



ORACLE
PRESS



CORE JAVA

Volume I: Fundamentals

FOURTEENTH EDITION

ORACLE

Cay S. Horstmann

FREE SAMPLE CHAPTER |



This page intentionally left blank

Core Java

Volume I: Fundamentals

Fourteenth Edition

Cay S. Horstmann

Cover image: emotionPicture/stock.adobe.com

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 2.3-2.5, 12.2: Eclipse Foundation AISBL

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.

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

Author websites are not owned or managed by Pearson.

Visit us on the Web: informit.com

Library of Congress Control Number: 2025945021

Copyright © 2026 Pearson Education, Inc.
Hoboken, New Jersey

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.

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

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 pearson.com/en-us/global-permission-granting.html.

ISBN-13: 978-0-13-555857-7

ISBN-10: 0-13-555857-3

Contents

<i>Preface</i>	xiii
<i>Acknowledgments</i>	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	5
1.2.9. High-Performance	5
1.2.10. Multithreaded	6
1.2.11. Dynamic	6
1.3. Java Applets and the Internet	7
1.4. A Short History of Java	8
1.5. Common Misconceptions about Java	11
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. Source Files and Documentation	18
2.2. Using the Command-Line Tools	18
2.3. Using an Integrated Development Environment	23
2.4. JShell	25
3. Fundamental Programming Structures in Java	29
3.1. A Simple Java Program	29
3.2. Comments	32
3.3. Data Types	33
3.3.1. Integer Types	33
3.3.2. Floating-Point Types	34
3.3.3. The char Type	36
3.3.4. Unicode and the char Type	37
3.3.5. The boolean Type	38
3.4. Variables and Constants	39
3.4.1. Declaring Variables	39
3.4.2. Initializing Variables	40
3.4.3. Constants	41
3.4.4. Enumerated Types	42
3.5. Operators	42
3.5.1. Arithmetic Operators	42

3.5.2. Mathematical Functions and Constants.....	43
3.5.3. Conversions between Numeric Types	45
3.5.4. Casts.....	46
3.5.5. Assignment.....	46
3.5.6. Increment and Decrement Operators	47
3.5.7. Relational and boolean Operators	48
3.5.8. The Conditional Operator.....	48
3.5.9. Switch Expressions	49
3.5.10. Bitwise Operators.....	50
3.5.11. Parentheses and Operator Hierarchy	51
3.6. Strings.....	52
3.6.1. Concatenation	53
3.6.2. Static and Instance Methods.....	54
3.6.3. Indexes and Substrings.....	55
3.6.4. Strings Are Immutable	56
3.6.5. Testing Strings for Equality	57
3.6.6. Empty and Null Strings.....	57
3.6.7. The String API	58
3.6.8. Reading the Online API Documentation.....	59
3.6.9. Building Strings	61
3.6.10. Text Blocks	64
3.7. Input and Output	66
3.7.1. Reading Input.....	66
3.7.2. Formatting Output	69
3.8. Control Flow	72
3.8.1. Block Scope	72
3.8.2. Conditional Statements.....	73
3.8.3. Loops	75
3.8.4. Determinate Loops.....	80
3.8.5. Multiple Selections with switch	83
3.8.6. Statements That Break Control Flow.....	87
3.9. Big Numbers	89
3.10. Arrays.....	92
3.10.1. Declaring Arrays	92
3.10.2. Accessing Array Elements.....	94
3.10.3. The “for each” Loop	94
3.10.4. Array Copying	95
3.10.5. Command-Line Arguments.....	96
3.10.6. Array Sorting.....	97
3.10.7. Multidimensional Arrays	99
3.10.8. Ragged Arrays.....	102
4. Objects and Classes	107
4.1. Introduction to Object-Oriented Programming	107
4.1.1. Classes	108
4.1.2. Objects	109
4.1.3. Identifying Classes	109
4.1.4. Relationships between Classes	110
4.2. Using Predefined Classes	112
4.2.1. Objects and Object Variables	112
4.2.2. The LocalDate Class of the Java Library.....	115
4.2.3. Mutator and Accessor Methods	116

4.3. Defining Your Own Classes	120
4.3.1. An Employee Class	120
4.3.2. Dissecting the Employee Class	122
4.3.3. First Steps with Constructors	123
4.3.4. Declaring Local Variables with var	124
4.3.5. Working with null References	125
4.3.6. Implicit and Explicit Parameters	126
4.3.7. Benefits of Encapsulation	127
4.3.8. Class-Based Access Privileges	129
4.3.9. Private Methods	130
4.3.10. Final Instance Fields	130
4.4. Static Fields and Methods	131
4.4.1. Static Fields	131
4.4.2. Static Constants	132
4.4.3. Static Methods	133
4.4.4. Factory Methods	134
4.4.5. The main Method	134
4.5. Method Parameters	138
4.6. Object Construction	143
4.6.1. Overloading	143
4.6.2. Default Field Initialization	144
4.6.3. The Constructor with No Arguments	145
4.6.4. Explicit Field Initialization	146
4.6.5. Parameter Names	147
4.6.6. Calling Another Constructor	147
4.6.7. Initialization Blocks	148
4.6.8. Static Initialization	149
4.7. Records	153
4.7.1. The Record Concept	154
4.7.2. Constructors: Canonical, Compact, and Custom	156
4.8. Packages	158
4.8.1. Encapsulation	158
4.8.2. Package Names	158
4.8.3. Class Importation	159
4.8.4. Module Imports	161
4.8.5. Static Imports	161
4.8.6. Addition of a Class into a Package	162
4.8.7. Compiling with Packages	164
4.8.8. Package Access	165
4.8.9. The Class Path	166
4.8.10. Setting the Class Path	168
4.9. JAR Files	169
4.9.1. Creating JAR files	170
4.9.2. The Manifest	171
4.9.3. Executable JAR Files	172
4.9.4. Multi-Release JAR Files	172
4.9.5. A Note about Command-Line Options	174
4.10. Documentation Comments	175
4.10.1. Comment Insertion	175
4.10.2. Class Comments	176
4.10.3. Method Comments	177

4.10.4. Field Comments	178
4.10.5. Package Comments	178
4.10.6. HTML Markup.....	178
4.10.7. Links.....	179
4.10.8. General Comments.....	181
4.10.9. Code Snippets	181
4.10.10. Comment Extraction	182
4.11. Class Design Hints.....	183
5. Inheritance.....	187
5.1. Classes, Superclasses, and Subclasses.....	187
5.1.1. Defining Subclasses	188
5.1.2. Overriding Methods	189
5.1.3. Subclass Constructors.....	190
5.1.4. Inheritance Hierarchies	193
5.1.5. Polymorphism.....	194
5.1.6. Understanding Method Calls	196
5.1.7. Preventing Inheritance: Final Classes and Methods.....	198
5.1.8. Casting	200
5.1.9. Pattern Matching for instanceof	202
5.1.10. Protected Access	205
5.2. Object: The Cosmic Superclass	205
5.2.1. Variables of Type Object	206
5.2.2. The equals Method.....	206
5.2.3. Equality Testing and Inheritance	208
5.2.4. The hashCode Method.....	211
5.2.5. The toString Method.....	215
5.3. Generic Array Lists	221
5.3.1. Declaring Array Lists	222
5.3.2. Accessing Array List Elements.....	224
5.3.3. Compatibility between Typed and Raw Array Lists	227
5.4. Object Wrappers and Autoboxing	228
5.5. Methods with a Variable Number of Arguments	232
5.6. Abstract Classes.....	233
5.7. Enumeration Classes	238
5.8. Sealed Classes	242
5.9. Pattern Matching	247
5.9.1. Null Handling	248
5.9.2. Guards.....	249
5.9.3. Exhaustiveness.....	249
5.9.4. Dominance.....	251
5.9.5. Patterns and Constants	251
5.9.6. Variable Scope and Fallthrough	252
5.10. Reflection	254
5.10.1. The Class Class	255
5.10.2. A Primer on Declaring Exceptions	257
5.10.3. Resources	258
5.10.4. Using Reflection to Analyze the Capabilities of Classes	260
5.10.5. Using Reflection to Analyze Objects at Runtime.....	266
5.10.6. Using Reflection to Write Generic Array Code	270
5.10.7. Invoking Arbitrary Methods and Constructors	273
5.11. Design Hints for Inheritance	277

6. Interfaces, Lambda Expressions, and Inner Classes	281
6.1. Interfaces	281
6.1.1. The Interface Concept.....	282
6.1.2. Properties of Interfaces.....	288
6.1.3. Interfaces and Abstract Classes	289
6.1.4. Static and Private Methods	291
6.1.5. Default Methods	291
6.1.6. Resolving Default Method Conflicts	292
6.1.7. Interfaces and Callbacks	294
6.1.8. The Comparator Interface	296
6.1.9. Object Cloning.....	298
6.2. Lambda Expressions	304
6.2.1. Why Lambdas?	304
6.2.2. The Syntax of Lambda Expressions	305
6.2.3. Functional Interfaces	307
6.2.4. Function Types	308
6.2.5. Method References	310
6.2.6. Constructor References	313
6.2.7. Variable Scope.....	314
6.2.8. Lambda Expressions and this	316
6.2.9. Processing Lambda Expressions	317
6.2.10. Creating Comparators.....	321
6.3. Inner Classes	322
6.3.1. Use of an Inner Class to Access Object State	323
6.3.2. Special Syntax Rules for Inner Classes.....	326
6.3.3. Are Inner Classes Useful? Actually Necessary? Secure?	327
6.3.4. Local Inner Classes	328
6.3.5. Accessing Variables from Outer Methods	329
6.3.6. Anonymous Inner Classes	330
6.3.7. Static Classes	334
6.3.8. Nested Records	337
6.4. Service Loaders	338
6.5. Proxies	340
6.5.1. When to Use Proxies	340
6.5.2. Creating Proxy Objects	341
6.5.3. Properties of Proxy Classes.....	344
7. Exceptions, Assertions, and Logging	347
7.1. Dealing with Errors.....	348
7.1.1. The Classification of Exceptions	349
7.1.2. Declaring Checked Exceptions.....	351
7.1.3. How to Throw an Exception	353
7.1.4. Creating Exception Classes.....	354
7.2. Catching Exceptions	355
7.2.1. Catching an Exception	355
7.2.2. Catching Multiple Exceptions	357
7.2.3. Rethrowing and Chaining Exceptions	358
7.2.4. The finally Clause.....	360
7.2.5. The try-with-Resources Statement.....	362
7.2.6. Analyzing Stack Trace Elements.....	364
7.3. Tips for Using Exceptions	367

7.4. Using Assertions	370
7.4.1. The Assertion Concept	370
7.4.2. Assertion Enabling and Disabling	371
7.4.3. Using Assertions for Parameter Checking	373
7.4.4. Using Assertions for Documenting Assumptions	374
7.5. Logging	375
7.5.1. Should You Use the Java Logging Framework?	375
7.5.2. Logging 101	376
7.5.3. The Platform Logging API	377
7.5.4. Logging Configuration	378
7.5.5. Log Handlers	379
7.5.6. Filters and Formatters	382
7.5.7. A Logging Recipe	383
7.6. Debugging Tips	388
8. Generic Programming	395
8.1. Type Parameters	395
8.1.1. The Advantage of Generic Programming	395
8.1.2. Who Wants to Be a Generic Programmer?	397
8.1.3. Defining a Simple Generic Class	397
8.1.4. Generic Methods	399
8.1.5. Bounds for Type Variables	401
8.1.6. Generic Exceptions	403
8.2. Generic Code and the Virtual Machine	404
8.2.1. Type Erasure	404
8.2.2. Translating Generic Expressions	405
8.2.3. Translating Generic Methods	406
8.2.4. Calling Legacy Code	408
8.3. Inheritance Rules for Generic Types	409
8.4. Wildcard Types	410
8.4.1. The Wildcard Concept	410
8.4.2. Supertype Bounds for Wildcards	412
8.4.3. Unbounded Wildcards	415
8.4.4. Wildcard Capture	415
8.5. Restrictions and Limitations	418
8.5.1. Type Parameters Cannot Be Instantiated with Primitive Types	418
8.5.2. Casts Only Work with Raw Types	418
8.5.3. You Cannot Create Arrays of Parameterized Types	419
8.5.4. Varargs Warnings	420
8.5.5. Generic Varargs Do Not Spread Primitive Arrays	421
8.5.6. You Cannot Instantiate Type Variables	422
8.5.7. You Cannot Construct a Generic Array	423
8.5.8. Type Variables Are Not Valid in Static Contexts of Generic Classes	424
8.5.9. You Can Defeat Checked Exception Checking	425
8.5.10. Beware of Clashes after Erasure	426
8.5.11. Type Inference in Generic Record Patterns is Limited	427
8.6. Reflection and Generics	429
8.6.1. The Generic Class Class	429
8.6.2. Using Class<T> Parameters for Type Matching	430
8.6.3. Generic Type Information in the Virtual Machine	431
8.6.4. Type Literals	434

9. Collections.....	439
9.1. The Java Collections Framework	439
9.1.1. Separating Collection Interfaces and Implementation	440
9.1.2. The Collection Interface	442
9.1.3. Iterators	442
9.1.4. Generic Utility Methods	445
9.2. Interfaces in the Collections Framework.....	448
9.3. Concrete Collections.....	451
9.3.1. Linked Lists	453
9.3.2. Array Lists	461
9.3.3. Hash Sets	461
9.3.4. Tree Sets	465
9.3.5. Queues and Deques.....	468
9.3.6. Priority Queues	470
9.4. Maps	471
9.4.1. Basic Map Operations	471
9.4.2. Updating Map Entries	474
9.4.3. Map Views	476
9.4.4. Weak Hash Maps.....	478
9.4.5. Linked Hash Sets and Maps.....	478
9.4.6. Enumeration Sets and Maps	480
9.4.7. Identity Hash Maps	481
9.5. Copies and Views	483
9.5.1. Small Collections.....	483
9.5.2. Unmodifiable Copies and Views	485
9.5.3. Subranges	487
9.5.4. Sets From Boolean-Valued Maps.....	487
9.5.5. Reversed Views	488
9.5.6. Checked Views	488
9.5.7. Synchronized Views	489
9.5.8. A Note on Optional Operations	489
9.6. Algorithms.....	493
9.6.1. Why Generic Algorithms?.....	493
9.6.2. Sorting and Shuffling	495
9.6.3. Binary Search	497
9.6.4. Simple Algorithms	499
9.6.5. Bulk Operations.....	501
9.6.6. Converting between Collections and Arrays.....	502
9.6.7. Writing Your Own Algorithms	502
9.7. Legacy Collections	504
9.7.1. The Hashtable Class.....	504
9.7.2. Enumerations	504
9.7.3. Property Maps.....	505
9.7.4. System Properties	508
9.7.5. Stacks	510
9.7.6. Bit Sets	510
10. Concurrency	515
10.1. Running Threads.....	515
10.2. Thread States.....	520
10.2.1. New Threads	521
10.2.2. Runnable Threads	521

10.2.3. Blocked and Waiting Threads.....	521
10.2.4. Terminated Threads	522
10.3. Thread Properties	523
10.3.1. Virtual Threads.....	523
10.3.2. Thread Interruption	524
10.3.3. Daemon Threads	527
10.3.4. Thread Names and Ids	527
10.3.5. Handlers for Uncaught Exceptions	528
10.3.6. Thread Priorities	529
10.3.7. Thread Factories and Builders	530
10.4. Coordinating Tasks	531
10.4.1. Callables and Futures	531
10.4.2. Executor Services.....	534
10.4.3. Invoking a Group of Tasks.....	537
10.4.4. Thread-Local Variables.....	542
10.4.5. Scoped Values	543
10.4.6. The Fork-Join Framework.....	545
10.5. Synchronization	547
10.5.1. An Example of a Race Condition.....	548
10.5.2. The Race Condition Explained	550
10.5.3. Lock Objects.....	551
10.5.4. Condition Objects.....	554
10.5.5. Deadlocks	559
10.5.6. The synchronized Keyword.....	561
10.5.7. Synchronized Blocks	565
10.5.8. The Monitor Concept	567
10.5.9. Volatile Fields.....	568
10.5.10. Final Fields.....	569
10.5.11. Atomics.....	570
10.5.12. On-Demand Initialization	572
10.5.13. Safe Publication	573
10.5.14. Sharing with Thread-Local Variables	573
10.6. Thread-Safe Collections.....	574
10.6.1. Blocking Queues.....	575
10.6.2. Efficient Maps, Sets, and Queues	580
10.6.3. Atomic Update of Map Entries	582
10.6.4. Bulk Operations on Concurrent Hash Maps	585
10.6.5. Concurrent Set Views.....	587
10.6.6. Copy on Write Arrays	587
10.6.7. Parallel Array Algorithms	587
10.6.8. Older Thread-Safe Collections	588
10.7. Asynchronous Computations	589
10.7.1. Completable Futures.....	589
10.7.2. Composing Completable Futures	591
10.7.3. Long-Running Tasks in User Interface Callbacks	597
10.8. Processes	603
10.8.1. Building a Process.....	603
10.8.2. Running a Process.....	605
10.8.3. Process Handles	607

11. Annotations	611
11.1. Using Annotations.....	611
11.1.1. Annotation Elements	612
11.1.2. Multiple and Repeated Annotations	613
11.1.3. Annotating Declarations.....	613
11.1.4. Annotating Type Uses	614
11.1.5. Receiver Parameters	615
11.2. Defining Annotations	616
11.3. Annotations in the Java API	619
11.3.1. Annotations for Compilation	620
11.3.2. Meta-Annotations	621
11.4. Processing Annotations at Runtime	623
11.5. Source-Level Annotation Processing	627
11.5.1. Annotation Processors.....	627
11.5.2. The Language Model API	628
11.5.3. Using Annotations to Generate Source Code.....	628
11.6. Bytecode Engineering	632
11.6.1. Modifying Class Files	633
11.6.2. Modifying Bytecodes at Load Time	638
12. The Java Platform Module System	641
12.1. The Module Concept	641
12.2. Naming Modules	642
12.3. The Modular “Hello, World!” Program	643
12.4. Requiring Modules.....	645
12.5. Exporting Packages	646
12.6. Modular JARs	649
12.7. Modules and Reflective Access	651
12.8. Automatic Modules	654
12.9. The Unnamed Module.....	656
12.10. Command-Line Flags for Migration.....	657
12.11. Transitive and Static Requirements	658
12.12. Importing Modules.....	659
12.13. Qualified Exporting and Opening	660
12.14. Service Loading	661
12.15. Tools for Working with Modules	663
Appendix.....	667
<i>Index</i>	<i>671</i>

This page intentionally left blank

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 fourteenth 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 25.

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
- Foreign functions and memory

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

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:

java.lang.*IO* 25

- `println(Object obj)`
Converts the object to a string and prints it on the console, followed by a line separator.

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     IO.println("We will not use 'Hello, World!'");  
3 }
```

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.

This page intentionally left blank

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.

My thanks go to my editor, Harry Misthos, and to Julie Nahil from Pearson for steering the book through the production process. I wrote the book using HTML and CSS, and Prince (<https://princexml.com>) turned it into PDF—a workflow that I highly recommend.

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), Alan Bateman (Oracle), Alec Beaton (IBM), Cliff Berg, Andrew Binstock (Oracle), Joshua Bloch, David Brown, Brian Burkhalter (Oracle), Corky Cartwright, Hillmer Chona, Frank Cohen (PushToTest), Chris Crane (devXsolution), Joe Darcy (Oracle), Nicholas J. De Lillo (Manhattan College), Rakesh Dhoopar (Oracle), Ahmad R. Elkomey, Hanno Embregts (Info Support), 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, Majid Hameed, 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), Josh Juneau, Heinz Kabutz (The Java Specialists' Newsletter, <https://javaspecialists.eu>), 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), Jason Lee (IBM), Gregory Longshore, Bob Lynch (Lynch Associates), Michael McMahon (Oracle), Rustam Mehmandarov, Philip Milne (consultant), Mark Morrissey (The Oregon Graduate Institute), Maurice Naftalin, Mahesh Neelakanta (Florida Atlantic University), José Paumard (Oracle), Hao Pham, Paul Pillion, 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), Naoto Sato (Oracle), 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, Henri Tremblay, Paul Tyma (consultant), Christian Ullenboom, Peter van der Linden, Joe Wang (Oracle), Sven Woltmann, Burt Walsh, Dan Xu (Oracle), and John Zavgren (Oracle).

Finally, a warm thank you to my coauthor of earlier editions, Gary Cornell, and to Greg Doench who was my editor for almost thirty years.

*Cay Horstmann
Düsseldorf, Germany
September 2025*

Fundamental Programming Structures in Java

At this point, you should have successfully installed the JDK and executed the sample programs from [Chapter 2](#). It's time to start programming. This chapter shows you how the basic programming concepts such as data types, branches, and loops are implemented in Java.

3.1. A Simple Java Program

Let's look more closely at one of the simplest Java programs you can have—one that merely prints a message to console:

```
void main() {  
    IO.println("We will not use 'Hello, World!'");  
}
```

First and foremost, *Java is case sensitive*. If you made any mistakes in capitalization (such as typing `Main` instead of `main`), the program will not run.

The program declares a method called `main`. The term “method” is Java-speak for a function—a block of code that carries out a specific task. You *must* have a `main` method in every program. You can, of course, add your own methods and call them from the `main` method.

Notice the braces `{ }` in the source code. In Java, as in C/C++, braces are used to form a group of statements (called a *block*). In Java, the code for any method must be started by an opening brace `{` and ended by a closing brace `}`.

Brace styles have inspired an inordinate amount of useless controversy. This book follows a compact style that is common among Java programmers, sometimes called the “Kernighan and Ritchie” style. In other styles, matching braces line up. As whitespace is irrelevant to the Java compiler, you can use whatever brace style you like.

The main method calls another method, called `println`, defined in the `I0` class. You will learn a lot more about classes in the next chapter. For now, think of a class as a container for the program logic that defines the behavior of an application. Classes are the building blocks with which all Java applications are built.

In fact, *everything* in a Java program lives inside a class, even our main method. It is placed inside a class whose name is the name of the file, without the extension. If we place the code in a file named `FirstSample.java`, `main` is a method of a class `FirstSample`.

The standard naming convention (used in the name `FirstSample`) is that class names are nouns that start with an uppercase letter. If a name consists of multiple words, use an initial uppercase letter in each of the words. This use of uppercase letters in the middle of a name is sometimes called “camel case” or, self-referentially, “`CamelCase`.”



Note: Prior to Java 25, you had to explicitly declare the class containing the main method. This is no longer necessary.

It used to be a requirement to declare the main method as

```
public static void main(String[] args)
```

You will learn in [Chapter 4](#) what the keywords `public` and `static` mean. The `String[] args` parameter holds command line arguments—see [Section 3.10.5](#).

Moreover, Java 25 introduced the `I0` class to simplify console input and output. Previously, you had to use the special `System.out` object, which was yet another concept that was confusing to beginners.

To run any of the programs in this chapter with an older version of Java, do the following:

1. Place all code inside a class whose name equals the file name, without the extension.
2. Declare `main` in the old style.
3. Replace `I0.println` with `System.out.println`

For example:

```
public class FirstSample {  
    public static void main(String[] args) {  
        System.out.println("We will not use 'Hello, World!'");  
    }  
}
```



Note: Version 1.0 of the Java Language Specification decreed that the main method must be declared public, static, and void. (The Java Language Specification is the official document that describes the Java language. You can view or download it from <https://docs.oracle.com/javase/specs>.)

However, early versions of the Java launcher were willing to execute Java programs even when the main method was not public. A programmer filed a bug report. To see it, visit <https://bugs.openjdk.org/browse/JDK-4252539>. In 1999, that bug was marked as “closed, will not be fixed.” An engineer added an explanation that the Java Virtual Machine Specification does not mandate that main is public and that “fixing it will cause potential troubles.” In the end, sanity prevailed. As of Java 1.4, the Java launcher enforces that the main method is public, as intended in the language specification. That behavior was in place until Java 25, which allows other forms of the main method.

It is remarkable that the bug reports and their resolutions have been available for anyone to scrutinize for as long as Java existed, even before it became open source.

Now turn your attention to the contents inside the braces of the main method,

```
I0.println("We will not use 'Hello, World!'");
```

This is the *body* of the method. The body of most methods contains multiple statements, but here we have just one. As with most programming languages, you can think of Java statements as sentences of the language. In Java, every statement must end with a semicolon. In particular, carriage returns do not mark the end of a statement, so statements can span multiple lines if need be.

Here, we are calling the `println` method that is declared in a class called `I0`. Notice the period that separates the name of the `I0` class and the `println` method.

The `println` method receives a string argument. The method displays the string argument on the console. It then terminates the output line, so that each call to `println` displays its output on a new line. Notice that Java, like C/C++, uses double quotes to delimit strings. (You can find more information about strings later in this chapter.)

Methods in Java, like functions in any programming language, can use zero, one, or more *arguments*, which are enclosed in parentheses. Even if a method has no arguments, you must still use empty parentheses. For example, a variant of the `println` method with no arguments just prints a blank line. You invoke it with the call

```
I0.println();
```



Note: The `I0` class also has a `print` method that doesn’t add a newline character to the output. For example, `I0.print("Hello")` prints Hello without a newline. The next output appears immediately after the letter o.

You compile the file with the command

You run the sample program with this command:

```
java FirstSample.java
```

When the program executes, it simply displays the string We will not use 'Hello, World!' on the console.

If you intend to run a program multiple times, it is more efficient to compile it first:

```
javac FirstSample.java
```

You end up with a file containing the *bytecodes* for this class. These are instructions for the Java virtual machine. The Java compiler names the bytecode file `FirstSample.class` and stores it in the same directory as the source file. Whenever you want to launch the program, issue the following command:

```
java FirstSample
```

Remember to leave off the `.class` extension.

When you use

```
java ClassName
```

to run a compiled program, the Java virtual machine is launched, and execution starts with the code in the `main` method of the class you indicate.

3.2. Comments

Comments in Java, as in most programming languages, do not show up in the executable program. Thus, you can add as many comments as needed without fear of bloating the code. Java has three ways of marking comments. The most common form is a `//`. Use this for a comment that runs from the `//` to the end of the line.

