic arrays 258
 - get, set methods 658
 - synchronization in 537
- @version annotation 212, 214
- Views 562
 - bulk operations for 583
 - checked 568
 - restricted 569
 - subranges of 566
 - synchronized 569
 - unmodifiable 565
- Visual Basic
 - built-in date type in 133
 - syntax of 2
- Visual Studio 19
- void keyword 33, 34, 778
- volatile keyword 660, 662, 778
- Volatile fields 660
- von der Ahé, Peter 465

W

- wait method
 - of Object 606, 653, 657
- Wait sets 645
- waitFor method 705, 709
- walk method
 - of StackWalker 423, 426
- Warning messages 721
- Warning messages, suppressing 722
- Warnings
 - fallthrough behavior and 102
 - generic 265, 473, 487, 493
 - suppressing 487, 493
 - when using reflection 312
- Weak hash maps 557
- Weak references 557
- WeakHashMap class 557, 560
 - as a concrete collection type 525
- Weakly consistent iterators 675
- WeakReference class 557
- Web pages
 - dynamic 7
 - extracting links from 688
 - reading 695
- whenComplete method 689
- while keyword 90, 778
- Whitespace
 - escape sequence for 40, 75
 - in text blocks 75
 - irrelevant to compiler 33
 - leading/trailing 68, 75
- Wildcard types 461, 477
 - annotating 715
 - arrays of 487
 - capturing 482
 - supertype bounds for 478
 - unbounded 482

WildcardType interface 500, 501
 getLowerBounds, getUpperBounds methods 508
Windows operating system
 executing JARs in 203
 IDEs for 24
 JDK in 15
 paths in 197, 199
 thread priority levels in 616
Windows. See Dialogs
Wirth, Niklaus 5, 9, 127
with keyword 779
withInitial method 667
Work stealing 634
Worker threads 694
Working directory, for a process 702
Wrappers 265
 class constructors for 267
 equality testing for 266
 immutability of 265
 locks and 267, 658

X

X, x conversion characters 80
XML 10, 12
XML descriptors, generating 733
XML/JSON binding 760
xor method (BitSet) 594

Y

Yasson 761
yield keyword 103, 779
yield method (Thread) 606

Z

ZIP archives
 for JMOD files 774
ZIP format 197, 200