```
IO.println("We will not use 'Hello, World!'); // is this too cute?
```

When longer comments are needed, you can mark each line with a `//`, or you can use the `/*` and `*/` comment delimiters that let you block off a longer comment.

Finally, a third kind of comment is used to generate documentation automatically. This comment uses a `/**` to start and a `*/` to end. You can see this type of comment in [Listing 3.1](#). For more on this type of comment and on automatic documentation generation, see [Chapter 4](#).

Listing 3.1: FirstSample.java

```
1  /**
2   * This is the first sample program in Core Java Chapter 3
3   */
4  void main() {
5      IO.println("We will not use 'Hello, World!');
6  }
```



Caution: `/* */` comments do not nest in Java. That is, you might not be able to deactivate code simply by surrounding it with `/*` and `*/` because the code you want to deactivate might itself contain a `*/` delimiter.

3.3. Data Types

Java is a *strongly typed language*. This means that every variable must have a declared type. There are eight *primitive types* in Java. Four of them are integer types; two are floating-point number types; one is the character type `char`, used for UTF-16 code units in the Unicode encoding scheme (see [Section 3.3.3](#)); and one is a boolean type for truth values.



Note: Java has an arbitrary-precision arithmetic package. However, “big numbers,” as they are called, are Java *objects* and not a primitive Java type. You will see how to use them later in this chapter.

3.3.1. Integer Types

The integer types are for numbers without fractional parts. Negative values are allowed. Java provides the four integer types shown in [Table 3.1](#).

Table 3.1: Java Integer Types

Type	Storage Requirement	Range (Inclusive)
byte	1 byte	-128 to 127
short	2 bytes	-32,768 to 32,767
int	4 bytes	-2,147,483,648 to 2,147,483,647 (just over 2 billion)
long	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

In most situations, the `int` type is the most practical. If you want to represent the number of inhabitants of our planet, you’ll need to resort to a `long`. The `byte` and `short` types are mainly intended for specialized applications, such as low-level file handling, or for large arrays when storage space is at a premium.

Under Java, the ranges of the integer types do not depend on the machine on which you will be running the Java code. This alleviates a major pain for the programmer who wants to move software from one platform to another, or even between operating systems on the same platform. In contrast, C and C++ programs use the most efficient integer type for each processor. As a result, a C program that runs well on a 32-bit processor may exhibit integer overflow on a 16-bit system. Since Java programs must run with the same results on all machines, the ranges for the various types are fixed.

Long integer numbers have a suffix `L` or `l` (for example, `4000000000L`). Hexadecimal numbers have a prefix `0x` or `0X` (for example, `0xCAFE`). Octal numbers have a prefix `0` (for example, `010` is 8)—naturally, this can be confusing, and few programmers use octal constants.

You can write numbers in binary, with a prefix `0b` or `0B`. For example, `0b1001` is 9. You can add underscores to number literals, such as `1_000_000` (or `0b1111_0100_0010_0100_0000`) to denote one million. The underscores are for human eyes only. The Java compiler simply removes them.



Note: In C and C++, the sizes of types such as `int` and `long` depend on the target platform. On a 32-bit processor, integers have 4 bytes, but on a 64-bit processor they may have 4 bytes or 8 bytes. These differences make it challenging to write cross-platform programs. In Java, the sizes of all numeric types are platform-independent.

Note that Java does not have any unsigned versions of the `int`, `long`, `short`, or `byte` types.



Note: If you work with integer values that can never be negative and you really need an additional bit, you can, with some care, interpret signed integer values as unsigned. For example, instead of having a `byte` value `b` represent the range from -128 to 127, you may want a range from 0 to 255. You can store it in a `byte`. Due to the nature of binary arithmetic, addition, subtraction, and multiplication will work provided they don't overflow. For other operations, call `Byte.toUnsignedInt(b)` to get an `int` value between 0 and 255, then process the integer value and cast back to `byte`. The `Integer` and `Long` classes have methods for unsigned division and remainder.

3.3.2. Floating-Point Types

The floating-point types denote numbers with fractional parts. The two floating-point types are shown in [Table 3.2](#).

Table 3.2: Floating-Point Types

Type	Storage Requirement	Range
<code>float</code>	4 bytes	Approximately $\pm 3.40282347 \times 10^{38}$ (6-7 significant decimal digits)
<code>double</code>	8 bytes	Approximately $\pm 1.79769313486231570 \times 10^{308}$ (15 significant decimal digits)

The name `double` refers to the fact that these numbers have twice the precision of the `float` type. (Some people call these *double-precision* numbers.) The limited precision of `float` (6-7 significant digits) is simply not sufficient for many situations. Use `float` values only when you work with a library that requires them, or when you need to store a very large number of them.

Java 20 adds a couple of methods (`Float.floatToFloat16` and `Float.float16toFloat`) for storing “half-precision” 16-bit floating-point numbers in short values. These are used for implementing neural networks.

Numbers of type `float` have a suffix `F` or `f` (for example, `3.14F`). Floating-point numbers without an `F` suffix (such as `3.14`) are always considered to be of type `double`. You can optionally supply the `D` or `d` suffix (for example, `3.14D`).

An `E` or `e` denotes a decimal exponent. For example, `1.729E3` is the same as `1729`.



Note: You can specify floating-point literals in hexadecimal. For example, $0.125 = 2^{-3}$ can be written as `0x1.0p-3`. In hexadecimal notation, you use a `p`, not an `e`, to denote the exponent. (An `e` is a hexadecimal digit.) Note that the mantissa is written in hexadecimal and the exponent in decimal. The base of the exponent is 2, not 10.

All floating-point computations follow the IEEE 754 specification. In particular, there are three special floating-point values to denote overflows and errors:

- Positive infinity
- Negative infinity
- NaN (not a number)

For example, the result of dividing a positive floating-point number by 0 is positive infinity. Dividing 0.0 by 0 or the square root of a negative number yields NaN.



Note: The constants `Double.POSITIVE_INFINITY`, `Double.NEGATIVE_INFINITY`, and `Double.NaN` (as well as corresponding `Float` constants) represent these special values, but they are rarely used in practice. In particular, you cannot test

```
if (x == Double.NaN) // is never true
```

to check whether a particular result equals `Double.NaN`. All “not a number” values are considered distinct. However, you can use the `Double.isNaN` method:

```
if (Double.isNaN(x)) // check whether x is "not a number"
```



Note: There are both positive and negative floating-point zeroes, `0.0` and `-0.0`, but you can’t tell them apart with `==`. To check whether a value is negative zero, use this test:

```
if (Double.compare(x, -0.0) == 0)
```



Caution: Floating-point numbers are *not* suitable for financial calculations in which roundoff errors cannot be tolerated. For example, the command `I0.println(2.0 - 1.1)` prints `0.8999999999999999`, not `0.9` as you would expect. Such roundoff errors are caused by the fact that floating-point numbers are represented in the binary number system. There is no

precise binary representation of the fraction 9/10, just as there is no accurate representation of the fraction 1/3 in the decimal system. If you need precise numerical computations without roundoff errors, use the `BigDecimal` class, which is introduced later in this chapter.

3.3.3. The char Type

The `char` type was originally intended to describe individual characters. However, this is no longer the case. Nowadays, some Unicode characters can be described with one `char` value, and other Unicode characters require two `char` values. Read the next section for the gory details.

Literal values of type `char` are enclosed in single quotes. For example, `'A'` is a character constant with value 65. It is different from `"A"`, a string containing a single character. Values of type `char` can be expressed as hexadecimal values that run from `\u0000` to `\uFFFF`.

Besides the `\u` escape sequences, there are several escape sequences for special characters, as shown in [Table 3.3](#). You can use these escape sequences inside quoted character literals and strings, such as `'\u005B'` or `"Hello\n"`. The `\u` escape sequence (but none of the other escape sequences) can even be used *outside* quoted character constants and strings. For example,

```
void main()\u007BI0.println("Hello, World!");\u007D
```

is perfectly legal—`\u007B` and `\u007D` are the encodings for `{` and `}`.

Table 3.3: Escape Sequences for Special Characters

Escape Sequence	Name	Unicode Value
<code>\b</code>	Backspace	<code>\u0008</code>
<code>\t</code>	Tab	<code>\u0009</code>
<code>\n</code>	Line feed	<code>\u000a</code>
<code>\r</code>	Carriage return	<code>\u000d</code>
<code>\f</code>	Form feed	<code>\u000c</code>
<code>\"</code>	Double quote	<code>\u0022</code>
<code>\'</code>	Single quote	<code>\u0027</code>
<code>\\</code>	Backslash	<code>\u005c</code>
<code>\s</code>	Space. Used in text blocks to retain trailing whitespace.	<code>\u0020</code>
<code>\newline</code>	In text blocks only: Join this line with the next	—



Caution: Unicode escape sequences are processed before the code is parsed. For example, `"\u0022+\u0022"` is *not* a string consisting of a plus sign surrounded by quotation marks (U+0022). Instead, the `\u0022` are converted into `"` before parsing, yielding `""`, or an empty string.

Even more insidiously, you must beware of `\u` inside comments. The comment

```
// \u000A is a newline
```

yields a syntax error since `\u000A` is replaced with a newline when the program is read. Similarly, a comment

```
// look inside c:\users
```

yields a syntax error because the `\u` is not followed by four hex digits.



Note: You can have any number of `u` in a Unicode escape sequence: `\u00E9` and `\uuu00E9` both denote the character `é`. There is a reason for this oddity. Consider a programmer happily coding in Unicode who is forced, for some archaic reason, to check in code as ASCII only. A conversion tool can turn any character `> U+007F` into a Unicode escape and add a `u` to every existing Unicode escape. That makes the conversion reversible. For example, `\uD800 é` is turned into `\uuD800 \u00E9` and can be converted back to `\uD800 é`.

3.3.4. Unicode and the char Type

To fully understand the `char` type, you have to know about the Unicode encoding scheme. Before Unicode, there were many different character encoding standards: ASCII in the United States, ISO 8859-1 for Western European languages, KOI-8 for Russian, GB18030 and BIG-5 for Chinese, and so on. This caused two problems. First, a particular code value corresponds to different letters in the different encoding schemes. Second, the encodings for languages with large character sets have variable length: Some common characters are encoded as single bytes, others require two or more bytes.

Unicode was designed to solve both problems. When the unification effort started in the 1980s, a fixed 2-byte code was more than sufficient to encode all characters used in all languages in the world, with room to spare for future expansion—or so everyone thought at the time. In 1991, Unicode 1.0 was released, using slightly less than half of the available 65,536 code values. Java was designed from the ground up to use 16-bit Unicode characters, which was a major advance over other programming languages that used 8-bit characters.

Unfortunately, over time, the inevitable happened. Unicode grew beyond 65,536 characters, primarily due to the addition of a very large set of ideographs used for Chinese, Japanese, and Korean. Now, the 16-bit `char` type is insufficient to describe all Unicode characters.

We need a bit of terminology to explain how this problem is resolved in Java. A *code point* is an integer value associated with a character in an encoding scheme. In the Unicode standard, code points are written in hexadecimal and prefixed with `U+`, such as `U+0041` for the code point of the Latin letter A. Unicode has code points that are grouped into 17 *code planes*, each holding 65536

characters. The first code plane, called the *basic multilingual plane*, consists of the “classic” Unicode characters with code points U+0000 to U+FFFF. Sixteen additional planes, with code points U+10000 to U+10FFFF, hold many more characters called *supplementary* characters.

How a Unicode code point (that is, an integer ranging from 0 to hexadecimal 10FFFF) is represented in bits depends on the *character encoding*. You could encode each character as a sequence of 21 bits, but that is impractical for computer hardware. The UTF-32 encoding simply places each code point into 32 bits, where the top 11 bits are zero. That is rather wasteful. The most common encoding on the Internet is UTF-8, using between one and four bytes per character. See [Chapter 2 of Volume II](#) for details of that encoding.

Java strings use the UTF-16 encoding. It encodes all Unicode code points in a variable-length code of 16-bit units, called *code units*. The characters in the basic multilingual plane are encoded as a single code unit. All other characters are encoded as consecutive pairs of code units. Each of the code units in such an encoding pair falls into a range of 2048 unused values of the basic multilingual plane, called the *surrogates area* ('`\uD800`' to '`\uDBFF`' for the first code unit, '`\uDC00`' to '`\uDFFF`' for the second code unit). This is rather clever, because you can immediately tell whether a code unit encodes a single character or it is the first or second part of a supplementary character. For example, the beer mug emoji 🍺 has code point U+1F37A and is encoded by the two code units '`\uD83C`' and '`\uDF7A`'. (See <https://tools.ietf.org/html/rfc2781> for a description of the encoding algorithm.) Each code unit is stored as a `char` value. The details are not important. All you need to know is that a single Unicode character may require one or two `char` values.

You cannot ignore characters with code units above U+FFFF. Your customers may well write in a language where these characters are needed, or they may be fond of putting emojis such as 🍺 into their messages.

Nowadays, Unicode has become so complex that even code points no longer correspond to what a human viewer would perceive as a single character or symbol. This happens with languages whose characters are made from smaller building blocks, with emojis that can have modifiers for gender and skin tone, and with an ever-growing number of other compositions.

Consider the pirate flag 🏴. You perceive a single symbol: the flag. However, this symbol is composed of four Unicode code points: U+1F3F4 (waving black flag), U+200D (zero width joiner), U+2620 (skull and crossbones), and U+FE0F (variation selector-16). In Java, you need five `char` values to represent the flag: two `char` for the first code point, and one each for the other three.

In summary, a visible character or symbol is encoded as a sequence of some number of `char` values, and there is almost never a need to look at the individual values. Always work with strings (see [Section 3.6](#)) and don't worry about their representation as `char` sequences.

3.3.5. The boolean Type

The boolean type has two values, `false` and `true`. It is used for evaluating logical conditions. You cannot convert between integers and boolean values.



Note: In languages such as C++ and JavaScript, other values, such as numbers and even strings, can be used in place of boolean values. The value `0` is equivalent to the `bool` value

false, and a nonzero value is equivalent to true. This is *not* the case in Java. Thus, Java programmers are shielded from accidents such as

```
if (x = 0) // oops... meant x == 0
```

In C++ and JavaScript, this test compiles and runs, always evaluating to false. In Java, the test does not compile because the integer expression `x = 0` cannot be converted to a boolean value.

3.4. Variables and Constants

As in every programming language, variables are used to store values. Constants are variables whose values don't change. In the following sections, you will learn how to declare variables and constants.

3.4.1. Declaring Variables

In Java, every variable has a *type*. You declare a variable by placing the type first, followed by the name of the variable. Here are some examples:

```
double salary;  
int vacationDays;  
long earthPopulation;  
boolean done;
```

Notice the semicolon at the end of each declaration. The semicolon is necessary because a declaration is a complete Java statement, which must end in a semicolon.

The identifier for a variable name (as well as for other names) is made up of letters, digits, currency symbols, and “punctuation connectors.” The first character cannot be a digit.

Symbols like '+' or '@' cannot be used inside variable names, nor can spaces. Letter case is significant: `main` and `Main` are distinct identifiers. The length of an identifier is essentially unlimited.

The terms “letter,” “digit,” and “currency symbol” are much broader in Java than in most languages. A letter is *any* Unicode character that denotes a letter in a language. For example, German users can use umlauts such as `ä` in variable names; Greek speakers could use a `π`. Similarly, digits are 0–9 and *any* Unicode characters that denote a digit. Currency symbols are `$`, `€`, `¥`, and so on. Punctuation connectors include the underscore character `_`, a “wavy low line” `~`, and a few others. In practice, most programmers stick to A–Z, a–z, 0–9, and the underscore `_`.



Tip: If you are really curious as to what Unicode characters can be used in identifiers, you can use the `isJavaIdentifierStart` and `isJavaIdentifierPart` methods in the `Character` class to check.



Tip: Even though \$ is a valid character in an identifier, you should not use it in your own code. It is intended for names that are generated by the Java compiler and other tools.

You also cannot use a Java keyword such as `class` as a variable name.

Underscores can be parts of identifiers. This is common for constant names, such as `Double.POSITIVE_INFINITY`. However, a single underscore `_` is a keyword.



Note: As of Java 21, a single underscore `_` denotes a variable that is syntactically required but never used. You will see examples in Chapters 6 and 7.

You can declare multiple variables on a single line:

```
int i, j; // both are integers
```

I don't recommend this style. If you declare each variable separately, your programs are easier to read.



Note: As you saw, names are case sensitive, for example, `hireday` and `hireDay` are two separate names. In general, you should not have two names that only differ in their letter case. However, sometimes it is difficult to come up with a good name for a variable. Many programmers then give the variable the same name as the type, for example

```
Box box; // "Box" is the type and "box" is the variable name
```

Other programmers prefer to use an "a" prefix for the variable:

```
Box aBox;
```

3.4.2. Initializing Variables

After you declare a variable, you must explicitly initialize it by means of an assignment statement—you can never use the value of an uninitialized variable. For example, the Java compiler flags the following sequence of statements as an error:

```
int vacationDays;  
IO.println(vacationDays); // ERROR--variable not initialized
```

You assign to a previously declared variable by using the variable name on the left, an equal sign (`=`), and then some Java expression with an appropriate value on the right.

```
int vacationDays;  
vacationDays = 12;
```

You can both declare and initialize a variable on the same line. For example:

```
int vacationDays = 12;
```

Finally, in Java you can put declarations anywhere in your code. For example, the following is valid code in Java:

```
double salary = 65000.0;
IO.println(salary);
int vacationDays = 12; // OK to declare a variable here
```

In Java, it is considered good style to declare variables as closely as possible to the point where they are first used.



Note: You do not need to declare the types of local variables if they can be inferred from the initial value. Simply use the keyword `var` instead of the type:

```
var vacationDays = 12; // vacationDays is an int
var greeting = "Hello"; // greeting is a String
```

This is not too important for number and string types, but, as you will see in the next chapter, this feature can make the declaration of objects less verbose.

3.4.3. Constants

In Java, you use the keyword `final` to denote a constant. For example:

```
void main() {
    final double CM_PER_INCH = 2.54;
    double paperWidth = 8.5;
    double paperHeight = 11;
    IO.println("Paper size in centimeters: "
        + paperWidth * CM_PER_INCH + " by " + paperHeight * CM_PER_INCH);
}
```

The keyword `final` indicates that you can assign to the variable once, and then its value is set once and for all. It is customary to name constants in all uppercase.

It is probably more common in Java to create a constant so it's available to all methods of a class:

```
final double CM_PER_INCH = 2.54;

void main() {
    double paperWidth = 8.5;
    double paperHeight = 11;
    IO.println("Paper size in centimeters: "
        + paperWidth * CM_PER_INCH + " by " + paperHeight * CM_PER_INCH);
}

// CM_PER_INCH also accessible in other methods
```

You will see in [Chapter 4](#) how a class can declare constants that are usable in other classes. For example, the `Math` class declares a constant `PI` that you can use as `Math.PI`.



Caution: Some coding style guides state that uppercase letters should only be used for class constants, not local ones. If you need to follow such a style guide, and you have a local constant, decide what is more important to you—the fact that it is local (and lowercase), or that it is visibly a constant (in uppercase).



Note: `const` is a Java keyword, but it is not currently used for anything. You must use `final` for a constant.

3.4.4. Enumerated Types

Sometimes, a variable should only hold a restricted set of values. For example, you may sell clothes or pizza in four sizes: small, medium, large, and extra large. Of course, you could encode these sizes as integers 1, 2, 3, 4 or characters S, M, L, and X. But that is an error-prone setup. It is too easy for a variable to hold a wrong value (such as 0 or m).

You can define your own *enumerated type* whenever such a situation arises. An enumerated type has a finite number of named values. For example,

```
enum Size { SMALL, MEDIUM, LARGE, EXTRA_LARGE };
```

Now you can declare variables of this type:

```
Size s = Size.MEDIUM;
```

A variable of type `Size` can hold only one of the values listed in the type declaration, or the special value `null` that indicates that the variable is not set to any value at all. (See [Chapter 4](#) for more information about `null`.)

Enumerated types are discussed in greater detail in [Chapter 5](#).

3.5. Operators

Operators are used to combine values. As you will see in the following sections, Java has a rich set of arithmetic and logical operators and mathematical functions.

3.5.1. Arithmetic Operators

The usual arithmetic operators `+`, `-`, `*`, and `/` are used in Java for addition, subtraction, multiplication, and division.

The `/` operator denotes integer division if both operands are integers, and floating-point division otherwise. Integer division by 0 raises an exception, whereas floating-point division by 0 yields an infinite or NaN result.

Integer remainder (sometimes called *modulus*) is denoted by %. For example, $15 / 2$ is 7, $15 \% 2$ is 1, and $15.0 / 2$ is 7.5.



Caution: When one of the operands of % is negative, so is the result. For example, $n \% 2$ is 0 if n is even, 1 if n is odd and positive, and -1 if n is odd and negative. Why? When the first computers were built, someone had to make rules for how integer remainder should work for negative operands. Mathematicians had known the optimal (or “Euclidean”) rule for a few hundred years: always leave the remainder ≥ 0 . But, rather than open a math textbook, those pioneers came up with rules that seemed reasonable but are actually inconvenient.

Consider this problem. You compute the position of the hour hand of a clock. An adjustment is applied, and you want to normalize to a number between 0 and 11. That is easy: $(\text{position} + \text{adjustment}) \% 12$. But what if the adjustment is negative? Then you might get a negative number. So you have to introduce a branch, or use $((\text{position} + \text{adjustment}) \% 12 + 12) \% 12$. Either way, it is a hassle.

A better remedy is to use the `floorMod` method: `Math.floorMod(position + adjustment, 12)` always yields a value between 0 and 11. Unfortunately, `floorMod` still gives negative remainders for negative divisors, but that situation doesn’t often occur in practice.



Note: One of the stated goals of the Java programming language is portability. A computation should yield the same results no matter which virtual machine executes it. For that reason, the Java 1.0 language specification requires adherence to the IEEE 754 standard for 32- and 64-bit floating-point numbers. However, for many years, Intel processors used “extended” 80-bit floating-point registers for 64-bit floating-point operations, occasionally yielding more accurate but non-standard results.

To get the standard results on those processors was slower. As a pragmatic compromise, Java 1.2 allowed extended precision for intermediate computations. The keyword was introduced to force portable results. Modern processors can carry out 64-bit arithmetic efficiently. As of Java 17, the virtual machine is again required to use standard 64-bit arithmetic, and the keyword is now obsolete.

3.5.2. Mathematical Functions and Constants

The `Math` class contains an assortment of mathematical functions that you may occasionally need, depending on the kind of programming that you do.

To take the square root of a number, use the `sqrt` method:

```
double x = 4;
double y = Math.sqrt(x);
IO.println(y); // prints 2.0
```

The Java programming language has no operator for raising a quantity to a power: You must use the `pow` method in the `Math` class. The statement

```
double y = Math.pow(x, a);
```

sets y to be x raised to the power a (x^a). The `pow` method's arguments are both of type `double`, and it returns a `double` as well.

The `Math` class supplies the usual trigonometric functions:

```
Math.sin  
Math.cos  
Math.tan  
Math.atan  
Math.atan2
```

and the exponential function with its inverse, the natural logarithm, as well as the decimal logarithm:

```
Math.exp  
Math.log  
Math.log10
```

Java 21 adds a method `Math.clamp` that forces a number to fit within given bounds. For example:

```
Math.clamp(-1, 0, 10) // too small, yields lower bound 0  
Math.clamp(11, 0, 10) // too large, yields upper bound 10  
Math.clamp(3, 0, 10) // within bounds, yields value 3
```

Finally, three constants denote the closest possible approximations to the mathematical constants π , $\tau = 2\pi$, and e :

```
Math.PI  
Math.TAU  
Math.E
```



Tip: You can avoid the `Math` prefix for the mathematical methods and constants by adding the following line to the top of your source file:

```
import static java.lang.Math.*;
```

For example:

```
IO.println("The square root of  $\pi$  is " + sqrt(PI));
```

Static imports are covered in [Chapter 4](#).



Note: The methods in the `Math` class use the routines in the computer's floating-point unit for fastest performance. If completely predictable results are more important than performance, use the `StrictMath` class instead. It implements the algorithms from the “Freely Distributable Math Library” (<https://www.netlib.org/fdlibm>), guaranteeing identical results on all platforms.



Note: The Math class provides several methods to make integer arithmetic safer. The mathematical operators quietly return wrong results when a computation overflows. For example, one billion times three ($1000000000 * 3$) evaluates to -1294967296 because the largest int value is just over two billion. If you call `Math.multiplyExact(1000000000, 3)` instead, an exception is generated. You can catch that exception or let the program terminate rather than quietly continue with a wrong result. There are additional methods, including `addExact`, `subtractExact`, `incrementExact`, `decrementExact`, `negateExact`, `absExact`, `powExact`, all with arguments of type `int` and `long`.

3.5.3. Conversions between Numeric Types

It is often necessary to convert from one numeric type to another. [Figure 3.1](#) shows the legal conversions.

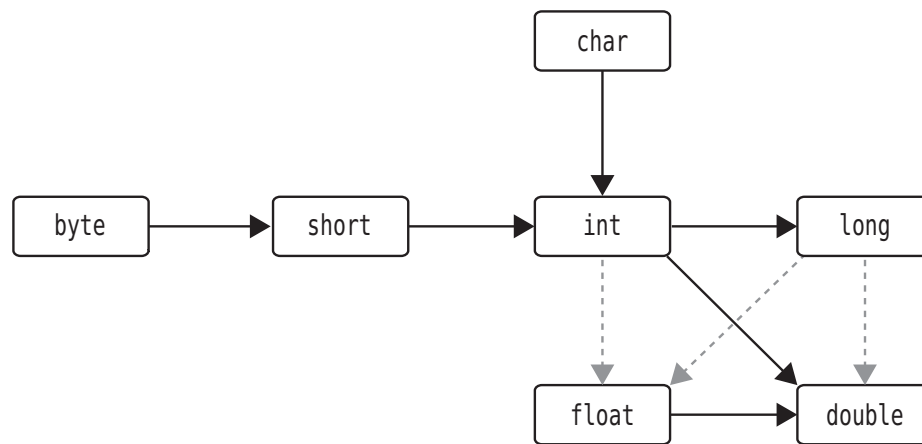


Figure 3.1: Legal conversions between numeric types

The six solid arrows in [Figure 3.1](#) denote conversions without information loss. The three dotted arrows denote conversions that may lose precision. For example, a large integer such as 123456789 has more digits than the float type can represent. When the integer is converted to a float, the resulting value has the correct magnitude but loses some precision.

```
int n = 123456789;
float f = n; // f is 1.23456792E8
```

When two values are combined with a binary operator (such as $n + f$ where n is an integer and f is a floating-point value), both operands are converted to a common type before the operation is carried out.

- If either of the operands is of type `double`, the other one will be converted to a `double`.
- Otherwise, if either of the operands is of type `float`, the other one will be converted to a `float`.
- Otherwise, if either of the operands is of type `long`, the other one will be converted to a `long`.
- Otherwise, both operands will be converted to an `int`.

3.5.4. Casts

In the preceding section, you saw that `int` values are automatically converted to `double` values when necessary. On the other hand, there are obviously times when you want to consider a `double` as an integer. Numeric conversions are possible in Java, but of course information may be lost. Conversions in which loss of information is possible are done by means of *casts*. The syntax for casting is to give the target type in parentheses, followed by the variable name. For example:

```
double x = 9.997;  
int nx = (int) x;
```

Now, the variable `nx` has the value 9 because casting a floating-point value to an integer discards the fractional part.

If you want to *round* a floating-point number to the *nearest* integer (which in most cases is a more useful operation), use the `Math.round` method:

```
double x = 9.997;  
int nx = (int) Math.round(x);
```

Now the variable `nx` has the value 10. You still need to use the cast `(int)` when you call `round`. The reason is that the return value of the `round` method is a `long`, and a `long` can only be assigned to an `int` with an explicit cast because there is the possibility of information loss.



Caution: If you try to cast a number of one type to another that is out of range for the target type, the result will be a truncated number that has a different value. For example, `(byte) 300` is actually 44.



Preview: Safe casts are a preview feature of Java 25. The syntax is as follows:

```
if (n instanceof byte b) . . .
```

If `n` fits into a byte without loss, then `b` is set to `(byte) n`.

3.5.5. Assignment

There is a convenient shortcut for using binary operators in an assignment. For example, the *compound assignment operator*

```
x += 4;
```

is equivalent to

```
x = x + 4;
```

(In general, place the operator to the left of the `=` sign, such as `*=` or `%=`.)



Caution: If a compound assignment operator yields a value whose type is different from that of the left-hand side, then it is coerced to fit. For example, if `x` is an `int`, then the statement

```
x += 3.5;
```

is valid. It sets `x` to `(int)(x + 3.5)`, that is, `x + 3`, with no warning!

As of Java 20, you get a warning if you compile with the `-Xlint:lossy-conversions` command line option, like this:

```
javac -Xlint:lossy-conversions MyProg.java
```

Note that in Java, an assignment is an *expression*. That is, it has a value—namely, the value that is being assigned. You can use that value—for example, to assign it to another variable. Consider these statements:

```
int x = 1;  
int y = x += 4;
```

The value of `x += 4` is 5, since that's the value that is being assigned to `x`. Next, that value is assigned to `y`.

Many programmers find such nested assignments confusing and prefer to write them more clearly, like this:

```
int x = 1;  
x += 4;  
int y = x;
```

3.5.6. Increment and Decrement Operators

Programmers, of course, know that one of the most common operations with a numeric variable is to add or subtract 1. Java, following in the footsteps of C and C++, has both increment and decrement operators: `n++` adds 1 to the current value of the variable `n`, and `n--` subtracts 1 from it. For example, the code

```
int n = 12;  
n++;
```

changes `n` to 13. Since these operators change the value of a variable, they cannot be applied to numbers themselves. For example, `4++` is not a legal statement.

There are two forms of these operators; you've just seen the postfix form of the operator that is placed after the operand. There is also a prefix form, `++n`. Both change the value of the variable by 1. The difference between the two appears only when they are used inside expressions. The prefix form does the addition first; the postfix form evaluates to the old value of the variable.

```
int m = 7;
int n = 7;
int a = 2 * ++m; // now a is 16, m is 8
int b = 2 * n++; // now b is 14, n is 8
```

Many programmers find this behavior confusing. In Java, using ++ inside expressions is uncommon.

3.5.7. Relational and boolean Operators

Java has the full complement of relational operators. To test for equality, use a double equal sign, ==. For example, the value of

```
3 == 7
```

is false.

Use a != for inequality. For example, the value of

```
3 != 7
```

is true.

Finally, you have the usual < (less than), > (greater than), <= (less than or equal), and >= (greater than or equal) operators.

Java, following C++, uses && for the logical “and” operator and || for the logical “or” operator. As you can easily remember from the != operator, the exclamation point ! is the logical negation operator. The && and || operators are evaluated in “short-circuit” fashion: The second operand is not evaluated if the first operand already determines the value. If you combine two expressions with the && operator,

```
expression1 && expression2
```

and the truth value of the first expression has been determined to be false, then it is impossible for the result to be true. Thus, the value for the second expression is *not* calculated. This behavior can be exploited to avoid errors. For example, in the expression

```
x != 0 && 1 / x > x + y // no division by 0
```

the second operand is never evaluated if x equals zero. Thus, 1 / x is not computed if x is zero, and no divide-by-zero error can occur.

Similarly, the value of *expression*₁ || *expression*₂ is automatically true if the first expression is true, without evaluating the second expression.

3.5.8. The Conditional Operator

Java provides the *conditional* ?: operator that selects a value, depending on a Boolean expression. The expression

condition ? expression₁ : expression₂

evaluates to the first expression if the condition is true, and to the second expression otherwise. For example,

```
x < y ? x : y
```

gives the smaller of x and y.

3.5.9. Switch Expressions

If you need to choose among more than two values, then you can use a switch expression, which was introduced in Java 14. It looks like this:

```
String seasonName = switch (seasonCode) {  
    case 0 -> "Spring";  
    case 1 -> "Summer";  
    case 2 -> "Fall";  
    case 3 -> "Winter";  
    default -> "???";  
};
```

The expression following the switch keyword is called the *selector expression*, and its value is the *selector*. For now, we only consider selectors and case labels that are numbers, strings, or constants of an enumerated type. In [Chapter 5](#), you will see how to use switch expressions with other types for *pattern matching*.



Note: The switch expression, like every expression, has a value. Note the -> arrow preceding the value in each branch.



Note: As of Java 14, there are *four* forms of switch. This section focuses on the most useful one. See [Section 3.8.5](#) for a thorough discussion of all forms of switch expressions and statements.



Preview: As a preview feature since Java 23, the switch selector can have type float, double, long, or boolean. These selector types were previously invalid.

A case label must be a compile-time constant whose type matches the selector type. You can provide multiple labels for each case, separated by commas:

```
int numLetters = switch (seasonName) {  
    case "Spring", "Summer", "Winter" -> 6;  
    case "Fall" -> 4;  
    default -> -1;  
};
```

When you use the switch expression with enumerated constants, you need not supply the name of the enumeration in each label—it is deduced from the switch value. For example:

```
enum Size { SMALL, MEDIUM, LARGE, EXTRA_LARGE };
...
Size itemSize = ...;
String label = switch (itemSize) {
    case SMALL -> "S"; // no need to use Size.SMALL
    case MEDIUM -> "M";
    case LARGE -> "L";
    case EXTRA_LARGE -> "XL";
};
```

In the example, it was legal to omit the default since there was a case for each possible value.



Caution: When the selector is an enum, and you don't have cases for all constants, you need a default. A switch expression with a numeric or String selector must always have a default.



Caution: If the selector is null, a `NullPointerException` is thrown. If you want to avoid this possibility, add a case null, like this:

```
String label = switch (itemSize) {
    ...
    case null -> "???";
};
```

This is a feature of Java 21. Note that default does *not* match null!

3.5.10. Bitwise Operators

For any of the integer types, you have operators that can work directly with the bits that make up the integers. This means that you can use masking techniques to get at individual bits in a number. The bitwise operators are

& ("and") | ("or") ^ ("xor") ~ ("not")

These operators work on bit patterns. For example, if *n* is an integer variable, then

```
int fourthBitFromRight = (n & 0b1000) / 0b1000;
```

gives you a 1 if the fourth bit from the right in the binary representation of *n* is 1, and 0 otherwise. Using & with the appropriate power of 2 lets you mask out all but a single bit.



Note: When applied to boolean values, the & and | operators yield a boolean value. These operators are similar to the && and || operators, except that the & and | operators are not

evaluated in “short-circuit” fashion—that is, both operands are evaluated before the result is computed.

There are also `>>` and `<<` operators which shift a bit pattern right or left. These operators are convenient when you need to build up bit patterns to do bit masking:

```
int fourthBitFromRight = (n & (1 << 3)) >> 3;
```

Finally, a `>>>` operator fills the top bits with zero, unlike `>>` which extends the sign bit into the top bits. There is no `<<<` operator.



Caution: The right-hand operand of the shift operators is reduced modulo 32 (unless the left-hand operand is a long, in which case the right-hand operand is reduced modulo 64). For example, the value of `1 << 35` is the same as `1 << 3` or `8`.



Note: In C and C++, there is no guarantee as to whether `>>` performs an arithmetic shift (extending the sign bit) or a logical shift (filling in with zeroes). Implementors are free to choose whichever is more efficient. That means the `>>` operator may yield implementation-dependent results for negative numbers. Java removes that uncertainty.



Note: The `Integer` class has a number of methods for bit-level operations. For example, `Integer.bitCount(n)` yields the number of bits that are 1 in the binary representation of `n`, and `Integer.reverse(n)` yields the number obtained by reversing the bits of `n`. Not many programmers need bit-level operations, but if you do, have a look at the `Integer` class to see whether there is a method for the task that you need to accomplish.

3.5.11. Parentheses and Operator Hierarchy

[Table 3.4](#) shows the precedence of operators. If no parentheses are used, operations are performed in the hierarchical order indicated. Operators on the same level are processed from left to right, except for those that are right-associative, as indicated in the table. For example, `&&` has a higher precedence than `||`, so the expression

```
a && b || c
```

means

```
(a && b) || c
```

Since `+=` associates right to left, the expression

```
a += b += c
```

means

```
a += (b += c)
```

That is, the value of `b += c` (which is the value of `b` after the addition) is added to `a`.

Table 3.4: Operator Precedence

Operators	Associativity
<code>[] . ()</code> (method call)	Left to right
<code>! ~ ++ -- +</code> (unary) <code>-</code> (unary) <code>()</code> (cast) <code>new</code>	Right to left
<code>*</code> <code>/</code> <code>%</code>	Left to right
<code>+</code> <code>-</code>	Left to right
<code><<</code> <code>>></code> <code>>>></code>	Left to right
<code><</code> <code><=</code> <code>></code> <code>>=</code> <code>instanceof</code>	Left to right
<code>==</code> <code>!=</code>	Left to right
<code>&</code>	Left to right
<code>^</code>	Left to right
<code> </code>	Left to right
<code>&&</code>	Left to right
<code> </code>	Left to right
<code>?:</code>	Right to left
<code>=</code> <code>+=</code> <code>-=</code> <code>*=</code> <code>/=</code> <code>%=</code> <code>&=</code> <code> =</code> <code>^=</code> <code><<=</code> <code>>>=</code> <code>>>>=</code>	Right to left



Note: Some programming languages (such as C++ and JavaScript) have a *comma operator* that evaluates one expression (only for its side effect), then another. Java does not have such an operator. However, you can use a *comma-separated list of expressions* in the first and third slot of a `for` statement (see [Section 3.8.4](#)).

3.6. Strings

Conceptually, Java strings are sequences of Unicode characters. As you have seen in [Section 3.3.4](#), the concept of what exactly a character is has become complicated. And the encoding of the characters into `char` values has also become complicated.

However, most of the time, you don't care. You get strings from string literals or from methods, and you operate on them with methods of the `String` class. The following sections cover the details.



Note: You have already seen string *literals* such as "Hello, World!", which are instances of the String class.

To include “complicated” characters in string literals, be sure that you use the UTF-8 encoding for source files (which is the default for most IDEs). Then you can just paste them from web pages, and produce string literals such as "Ahoy 🏴‍☠️".

In the past, programmers were more concerned that their collaborators might use a different file encoding, and instead provided escape sequences for the UTF-16 encoding: "Ahoy \uD83C\uDFF4\u200D\u2620\uFE0F".

3.6.1. Concatenation

Java, like most programming languages, allows you to use + to join (concatenate) two strings.

```
String expletive = "Expletive";
String PG13 = "deleted";
String message = expletive + PG13;
```

The preceding code sets the variable message to the string "Expletivedeleted". (Note the lack of a space between the words: The + operator joins two strings in the order received, *exactly* as they are given.)

When you concatenate a string with a value that is not a string, the latter is converted to a string. For example,

```
int age = 13;
String rating = "PG" + age;
```

sets rating to the string "PG13".

This feature is commonly used in output statements. For example,

```
IO.println("The answer is " + answer);
```

is perfectly acceptable and prints what you would expect (and with correct spacing because of the space after the word is).



Caution: Beware of string concatenations with expressions that have a + operator, such as:

```
int age = 42;
String output = "Next year, you'll be " + age + 1 + "."; // ERROR
```

Because the + operators are evaluated from left to right, the result is "Next year, you'll be 421.". The remedy is to use parentheses:

```
String output = "Next year, you'll be " + (age + 1) + "."; // OK
```



Caution: Concatenation only works with strings, not char literals. For example, the expression `':' + 8080` is *not* a string, but the integer 8138. (The colon character has Unicode value 58.)

If you need to put multiple strings together, separated by a delimiter, use the `join` method:

```
String all = String.join(" / ", "S", "M", "L", "XL");  
// all is the string "S / M / L / XL"
```

The `repeat` method produces a string that repeats a given string a number of times:

```
String repeated = "Java".repeat(3); // repeated is "JavaJavaJava"
```

3.6.2. Static and Instance Methods

At the end of the preceding section, you saw two methods of the `String` class, `join` and `repeat`. There is a crucial difference between these two methods. When you call

```
String all = String.join(" / ", "S", "M", "L", "XL");
```

you provide all arguments that the method needs inside the parentheses. Contrast this with the call

```
String repeated = "Java".repeat(3);
```

To compute the repetition of a string, two pieces of information are required: the string itself, and the number of times that it should be repeated.

Note that the string is written before the name of the method, with a dot (`.`) separating the two. The `repeat` method is an example of an *instance* method. As you will see in [Chapter 4](#), an instance method has one special argument; in this case, a string. That value precedes the method name. Supplementary arguments are provided after the method name in parentheses.

The `String.join` method, on the other hand, is a *static* method. It doesn't have a special argument. The dot serves a different function, separating the name of the class in which the method is declared from the method name.

To tell the two apart, locate the dot. Is it preceded by a value (such as the string `"Java"`)? Then you are looking at the call to an instance method. Or is it preceded by the name of a class (such as `String`)? Then it is a static method.

Many of the methods that you have seen so far, including `IO.println`, `Integer.parseInt`, and `Math.sqrt`, are static methods. However, as you learn more about Java, you will mostly use instance methods.



Note: The choice between static and instance methods may feel arbitrary at times. For example, why do we call `Integer.parseInt("42")` and not `"42".parseInt()`? The designers of

the Java API had to decide. They preferred that the conversion of strings to integers should be the responsibility of the Integer class, and not the String class.

3.6.3. Indexes and Substrings

Java strings are sequences of char values. As you saw in [Section 3.3.4](#), the char data type is used for representing Unicode code points in the UTF-16 encoding. Some characters can be represented with a single char value, but many characters and symbols require more than one char value.



Note: The virtual machine is not required to store strings as sequences of char values. For efficiency, strings that hold only single-byte code units store byte sequences, and all others char sequences. This is an implementation detail that has changed in the past and may again change in the future.

The length instance method yields the number of char values required for a given string. For example:

```
String greeting = "Ahoy 🇵🇸";  
int n = greeting.length(); // is 10
```

The call `s.charAt(n)` returns the char value at position `n`, where `n` is between 0 and `s.length() - 1`. (Like C and C++, Java counts positions in a string starting with 0.) For example:

```
char first = greeting.charAt(0); // first is 65 or 'A'  
char last = greeting.charAt(9); // last is 65039
```

However, these calls are not very useful. The last char value is just a part of the flag symbol, and you won't generally care what these values are.

Still, you sometimes need to know where a substring is located in a string. Use the `indexOf` method:

```
String sub = " ";  
int start = greeting.indexOf(sub); // 4
```

As it happens, the position or *index* of the space is 4, but the exact value doesn't matter. It depends on the characters preceding the substring, and the number of char values needed to encode each of them. Always treat an index as an opaque number, not the count of perceived characters preceding it.

You can compute where the next character starts:

```
int nextStart = start + sub.length(); // 5
```

The string " " has length 1, but do not hard-code the length of a string. Always use the `length` method instead.

You can extract a substring from a larger string with the `substring` method of the `String` class. For example,

```
String greeting = "Hello, World!";
int a = greeting.indexOf(",") + 2; // 7
int b = greeting.indexOf("!"); // 12
String s = greeting.substring(a, b);
```

creates a string consisting of the characters "World".

The second argument of `substring` is the first position that you *do not* want to copy. In our case, we copy everything from the beginning up to, but not including, the comma.

Note that the string `s.substring(a, b)` always has length $b - a$. For example, the substring "World" has length $12 - 7 = 5$.

3.6.4. Strings Are Immutable

The `String` class gives no methods that let you *change* a character in an existing string. If you want to turn `greeting` into "Help!", you cannot directly change the last positions of `greeting` into 'p' and '!'. If you are a C programmer, this can make you feel pretty helpless. How are we going to modify the string? In Java, it is quite easy: Concatenate the substring that you want to keep with the characters that you want to replace.

```
String greeting = "Hello";
int n = greeting.indexOf("lo");
greeting = greeting.substring(0, n) + "p!";
```

This declaration changes the current value of the `greeting` variable to "Help!".

Since you cannot change the individual characters in a Java string, the documentation refers to the objects of the `String` class as *immutable*. Just as the number 3 is always 3, the string "Hello" will always contain the code-unit sequence for the characters H, e, l, l, o. You cannot change these values. Yet you can, as you just saw, change the contents of the string *variable* `greeting` and make it refer to a different string, just as you can make a numeric variable currently holding the value 3 hold the value 4.

Isn't that a lot less efficient? It would seem simpler to change the characters than to build up a whole new string from scratch. Well, yes and no. Indeed, it is some amount of work to generate a new string that holds the concatenation of "Hel" and "p!". But immutable strings have one great advantage: The compiler can arrange that strings are *shared*.

To understand how this works, think of the various strings as sitting in a common pool. String variables then point to locations in the pool. If you copy a string variable, both the original and the copy share the same characters.

Overall, the designers of Java decided that the efficiency of sharing outweighs the inefficiency of string creation. Look at your own programs; most of the time, you probably don't change strings—you just compare them. (There is one common exception—assembling strings from individual characters or from shorter strings that come from the keyboard or a file. For these situations, Java provides a separate class—see [Section 3.6.9](#).)

3.6.5. Testing Strings for Equality

To test whether two strings are equal, use the `equals` method. The expression

```
s.equals(t)
```

returns `true` if the strings `s` and `t` are equal, `false` otherwise. Note that `s` and `t` can be string variables or string literals. For example, the expression

```
"Hello".equals(greeting)
```

is perfectly legal. To test whether two strings are identical except for the upper/lowercase letter distinction, use the `equalsIgnoreCase` method.

```
"Hello".equalsIgnoreCase("hello")
```

Do *not* use the `==` operator to test whether two strings are equal! It only determines whether or not the strings are stored in the same location. Sure, if strings are in the same location, they must be equal. But it is entirely possible to store multiple copies of identical strings in different places.

```
String greeting = "Hello"; // initialize greeting to a string
greeting == "Hello" // true
greeting.substring(0, greeting.indexOf("l")) == "He" // false
greeting.substring(0, greeting.indexOf("l")).equals("He") // true
```

If the virtual machine always arranges for equal strings to be shared, then you could use the `==` operator for testing equality. But only string *literals* are shared, not strings that are computed at runtime. Therefore, *never* use `==` to compare strings. Always use `equals` instead.



Caution: In most programming languages, such as Python, JavaScript, or C++, the `==` operator compares strings by their content. If you come from one of those languages, be particularly careful about string comparisons.

3.6.6. Empty and Null Strings

The empty string `""` is a string of length 0. You can test whether a string is empty by calling

```
if (str.length() == 0)
```

or

```
if (str.equals(""))
```

or, for optimum efficiency

```
if (str.isEmpty())
```

An empty string is a Java object which holds the string length (namely, 0) and an empty contents. However, a `String` variable can also hold a special value, called `null`, that indicates that no object is currently associated with the variable. To test whether a string is `null`, use

```
if (str == null)
```

Sometimes, you need to test that a string is neither null nor empty. Then use

```
if (str != null && str.length() != 0)
```

You need to test that `str` is not null first. As you will see in [Chapter 4](#), it is an error to invoke a method on a null value.

3.6.7. The String API

The `String` class in Java contains close to 100 methods. The following API note summarizes the most useful ones.

These API notes, found throughout the book, will help you understand the Java Application Programming Interface (API). Each API note starts with the name of a class, such as `java.lang.String`. (The significance of the so-called *package* name `java.lang` is explained in [Chapter 4](#).) The class name is followed by the names, explanations, and parameter descriptions of one or more methods. A *parameter variable* of a method is the variable that receives a method argument. For example, as you will see in the first API note below, the `charAt` method has a parameter called `index` of type `int`. If you call the method, you supply an argument of that type, such as `str.charAt(0)`.

The API notes do not list all methods of a particular class but present the most commonly used ones in a concise form. For a full listing, consult the online documentation (see [Section 3.6.8](#)).

The number following the class name is the JDK version number in which it was introduced. If a method has been added later, it has a separate version number.

java.lang.String 1.0

- `char charAt(int index)`
returns the code unit at the specified location. You probably don't want to call this method unless you are interested in low-level code units.
- `int length()`
returns the number of code units of the string.
- `boolean equals(Object other)`
returns true if the string equals other.
- `boolean equalsIgnoreCase(String other)`
returns true if the string equals other, except for upper/lowercase distinction.
- `int compareTo(String other)`
returns a negative value if the string comes before other in dictionary order, a positive value if the string comes after other in dictionary order, or 0 if the strings are equal.
- `boolean isEmpty()` **6**
`boolean isBlank()` **11**
return true if the string is empty or consists of whitespace.
- `boolean startsWith(String prefix)`
- `boolean endsWith(String suffix)`
return true if the string starts with prefix or ends with suffix.

- `int indexOf(String str)`
- `int indexOf(String str, int fromIndex)`
- `int indexOf(String str, int fromIndex, int toIndex)` **21**
return the start of the first substring equal to the string `str`, starting at index 0 or at `fromIndex`, and ending at the end of the string or at `toIndex`. Return -1 if `str` does not occur in this string or the specified substring.
- `int lastIndexOf(String str)`
- `int lastIndexOf(String str, int fromIndex)`
return the start of the last substring equal to the string `str`, starting at the end of the string or at `fromIndex`, or -1 if `str` does not occur.
- `String replace(CharSequence oldString, CharSequence newString)`
returns a new string that is obtained by replacing all substrings matching `oldString` in the string with the string `newString`. You can supply `String` or `StringBuilder` arguments for the `CharSequence` parameters.
- `String substring(int beginIndex)`
- `String substring(int beginIndex, int endIndex)`
return a new string consisting of all code units from `beginIndex` until the end of the string or until `endIndex-1`.
- `String toLowerCase()`
- `String toUpperCase()`
return a new string containing all characters in the original string, with uppercase characters converted to lowercase, or lowercase characters converted to uppercase.
- `String strip()` **11**
`String stripLeading()` **11**
`String stripTrailing()` **11**
return a new string by eliminating leading and trailing, or just leading or trailing whitespace in the original string. Use these methods instead of the archaic `trim` method that eliminates characters \leq U+0020.
- `String join(CharSequence delimiter, CharSequence... elements)` **8**
returns a new string joining all elements with the given delimiter.
- `String repeat(int count)` **11**
returns a string that repeats this string count times.



Note: In the API notes, there are a few parameters of type `CharSequence`. This is an *interface* type to which all strings belong. You will learn about interface types in [Chapter 6](#). For now, you just need to know that you can pass arguments of type `String` whenever you see a `CharSequence` parameter.

3.6.8. Reading the Online API Documentation

As you just saw, the `String` class has lots of methods. Furthermore, there are thousands of classes in the standard libraries, with many more methods. It is plainly impossible to remember all useful classes and methods. Therefore, it is essential that you become familiar with the online API documentation that lets you look up all classes and methods in the standard library. You can download the API documentation from Oracle and save it locally, or you can point your browser to <https://docs.oracle.com/en/java/javase/25/docs/api>.

The API documentation has a search box (see [Figure 3.2](#)). Older versions have frames with lists of packages and classes. You can still get those lists by clicking on the Frames menu item. For example, to get more information on the methods of the String class, type “String” into the search box and select the type `java.lang.String`, or locate the link in the frame with class names and click it. You get the class description, as shown in [Figure 3.3](#).

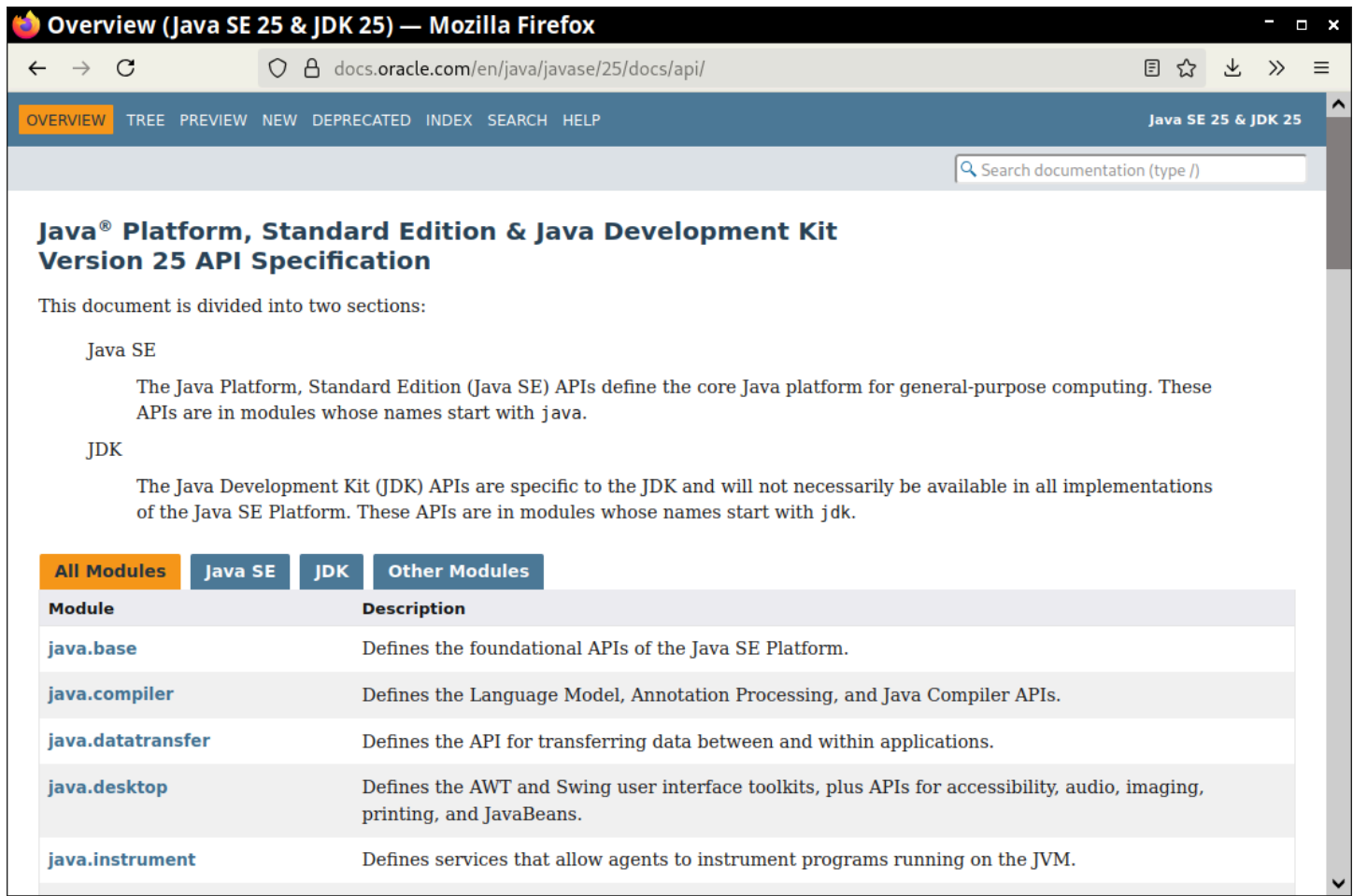


Figure 3.2: The Java API documentation

When you scroll down, you reach a summary of all methods, sorted in alphabetical order (see [Figure 3.4](#)). Click on any method name for a detailed description of that method (see [Figure 3.5](#)). For example, if you click on the `compareToIgnoreCase` link, you’ll get the description of the `compareToIgnoreCase` method.



Tip: If you have not already done so, download the JDK documentation, as described in [Chapter 2](#). Bookmark the `index.html` page of the documentation in your browser right now!

You can also add a new search engine to your browser with the query string

`https://docs.oracle.com/en/java/javase/25/docs/api/search.html?q=%s`

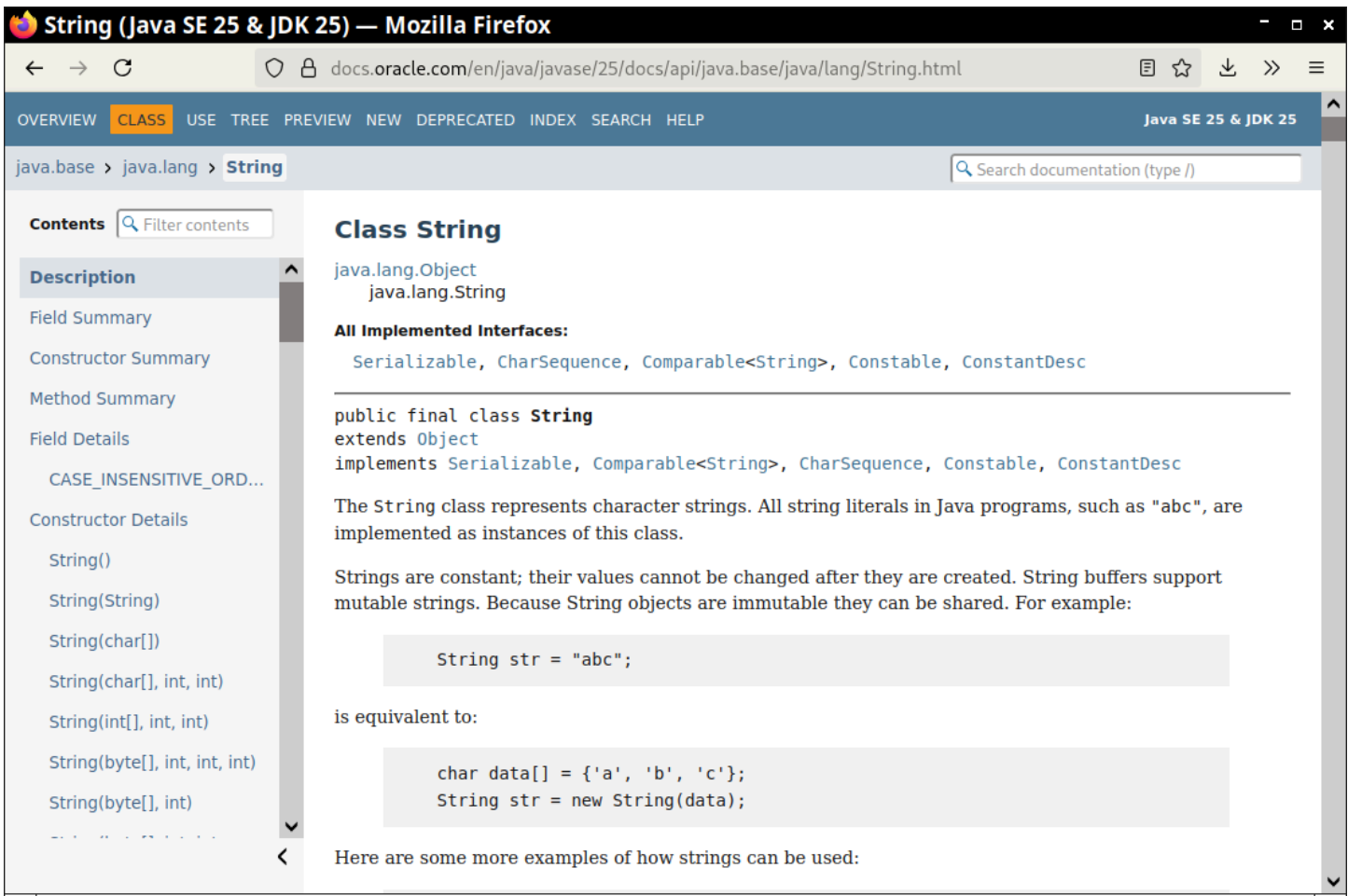


Figure 3.3: Class description for the `String` class

3.6.9. Building Strings

Occasionally, you need to build up strings from shorter strings, such as keystrokes or words from a file. It would be inefficient to use string concatenation for this purpose. Every time you concatenate strings, a new `String` object is constructed. This is time consuming and wastes memory. Using the `StringBuilder` class avoids this problem.

Follow these steps if you need to build a string from many small pieces. First, construct an empty string builder:

```
StringBuilder builder = new StringBuilder();
```

You can also provide initial content:

```
StringBuilder builder = new StringBuilder("INVOICE\n");
```

Each time you need to add another part, call the `append` method.

```
builder.append(str); // appends a string
builder.appendCodePoint(cp); // appends a single code point
```

The latter method is occasionally useful when you need to compute a code point. Here is an example. Flag emojis are made up of two code points, each in the range between 127462

The screenshot shows the Java SE 25 & JDK 25 API documentation for the `String` class. The left sidebar contains a 'Contents' section with a search bar and a list of sections: Description, Field Summary, Constructor Summary, Method Summary (selected), Field Details, Constructor Details, and a list of constructors. The main area displays the 'Method Summary' table, which is filtered to show 'All Methods'. The table has three columns: 'Modifier and Type', 'Method', and 'Description'. It lists various methods including `charAt`, `chars`, `codePointAt`, `codePointBefore`, `codePointCount`, `codePoints`, `compareTo`, `compareToIgnoreCase`, `concat`, and `contains`.

Modifier and Type	Method	Description
char	<code>charAt(int index)</code>	Returns the char value at the specified index.
<code>IntStream</code>	<code>chars()</code>	Returns a stream of int zero-extending the char values from this sequence.
int	<code>codePointAt(int index)</code>	Returns the character (Unicode code point) at the specified index.
int	<code>codePointBefore(int index)</code>	Returns the character (Unicode code point) before the specified index.
int	<code>codePointCount(int beginIndex, int endIndex)</code>	Returns the number of Unicode code points in the specified text range of this String.
<code>IntStream</code>	<code>codePoints()</code>	Returns a stream of code point values from this sequence.
int	<code>compareTo(String anotherString)</code>	Compares two strings lexicographically.
int	<code>compareToIgnoreCase(String str)</code>	Compares two strings lexicographically, ignoring case differences.
<code>String</code>	<code>concat(String str)</code>	Concatenates the specified string to the end of this string.
boolean	<code>contains(CharSequence s)</code>	Returns true if and only if this string contains the specified sequence of char values.

Figure 3.4: Method summary of the `String` class

(regional indicator symbol letter A) to 127487 (regional indicator symbol letter Z). Now suppose you have a country string such as "IT". Then you can compute the code points as follows:

```
final int REGIONAL_INDICATOR_SYMBOL_LETTER_A = 127462;
String country = "...";
builder.appendCodePoint(country.charAt(0) - 'A' + REGIONAL_INDICATOR_SYMBOL_LETTER_A);
builder.appendCodePoint(country.charAt(1) - 'A' + REGIONAL_INDICATOR_SYMBOL_LETTER_A);
```

When you are done building the string, call the `toString` method. You will get a `String` object with the character sequence contained in the builder.

```
String completedString = builder.toString();
```

Cleverly, the `StringBuilder` methods return the builder object, so that you can chain multiple method calls:

```
String completedString = new StringBuilder()
    .append(str)
    .appendCodePoint(cp)
    .toString();
```

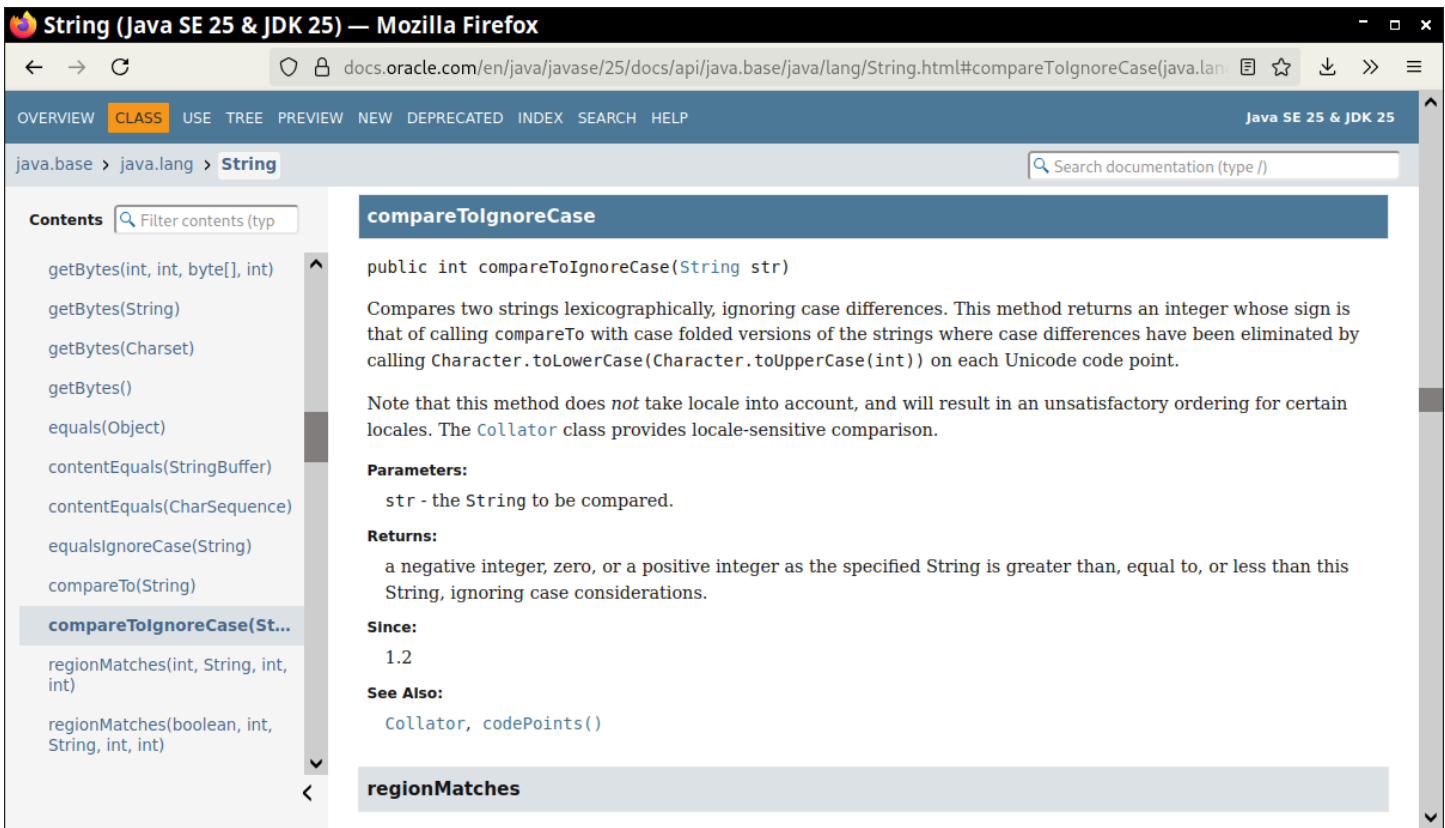



Figure 3.5: Detailed description of a String method

The `String` class doesn't have a method to reverse the Unicode characters of a string, but `StringBuilder` does. To reverse a string, use this code snippet:

```
String reversed = new StringBuilder(original).reverse().toString();
```



Caution: Reversing works correctly for characters that are encoded with two char values, but it fails when a symbol is composed of multiple code points. For example, reversing a string containing the pirate flag described in [Section 3.3.4](#) does not preserve the flag.



Note: The legacy `StringBuffer` class is less efficient than `StringBuilder`, but it allows multiple threads to add or remove characters. If all string editing happens in a single thread (which is usually the case), you should use `StringBuilder`. The APIs of both classes are identical.

The following API notes contain the most important methods for the `StringBuilder` class.

`java.lang.StringBuilder` 5.0

- `StringBuilder()`
- `StringBuilder(CharSequence seq)`
constructs an empty string builder, or one with the given initial content.

- `int length()`
returns the number of code units of the builder or buffer.
- `StringBuilder append(String str)`
appends a string and returns the string builder.
- `StringBuilder appendCodePoint(int cp)`
appends a code point, converting it into one or two code units, and returns this.
- `StringBuilder insert(int offset, String str)`
inserts a string at position `offset` and returns the string builder.
- `StringBuilder delete(int startIndex, int endIndex)`
deletes the code units with offsets `startIndex` to `endIndex- 1` and returns the string builder.
- `StringBuilder repeat(CharSequence cs, int count)` **21**
Appends `count` copies of `cs` and returns the string builder.
- `StringBuilder reverse()`
Reverses the code points in this string builder and returns the builder.
- `String toString()`
returns a string with the same data as the builder or buffer contents.

3.6.10. Text Blocks

The text block feature, added in Java 15, makes it easy to provide string literals that span multiple lines. A text block starts with `"""`, followed by a line feed. The block ends with another `"""`:

```
String greeting = """
Hello
World
""";
```

A text block is easier to read and write than the equivalent string literal:

```
"Hello\nWorld\n"
```

This string contains two `\n`: one after `Hello` and one after `World`. The newline after the opening `"""` is not included in the string literal.

If you don't want a newline after the last line, put the closing `"""` immediately after the last character:

```
String prompt = """
Hello, my name is Hal.
Please enter your name:""";
```

Text blocks are particularly suited for including code in some other language, such as SQL or HTML. You can just paste it between the triple quotes:

```
String html = """
<div class="Warning">
  Beware of those who say "Hello" to the world
</div>
""";
```

All escape sequences from regular strings work the same way in text blocks.

Note that you don't have to use escape sequences with the quotation marks around `Hello`. There are just two situations where you need to use the `\` escape sequence in a text block:

- If the text block *ends* in a quotation mark
- If the text block contains a sequence of three or more quotation marks

Unfortunately, you still need the escape sequence `\\` to denote a backslash in a text block.

There is one escape sequence that only works in text blocks. A `\` directly before the end of a line joins this line and the next. For example,

```
"""
Hello, my name is Hal. \
Please enter your name:"";
```

is the same as

```
"Hello, my name is Hal. Please enter your name:""
```

Line endings are normalized by removing trailing whitespace and changing any Windows line endings (`\r\n`) to simple newlines (`\n`). If you need to preserve trailing spaces, turn the last one into a `\s` escape. In fact, that's what you probably want for prompt strings. The following string ends in a space:

```
"""
Hello, my name is Hal. \
Please enter your name:\s""";
```

The story is more complex for leading whitespace. Consider a typical variable declaration that is indented from the left margin. You can indent the text block as well:

```
String html = """
    <div class="Warning">
        Beware of those who say "Hello" to the world
    </div>
    """;
```

The indentation that is common to all lines in the text block is subtracted. The actual string is

```
"<div class=\"Warning\">\n  Beware of those who say \"Hello\" to the world\n</div>\n"
```

Note that there are no indentations in the first and third lines.

You can always avoid this indentation stripping by having no whitespace in the last line, before the closing `"""`. But many programmers seem to find that it looks neater when text blocks are indented. Your IDE may cheerfully offer to indent all text blocks, using tabs or spaces.

Java wisely does not prescribe the width of a tab. The whitespace prefix has to match *exactly* for all lines in the text block.

Entirely blank lines are not considered when stripping common indentation. However, the whitespace before the closing `"""` is significant. Be sure to indent to the end of the whitespace that you want to have stripped.



Caution: Be careful about mixed tabs and spaces in indentations. An overlooked space can easily yield a wrongly indented string.



Tip: If a text block contains code that isn't Java, you may actually prefer to place it at the left margin. It stands out from the Java code, and you have more room for long lines.

3.7. Input and Output

To make our example programs more interesting, we want to accept input and properly format the program output. Of course, modern programs use a GUI for collecting user input. However, programming such an interface requires more tools and techniques than we have at our disposal at this time. Our first order of business is to become more familiar with the Java programming language, so we use the humble console for input and output.

3.7.1. Reading Input

You saw that it is easy to print output to the console window just by calling `IO.println`. Reading from the console is just as simple.

The `readln` method reads one line of input and returns it as a string value. You can optionally pass a prompt string as an argument.

```
String name = IO.readln("What is your name? ");
```

To read an integer, use the `Integer.parseInt` method to convert the entered string into an integer.

```
int age = Integer.parseInt(IO.readln("How old are you? "));
```

Similarly, the `parseDouble` method converts a string to a floating-point number.

```
double rate = Double.parseDouble(IO.readln("Interest rate: "));
```

The program in [Listing 3.2](#) asks for the user's name and age and then prints a message like

```
Hello, Cay. Next year, you'll be 65.
```

Listing 3.2: InputDemo.java

```
1  /**
2   * This program demonstrates console input.
3   */
4  void main() {
5      // get first input
```

```
6 String name = IO.readLine("What is your name? ");
7
8 // get second input
9 int age = Integer.parseInt(IO.readLine("How old are you? "));
10
11 // display output on console
12 IO.println("Hello, " + name + ". Next year, you'll be " + (age + 1) + ".");
13 }
```



Caution: If you run this program from a Windows terminal, special characters in your name may not show up correctly. By default, Windows terminals use an archaic character encoding. To fix this, switch the terminal to the UTF-8 encoding, by issuing the following command prior to running the program:

```
chcp 65001
```

Then, if you use Java 18 or above, all will be well. With older versions of Java, run the program as:

```
java -Dfile.encoding=utf-8 InputDemo
```

If you use a development environment, you should not have to worry about this issue.



Note: Prior to Java 25, reading console input was not so easy. To use an older version of Java, make these adaptations:

First first construct a Scanner object that is attached to `System.in`:

```
Scanner in = new Scanner(System.in);
```

(Objects, constructors, and the new operator are discussed in detail in [Chapter 4](#).)

The `nextLine` method reads a line of input.

```
System.out.print("What is your name? ");
String name = in.nextLine();
```

To read an integer, use the `nextInt` method.

```
System.out.print("How old are you? ");
int age = in.nextInt();
```

Similarly, the `nextDouble` method reads the next floating-point number.

Finally, include the line

```
import java.util.Scanner;
```

at the beginning of the program, to tell the compiler that the `Scanner` class is defined in the `java.util` package. Packages and import directives are covered in more detail in [Chapter 4](#).



Caution: The `parseInt` and `parseDouble` methods are not intended for parsing user input, and `IO.println` is not intended for presenting numbers to a general audience. They use the number format for decimal Java literals. That's ok for sample programs in a programming book. However, most users expect to see the decimal digits and separators to which they are accustomed.

To parse human input, use the `nextInt` and `nextDouble` methods of the `Scanner` class. For output, use the formatted method that you will see in [Section 3.7.2](#). These methods use the number format of the host system.



Note: The `IO.readLine` method is not suitable for reading a password from a console since the input is plainly visible to anyone. Use the `readPassword` method of the `Console` class to read a password while hiding the user input:

```
String username = System.console().readLine("User name: ");
char[] passwd = System.console().readPassword("Password: ");
...
Arrays.fill(passwd, '*');
```

For security reasons, the password is returned in an array of characters rather than a string. After you are done processing the password, you should immediately overwrite the array elements with a filler value.

java.lang.IO 25

- `println(Object obj)`
Converts the object to a string and prints it on the console, followed by a line separator.
- `print(Object ob)`
Converts the object to a string and prints it on the console without a line separator.
- `println()`
Prints a line separator.
- `String readln(String prompt)`
Prints a prompt on the console and returns one line of user input.
- `String readln()`
Returns one line of user input without printing a prompt.

java.util.Scanner 5.0

- `Scanner(InputStream in)`
constructs a `Scanner` object from the given input stream.
- `String nextLine()`
reads the next line of input.
- `String next()`
reads the next word of input (delimited by whitespace).

- `int nextInt()`
- `double nextDouble()`
read and convert the next character sequence that represents an integer or floating-point number.
- `boolean hasNext()`
tests whether there is another word in the input.
- `boolean hasNextInt()`
- `boolean hasNextDouble()`
test whether the next character sequence represents an integer or floating-point number.

java.lang.System 1.0

- `static Console console() 6`
returns a `Console` object for interacting with the user through a console window if such interaction is possible, null otherwise. A `Console` object is available for any program that is launched in a console window. Otherwise, the availability is system-dependent.

java.io.Console 6

- `char[] readPassword(String prompt, Object... args)`
- `String readLine(String prompt, Object... args)`
display the prompt and read the user input until the end of the input line. The optional args parameters are used to supply formatting arguments, as described in the next section.

3.7.2. Formatting Output

You can print a number `x` to the console with the statement `I0.print(x)`. That command will print `x` with the maximum number of nonzero digits for that type. For example,

```
double x = 10000.0 / 3.0;
I0.print(x);
```

prints

```
3333.3333333333335
```

That is a problem if you want to display, for example, dollars and cents.

The remedy is the formatted method, which follows the venerable conventions from the C library. For example, the call

```
I0.print("%8.2f".formatted(x));
```

prints `x` with a *field width* of 8 characters and a *precision* of 2 characters. That is, the printout contains a leading space and the seven characters

```
3333.33
```

You can supply multiple arguments to `formatted`. For example:

```
I0.print("Hello, %s. Next year, you'll be %d.".formatted(name, age + 1));
```

Each of the *format specifiers* that start with a `%` character is replaced with the corresponding argument. The *conversion character* that ends a format specifier indicates the type of the value to be formatted: `f` is a floating-point number, `s` a string, and `d` a decimal integer. [Table 3.5](#) shows all conversion characters.

The uppercase variants produce uppercase letters. For example, `"%8.2E"` formats `3333.33` as `3.33E+03`, with an uppercase `E`.

Table 3.5: Conversions for `formatted`

Conversion Character	Type	Example
<code>d</code>	Decimal integer	159
<code>x</code> or <code>X</code>	Hexadecimal integer. For more control over hexadecimal formatting, use the <code>HexFormat</code> class.	9f
<code>o</code>	Octal integer	237
<code>f</code> or <code>F</code>	Fixed-point floating-point	15.9
<code>e</code> or <code>E</code>	Exponential floating-point	1.59e+01
<code>g</code> or <code>G</code>	General floating-point (the shorter of <code>e</code> and <code>f</code>)	—
<code>a</code> or <code>A</code>	Hexadecimal floating-point	0x1.fccdp3
<code>s</code> or <code>S</code>	String	Hello
<code>c</code> or <code>C</code>	Character	H
<code>b</code> or <code>B</code>	boolean	true
<code>h</code> or <code>H</code>	Hash code	42628b2
<code>tx</code> or <code>Tx</code>	Legacy date and time formatting. Use the <code>java.time</code> classes instead—see Chapter 6 of Volume II .	—
<code>%</code>	The percent symbol	%
<code>n</code>	The platform-dependent line separator	—



Note: You can use the `s` conversion to format arbitrary objects. If an arbitrary object implements the `Formattable` interface, the object's `formatTo` method is invoked. Otherwise, the `toString` method is invoked to turn the object into a string. The `toString` method is discussed in [Chapter 5](#) and interfaces in [Chapter 6](#).

In addition, you can specify *flags* that control the appearance of the formatted output. [Table 3.6](#) shows all flags. For example, the comma flag adds group separators. That is,

```
IO.println("%,.2f".formatted(10000.0 / 3.0));
```

prints

```
3,333.33
```

You can use multiple flags, for example "%(.2f" to use group separators and enclose negative numbers in parentheses.

Table 3.6: Flags for printf

Flag	Purpose	Example
+	Prints sign for positive and negative numbers.	+3333.33
space	Adds a space before positive numbers.	3333.33
0	Adds leading zeroes.	003333.33
-	Left-justifies field.	3333.33
(Encloses negative numbers in parentheses.	(3333.33)
,	Adds group separators.	3,333.33
# (for f format)	Always includes a decimal point.	3,333.
# (for x or o format)	Adds 0x or 0 prefix.	0xcafe
\$	Specifies the index of the argument to be formatted. For example, %1\$d %1\$x prints the first argument in decimal and hexadecimal.	159 9F
<	Formats the same value as the previous specification. For example, %d %<x prints the same number in decimal and hexadecimal.	159 9F

[Figure 3.6](#) shows a syntax diagram for format specifiers.



Note: Formatting is *locale-specific*. For example, in Germany, the group separator is a period, not a comma. On a computer with a German locale, the call

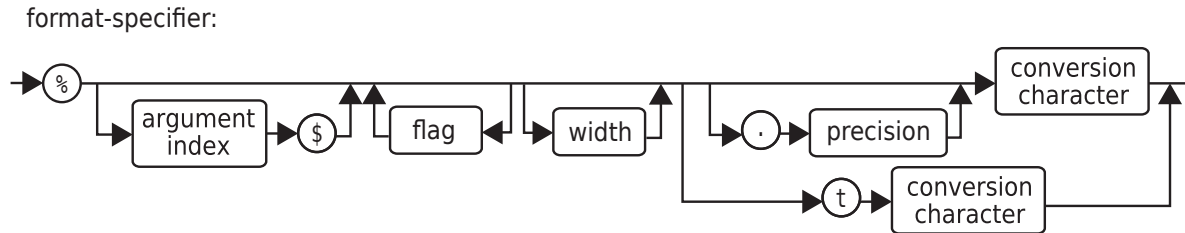


Figure 3.6: Format specifier syntax

```
double x = 10000.0 / 3.0;
IO.print("%8.2f".formatted(x));
```

yields the output

```
3333,33
```

This locale-specific behavior is normally what you want when you communicate with users. However, if you produce a file that is later consumed by a computer program, you may need to choose a fixed locale for the output. Specify the locale as the first argument to the static format method of the `String` class:

```
IO.print(String.format(Locale.US, "%8.2f", x));
```

3.8. Control Flow

Java, like any programming language, supports both conditional statements and loops to determine control flow. I will start with the conditional statements, then move on to loops, to end with a thorough discussion of the four forms of `switch`.



Note: The Java control flow constructs are similar to those in C, C++, or JavaScript. There is no `goto`, but there is a “labeled” version of `break` that you can use to break out of a nested loop (where, in C, you perhaps would have used a `goto`). Finally, there is a variant of the `for` loop that is similar to the range-based `for` loop in C++ and the `for of` loop in JavaScript.

3.8.1. Block Scope

Before learning about control structures, you need to know more about *blocks*.

A block, or compound statement, consists of a number of Java statements, surrounded by a pair of braces. Blocks define the scope of your variables. A block can be *nested* inside another block. Here is a block that is nested inside the block of the `main` method:

```
void main() {
    int n;
    . . .
    {
        int k;
```

```

    . . .
} // k is only defined up to here
}

```

You may not declare identically named local variables in two nested blocks. For example, the following is an error and will not compile:

```

void main() {
    int n;
    . . .
    {
        int k;
        int n; // ERROR--can't redeclare n in inner block
        . . .
    }
}

```



Note: In many programming languages, it is possible to redefine a variable inside a nested block. The inner definition then shadows the outer one. This can be a source of programming errors; hence, Java does not allow it.

3.8.2. Conditional Statements

The conditional statement in Java has the form

```
if (condition) statement
```

The condition must be surrounded by parentheses.

In Java, as in most programming languages, you will often want to execute multiple statements when a single condition is true. In this case, use a *block statement* that takes the form

```

{
    statement1
    statement2
    . . .
}

```

For example:

```

if (yourSales >= target) {
    performance = "Satisfactory";
    bonus = 100;
}

```

In this code all the statements surrounded by the braces will be executed when `yourSales` is greater than or equal to `target` (see [Figure 3.7](#)).

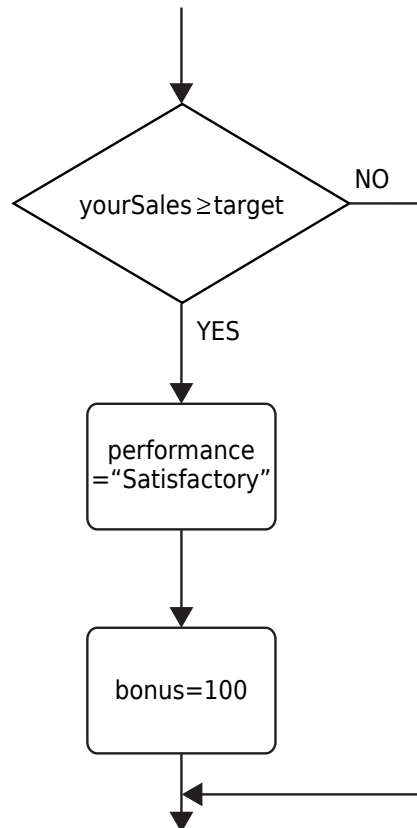


Figure 3.7: Flowchart for the if statement



Note: A block (sometimes called a *compound statement*) enables you to have more than one (simple) statement in any Java programming structure that otherwise allows for a single (simple) statement.

The more general conditional in Java looks like this (see [Figure 3.8](#)):

```
if (condition) statement1 else statement2
```

For example:

```
if (yourSales >= target) {  
    performance = "Satisfactory";  
    bonus = 100 + 0.01 * (yourSales - target);  
}  
else {  
    performance = "Unsatisfactory";  
    bonus = 0;  
}
```

The else part is always optional. An else groups with the closest if. Thus, in the statement

```
if (x <= 0) if (x == 0) sign = 0; else sign = -1;
```

the else belongs to the second if. Of course, it is a good idea to use braces to clarify this code:

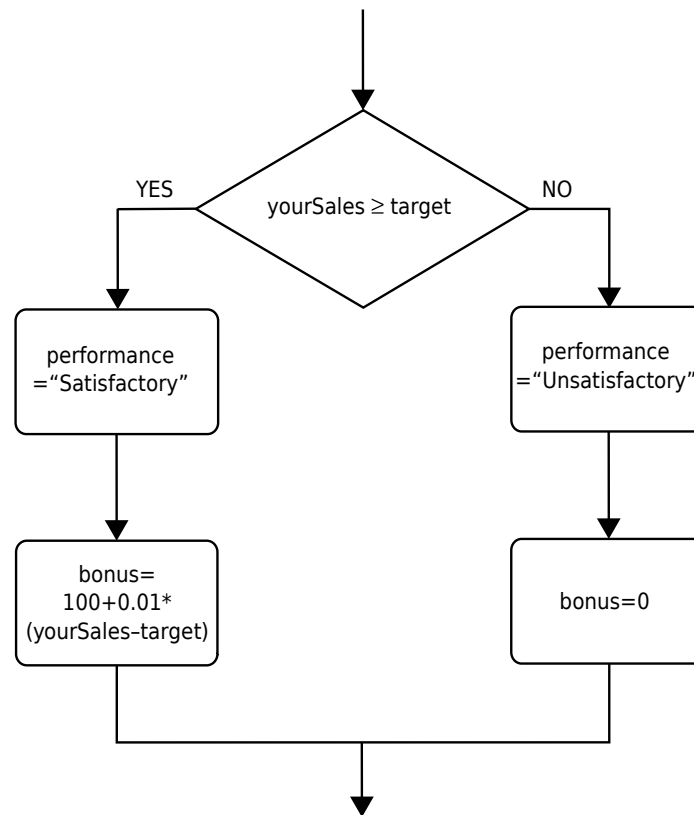


Figure 3.8: Flowchart for the if/else statement

```
if (x <= 0) { if (x == 0) sign = 0; else sign = -1; }
```

Repeated if . . . else if . . . alternatives are common (see [Figure 3.9](#)). For example:

```
if (yourSales >= 2 * target) {
    performance = "Excellent";
    bonus = 1000;
}
else if (yourSales >= 1.5 * target) {
    performance = "Fine";
    bonus = 500;
}
else if (yourSales >= target) {
    performance = "Satisfactory";
    bonus = 100;
}
else {
    IO.println("You're fired");
}
```

3.8.3. Loops

The while loop executes a statement (which may be a block statement) while a condition is true. The general form is

```
while (condition) statement
```

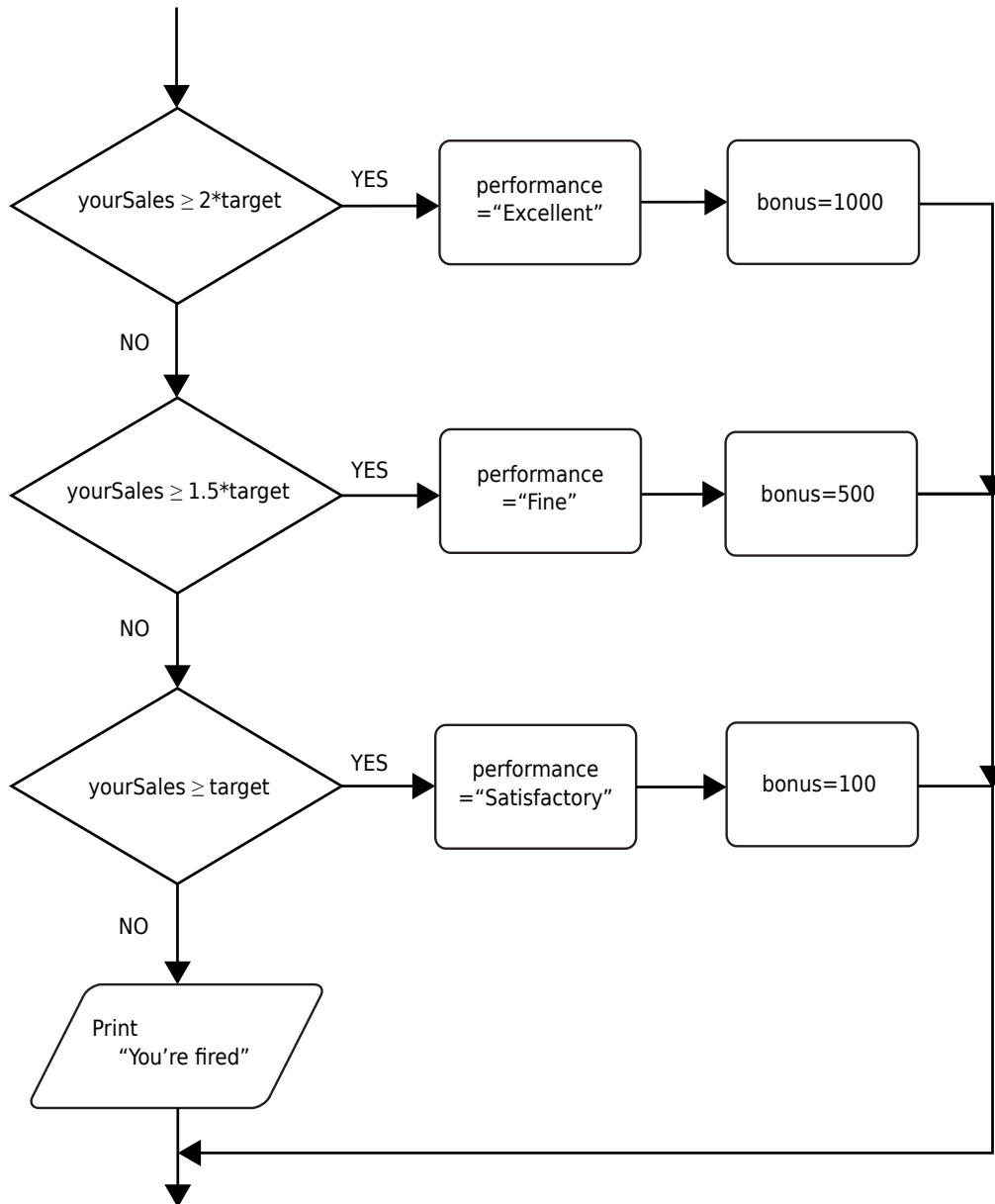


Figure 3.9: Flowchart for the if/else if (multiple branches)

The while loop will never execute if the condition is false at the outset (see [Figure 3.10](#)).

The program in [Listing 3.3](#) determines how long it will take to save a specific amount of money for your well-earned retirement, assuming you deposit the same amount of money per year and the money earns a specified interest rate.

In the example, we are incrementing a counter and updating the amount currently accumulated in the body of the loop until the total exceeds the targeted amount.

```

while (balance < goal) {
    balance += payment;
    double interest = balance * interestRate / 100;
    balance += interest;
    years++;
}
IO.println(years + " years.");

```

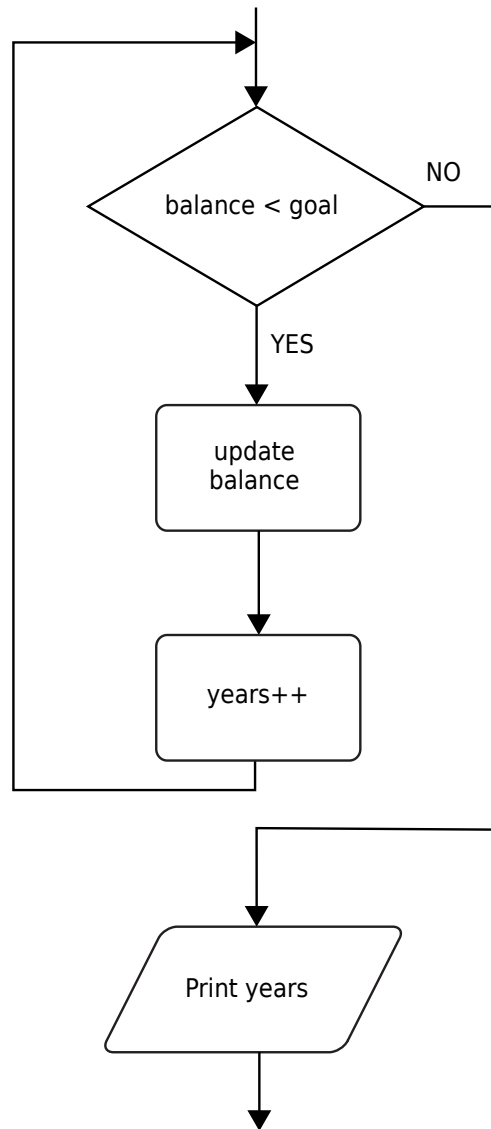


Figure 3.10: Flowchart for the while statement

(Don't rely on this program to plan for your retirement. It lacks a few niceties such as inflation and your life expectancy.)

A while loop tests at the top. Therefore, the code in the block might never be executed. If you want to make sure a block is executed at least once, you need to move the test to the bottom, using the do/while loop. Its syntax looks like this:

```
do statement while (condition);
```

This loop executes the statement (which is typically a block) and only then tests the condition. If it's true, it repeats the statement and retests the condition, and so on. The code in [Listing 3.4](#) computes the new balance in your retirement account and then asks if you are ready to retire:

```
do {  
    balance += payment;  
    double interest = balance * interestRate / 100;  
    balance += interest;  
    years++;  
}
```

```

// print current balance
...
// ask if ready to retire and get input
...
} while (input.equals("N"));

```

As long as the user answers "N", the loop is repeated (see [Figure 3.11](#)). This program is a good example of a loop that needs to be entered at least once, because the user needs to see the balance before deciding whether it is sufficient for retirement.

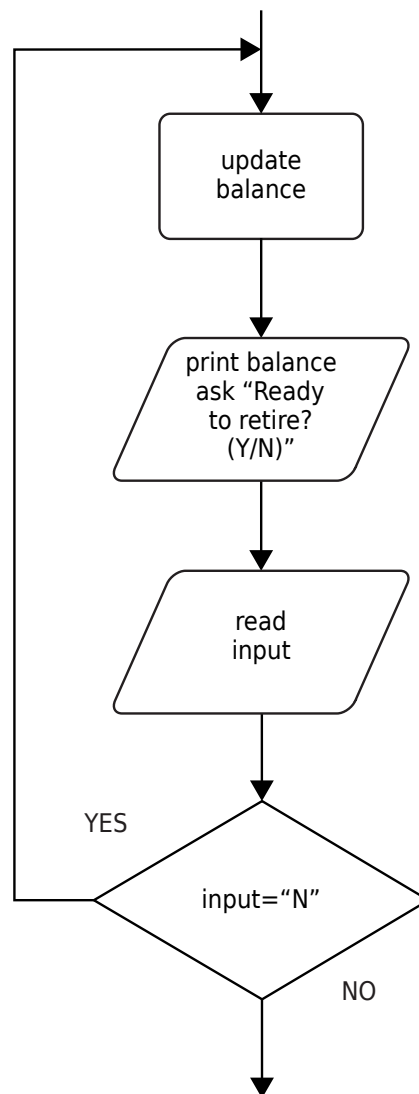


Figure 3.11: Flowchart for the do/while statement

Listing 3.3: Retirement.java

```

1  /**
2   * This program demonstrates a <code>while</code> loop.
3   */
4  void main() {
5      // read inputs
6      double goal = Double.parseDouble(IO.readLine("How much money do you need to retire? "));

```



```

7      double payment
8          = Double.parseDouble(IO.readLine("How much money will you contribute every year? "));
9      double interestRate = Double.parseDouble(IO.readLine("Interest rate in %: "));
10
11      double balance = 0;
12      int years = 0;
13
14      // update account balance while goal isn't reached
15      while (balance < goal) {
16          // add this year's payment and interest
17
18          balance += payment;
19          double interest = balance * interestRate / 100;
20          balance += interest;
21          years++;
22      }
23
24      IO.println("You can retire in " + years + " years.");
25  }

```

Listing 3.4: Retirement2.java

```

1  /**
2   * This program demonstrates a <code>do/while</code> loop.
3   */
4  void main() {
5      double payment = Double.parseDouble(
6          IO.readLine("How much money will you contribute every year? "));
7      double interestRate = Double.parseDouble(IO.readLine("Interest rate in %: "));
8
9      double balance = 0;
10     int year = 0;
11
12     String input;
13
14     // update account balance while user isn't ready to retire
15     do {
16         // add this year's payment and interest
17         balance += payment;
18         double interest = balance * interestRate / 100;
19         balance += interest;
20
21         year++;
22
23         // print current balance
24         IO.println("After year %d, your balance is %, .2f".formatted(year,
25             balance));
26
27         // ask if ready to retire and get input
28         input = IO.readLine("Ready to retire? (Y/N) ");
29     }
30     while (input.equals("N"));
31 }

```

3.8.4. Determinate Loops

The for loop is a general construct to support iteration controlled by a counter or similar variable that is updated after every iteration. As [Figure 3.12](#) shows, the following loop prints the numbers from 1 to 10 on the screen:

```
for (int i = 1; i <= 10; i++)  
    IO.println(i);
```

The first slot of the for statement usually holds the counter initialization. The second slot gives the condition that will be tested before each new pass through the loop, and the third slot specifies how to update the counter.

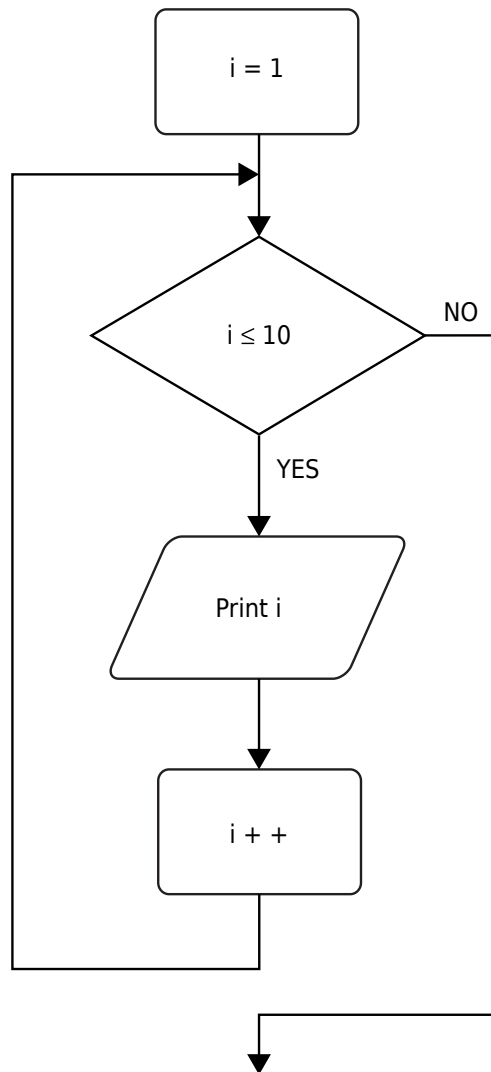


Figure 3.12: Flowchart for the for statement

Although Java, like C++, allows almost any expression in the various slots of a for loop, it is an unwritten rule of good taste that the three slots should only initialize, test, and update the same counter variable. One can write very obscure loops by disregarding this rule.

Even within the bounds of good taste, much is possible. For example, you can have loops that count down:

```
for (int i = 10; i > 0; i--)
    IO.println("Counting down . . . " + i);
IO.println("Blastoff!");
```



Caution: Be careful with testing for equality of floating-point numbers in loops. A for loop like this one

```
for (double x = 0; x != 10; x += 0.1) . . .
```

might never end. Because of roundoff errors, the final value might not be reached exactly. In this example, *x* jumps from 9.9999999999998 to 10.0999999999998 because there is no exact binary representation for 0.1.

When you declare a variable in the first slot of the for statement, the scope of that variable extends until the end of the body of the for loop.

```
for (int i = 1; i <= 10; i++) {
    . . .
}
// i no longer defined here
```

In particular, if you define a variable inside a for statement, you cannot use its value outside the loop. Therefore, if you wish to use the final value of a loop counter outside the for loop, be sure to declare it outside the loop header.

```
int i;
for (i = 1; i <= 10; i++) {
    . . .
}
// i is still defined here
```

On the other hand, you can define variables with the same name in separate for loops:

```
for (int i = 1; i <= 10; i++) {
    . . .
}
. . .
for (int i = 11; i <= 20; i++) { // OK to define another variable named i
    . . .
}
```

A for loop is merely a convenient shortcut for a while loop. For example,

```
(i = 10; i > 0; i--)
    IO.println("Counting down . . . " + i);
```

can be rewritten as follows:

```

i = 10;
while (i > 0) {
    IO.println("Counting down . . . " + i);
    i--;
}

```

The first slot of a for loop can declare multiple variables, provided they are of the same type. And the third slot can contain multiple comma-separated expressions:

```
for (int i = 1, j = 10; i <= 10; i++, j--) { . . . }
```

While technically legal, this stretches the intuitive meaning of the for loop, and you should consider a while loop instead.

[Listing 3.5](#) shows a typical example of a for loop.

The program computes the odds of winning a lottery. For example, if you must pick six numbers from the numbers 1 to 50 to win, then there are $(50 \times 49 \times 48 \times 47 \times 46 \times 45) / (1 \times 2 \times 3 \times 4 \times 5 \times 6)$ possible outcomes, so your chance is 1 in 15,890,700. Good luck!

In general, if you pick k numbers out of n , there are

$$\frac{n \times (n - 1) \times (n - 2) \times \dots \times (n - k + 1)}{1 \times 2 \times 3 \times 4 \times \dots \times k}$$

possible outcomes. The following for loop computes this value:

```

int lotteryOdds = 1;
for (int i = 1; i <= k; i++)
    lotteryOdds = lotteryOdds * (n - i + 1) / i;

```



Note: [Section 3.10.3](#) describes the “generalized for loop” (also called “for each” loop) that makes it convenient to visit all elements of an array or collection.

Listing 3.5: LotteryOdds.java

```

1  /**
2   * This program demonstrates a <code>for</code> loop.
3   */
4  void main() {
5      int k = Integer.parseInt(IO.readLine("How many numbers do you need to draw? "));
6      int n = Integer.parseInt(IO.readLine("What is the highest number you can draw? "));
7
8      // Binomial coefficient n*(n-1)*(n-2)*...*(n-k+1)/(1*2*3*...*k)
9
10     int lotteryOdds = 1;
11     for (int i = 1; i <= k; i++)
12         lotteryOdds = lotteryOdds * (n - i + 1) / i;

```

```

13 |
14 |     IO.println("Your odds are 1 in " + lotteryOdds + ". Good luck!");
15 | }

```

3.8.5. Multiple Selections with switch

The if/else construct can be cumbersome when you have to deal with multiple alternatives for the same expression. The switch statement makes this easier, particularly with the form that has been introduced in Java 14.

For example, if you set up a menu system with four alternatives like that in [Figure 3.13](#), you could use code that looks like this:

```

int choice = Integer.parseInt(IO.readLine("Select an option (1, 2, 3, 4) "));
switch (choice) {
    case 1 ->
        . . .
    case 2 ->
        . . .
    case 3 ->
        . . .
    case 4 ->
        . . .
    default ->
        IO.println("Bad input");
}

```

Note the similarity to the switch expressions that you saw in [Section 3.5.9](#). Unlike a switch expression, a switch statement has no value. Each case carries out an action.

The “classic” form of the switch statement, which dates all the way back to the C language, has been supported since Java 1.0. It has the form:

```

int choice = . . . ;
switch (choice) {
    case 1:
        . . .
        break;
    case 2:
        . . .
        break;
    case 3:
        . . .
        break;
    case 4:
        . . .
        break;
    default:
        IO.println("Bad input");
}

```

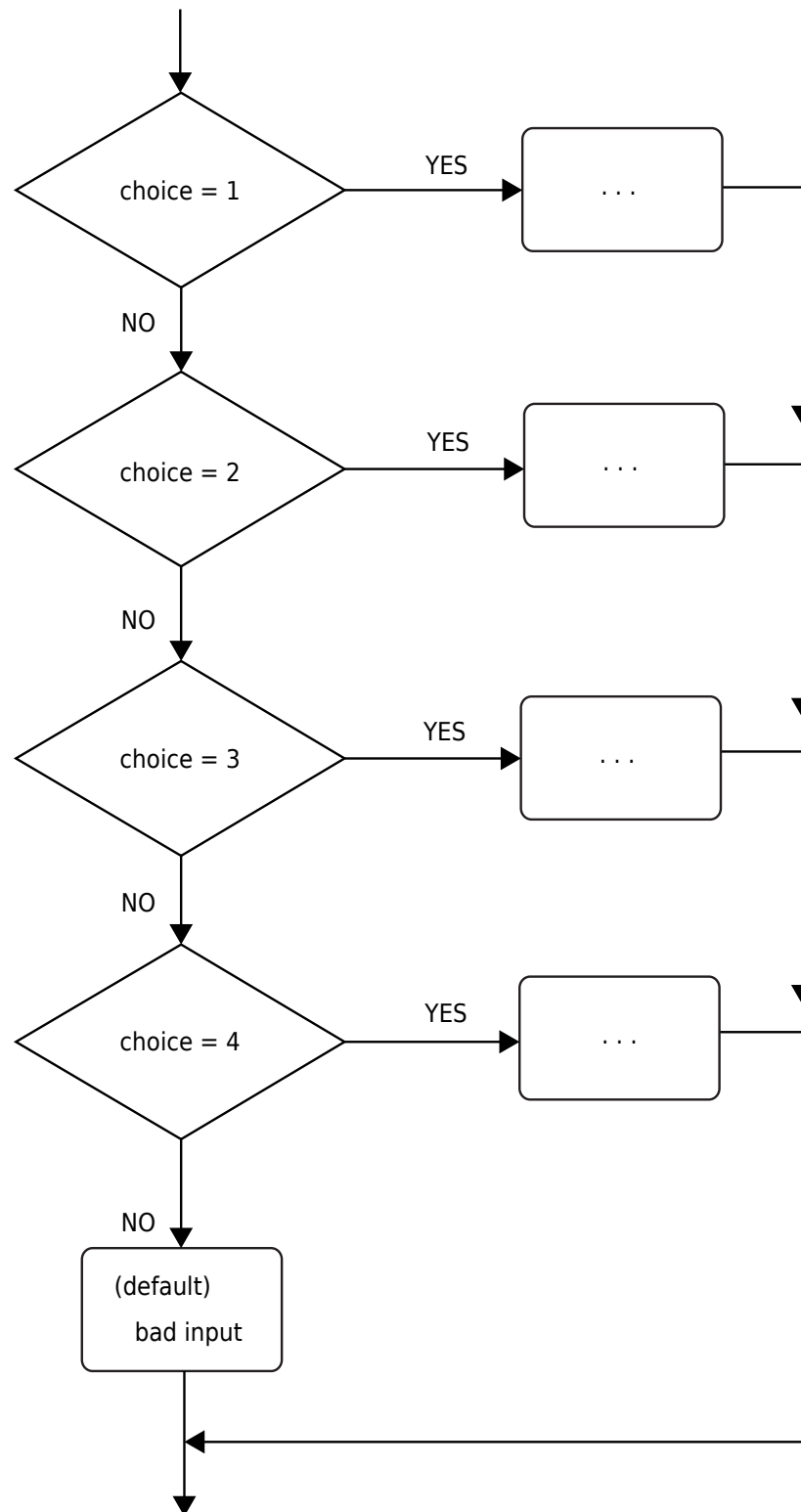


Figure 3.13: Flowchart for the switch statement

Execution starts at the case label that matches the value on which the selection is performed and continues until the next break or the end of the switch. If none of the case labels match, then the default clause is executed, if it is present.



Caution: It is possible for multiple alternatives to be triggered. If you forget to add a break at the end of an alternative, execution falls through to the next alternative! This behavior is plainly dangerous and a common cause for errors.

To detect such problems, compile your code with the `-Xlint:fallthrough` option. Then the compiler will issue a warning whenever an alternative does not end with a break statement.

If you actually want to use the fallthrough behavior, tag the surrounding method with the annotation `@SuppressWarnings("fallthrough")`. Then no warnings will be generated for that method. (An annotation is a mechanism for supplying information to the compiler or a tool that processes Java source or class files. Volume II has an in-depth coverage of annotations.)

For symmetry, Java 14 also introduced a switch expression with fallthrough, for a total of four forms of switch. [Table 3.7](#) shows them all.

Table 3.7: The four forms of switch

	Expression	Statement
No Fallthrough	<pre>int numLetters = switch (seasonName) { case "Spring" -> { IO.println("spring time!"); yield 6; } case "Summer", "Winter" -> 6; case "Fall" -> 4; default -> -1; };</pre>	<pre>switch (seasonName) { case "Spring" -> { IO.println("spring time!"); numLetters = 6; } case "Summer", "Winter" -> numLetters = 6; case "Fall" -> numLetters = 4; default -> numLetters = -1; }</pre>
Fallthrough	<pre>int numLetters = switch (seasonName) { case "Spring": IO.println("spring time!"); case "Summer", "Winter": yield 6; case "Fall": yield 4; default: yield -1; };</pre>	<pre>switch (seasonName) { case "Spring": IO.println("spring time!"); case "Summer", "Winter": numLetters = 6; break; case "Fall": numLetters = 4; break; default: numLetters = -1; }</pre>

In the fallthrough variants, each case ends with a colon. If the cases end with arrows `->`, then there is no fallthrough. You can't mix colons and arrows in a single switch statement.

Each branch of a switch *expression* must yield a value. Most commonly, each value follows an `->` arrow:

```
case "Summer", "Winter" -> 6;
```

If you cannot compute the result in a single expression, use braces and a `yield` statement. Like `break`, it terminates execution. Unlike `break`, it also yields a value—the value of the expression:

```
case "Spring" -> {  
    IO.println("spring time!");  
    yield 6;  
}
```



Note: It is legal to throw an exception in a branch of a switch expression. For example:

```
default -> throw new IllegalArgumentException("Not a valid season");
```

Exceptions are covered in detail in [Chapter 7](#).



Caution: The point of a switch expression is to produce a value (or to fail with an exception). You are not allowed to "jump out":

```
default -> { return -1; } // ERROR
```

Specifically, you cannot use `return`, `break`, or `continue` statements in a switch expression. (See [Section 3.8.6](#) for the latter two.)

With so many variations of switch, which one should you choose?

1. Avoid the fallthrough forms. It is very uncommon to need fallthrough.
2. Prefer switch expressions over statements.

For example, consider:

```
switch (seasonName) {  
    case "Spring", "Summer", "Winter":  
        numLetters = 6;  
        break;  
    case "Fall":  
        numLetters = 4;  
        break;  
    default:  
        numLetters = -1;  
}
```

Since every case ends with a `break`, there is no need to use the fallthrough form. The following is an improvement:


```
switch (seasonName) {
    case "Spring", "Summer", "Winter" ->
        numLetters = 6;
    case "Fall" ->
        numLetters = 4;
    default ->
        numLetters = -1;
}
```

Now note that each branch assigns a value to the same variable. It is much more elegant to use a switch expression here:

```
numLetters = switch (seasonName) {
    case "Spring", "Summer", "Winter" -> 6
    case "Fall" -> 4
    default -> -1
};
```

3.8.6. Statements That Break Control Flow

Although the designers of Java kept `goto` as a keyword, they decided not to include it in the language. In general, `goto` statements are considered poor style. Some programmers feel the anti-`goto` forces have gone too far (see, for example, the famous article of Donald Knuth called “Structured Programming with `goto` statements”). They argue that unrestricted use of `goto` is error-prone but that an occasional jump *out of a loop* is beneficial. The Java designers agreed and even added a new statement, the labeled `break`, to support this programming style.

Let us first look at the unlabeled `break` statement. The same `break` statement that you use to exit a `switch` statement can also be used to break out of a loop. For example:

```
while (years <= 100) {
    balance += payment;
    double interest = balance * interestRate / 100;
    balance += interest;
    if (balance >= goal) break;
    years++;
}
```

Now the loop is exited if either `years > 100` occurs at the top of the loop or `balance >= goal` occurs in the middle of the loop. Of course, you could have computed the same value for `years` without a `break`, like this:

```
while (years <= 100 && balance < goal) {
    balance += payment;
    double interest = balance * interestRate / 100;
    balance += interest;

    if (balance < goal) years++;
}
```

But note that the test `balance < goal` is repeated twice in this version. To avoid this repeated test, some programmers prefer the `break` statement.

The *labeled break* statement lets you break out of multiple nested loops. Occasionally something weird happens inside a deeply nested loop. In that case, you may want to break completely out of all the nested loops. It is inconvenient to program that simply by adding extra conditions to the various loop tests.

Here's an example that shows the labeled break statement at work. Notice that the label must precede the outermost loop out of which you want to break. It also must be followed by a colon.

```
int n;
read_data:
while ( . . . ) { // this loop statement is tagged with the label
    . . .
    for ( . . . ) { // this inner loop is not labeled
        IO.print();
        n = Integer.parseInt(IO.readLine("Enter a number >= 0: "));
        if (n < 0) { // should never happen—can't go on
            break read_data; // break out of read_data loop
        }
        . . .
    }
}
// this statement is executed immediately after the labeled break

if (n < 0) { // check for bad situation
    // deal with bad situation
}
else {
    // carry out normal processing
}
```

If there is a bad input, the labeled break moves past the end of the labeled block. As with any use of the `break` statement, you then need to test whether the loop exited normally or as a result of a break.



Note: Curiously, you can apply a label to any statement, even an `if` statement or a block statement, like this:

```
label: {
    . . .
    if (condition) break label; // exits block
    . . .
}
// jumps here when the break statement executes
```

Thus, if you are lusting after a `goto` and you can place a block that ends just before the place to which you want to jump, you can use a `break` statement! Naturally, I don't

recommend this approach. Note, however, that you can only jump *out of* a block, never *into* a block.

Finally, there is a `continue` statement that, like the `break` statement, breaks the regular flow of control. The `continue` statement transfers control to the header of the innermost enclosing loop. Here is an example:

```
while (sum < goal) {  
    n = Integer.parseInt(IO.readLine("Enter a number: "));  
    if (n < 0) continue;  
    sum += n; // not executed if n < 0  
}
```

If $n < 0$, then the `continue` statement jumps immediately to the loop header, skipping the remainder of the current iteration.

If the `continue` statement is used in a `for` loop, it jumps to the “update” part of the `for` loop. For example:

```
for (count = 1; count <= 100; count++) {  
    n = Integer.parseInt(IO.readLine("Enter a number, -1 to quit: "));  
    if (n < 0) continue;  
    sum += n; // not executed if n < 0  
}
```

If $n < 0$, then the `continue` statement jumps to the `count++` statement.

There is also a labeled form of the `continue` statement that jumps to the header of the loop with the matching label.



Tip: Many programmers find the `break` and `continue` statements confusing. These statements are entirely optional—you can always express the same logic without them. None of the programs in this book use `break` or `continue`.

3.9. Big Numbers

If the precision of the basic integer and floating-point types is not sufficient, you can turn to a couple of handy classes in the `java.math` package: `BigInteger` and `BigDecimal`. These are classes for manipulating numbers with an arbitrarily long sequence of digits. The `BigInteger` class implements arbitrary-precision integer arithmetic, and `BigDecimal` does the same for floating-point numbers.

Use the static `valueOf` method to turn an ordinary number into a big number:

```
BigInteger a = BigInteger.valueOf(100);
```

For longer numbers, use a constructor with a string argument:

```
BigInteger reallyBig
    = new BigInteger("222232244629420445529739893461909967206666939096499764990979600");
```

There are also constants `BigInteger.ZERO`, `BigInteger.ONE`, `BigInteger.TWO`, and `BigInteger.TEN`.



Caution: Always construct `BigDecimal` objects from integers or strings. Avoid the constructor `BigDecimal(double)` that is inherently prone to roundoff. For example, new `BigDecimal(0.1)` has digits

```
0.10000000000000000055511151231257827021181583404541015625
```

Unfortunately, you cannot use the familiar mathematical operators such as `+` and `*` to combine big numbers. Instead, you must use methods such as `add` and `multiply` in the big number classes.

```
BigInteger c = a.add(b); // c = a + b
BigInteger d = c.multiply(b.add(BigInteger.valueOf(2))); // d = c * (b + 2)
```



Note: Java has no programmable operator overloading. There was no way for the programmers of the `BigInteger` class to redefine the `+` and `*` operators to give the `add` and `multiply` operations of the `BigInteger` classes. The language designers did overload the `+` operator to denote concatenation of strings. They chose not to overload other operators, and they did not give Java programmers the opportunity to overload operators in their own classes.



Note: In Java 19, the `BigInteger` class provides a `parallelMultiply` method that yields the same result as `multiply` but can potentially compute the result faster by using multiple processor cores. Use this method if you have to do a lot of multiplications and you know that your application does not need the CPU resources for other computations.

[Listing 3.6](#) shows a modification of the lottery odds program of [Listing 3.5](#), updated to work with big numbers. For example, if you are invited to participate in a lottery in which you need to pick 60 numbers out of a possible 490 numbers, you can use this program to tell you your odds of winning. They are 1 in 716395843461995557415116222540092933411717612789263493493351013459481104668848. Good luck!

The program in [Listing 3.5](#) computed the statement

```
lotteryOdds = lotteryOdds * (n - i + 1) / i;
```

When big integers are used for `lotteryOdds` and `n`, the equivalent statement becomes

```
lotteryOdds = lotteryOdds
    .multiply(n.subtract(BigInteger.valueOf(i - 1)))
    .divide(BigInteger.valueOf(i));
```



Note: To run this program with a version prior to Java 25, add the line

```
import java.math.BigInteger;
```

to the top of the program, in addition to the general modifications described in the notes in [Section 3.1](#) and [Section 3.7.1](#).

Listing 3.6: BigIntegerDemo.java

```

1  /**
2   * This program uses big numbers to compute the odds of winning the grand prize
3   * in a lottery.
4   */
5  void main() {
6      IO.print("How many numbers do you need to draw? ");
7      int k = Integer.parseInt(IO.readLine());
8
9      IO.print("What is the highest number you can draw? ");
10     BigInteger n = new BigInteger(IO.readLine());
11
12     // Binomial coefficient n*(n-1)*(n-2)*...*(n-k+1)/(1*2*3*...*k)
13
14     BigInteger lotteryOdds = BigInteger.ONE;
15
16     for (int i = 1; i <= k; i++)
17         lotteryOdds = lotteryOdds
18             .multiply(n.subtract(BigInteger.valueOf(i - 1)))
19             .divide(BigInteger.valueOf(i));
20
21     IO.println("Your odds are 1 in " + lotteryOdds + " Good luck!");
22 }
```

java.math.BigInteger 1.1

- `BigInteger add(BigInteger other)`
- `BigInteger subtract(BigInteger other)`
- `BigInteger multiply(BigInteger other)`
- `BigInteger divide(BigInteger other)`
- `BigInteger mod(BigInteger other)`
- `BigInteger pow(int exponent)`
return the sum, difference, product, quotient, remainder, and power of this big integer and other.
- `BigInteger sqrt()` 9
yields the square root of this `BigInteger`.
- `int compareTo(BigInteger other)`
returns 0 if this big integer equals other, a negative result if this big integer is less than other, and a positive result otherwise.
- `static BigInteger valueOf(long x)`
returns a big integer whose value equals x.

java.math.BigDecimal 1.1

- `BigDecimal(String digits)`
constructs a big decimal with the given digits.
- `BigDecimal add(BigDecimal other)`
- `BigDecimal subtract(BigDecimal other)`
- `BigDecimal multiply(BigDecimal other)`
- `BigDecimal divide(BigDecimal other)` **5.0**
- `BigDecimal divide(BigDecimal other, RoundingMode mode)` **5.0**
return the sum, difference, product, or quotient of this big decimal and other. The first divide method throws an exception if the quotient does not have a finite decimal expansion. To obtain a rounded result, use the second method. The mode `RoundingMode.HALF_UP` is the rounding mode that you learned in school: round down the digits 0 to 4, round up the digits 5 to 9. It is appropriate for routine calculations. See the API documentation for other rounding modes.
- `int compareTo(BigDecimal other)`
returns 0 if this big decimal equals other, a negative result if this big decimal is less than other, and a positive result otherwise.

3.10. Arrays

Arrays hold sequences of values of the same type. In the following sections, you will see how to work with arrays in Java.

3.10.1. Declaring Arrays

Declare an array variable by specifying the array type—which is the element type followed by `[]`—and the array variable name. For example, here is the declaration of an array `a` of integers:

```
int[] a;
```

However, this statement only declares the variable `a`. It does not yet initialize `a` with an actual array. Use the `new` operator to create the array.

```
int[] a = new int[100]; // or var a = new int[100];
```

This statement declares and initializes an array of 100 integers.

The array length need not be a constant: `new int[n]` creates an array of length `n`.

Once you create an array, you cannot change its length (although you can, of course, change an individual array element). If you frequently need to expand the length of arrays while your program is running, you should use *array lists*, which are covered in [Chapter 5](#).

The type of an array variable does not include the length. For example, the variable `a` in the preceding example has type `int[]` and can be set to an `int` array of any length.



Note: You can define an array variable either as

```
int[] a;
```

or as

```
int a[];
```

Most Java programmers prefer the former style because it neatly separates the type `int[]` (integer array) from the variable name.

Java has a shortcut for creating an array object and supplying initial values:

```
int[] smallPrimes = { 2, 3, 5, 7, 11, 13 };
```

Notice that you do not use `new` with this syntax, and you don't specify the length.

A comma after the last value is allowed, which can be convenient for an array to which you keep adding values over time:

```
String[] authors = {  
    "James Gosling",  
    "Bill Joy",  
    "Guy Steele",  
    // add more names here and put a comma after each name  
};
```

You can declare an *anonymous array*:

```
new int[] { 17, 19, 23, 29, 31, 37 }
```

This expression allocates a new array and fills it with the values inside the braces. It counts the number of initial values and sets the array length accordingly. You can use this syntax to reinitialize an array without creating a new variable. For example,

```
smallPrimes = new int[] { 17, 19, 23, 29, 31, 37 };
```

is shorthand for

```
int[] anonymous = { 17, 19, 23, 29, 31, 37 };  
smallPrimes = anonymous;
```



Note: It is legal to have arrays of length 0. Such an array can be useful if you write a method that computes an array result and the result happens to be empty. Construct an array of length 0 as

```
new elementType[0]
```

or

```
new elementType[] {}
```

Note that an array of length 0 is not the same as `null`.

3.10.2. Accessing Array Elements

You access each individual element of an array through an integer *index*, using the bracket operator. For example, if `a` is an array of integers, then `a[i]` is the element with index `i` in the array.

The array elements are *numbered starting from 0*. The last valid index is one less than the length. In the example below, the index values range from 0 to 99. Once the array is created, you can fill the elements in an array, for example, by using a loop:

```
int[] a = new int[100];
for (int i = 0; i < 100; i++)
    a[i] = i; // fills the array with numbers 0 to 99
```

When you create an array of numbers, all elements are initialized with zero. Arrays of boolean are initialized with false. Arrays of objects are initialized with the special value `null`, which indicates that they do not (yet) hold any objects. This can be surprising for beginners. For example,

```
String[] names = new String[10];
```

creates an array of ten strings, all of which are `null`. If you want the array to hold empty strings, you must supply them:

```
for (int i = 0; i < 10; i++) names[i] = "";
```



Caution: If you construct an array with 100 elements and then try to access the element `a[100]` (or any other index outside the range from 0 to 99), an “array index out of bounds” exception will occur.

To find the number of elements of an array, use `array.length`. For example:

```
for (int i = 0; i < a.length; i++)
    IO.println(a[i]);
```

3.10.3. The “for each” Loop

Java has a powerful looping construct that allows you to loop through each element in an array (or any other collection of elements) without having to fuss with index values.

The *enhanced* for loop

```
for (variable : collection) statement
```

sets the given variable to each element of the collection and then executes the statement (which, of course, may be a block). The *collection* expression must be an array or an object of a class that

implements the `Iterable` interface, such as `ArrayList`. Array lists are covered in [Chapter 5](#) and the `Iterable` interface in [Chapter 9](#).

For example,

```
for (int element : a)
    IO.println(element);
```

prints each element of the array `a` on a separate line.

You should read this loop as “for each element in `a`.” The designers of the Java language considered using keywords, such as `foreach` and `in`. But this loop was a late addition to the Java language, and in the end nobody wanted to break the old code that already contained methods or variables with these names (such as `System.in`).

Of course, you could achieve the same effect with a traditional for loop:

```
for (int i = 0; i < a.length; i++)
    IO.println(a[i]);
```

However, the “for each” loop is more concise and less error-prone, as you don’t have to worry about those pesky start and end index values.



Note: The loop variable of the “for each” loop traverses the *elements* of the array, not the index values.

The “for each” loop is a pleasant improvement over the traditional loop if you need to process all elements in a collection. However, there are still plenty of opportunities to use the traditional for loop. For example, you might not want to traverse the entire collection, or you may need the index value inside the loop.



Tip: There is an even easier way to print all values of an array, using the `toString` method of the `Arrays` class. The call `Arrays.toString(a)` returns a string containing the array elements, enclosed in brackets and separated by commas, such as “[2, 3, 5, 7, 11, 13]”. To print the array, simply call

```
IO.println(Arrays.toString(a));
```

3.10.4. Array Copying

You can copy one array variable into another, but then *both variables refer to the same array*:

```
int[] luckyNumbers = smallPrimes;
luckyNumbers[5] = 12; // now smallPrimes[5] is also 12
```

[Figure 3.14](#) shows the result.

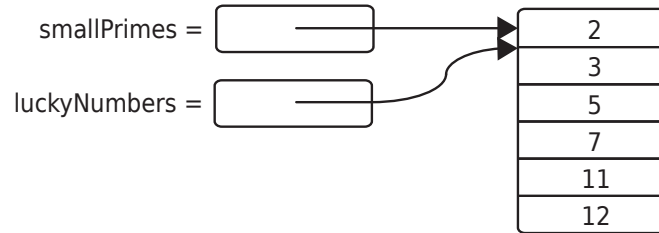


Figure 3.14: Copying an array variable

If you actually want to copy all values of one array into a new array, use the `copyOf` method in the `Arrays` class:

```
int[] copiedLuckyNumbers = Arrays.copyOf(luckyNumbers, luckyNumbers.length);
```

The second argument is the length of the new array. A common use of this method is to increase the length of an array:

```
luckyNumbers = Arrays.copyOf(luckyNumbers, 2 * luckyNumbers.length);
```

The additional elements are filled with 0 if the array contains numbers, false if the array contains boolean values. Conversely, if the length is less than the length of the original array, only the initial values are copied.



Note: As in Python and JavaScript, Java arrays are allocated on the heap. This is quite different from a C array or C++ vector on the stack. If you come from C or C++, you should think of a Java arrays as a pointer to an array allocated on the heap. That is,

```
int[] a = new int[100]; // Java
```

is not the same as

```
int a[100]; // C++
```

but rather

```
int* a = new int[100]; // C++
```

3.10.5. Command-Line Arguments

If you want to process arguments that a user of your program specified on the command line, your `main` method needs a parameter that is an array of strings.

For example, consider this program in a file `Message.java`:

```
void main(String[] args) {
    IO.print(switch (args[0])) {
        case "-a" -> "🐼";
        case "-b" -> "🐼";
        case "-h" -> "Hello,";
```

```
        default -> args[0];
    }
    IO.print(" " + args[1]);
    IO.println("!");
}
```

If the program is called as

```
java Message.java -h World
```

or

```
javac Message.java
java Message -h World
```

then `args[0]` is the string `"-h"`, and `args[1]` is `"World"`.



Note: Unlike in Python or C, the name of the program is not stored in the array of command-line arguments. When you start up a program as

```
java Message.java -h World
```

from the command line, then the `args` array does not contain `java` or `"Message.java"`.

3.10.6. Array Sorting

To sort an array of numbers, you can use one of the sort methods in the `Arrays` class:

```
int[] a = new int[10000];
...
Arrays.sort(a)
```

This method uses a tuned version of the QuickSort algorithm that is claimed to be very efficient on most data sets. The `Arrays` class provides several other convenience methods for arrays that are included in the API notes at the end of this section.

The program in [Listing 3.7](#) puts arrays to work. This program draws a random combination of numbers for a lottery game. For example, if you play a “choose 6 numbers from 49” lottery, the program might print this:

```
Bet the following combination. It'll make you rich!
4
7
8
19
30
44
```

To select such a random set of numbers, first fill an array `numbers` with the values `1, 2, . . . , n`:

```
int[] numbers = new int[n];
for (int i = 0; i < numbers.length; i++)
    numbers[i] = i + 1;
```

A second array holds the numbers to be drawn:

```
int[] result = new int[k];
```

Now draw k numbers. The `Math.random` method returns a random floating-point number that is between 0 (inclusive) and 1 (exclusive). Multiplying the result with n yields a random number between 0 and $n - 1$.

```
int r = (int) (Math.random() * n);
```

Set the i th result to be the number at that index. Initially, that is just $r + 1$, but as you'll see presently, the contents of the numbers array are changed after each draw.

```
result[i] = numbers[r];
```

Now, you must be sure never to draw that number again—all lottery numbers must be distinct. Therefore, overwrite `numbers[r]` with the *last* number in the array and reduce n by 1.

```
numbers[r] = numbers[n - 1];
n--;
```

The point is that in each draw we pick an *index*, not the actual value. The index points into an array that contains the values that have not yet been drawn.

After drawing k lottery numbers, sort the result array for a more pleasing output:

```
Arrays.sort(result);
for (int r : result)
    IO.println(r);
```

Listing 3.7: LotteryDrawing.java

```
1  /**
2   * This program demonstrates array manipulation.
3   */
4  void main() {
5      int k = Integer.parseInt(IO.readLine("How many numbers do you need to draw? "));
6      int n = Integer.parseInt(IO.readLine("What is the highest number you can draw? "));
7
8      // fill an array with numbers 1 2 3 . . . n
9      int[] numbers = new int[n];
10     for (int i = 0; i < numbers.length; i++)
11         numbers[i] = i + 1;
12
13     // draw k numbers and put them into a second array
14     int[] result = new int[k];
15     for (int i = 0; i < result.length; i++) {
16         // make a random index between 0 and n - 1
17         int r = (int) (Math.random() * n);
```

```

18
19     // pick the element at the random location
20     result[i] = numbers[r];
21
22     // move the last element into the random location
23     numbers[r] = numbers[n - 1];
24     n--;
25 }
26
27 // print the sorted array
28 Arrays.sort(result);
29 IO.println("Bet the following combination. It'll make you rich!");
30 for (int r : result)
31     IO.println(r);
32 }

```

java.util.Arrays 1.2

- static String toString(T[] a) **5.0**
returns a string with the elements of *a*, enclosed in brackets and delimited by commas. In this and the following methods, the component type *T* of the array can be int, long, short, char, byte, boolean, float, or double.
- static T[] copyOf(T[] a, int end) **6**
- static T[] copyOfRange(T[] a, int start, int end) **6**
return an array of the same type as *a*, of length either *end* or *end*-*start*, filled with the values of *a*. If *end* is larger than *a.length*, the result is padded with 0 or false values.
- static void sort(T[] a)
sorts the array, using a tuned QuickSort algorithm.
- static void fill(T[] a, T v)
sets all elements of the array to *v*.
- static boolean equals(T[] a, T[] b)
returns true if the arrays have the same length and if the elements at corresponding indexes match.

3.10.7. Multidimensional Arrays

Multidimensional arrays use more than one index to access array elements. They are used for tables and other more complex arrangements. You can safely skip this section until you have a need for this storage mechanism.

Suppose you want to make a table of numbers that shows how much an investment of \$10,000 will grow under different interest rate scenarios in which interest is paid annually and reinvested.

	5%	6%	7%	8%	9%	10%
10000.00	10000.00	10000.00	10000.00	10000.00	10000.00	10000.00
10500.00	10600.00	10700.00	10800.00	10900.00	11000.00	
11025.00	11236.00	11449.00	11664.00	11881.00	12100.00	
11576.25	11910.16	12250.43	12597.12	12950.29	13310.00	
12155.06	12624.77	13107.96	13604.89	14115.82	14641.00	
12762.82	13382.26	14025.52	14693.28	15386.24	16105.10	
13400.96	14185.19	15007.30	15868.74	16771.00	17715.61	

14071.00	15036.30	16057.81	17138.24	18280.39	19487.17
14774.55	15938.48	17181.86	18509.30	19925.63	21435.89
15513.28	16894.79	18384.59	19990.05	21718.93	23579.48

You can store this information in a two-dimensional array named `balances`.

Declaring a two-dimensional array in Java is simple enough. For example:

```
double[][] balances;
```

You cannot use the array until you initialize it. In this case, you can do the initialization as follows:

```
balances = new double[NYEARS][NRATES];
```

In other cases, if you know the array elements, you can use a shorthand notation for initializing a multidimensional array without a call to `new`. For example:

```
int[][] magicSquare = {  
    { 16, 3, 2, 13 },  
    { 5, 10, 11, 8 },  
    { 9, 6, 7, 12 },  
    { 4, 15, 14, 1 }  
};
```

Once the array is initialized, you can access individual elements by supplying two pairs of brackets—for example, `balances[i][j]`.

The example program stores a one-dimensional array `interestRates` of interest rates and a two-dimensional array `balances` of account balances, one for each year and interest rate. Initialize the first row of the array with the initial balance:

```
for (int j = 0; j < balances[0].length; j++)  
    balances[0][j] = 10000;
```

Then compute the other rows, as follows:

```
for (int i = 1; i < balances.length; i++) {  
    for (int j = 0; j < balances[i].length; j++) {  
        double oldBalance = balances[i - 1][j];  
        double interest = . . . ;  
        balances[i][j] = oldBalance + interest;  
    }  
}
```

[Listing 3.8](#) shows the full program. In this program, you can see how to use multiple methods. The main method calls a `printTable` method that prints the table of balances.



Note: A “for each” loop does not automatically loop through all elements in a two-dimensional array. Instead, it loops through the rows, which are themselves one-

dimensional arrays. To visit all elements of a two-dimensional array *a*, nest two loops, like this:

```
for (double[] row : values)
    for (double value : row)
        do something with value
```



Tip: To print out a quick-and-dirty list of the elements of a two-dimensional array, call

```
IO.println(Arrays.deepToString(a));
```

The output is formatted like this:

```
[[16, 3, 2, 13], [5, 10, 11, 8], [9, 6, 7, 12], [4, 15, 14, 1]]
```

Listing 3.8: CompoundInterest.java

```
1  /**
2   * This program shows how to store tabular data in a 2D array.
3   */
4  void main() {
5      final double STARTRATE = 5;
6      final int NRATES = 6;
7      final int NYEARS = 10;
8
9      // set interest rates to 5 . . . 10%
10     double[] interestRates = new double[NRATES];
11     for (int j = 0; j < interestRates.length; j++)
12         interestRates[j] = (STARTRATE + j) / 100.0;
13
14     double[][] balances = new double[NYEARS][NRATES];
15
16     // set initial balances to 10000
17     for (int j = 0; j < balances[0].length; j++)
18         balances[0][j] = 10000;
19
20     // compute interest for future years
21     for (int i = 1; i < balances.length; i++) {
22         for (int j = 0; j < balances[i].length; j++) {
23             // get last year's balances from previous row
24             double oldBalance = balances[i - 1][j];
25
26             // compute interest
27             double interest = oldBalance * interestRates[j];
28
29             // compute this year's balances
30             balances[i][j] = oldBalance + interest;
31         }
32     }
33
34     printTable(interestRates, balances);
35 }
```

```

36
37 void printTable(double[] headers, double[][] values) {
38     for (double header : headers) {
39         IO.print("%10.2f".formatted(header));
40     }
41     IO.println();
42     IO.println("-".repeat(10 * headers.length));
43     // print balance table
44     for (double[] row : values) {
45         // print table row
46         for (double value : row)
47             IO.print("%10.2f".formatted(value));
48
49         IO.println();
50     }
51 }

```

3.10.8. Ragged Arrays

So far, what you have seen is not too different from other programming languages. But there is actually something subtle going on behind the scenes that you can sometimes turn to your advantage: Java has *no* multidimensional arrays at all, only one-dimensional arrays. Multidimensional arrays are faked as “arrays of arrays.”

For example, the `balances` array in the preceding example is actually an array that contains ten elements, each of which is an array of six floating-point numbers ([Figure 3.15](#)).

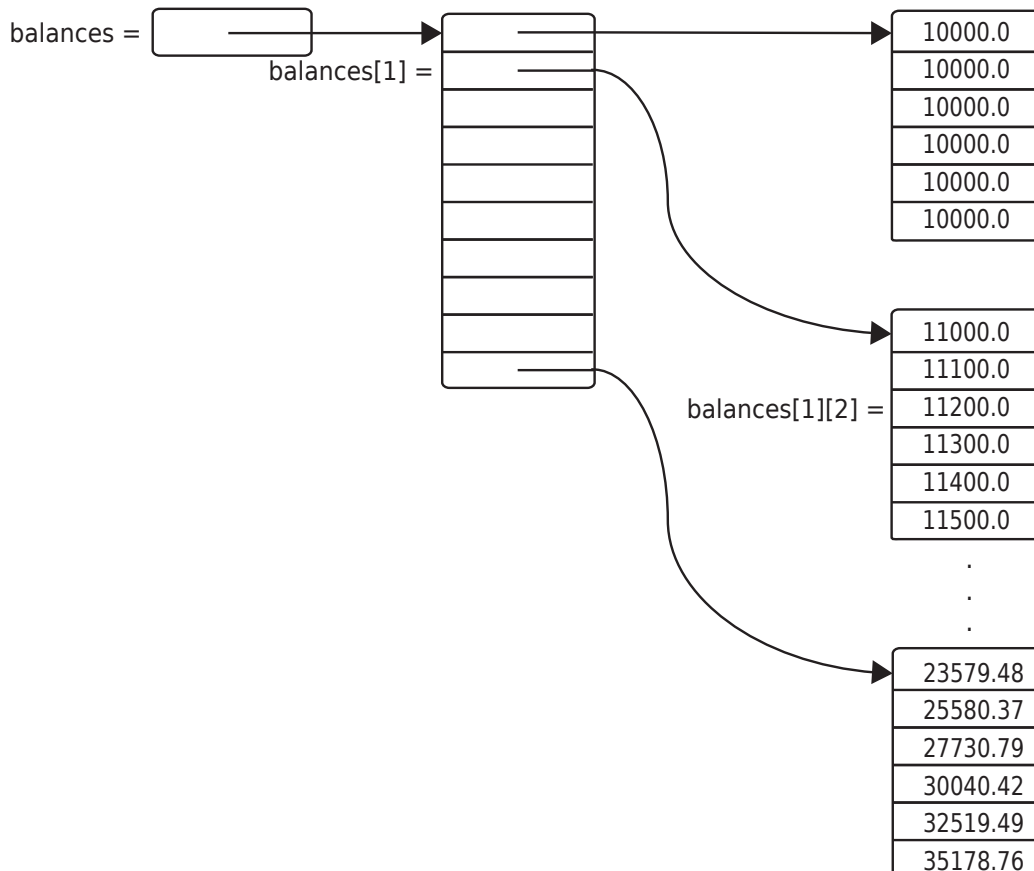


Figure 3.15: A two-dimensional array

The expression `balances[i]` refers to the *i*th subarray—that is, the *i*th row of the table. It is itself an array, and `balances[i][j]` refers to the *j*th element of that array.

Since rows of arrays are individually accessible, you can actually swap them!

```
double[] temp = balances[i];
balances[i] = balances[i + 1];
balances[i + 1] = temp;
```

Note that the number of rows and columns is not a part of the type of an array variable. The variable `balances` has type `double[][]`: an array of double arrays.

Therefore, you can make “ragged” arrays—that is, arrays in which different rows have different lengths. Here is the standard example. Let us make an array in which the element at row *i* and column *j* equals the number of possible outcomes of a “choose *j* numbers from *i* numbers” lottery.

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
```

As *j* can never be larger than *i*, the matrix is triangular. The *i*th row has *i* + 1 elements. (It is OK to choose 0 elements; there is one way to make such a choice.) To build this ragged array, first allocate the array holding the rows:

```
final int NMAX = 10;
int[][] odds = new int[NMAX + 1][];
```

Next, allocate the rows:

```
for (int n = 0; n <= NMAX; n++)
    odds[n] = new int[n + 1];
```

Now that the array is allocated, you can access the elements in the normal way, provided you do not overstep the bounds:

```
for (int n = 0; n < odds.length; n++) {
    for (int k = 0; k < odds[n].length; k++) {
        // compute lotteryOdds
        . . .
        odds[n][k] = lotteryOdds;
    }
}
```

[Listing 3.9](#) gives the complete program.



Note: Just as with one-dimensional arrays, it is legal to construct multi-dimensional arrays where a dimension is zero. For example,

```
new int[3][0]
```

has three rows, each of which happen to have length zero. In contrast,

```
new int[0][3]
```

has no rows. The row length is immaterial, since no rows are actually allocated. In other words, `new int[0][3]`, `new int[0][4]`, and `new int[0][]` are all the same.



Note: The Java declaration

```
double[][] balances = new double[10][6]; // Java
```

is very different from declaring a two-dimensional array in C or C++.

```
double balances[10][6]; // C/C++
```

The latter declares a contiguous block of 60 floating-point numbers on the stack. In Java, each row is stored separately on the heap, as you have seen in [Figure 3.15](#).

Listing 3.9: LotteryArray.java

```
1  /**
2   * This program demonstrates a triangular array.
3   */
4  void main() {
5      final int NMAX = 10;
6
7      // allocate triangular array
8      int[][] odds = new int[NMAX + 1][];
9      for (int n = 0; n <= NMAX; n++)
10         odds[n] = new int[n + 1];
11
12     // fill triangular array
13     for (int n = 0; n < odds.length; n++)
14         for (int k = 0; k < odds[n].length; k++) {
15             /*
16              * compute binomial coefficient
17              * n*(n-1)*(n-2)*...*(n-k+1)/(1*2*3*...*k)
18              */
19             int lotteryOdds = 1;
20             for (int i = 1; i <= k; i++)
21                 lotteryOdds = lotteryOdds * (n - i + 1) / i;
22
23             odds[n][k] = lotteryOdds;
24         }
25 }
```

```
26 // print triangular array
27 for (int[] row : odds) {
28     for (int odd : row)
29         IO.print("%4d".formatted(odd));
30     IO.println();
31 }
32 }
```

You have now seen the fundamental programming structures of the Java language. The next chapter covers object-oriented programming in Java.

This page intentionally left blank

Index

Symbols

- ! operator [48, 52](#)
- != operator [48, 52, 81](#)
- """ . . . """ (triple quotes, for text blocks) [64](#)
- " . . ." (single quotes, for strings) [31](#)
- # (number sign)
 - in JavaDoc hyperlinks [179](#)
 - printf flag [71](#)
- \$ (dollar sign)
 - delimiter, for inner classes [327](#)
 - in variable names [40](#)
 - printf flag [71](#)
- % (percent sign)
 - arithmetic operator [42, 52](#)
 - conversion character [70](#)
- & (ampersand)
 - bitwise operator [50, 52](#)
 - in bounding types [402](#)
 - in reference parameters (C++) [142](#)
- && operator [48, 52](#)
- > (right angle bracket)
 - in shell syntax [390](#)
 - relational operator [48, 52](#)
- >& (shell syntax) [390](#)
- >>, >>> operators [51, 52](#)
- >= operator [48, 52](#)
- < (left angle bracket)
 - printf flag [71](#)
 - relational operator [48, 52](#)
- << operator [51, 52](#)
- <. . . > (angle brackets) [222, 398](#)
- <= operator [48, 52](#)
- ' , " (single, double quote), escape sequences for [36](#)
- ((left parenthesis) [71](#)
 - printf flag [71](#)
- (. . .) (parentheses)
 - empty, in method calls [31](#)
 - for casts [46, 52, 201](#)
 - for operator hierarchy [51](#)
- * (asterisk)
 - arithmetic operator [42, 52](#)
 - for annotation processors [627](#)
 - in class path [167](#)
 - in imports [159](#)
- + (plus sign)
 - arithmetic operator [42, 45, 52](#)
 - for objects and strings [53, 216](#)
 - printf flag [71](#)
- ++ operator [47, 52](#)
- , (comma)
 - operator (C++) [52](#)
 - printf flag [71](#)
- (minus sign)
 - arithmetic operator [42, 52](#)
 - printf flag [71](#)
- > operator
 - in lambda expressions [305](#)
 - in switch expressions [85](#)
- operator [47, 52](#)
- . (period) [167](#)
- ... (ellipsis) [232](#)
- .class extension [31](#)
- .exe extension [172](#)
- / (slash) [42, 52](#)
- /* . . . */ comments [32](#)
- /** . . . */ (Javadoc comment delimiters) [32, 175, 176](#)
- // comments [32](#)
- 0, 0b, 0B, 0x, 0X prefixes (in integers) [34](#)
- 0, printf flag [71](#)
- 2> (shell syntax) [390](#)
- : (colon)
 - in assertions [371](#)
 - in class path (UNIX) [167](#)
 - inheritance token (C++) [188](#)
- :: (C++ operator) [310](#)
- ; (semicolon)
 - in class path (Windows) [167](#)
 - in statements [31, 39](#)
- = operator [40, 46](#)
- == operator [48, 52](#)
 - for class objects [256](#)
 - for enumerated types [239](#)
 - for floating-point numbers [81](#)
 - for identity hash maps [481](#)
 - for strings [57](#)
 - wrappers and [229](#)
- ? (question mark)
 - for wildcard types [410](#)
- ? : operator [48, 52](#)
 - with pattern matching [203](#)
- @ (at sign) [176](#)
 - in java command-line options [657](#)
- @author
 - JavaDoc tag [181](#)
- @Deprecated annotation [620](#)
- @deprecated JavaDoc tag [620](#)
- @Documented annotation [620, 621](#)
- @FunctionalInterface annotation [320, 620, 621](#)
- @Generated annotation [620, 621](#)
- @index
 - JavaDoc tag [181](#)
- @Inherited annotation [620, 621](#)
- @link
 - JavaDoc tag [180](#)
- @LogEntry annotation [633](#)
- @NonNull annotation [614](#)
- @Override annotation [211, 619, 620](#)
- @param
 - JavaDoc tag [177](#)

- @Persistent annotation [622](#)
 - @Property annotation [630](#)
 - @Repeatable annotation [620](#), [622](#), [623](#)
 - @RepeatedTest annotation [613](#)
 - @Retention annotation [617](#), [620](#)
 - @return
 - JavaDoc tag [177](#)
 - @SafeVarargs annotation [420](#), [620](#), [621](#)
 - @see
 - JavaDoc tag [179](#), [180](#)
 - @Serial annotation [619](#), [621](#)
 - @since
 - JavaDoc tag [181](#)
 - @SuppressWarnings annotation [85](#), [228](#), [408](#), [420](#), [425](#), [620](#), [621](#)
 - @Target annotation [617](#), [620](#)
 - @Test annotation [612](#), [616](#), [617](#)
 - @throws
 - JavaDoc tag [177](#)
 - @version
 - JavaDoc tag [181](#), [182](#)
 - [. . .] (brackets)
 - empty, in generics [399](#)
 - for arrays [94](#)
 - \ (backslash)
 - escape sequence for [36](#)
 - in text blocks [65](#)
 - \b (backslash character literal) [36](#)
 - \f (form feed character literal) [36](#)
 - \n (newline character literal) [36](#), [64](#)
 - \r (carriage return character literal) [36](#)
 - \s (space character literal) [36](#), [65](#)
 - \t (tab character literal) [36](#)
 - \u (Unicode character literal) [36](#), [37](#)
 - ^ (caret) [50](#), [52](#), [305](#)
 - _ (underscore)
 - as a reserved word [670](#)
 - delimiter, in number literals [34](#)
 - {. . .} (braces)
 - double, in inner classes [332](#)
 - for blocks [29](#), [31](#), [72](#)
 - for enumerated type [42](#)
 - in annotation elements [613](#)
 - in lambda expressions [305](#)
 - | operator [50](#), [52](#)
 - || operator [48](#), [52](#)
 - ~ operator [50](#), [52](#)
 - ☞ [38](#)
- ## A
- A, a conversion characters [70](#)
 - Abstract classes [233](#)
 - extending [235](#)
 - interfaces and [282](#), [289](#)
 - no instantiating for [235](#)
 - object variables of [235](#)
 - abstract keyword [233](#), [667](#)
 - Abstract methods [234](#)
 - in functional interfaces [307](#)
 - AbstractCollection class [292](#), [446](#), [457](#)
 - AbstractProcessor class [627](#)
 - acceptEither method [593](#), [594](#)
 - Access modifiers
 - checking [260](#)
 - final [41](#), [130](#), [198](#), [288](#), [329](#), [569](#)
 - private [123](#), [165](#), [324](#)
 - protected [205](#), [277](#), [302](#)
 - public [31](#), [122](#), [165](#), [282](#), [283](#)
 - public static final [289](#)
 - redundant [289](#)
 - static [131](#)
 - accessFlags method
 - of Constructor [265](#)
 - of Field [265](#)
 - of Method [265](#)
 - AccessibleObject class [270](#)
 - canAccess method [270](#)
 - setAccessible method [267](#), [270](#)
 - trySetAccessible method [270](#)
 - Accessor methods [116](#), [127](#), [128](#), [412](#)
 - accumulate method
 - of LongAccumulator [571](#)
 - accumulateAndGet method
 - of AtomicXxx [571](#)
 - ActionListener interface
 - actionPerformed method [294](#), [295](#), [304](#), [323](#), [324](#), [328](#), [330](#)
 - implementing [308](#)
 - ActiveX [4](#)
 - add method
 - of ArrayList [222](#), [224](#), [227](#)
 - of BigDecimal [92](#)
 - of BigInteger [91](#)
 - of BlockingQueue [575](#), [576](#)
 - of Collection [442](#), [446](#), [447](#), [449](#)
 - of GregorianCalendar [117](#)
 - of HashSet [463](#)
 - of List [449](#), [460](#)
 - of ListIterator [454](#), [456](#), [460](#)
 - of LongAdder [571](#)
 - of Queue [469](#)
 - of Set [450](#)
 - addAll method
 - of ArrayList [397](#)
 - of Collection [446](#), [447](#)
 - of Collections [500](#)
 - of List [460](#)
 - addExact method [45](#)
 - addFirst method
 - of LinkedList [461](#)
 - of SequencedCollection [469](#)
 - Addition [42](#), [52](#)
 - for different numeric types [45](#)
 - for objects and strings [53](#), [216](#)
 - addLast method
 - of LinkedList [461](#)
 - of SequencedCollection [469](#)
 - addSuppressed method [363](#)
 - of Throwable [366](#)
 - Adobe Flash [8](#)
 - Agent code [638](#), [639](#)
 - Aggregation [110](#)
 - Algorithms [107](#)
 - for binary search [497](#)
 - for shuffling [496](#)
 - for sorting [495](#)
 - QuickSort [97](#), [495](#)

- simple, in the Java Collections Framework [499](#)
- writing [502](#)
- Algorithms + Data Structures = Programs* (Wirth) [107](#)
- Algorithms in C++* (Sedgewick) [495](#)
- allOf method
 - of CompletableFuture [593, 594](#)
 - of EnumSet [482](#)
- allProcesses method
 - of ProcessHandle [609](#)
- Amazon [15](#)
- and method
 - of BitSet [511](#)
- andNot method
 - of BitSet [511](#)
- Andreessen, Mark [8](#)
- Android [13, 598](#)
- AnnotatedConstruct interface [628](#)
- AnnotatedElement interface [624, 626](#)
 - getAnnotation method [626](#)
 - getAnnotation, getAnnotationsByType methods [625](#)
 - getAnnotations method [626](#)
 - getAnnotationsByType method [626](#)
 - getDeclaredAnnotations method [626](#)
 - isAnnotationPresent method [626](#)
- Annotation interface [619](#)
 - annotationType method [619](#)
 - equals method [619](#)
 - extending [619](#)
 - hashCode method [619](#)
 - toString method [619](#)
- Annotation interfaces [616](#)
- Annotation processors [627](#)
 - at bytecode level [632](#)
- Annotations [408](#)
 - accessing [618](#)
 - applicability of [620](#)
 - container [623, 624](#)
 - declaration [613](#)
 - documented [620, 621](#)
 - generating source code with [628](#)
 - inherited [620, 621, 624](#)
 - key/value pairs in [612, 618](#)
 - meta [617, 623](#)
 - modifiers and [615](#)
 - multiple [613](#)
 - processing
 - at runtime [623](#)
 - source-level [627](#)
 - repeatable [613, 620, 622, 623, 624](#)
 - standard [619](#)
 - type use [614](#)
- annotationType method
 - of Annotation [619](#)
- Anonymous arrays [93](#)
- Anonymous inner classes [330](#)
- Antisymmetry rule [288](#)
- anyOf method [593](#)
 - of CompletableFuture [594](#)
- Apache
 - Commons CSV library [654, 655](#)
- append method
 - of StringBuilder [61, 64](#)
- appendCodePoint method
 - of StringBuilder [64](#)
- Applets [7, 12](#)
 - running in a browser [7](#)
- Application Programming Interfaces (APIs), online documentation for [58, 59](#)
- Applications
 - compiling/launching from the command line [19, 31](#)
 - debugging [20, 348](#)
 - executing
 - without a separate Java runtime [664](#)
 - extensible [197](#)
 - for different Java releases [172](#)
 - localizing [112, 259](#)
 - managing in JVM [391](#)
 - monitoring [638, 639](#)
 - responsive [597](#)
 - terminating [125](#)
 - testing [370](#)
- applyToEither method [593, 594](#)
- Arguments [31](#)
 - string [31](#)
 - variable number of [232](#)
- arguments method
 - of ProcessHandle.Info [610](#)
- Arguments. See Parameters
- Arithmetic operators [42](#)
 - accuracy of [43](#)
 - autoboxing with [229](#)
 - combining with assignment [46](#)
 - precedence of [52](#)
- Array class [270, 273](#)
 - get method [273](#)
 - getLength method [271, 273](#)
 - getXxx method [273](#)
 - newInstance method [271, 273](#)
 - set method [273](#)
 - setXxx method [273](#)
- Array lists [461](#)
 - anonymous [332](#)
 - capacity of [223](#)
 - elements of
 - accessing [224](#)
 - adding [222, 225](#)
 - removing [226](#)
 - traversing [226](#)
 - generic [221](#)
 - raw vs. typed [227](#)
- Array variables [92](#)
- ArrayBlockingQueue class [576, 579](#)
 - Constructor [579](#)
- ArrayDeque class [468, 470](#)
 - as a concrete collection type [451](#)
 - Constructor [470](#)
- ArrayIndexOutOfBoundsException class [94, 350, 352](#)
- ArrayList class [94, 221, 224, 227, 395, 453](#)
 - add method [222, 224, 227](#)
 - addAll method [397](#)
 - ArrayList<E> method [224](#)
 - as a concrete collection type [451](#)
 - declaring with var [222](#)
 - ensureCapacity method [223, 224](#)
 - get method [227](#)
 - get, set methods [224](#)
 - iterating over [443](#)
 - remove method [226, 227](#)

- removeIf method [309](#)
- set method [227](#)
- size method [224](#)
- size, trimToSize methods [223](#)
- synchronized [589](#)
- toArray method [424](#)
- trimToSize method [224](#)
- ArrayList<E> method
 - of ArrayList [224](#)
- Arrays [94](#)
 - annotating [614](#)
 - anonymous [93](#)
 - circular [442](#)
 - cloning [302](#)
 - converting to collections [502](#)
 - copying [95](#)
 - on write [587](#)
 - creating [92](#)
 - elements of
 - computing in parallel [588](#)
 - numbering [94](#)
 - remembering types of [195](#)
 - removing from the middle [453](#)
 - traversing [94, 100](#)
 - equality testing for [210, 211](#)
 - generic methods for [270](#)
 - hash codes of [213](#)
 - in command-line arguments [96](#)
 - initializing [93, 94](#)
 - length of [94](#)
 - equal to 0 [93](#)
 - increasing [96](#)
 - multidimensional [99, 103, 210, 217](#)
 - not of generic types [314, 409, 419, 423](#)
 - of integers [217](#)
 - of subclass/superclass references [195](#)
 - of wildcard types [419](#)
 - out-of-bounds access in [350](#)
 - parallel operations on [587](#)
 - printing [101, 217](#)
 - ragged [102](#)
 - size of [223, 271](#)
 - setting at runtime [221](#)
 - sorting [97, 285, 587](#)
- Arrays class [99, 211, 215, 287, 492, 498](#)
 - asList method [492](#)
 - binarySearch method [342, 498](#)
 - copyOf method [95, 99, 270](#)
 - copyOfRange method [99](#)
 - deepEquals method [210](#)
 - deepToString method [101, 217](#)
 - equals method [99, 210, 211](#)
 - fill method [99](#)
 - hashCode method [213, 215](#)
 - parallelXxx methods [587](#)
 - sort method [97, 99, 283, 285, 287, 304, 308](#)
 - toString method [95, 99](#)
- ArrayStoreException class [195, 410, 419, 421](#)
- arrayType method [271](#)
 - of Class [273](#)
- ASCII [37](#)
- asIterator method [505](#)
 - of Enumeration [505](#)
- asList method

- of Arrays [492](#)
- assert keyword [370, 667](#)
- Assertions [370](#)
 - checking [614](#)
 - checking parameters with [373](#)
 - defined [370](#)
 - documenting assumptions with [374](#)
 - enabling/disabling [371, 372](#)
- Assignment [40, 46](#)
- Asynchronous computations [589](#)
- Asynchronous methods [531](#)
- AsyncTask class (Android) [598](#)
- atan, atan2 methods
 - of Math [44](#)
- Atomic operations [570](#)
 - client-side locking for [566](#)
 - in concurrent hash maps [582](#)
 - performance of [571](#)
- AtomicType classes [570](#)
- @author [182](#)
- Autoboxing [228](#)
- AutoCloseable interface [362](#)
 - close method [362, 363](#)
- await method [521](#)
 - of Condition [555, 558](#)
- awaitTermination method
 - of ExecutorService [537](#)
- Azul [15](#)

B

- B, b conversion characters [70](#)
- Base classes. See Superclasses
- BASE64Encoder class [647](#)
- Basic multilingual planes [37](#)
- Batch files [168](#)
- Beans class [171](#)
- beep method
 - of Toolkit [296](#)
- BiConsumer interface [318](#)
- BiFunction interface [308, 318](#)
- BIG-5 [37](#)
- BigDecimal class [89, 92](#)
 - add method [92](#)
 - compareTo method [92](#)
 - Constructor [92](#)
 - divide method [92](#)
 - multiply method [92](#)
 - subtract method [92](#)
- BigInteger class [89, 91](#)
 - add method [91](#)
 - compareTo method [91](#)
 - divide method [91](#)
 - mod method [91](#)
 - multiply method [91](#)
 - pow method [91](#)
 - sqrt method [91](#)
 - subtract method [91](#)
 - valueOf method [89, 91](#)
- Binary search [497](#)
- BinaryOperator interface [318](#)
- binarySearch method
 - of Arrays [342, 498](#)
 - of Collections [497, 498](#)

BiPredicate interface [318](#)
 Bit masks [51](#)
 Bit sets [510](#)
 BitSet class [439, 510, 511](#)
 and method [511](#)
 andNot method [511](#)
 cardinality method [511](#)
 clear method [511](#)
 Constructor [511](#)
 get method [511](#)
 length method [511](#)
 or method [511](#)
 set method [511](#)
 size method [511](#)
 stream method [511](#)
 xor method [511](#)
 Bitwise operators [50, 52](#)
 Blank lines, printing [31](#)
 Blocking queues [575](#)
 BlockingDeque interface [580](#)
 offerFirst method [580](#)
 offerLast method [580](#)
 pollFirst method [580](#)
 pollLast method [580](#)
 putFirst method [580](#)
 putLast method [580](#)
 takeFirst method [580](#)
 takeLast method [580](#)
 BlockingQueue interface [579](#)
 add, element, peek, remove methods [575, 576](#)
 offer method [579](#)
 offer, poll, put, take methods [575, 576](#)
 poll method [579](#)
 put method [579](#)
 take method [579](#)
 Blocks [29, 31, 72](#)
 nested [72](#)
 synchronized [565](#)
 Boolean class [215, 509](#)
 converting from boolean [228](#)
 getBoolean method [509](#)
 hashCode method [215](#)
 boolean operators [48, 52](#)
 boolean type [38, 667](#)
 default initialization of [144](#)
 formatting output for [70](#)
 Bounded collections [442](#)
 Brace style
 Kernighan and Ritchie [30](#)
 break keyword [84, 89, 667](#)
 labeled/unlabeled [87, 88](#)
 not allowed in switch expressions [86](#)
 Bridge methods [406, 407, 427](#)
 Buckets (of hash tables) [462](#)
 Bulk operations [501](#)
 Byte class [215](#)
 converting from byte [228](#)
 hashCode method [215](#)
 toUnsignedInt method [34](#)
 byte type [33, 667](#)
 Bytecode files [31](#)
 Bytecodes
 engineering [632](#)
 at load time [638, 639](#)

C

C
 assert macro in [371](#)
 function pointers in [273](#)
 integer types in [5, 34](#)
 C# [6](#)
 polymorphism in [200](#)
 useful features of [9](#)
 C++
 #include in [160](#)
 >> operator in [51](#)
 , (comma) operator in [52](#)
 algorithms in [494](#)
 arrays in [96, 104](#)
 boolean values in [38](#)
 classes in [322](#)
 dynamic binding in [192](#)
 dynamic casts in [202](#)
 exceptions in [350, 353, 354](#)
 for loop in [80](#)
 function pointers in [273](#)
 inheritance in [188, 194, 290](#)
 integer types in [5, 33, 34](#)
 iterators as parameters in [505](#)
 methods in
 accessor [117](#)
 default [292](#)
 destructor [145](#)
 namespace directive in [160](#)
 NULL pointer in [114](#)
 object pointers in [114](#)
 operator overloading in [90](#)
 passing parameters in [140, 142](#)
 performance of, compared to Java [512](#)
 polymorphism in [200](#)
 protected modifier in [205](#)
 pure virtual functions (= 0) in [235](#)
 range-based for loop in [72](#)
 references in [114](#)
 Standard Template Library in [439, 443](#)
 syntax of [2](#)
 templates in [9, 404](#)
 type parameters in [400](#)
 using directive in [160](#)
 variables in
 redefining in nested blocks [73](#)
 vector template in [223](#)
 C, c conversion characters [70](#)
 CachedRowSetImpl class [647](#)
 Calendar class [115](#)
 get/setTime methods [198](#)
 Calendars
 displaying [117, 119](#)
 vs. time measurement [115](#)
 Call by reference [138](#)
 Call by value [138](#)
 call method
 of Callable [533](#)
 of ScopedValue.Carrier [545](#)
 Callable interface [533, 538](#)
 call method [531, 533](#)
 wrapper for [533](#)
 Callables [531](#)

- Callbacks [294](#)
- CamelCase [30](#)
- canAccess method
 - of AccessibleObject [270](#)
- Cancel [536](#)
- cancel method [532](#)
 - of Future [533](#), [599](#)
- CancellationException class [599](#)
- cardinality method
 - of BitSet [511](#)
- Carriage return character [36](#)
- case keyword [49](#), [83](#), [667](#)
- cast method
 - of Class [430](#)
- Casts [46](#), [200](#)
 - annotating [615](#)
 - bad [350](#)
 - checking before attempting [201](#)
- catch keyword [355](#), [667](#)
 - annotating parameters of [613](#)
- ceiling method
 - of NavigableSet [468](#)
- char type [36](#), [667](#)
- Character class [215](#)
 - converting from char [228](#)
 - hashCode method [215](#)
 - isJavaIdentifierXxx methods [39](#)
- Characters
 - escape sequences for [36](#)
 - exotic [38](#)
 - formatting output for [70](#)
- charAt method [55](#)
 - of String [58](#)
- CharSequence interface [59](#), [290](#)
- Checked exceptions [255](#), [258](#)
 - applicability of [369](#)
 - declaring [351](#)
 - suppressing with generics [425](#)
- Checked views [488](#)
- checkedCollection method
 - of Collections [491](#)
- checkedList method
 - of Collections [491](#)
- checkedMap method
 - of Collections [491](#)
- checkedNavigableMap method
 - of Collections [491](#)
- checkedNavigableSet method
 - of Collections [491](#)
- checkedQueue method
 - of Collections [491](#)
- checkedSet method
 - of Collections [491](#)
- checkedSortedMap method
 - of Collections [491](#)
- checkedSortedSet method
 - of Collections [491](#)
- Checker Framework [614](#)
- checkFromIndexSize, checkFromToIndex, checkIndex methods
 - of Objects [370](#)
- Child classes. See Subclasses
- children method
 - of ProcessHandle [607](#), [609](#)
- ChronoLocalDate [414](#)
- Church, Alonzo [305](#)
- Circular arrays [442](#)
- Clark, Jim [8](#)
- Class class [221](#), [255](#), [257](#), [260](#), [264](#), [270](#), [273](#), [430](#), [437](#)
 - arrayType method [271](#), [273](#)
 - cast method [430](#)
 - forName method [255](#), [257](#)
 - getClass method [255](#)
 - getComponentType method [271](#), [273](#)
 - getConstructor method [257](#), [430](#)
 - getConstructors method [261](#), [265](#)
 - getDeclaredConstructor method [430](#)
 - getDeclaredConstructors method [261](#), [265](#)
 - getDeclaredField method [270](#)
 - getDeclaredFields method [264](#), [270](#)
 - getDeclaredMethods method [261](#), [264](#), [274](#)
 - getEnumConstants method [430](#)
 - getField method [270](#)
 - getFields method [264](#), [270](#)
 - getFields, getDeclaredFields methods [261](#), [267](#)
 - getGenericInterfaces method [437](#)
 - getGenericSuperclass method [437](#)
 - getImage method [258](#)
 - getMethod method [274](#)
 - getMethods method [261](#), [264](#)
 - getName method [221](#), [255](#), [256](#)
 - getPackageName method [265](#)
 - getRecordComponents method [265](#)
 - getResource method [260](#)
 - getResource, getResourceAsStream methods [258](#), [259](#)
 - getResourceAsStream method [260](#), [653](#)
 - getSuperclass method [221](#), [430](#)
 - getTypeParameters method [437](#)
 - isArray method [273](#)
 - isEnum method [265](#)
 - isInterface method [265](#)
 - isRecord method [265](#)
 - newInstance method [257](#), [430](#)
- Class constants [41](#)
- Class declarations
 - annotations in [613](#), [621](#)
- Class diagrams [111](#)
- Class file API [632](#)
- Class files [162](#), [166](#)
 - compiling [31](#)
 - format of [632](#)
 - locating [167](#), [168](#)
 - modifying [633](#)
 - transformers for [638](#)
- class keyword [667](#)
 - implicitly declared [135](#)
- Class literals
 - no annotations for [615](#)
- Class loaders [341](#), [371](#)
- Class path [166](#), [169](#)
- Class wins rule [294](#)
- Class<T> parameters [430](#)
- ClassCastException class [201](#), [271](#), [288](#), [410](#), [424](#), [430](#), [488](#)
- Classes [108](#), [187](#)
 - abstract [233](#), [282](#), [289](#)
 - access privileges for [129](#)
 - adding to packages [162](#)
 - capabilities of [260](#)
 - companion [291](#), [292](#)

- constructors for [122](#)
- defining [120](#)
 - at runtime [340](#)
- deprecated [620](#)
- designing [109, 183](#)
- documentation comments for [175, 180](#)
- encapsulation of [108, 109, 127, 641](#)
- extending [109](#)
- final [198, 302](#)
- generic [221, 222, 397, 410, 422, 429, 432](#)
- immutable [130, 154, 278](#)
- implementing multiple interfaces [289, 290](#)
- importing [159](#)
- inner [322](#)
- instances of [108, 112](#)
- legacy [154](#)
- loading [391](#)
- names of [158, 184](#)
 - full package [159](#)
- nested [615](#)
- number of basic types in [183](#)
- objects of, at runtime [266](#)
- package scope of [165](#)
- parameters in [126](#)
- predefined [112](#)
- private methods in [130](#)
- protected [205](#)
- public [159, 175](#)
- relationships between [110](#)
- sealed [242](#)
- sharing, among programs [166](#)
- wrapper [228](#)
- ClassLoader class [375](#)
 - clearAssertionStatus method [375](#)
 - setClassAssertionStatus method [375](#)
 - setDefaultAssertionStatus method [375](#)
 - setPackageAssertionStatus method [375](#)
- CLASSPATH [168, 169](#)
- clear method
 - of BitSet [511](#)
 - of Collection [446, 447](#)
- clearAssertionStatus method
 - of ClassLoader [375](#)
- Client-side locking [565, 566](#)
- clone method
 - of array types [302](#)
 - of Object [129, 298, 307](#)
- Cloneable interface [298](#)
- CloneNotSupportedException class [301, 302](#)
- close method
 - of AutoCloseable [362, 363](#)
 - of Closeable [362](#)
 - of ExecutorService [537](#)
 - of Handler [387](#)
- Closures [315](#)
- Code errors [348](#)
- Code generator tools [621](#)
- Code planes [37](#)
- Code points, code units [37, 55](#)
- Collection interface [442, 447, 449, 457, 500](#)
 - add method [442, 446, 447, 449](#)
 - addAll method [446, 447](#)
 - clear method [446, 447](#)
 - contains method [447](#)
 - contains, containsAll methods [446, 457](#)
 - containsAll method [447](#)
 - equals method [446](#)
 - generic [445](#)
 - implementing [292](#)
 - isEmpty method [292, 446, 447](#)
 - iterator method [442, 447](#)
 - remove method [446, 447](#)
 - removeAll method [446, 447](#)
 - removeIf method [447, 500](#)
 - retain method [446](#)
 - retainAll method [447](#)
 - size method [446, 447](#)
 - stream method [292](#)
 - toArray method [225, 446, 447, 502](#)
- Collections class [439, 490, 496, 497, 498, 500, 505, 589](#)
 - addAll method [500](#)
 - algorithms for [493](#)
 - binarySearch method [497, 498](#)
 - bounded [442](#)
 - bulk operations in [501](#)
 - checkedCollection method [491](#)
 - checkedList method [491](#)
 - checkedMap method [491](#)
 - checkedNavigableMap method [491](#)
 - checkedNavigableSet method [491](#)
 - checkedQueue method [491](#)
 - checkedSet method [491](#)
 - checkedSortedMap method [491](#)
 - checkedSortedSet method [491](#)
 - concrete [451](#)
 - concurrent modifications of [457](#)
 - converting to arrays [502](#)
 - copy method [500](#)
 - debugging [457](#)
 - disjoint method [500](#)
 - elements of
 - inserting [449](#)
 - maximum [493](#)
 - removing [444](#)
 - traversing [443](#)
 - emptyEnumeration method [492](#)
 - emptyIterator method [492](#)
 - emptyList method [492](#)
 - emptyListIterator method [492](#)
 - emptyMap method [492](#)
 - emptyNavigableMap method [492](#)
 - emptyNavigableSet method [492](#)
 - emptySet method [492](#)
 - emptySortedMap method [492](#)
 - emptySortedSet method [492](#)
 - enumeration method [505](#)
 - fill method [500](#)
 - frequency method [500](#)
 - indexOfSubList method [500](#)
 - interfaces for [439](#)
 - lastIndexOfSubList method [500](#)
 - legacy [504](#)
 - list method [505](#)
 - max method [500](#)
 - min method [500](#)
 - mutable [484](#)
 - nCopies method [484, 491](#)
 - newSetFromMap method [490](#)

- ordered [449, 454](#)
- performance of [449, 463](#)
- replaceAll method [500](#)
- reverse method [500](#)
- rotate method [500](#)
- searching in [497](#)
- shuffle method [496, 497](#)
- singleton method [492](#)
- singletonList method [492](#)
- sort method [495, 497](#)
- sorted [465](#)
- swap method [500](#)
- synchronizedCollection method [491, 589](#)
- synchronizedCollection methods [489](#)
- synchronizedList method [491, 589](#)
- synchronizedMap method [491, 589](#)
- synchronizedNavigableMap method [491](#)
- synchronizedNavigableSet method [491](#)
- synchronizedSet method [491, 589](#)
- synchronizedSortedMap method [491, 589](#)
- synchronizedSortedSet method [491, 589](#)
- thread-safe [489, 574](#)
- unmodifiableCollection method [491](#)
- unmodifiableCollection methods [485, 486](#)
- unmodifiableList method [491](#)
- unmodifiableMap method [491](#)
- unmodifiableNavigableMap method [491](#)
- unmodifiableNavigableSet method [491](#)
- unmodifiableSequencedCollection method [491](#)
- unmodifiableSequencedMap method [491](#)
- unmodifiableSequencedSet method [491](#)
- unmodifiableSet method [491](#)
- unmodifiableSortedMap method [491](#)
- unmodifiableSortedSet method [491](#)
- using for method parameters [503](#)
- Command line
 - arguments in [96](#)
 - compiling/launching from [19, 31](#)
- command method
 - of ProcessHandle.Info [610](#)
- commandLine method
 - of ProcessHandle.Info [610](#)
- Comments [32](#)
 - automatic documentation and [32, 175](#)
 - blocks of [32](#)
 - not nesting [33](#)
 - to the end of line [32](#)
- Commons CSV library [654, 655](#)
- Compact compilation unit [135](#)
- Compact source file [161](#)
- Companion classes [291, 292](#)
- Comparable interface [282, 287, 342, 401, 402, 463, 495](#)
 - compareTo method [282, 287, 401, 413](#)
- Comparator interface [296, 304, 321, 495, 497](#)
 - chaining comparators in [321](#)
 - comparing method [321](#)
 - lambda expressions and [307](#)
 - naturalOrder method [321](#)
 - nullFirst/Last methods [321](#)
 - reversed method [497](#)
 - reversed, reverseOrder methods [322, 495](#)
 - reverseOrder method [497](#)
 - thenComparing method [321](#)
- comparator method
 - of SortedMap [474](#)
 - of SortedSet [468](#)
- compare method
 - of Double [287](#)
 - of Integer [287](#)
- compare method (integer types) [308](#)
- compareAndSet method [570](#)
- compareTo method
 - in subclasses [288](#)
 - of BigDecimal [92](#)
 - of BigInteger [91](#)
 - of Comparable [282, 287, 401, 413](#)
 - of Enum [242](#)
 - of String [58](#)
- Compilation unit
 - compact [135](#)
- Compiler
 - autoboxing in [230](#)
 - bridge methods in [406](#)
 - command-line options of [391](#)
 - creating bytecode files in [31](#)
 - deducting method types in [400](#)
 - enforcing throws specifiers in [356](#)
 - error messages in [352](#)
 - just-in-time [4, 5, 12, 200, 512](#)
 - launching [19](#)
 - optimizing method calls in [6, 200](#)
 - overloading resolution in [196](#)
 - shared strings in [56, 57](#)
 - translating typed array lists in [228](#)
 - type parameters in [396](#)
 - warnings in [85, 228](#)
 - whitespace in [30](#)
- CompletableFuture class [591](#)
 - acceptEither method [593, 594](#)
 - allOf, anyOf methods [593, 594](#)
 - applyToEither method [593, 594](#)
 - exceptionally, exceptionallyCompose methods [593](#)
 - handle method [593](#)
 - orTimeout method [593](#)
 - runAfterXxx methods [593, 594](#)
 - thenAccept, thenAcceptBoth, thenCombine, thenRun methods [593](#)
 - thenApply, thenApplyAsync, thenCompose methods [592, 593](#)
 - whenComplete method [593](#)
- CompletionStage interface [595](#)
- Components (of records) [154](#)
- Computations
 - asynchronous [589](#)
 - performance of [43, 44](#)
 - truncated [43](#)
- compute method
 - of Map [475](#)
- compute, computeIfXxx methods
 - of ConcurrentHashMap [583](#)
- computeIfAbsent method
 - of Map [476](#)
- computeIfPresent method
 - of Map [476](#)
- Concrete collections [451](#)
- Concrete methods [234](#)
- Concurrent hash maps
 - atomic updates in [582](#)
 - bulk operations on [585](#)

- efficiency of [581](#)
- size of [580](#)
- vs. synchronization wrappers [589](#)
- Concurrent modification detection [457](#)
- Concurrent programming [6](#), [515](#), [606](#)
 - records in [155](#)
 - synchronization in [547](#)
- Concurrent sets [587](#)
- ConcurrentHashMap class [580](#), [581](#)
 - atomic updates in [582](#)
 - compute, computeIfXxx methods [583](#)
 - forEach method [585](#)
 - forEach, forEachXxx methods [586](#)
 - get method [582](#)
 - keySet, newKeySet methods [587](#)
 - mappingCount method [580](#)
 - merge method [583](#)
 - organizing buckets as trees in [581](#)
 - put, putIfAbsent methods [582](#)
 - reduce, reduceXxx methods [585](#), [586](#)
 - replace method [582](#)
 - search, searchXxx methods [585](#), [586](#)
 - V> method [581](#)
- ConcurrentLinkedQueue class [580](#), [581](#)
 - ConcurrentLinkedQueue<E> method [581](#)
- ConcurrentLinkedQueue<E> method
 - of ConcurrentLinkedQueue [581](#)
- ConcurrentModificationException class [456](#), [457](#), [580](#), [589](#)
- ConcurrentSkipListMap class [582](#)
 - V> method [582](#)
- ConcurrentSkipListMap/Set classes [580](#)
- ConcurrentSkipListSet class [581](#)
 - ConcurrentSkipListSet<E> method [581](#)
- ConcurrentSkipListSet<E> method
 - of ConcurrentSkipListSet [581](#)
- Condition interface [558](#), [561](#)
 - await method [521](#), [558](#)
 - signal method [558](#)
 - signal, signalAll methods [560](#)
 - signalAll method [558](#)
 - vs. synchronization methods [563](#)
- Condition objects [554](#)
- Condition variables [554](#)
- Conditional operator [48](#)
 - with pattern matching [203](#)
- Conditional statements [73](#)
- Configuration files
 - editing [378](#)
- Console class [68](#), [69](#)
 - printing output to [29](#), [69](#)
 - reading input from [66](#)
 - readLine method [69](#)
 - readPassword method [69](#)
- console method
 - of System [69](#)
- ConsoleHandler class [379](#), [382](#), [387](#)
 - Constructor [387](#)
- const keyword [42](#), [667](#)
- Constants [41](#)
 - documentation comments for [178](#)
 - names of [41](#)
 - public [132](#)
 - static [132](#)
- Constructor class [257](#), [260](#), [265](#)
 - accessFlags method [265](#)
 - getDeclaringClass method [265](#)
 - getExceptionTypes method [265](#)
 - getModifiers method [265](#)
 - getModifiers, getName methods [260](#)
 - getName method [265](#)
 - getParameterTypes method [265](#)
 - getReturnType method [265](#)
 - newInstance method [257](#)
- Constructor expressions [422](#)
- Constructor references [313](#)
 - annotating [615](#)
- Constructors [122](#), [124](#), [143](#)
 - annotating [613](#), [615](#)
 - calling another constructor in [147](#)
 - canonical, compact, custom [156](#)
 - defined [112](#)
 - documentation comments for [175](#)
 - field initialization in [144](#), [146](#)
 - final [260](#)
 - initialization blocks in [148](#)
 - names of [112](#), [123](#)
 - no-argument [145](#), [191](#), [339](#)
 - overloading [143](#)
 - parameter names in [147](#)
 - private [260](#)
 - protected [175](#)
 - public [175](#), [260](#)
 - with super keyword [191](#)
- Consumer interface [318](#)
- Consumer threads [575](#)
- contains method
 - of Collection [446](#), [447](#), [457](#)
 - of HashSet [463](#)
- containsAll method [446](#), [457](#)
 - of Collection [447](#)
- containsKey method
 - of Map [473](#)
- containsValue method
 - of Map [473](#)
- Context
 - early execution [148](#)
- continue keyword [89](#), [667](#)
 - not allowed in switch expressions [86](#)
- Control flow [72](#)
 - block scope [72](#)
 - breaking [87](#)
 - conditional statements [73](#)
 - loops [75](#)
 - determinate [80](#)
 - “for each” [94](#)
 - multiple selections [83](#)
- Conversion characters [70](#)
- Coordinated Universal Time (UTC) [115](#)
- Copies [483](#)
 - unmodifiable [485](#)
- copy method
 - of Collections [500](#)
- copyOf method
 - of Arrays [95](#), [99](#), [270](#)
 - of EnumSet [482](#)
 - of List [490](#)
 - of List, Map, Set [485](#)
 - of Map [490](#)

- of Map.Entry [478](#)
- of Set [490](#)
- copyOfRange method
 - of Arrays [99](#)
- CopyOnWriteArrayList class [587](#), [589](#)
- CopyOnWriteArraySet class [587](#)
- CORBA [642](#)
- Cornell, Gary [1](#)
- Corruption of data [548](#), [551](#)
- cos method
 - of Math [44](#)
- Count of Monte Cristo, The* (Dumas) [598](#), [600](#)
- Covariant return types [407](#)
- CSV files [654](#), [655](#)
- Ctrl+\, for thread dump [560](#)
- Ctrl+C, for program termination [548](#), [557](#)
- current method
 - of ProcessHandle [607](#), [609](#)
 - of ThreadLocalRandom [574](#)
- currentThread method [524](#)
- of Thread [527](#)

D

- d conversion character [70](#)
- D, d suffixes (for double numbers) [35](#)
- daemon method
 - of Thread.Builder.OfPlatform [531](#)
- Daemon threads [527](#)
- Data types [33](#)
 - boolean type [38](#)
 - casting between [46](#)
 - char type [36](#)
 - conversions between [45](#), [200](#)
 - floating-point [34](#)
 - integer [33](#)
- Databases [611](#)
- DataFlavor class [642](#)
- Date and time
 - formatting output for [70](#)
 - hash codes for [213](#)
 - no built-in types for [112](#)
- Date class [115](#)
 - getDay/Month/Year methods (deprecated) [116](#)
 - toString method [113](#)
- Deadlocks [556](#), [559](#)
- Debugging [6](#), [388](#)
 - collections [457](#)
 - debuggers for [388](#)
 - generic types [488](#)
 - GUI programs [355](#)
 - including class names in [333](#)
 - messages for [354](#)
 - reflection for [267](#)
 - trapping program errors in a file for [390](#)
 - when running applications in terminal window [20](#)
- Decrement operators [47](#)
- decrementExact method [45](#)
- Deep copies [299](#)
- deepEquals method [210](#)
- deepToString method [101](#), [217](#)
- Default for annotation element [618](#)
- default keyword [84](#), [291](#), [667](#)
 - sealed classes and [244](#)

- Default methods [291](#)
 - conflicts in [292](#)
- Deferred execution [317](#)
- delete method
 - of StringBuilder [64](#)
- Dependence [110](#)
- Deprecated methods [116](#)
- Deque interface [468](#), [469](#)
 - offerFirst method [469](#)
 - offerLast method [469](#)
 - peekFirst method [469](#)
 - peekLast method [469](#)
 - pollFirst method [469](#)
 - pollLast method [469](#)
- Dequeues [468](#)
- Derived classes. See Subclasses
- descendants method
 - of ProcessHandle [607](#), [609](#)
- destroy method
 - of Process [609](#)
- destroy, destroyForcibly methods
 - of Process [606](#)
- destroyForcibly method
 - of Process [609](#)
- Determinate loops [80](#)
- Development environments
 - choosing [18](#)
 - in terminal window [20](#)
 - integrated [23](#)
- Device errors [348](#)
- Diamond syntax [222](#)
- Digital signatures [4](#)
- Directories
 - working, for a process [604](#)
- directory method
 - of ProcessBuilder [604](#), [608](#)
- disjoint method
 - of Collections [500](#)
- divide method
 - of BigDecimal [92](#)
 - of BigInteger [91](#)
- Division [42](#)
- do/while loop [77](#), [78](#), [667](#)
- Documentation comments [32](#), [175](#)
 - extracting [182](#)
 - for fields [178](#)
 - for methods [177](#)
 - for packages [178](#)
 - general [181](#)
 - HTML markup in [178](#)
 - hyperlinks in [180](#)
 - inserting [175](#)
 - links to other files in [180](#)
 - overview [183](#)
- doInBackground method [598](#), [599](#)
 - of SwingWorker [603](#)
- Double brace initialization [332](#)
- Double class [215](#), [287](#)
 - compare method [287](#)
 - converting from double [228](#)
 - hashCode method [215](#)
 - parseDouble method [66](#)
 - POSITIVE_INFINITY, NEGATIVE_INFINITY, NaN constants [35](#)
- double type [34](#), [667](#)

- arithmetic computations with [43](#)
- converting to other numeric types [45](#)
- Double-precision numbers [34](#)
- DoubleAccumulator, DoubleAdder classes [572](#)
- Doubly linked lists [453](#)
- Dynamic binding [192](#), [196](#)
- Dynamic languages [6](#)

E

- E
 - of Math [44](#)
- E, e conversion characters [35](#), [70](#)
- Early construction context [148](#)
- Eclipse [23](#), [388](#)
 - Adoptium [15](#)
 - configuring projects in [24](#)
 - IDE [649](#)
 - imports in [160](#)
 - running source files in [25](#)
 - Yasson framework [653](#)
- Effectively final variables [363](#)
- Eiffel programming language [290](#)
- Element interface [628](#)
- element method
 - of BlockingQueue [575](#), [576](#)
 - of Queue [469](#)
- Elements [612](#)
- else if [75](#), [76](#)
- else keyword [74](#), [667](#)
- Emoji characters [38](#)
- emptyEnumeration method
 - of Collections [492](#)
- emptyIterator method
 - of Collections [492](#)
- emptyList method
 - of Collections [492](#)
- emptyListIterator method
 - of Collections [492](#)
- emptyMap method
 - of Collections [492](#)
- emptyNavigableMap method
 - of Collections [492](#)
- emptyNavigableSet method
 - of Collections [492](#)
- emptySet method
 - of Collections [492](#)
- emptySortedMap method
 - of Collections [492](#)
- emptySortedSet method
 - of Collections [492](#)
- EmptyStackException class [368](#), [369](#)
- Encapsulation [108](#), [109](#), [641](#)
 - benefits of [127](#)
 - compile-time [657](#)
 - protected instance fields and [277](#)
- endsWith method
 - of String [58](#)
- ensureCapacity method [223](#)
 - of ArrayList [224](#)
- Enterprise Edition [9](#)
- entry method
 - of Map [484](#), [490](#)
- EntryLogger [638](#)
- EntryLoggingAgent.mf [638](#)
- entrySet method [476](#)
 - of Map [477](#)
- Enum class [238](#), [242](#)
 - compareTo method [242](#)
 - name method [242](#)
 - ordinal method [242](#)
 - toString, valueOf methods [239](#)
 - valueOf method [242](#)
- enum keyword [42](#), [667](#)
- Enumerated types [42](#)
 - equality testing for [239](#)
 - in switch statement [50](#)
- Enumeration interface [439](#), [504](#), [505](#)
 - asIterator method [505](#)
 - hasMoreElements method [505](#)
 - hasMoreElements, nextElement methods [443](#), [504](#)
 - nextElement method [505](#)
- Enumeration maps/sets [480](#)
- enumeration method
 - of Collections [505](#)
- Enumerations [238](#)
 - always final [200](#)
 - annotating [613](#)
 - declared inside a class [338](#)
 - implementing interfaces [289](#)
 - legacy [504](#)
- EnumMap class [480](#), [482](#)
 - as a concrete collection type [451](#)
 - Constructor [482](#)
- EnumSet class [480](#), [482](#)
 - allOf method [482](#)
 - as a concrete collection type [451](#)
 - copyOf method [482](#)
 - noneOf method [482](#)
 - of method [482](#)
 - range method [482](#)
- environment method
 - of ProcessBuilder [609](#)
- Environment variables, modifying [605](#)
- EOFException class [353](#)
- Epoch [115](#)
- Equals [294](#)
 - hashCode method and [212](#), [214](#)
 - implementing [210](#)
 - inheritance and [208](#)
 - of proxy classes [344](#)
 - of records [154](#), [208](#)
 - redefining [212](#), [214](#)
 - wrappers and [229](#)
- equals method
 - of Annotation [619](#)
 - of Arrays [99](#), [210](#), [211](#)
 - of Collection [446](#)
 - of Object [206](#), [220](#), [486](#)
 - of Objects [211](#)
 - of Set [450](#)
 - of String [57](#), [58](#)
- equalsIgnoreCase method [57](#)
 - of String [58](#)
- Error class [349](#)
- Errors
 - checking, in mutator methods [128](#)
 - code [348](#)

- device [348](#)
 - internal [349](#), [352](#), [373](#)
 - messages for [357](#)
 - physical limitations [348](#)
 - user input [348](#)
 - Escape sequences [36](#)
 - Exception class [350](#), [366](#)
 - Constructor [366](#)
 - Exception handlers [257](#), [348](#)
 - Exception specification [351](#)
 - exceptionally, exceptionallyCompose methods
 - of CompletableFuture [593](#)
 - exceptionally, exceptionallyCompose methods (CompletableFuture) [593](#)
 - exceptionNow method
 - of Future [533](#)
 - Exceptions [349](#)
 - annotating [615](#)
 - ArrayIndexOutOfBoundsException [94](#), [350](#), [352](#)
 - ArrayStoreException [195](#), [410](#), [419](#), [421](#)
 - CancellationException [599](#)
 - catching [125](#), [257](#), [301](#), [352](#), [355](#)
 - changing type of [358](#)
 - checked [255](#), [258](#), [350](#), [353](#), [367](#), [369](#)
 - ClassCastException [201](#), [271](#), [288](#), [410](#), [424](#), [430](#), [488](#)
 - CloneNotSupportedException [301](#), [302](#)
 - ConcurrentModificationException [456](#), [457](#), [580](#), [589](#)
 - creating classes for [353](#), [354](#)
 - documentation comments for [177](#)
 - EmptyStackException [368](#), [369](#)
 - EOFException [353](#)
 - FileNotFoundException [351](#), [352](#)
 - finally clause in [360](#)
 - generics in [425](#)
 - hierarchy of [349](#), [369](#)
 - IllegalAccessException [266](#)
 - IllegalStateException [444](#), [575](#)
 - InaccessibleObjectException [267](#)
 - InterruptedException [516](#), [524](#), [532](#)
 - InvocationTargetException [257](#)
 - IOException [351](#), [353](#), [356](#), [362](#)
 - logging [383](#)
 - micromanaging [368](#)
 - NoSuchElementException [442](#)
 - NullPointerException [125](#), [126](#), [230](#), [313](#), [350](#), [369](#), [370](#)
 - NumberFormatException [369](#)
 - out-of-bounds [370](#)
 - propagating [356](#), [369](#)
 - rethrowing and chaining [358](#), [390](#)
 - RuntimeException [350](#), [369](#)
 - ServletException [358](#)
 - squelching [369](#)
 - stack trace for [364](#)
 - “throw early, catch late” [370](#)
 - throwing [257](#), [353](#)
 - TimeoutException [532](#)
 - tips for using [367](#)
 - type variables in [403](#)
 - uncaught [390](#), [522](#), [528](#)
 - unchecked [258](#), [350](#), [352](#), [369](#)
 - unexpected [383](#)
 - UnsupportedOperationException [477](#), [486](#), [489](#)
 - variables for, implicitly final [358](#)
 - vs. simple tests [367](#)
 - wrapping [359](#)
 - exec method
 - of Runtime [603](#)
 - Executable class [274](#)
 - Executable JAR files [172](#)
 - ExecutableElement interface [628](#)
 - Execute [598](#)
 - execute method
 - of SwingWorker [603](#)
 - ExecutorCompletionService class [538](#), [542](#)
 - Constructor [542](#)
 - poll method [542](#)
 - submit method [542](#)
 - take method [542](#)
 - Executors class [534](#), [537](#)
 - groups of tasks, controlling [537](#)
 - newCachedThreadPool method [537](#)
 - newFixedThreadPool method [537](#)
 - newSingleThreadExecutor method [537](#)
 - newThreadPerTaskExecutor method [537](#)
 - newVirtualThreadPerTaskExecutor method [537](#)
 - Executors class, newXxx methods [534](#)
 - ExecutorService interface [537](#), [542](#)
 - awaitTermination method [537](#)
 - close method [537](#)
 - invokeAll method [542](#)
 - invokeAny method [542](#)
 - invokeAny/All methods [538](#)
 - shutdown method [536](#), [537](#)
 - shutdownNow method [536](#), [537](#)
 - submit method [535](#), [537](#)
 - exitValue method [606](#)
 - of Process [609](#)
 - exp method
 - of Math [44](#)
 - Explicit parameters [126](#)
 - Exploratory programming [5](#)
 - exports keyword [646](#), [648](#), [649](#), [660](#), [668](#)
 - Expressions [47](#)
 - extends keyword [187](#), [401](#), [402](#), [668](#)
- ## F
- F, f conversion characters [70](#)
 - F, f suffixes (for float numbers) [35](#)
 - factory method
 - of Thread.Builder [531](#)
 - Factory methods [134](#)
 - Fair locks [554](#)
 - Fallthrough behavior [85](#)
 - false literal [668](#)
 - fdlibm library [44](#)
 - Field class [260](#), [265](#), [270](#)
 - accessFlags method [265](#)
 - get method [266](#), [270](#)
 - getDeclaringClass method [265](#)
 - getExceptionTypes method [265](#)
 - getModifiers method [265](#)
 - getModifiers, getName methods [260](#)
 - getName method [265](#)
 - getParameterTypes method [265](#)
 - getReturnType method [265](#)
 - getType method [260](#)
 - set method [270](#)

Fields

- adding, in subclasses [190](#)
- annotating [613](#)
- default initialization of [144](#)
- documentation comments for [175, 178](#)
- final [132, 198](#)
- instance [108, 123, 127, 130, 146, 183](#)
- private [183, 189, 190](#)
- protected [175, 205, 277](#)
- public [175, 178](#)
- public static final [289](#)
- static [131, 149, 161, 424](#)
- volatile [568](#)
- with the null value [125](#)

File handlers [380, 381](#)

FileHandler class [380, 382, 387](#)

- Constructor [387](#)

FileNotFoundException class [351, 352](#)

Files class

- reading
 - all words from [363](#)
 - in a separate thread [598](#)

fill method

- of Arrays [99](#)
- of Collections [500](#)

Filter class [382](#)

Filter interface [388](#)

- isLoggable method [388](#)

final keyword [41, 198, 668](#)

- checking [260](#)
- for fields in interfaces [289](#)
- for instance fields [130](#)
- for methods in superclass [288](#)
- for shared fields [569](#)
- inner classes and [329](#)

finalize method

- of Object [145](#)

finally keyword [360, 668](#)

- return statements in [362](#)
- unlock operation in [552](#)
- without catch [361](#)

Financial calculations [35](#)

findFirst method

- of ServiceLoader [340](#)

first method

- of SortedSet [468](#)

First Person, Inc. [8](#)

firstKey method

- of SortedMap [474](#)

Flags, for formatted output [71](#)

Float class [215](#)

- converting from float [228](#)
- hashCode method [215](#)
- POSITIVE_INFINITY, NEGATIVE_INFINITY, NaN constants [35](#)

float type [34, 668](#)

- converting to other numeric types [45](#)

Floating-point numbers [34](#)

- arithmetic computations with [43](#)
- converting from/to integers [200](#)
- equality of [81](#)
- formatting output for [70](#)
- rounding [35, 46](#)

floor method

- of NavigableSet [468](#)

floorMod method [43](#)

flush method

- of Handler [387](#)

“for each” loop [95](#)

- for array lists [226](#)
- for collections [443, 589](#)
- for multidimensional arrays [100](#)

for keyword [80, 668](#)

- comma-separated expressions in [52](#)
- defining variables inside [81](#)
- for collections [442](#)

forEach method

- of ConcurrentHashMap [585](#)
- of Map [473](#)
- of StackWalker [366](#)

forEach, forEachXxx methods

- of ConcurrentHashMap [586](#)

forEachRemaining method [442](#)

- of Iterator [448](#)

Fork-join framework [545](#)

Form feed character [36](#)

format method

- of Formatter [388](#)

Format specifiers [70, 72](#)

formatMessage method

- of Formatter [388](#)

Formattable interface [70](#)

formatted method

- of String [69](#)

Formatter class [382, 388](#)

- format method [388](#)
- formatMessage method [388](#)
- getHead method [388](#)
- getTail method [388](#)

forName method [255](#)

- of Class [257](#)

frequency method

- of Collections [500](#)

from method

- of Random [153](#)

Function interface [318, 320](#)

Functional interfaces [307, 620, 621](#)

- abstract methods in [307](#)
- annotating [320](#)
- conversion to [308](#)
- generic [308](#)
- using supertype bounds in [414](#)

Functions. See Methods

Future interface [533, 538](#)

- cancel method [533](#)
- cancel, get methods [532, 536, 599](#)
- exceptionNow method [533](#)
- get method [533](#)
- isCancelled method [533](#)
- isCancelled, isDone methods [532, 536](#)
- isDone method [533](#)
- resultNow method [533](#)
- state method [533](#)

Futures [531](#)

- combining [593, 594](#)
- completable [591](#)

FutureTask class [531, 534](#)

- Constructor [534](#)

G

- G, g conversion characters [70](#)
- Garbage collection [114](#)
 - hash maps and [478](#)
- GB18030 [37](#)
- General Public License (GPL) [12](#)
- Generic programming [395](#)
 - arrays and [314, 423](#)
 - classes in [221, 222, 397](#)
 - extending/implementing other generic classes [410](#)
 - no throwing or catching instances of [403](#)
 - collection interfaces in [502](#)
 - converting to raw types [410](#)
 - debugging [488](#)
 - expressions in [405](#)
 - in JVM [404, 431](#)
 - inheritance rules for [409, 411](#)
 - legacy code and [408](#)
 - methods in [399, 406, 445](#)
 - reflection and [429](#)
 - required skill levels for [396](#)
 - static fields or methods and [424](#)
 - type erasure in [404, 418, 423](#)
 - clashes after [426](#)
 - type matching in [430](#)
 - vs. inheritance [395](#)
 - wildcard types in [410](#)
- Generic types
 - annotating [614](#)
- GenericArrayType interface [431, 432, 438](#)
 - getGenericComponentType method [438](#)
- get method
 - of Array [273](#)
 - of ArrayList [224, 227](#)
 - of BitSet [511](#)
 - of ConcurrentHashMap [582](#)
 - of Field [266, 270](#)
 - of Future [532, 533, 536, 599](#)
 - of LinkedList [457](#)
 - of List [449, 460](#)
 - of LongAccumulator [571](#)
 - of Map [449, 472, 473](#)
 - of Paths [291](#)
 - of ScopedValue [544](#)
 - of ServiceLoader.Provider [339, 340](#)
 - of ThreadLocal [543](#)
 - of Vector [567](#)
- getAccessor method
 - of RecordComponent [266](#)
- getActualTypeArguments method
 - of ParameterizedType [438](#)
- getAndUpdate, getAndAccumulate methods
 - of AtomicXxx [571](#)
- getAnnotation method
 - of AnnotatedElement [626](#)
- getAnnotation, getAnnotationsByType methods
 - of AnnotatedConstruct [628](#)
 - of AnnotatedElement [624, 625](#)
- getAnnotations method
 - of AnnotatedElement [626](#)
- getAnnotationsByType method
 - of AnnotatedElement [626](#)
- getBoolean method
 - of Boolean [509](#)
- getBounds method
 - of TypeVariable [437](#)
- getCause method
 - of Throwable [365](#)
- getClass method
 - of Class [255](#)
 - of Object [220](#)
- getClassName method
 - of StackTraceElement [367](#)
 - of StackWalker.StackFrame [366](#)
- getComponentType method [271](#)
 - of Class [273](#)
- getConstructor method
 - of Class [257, 430](#)
- getConstructors method [261](#)
 - of Class [265](#)
- getDay method
 - of Date [116](#)
- getDayOfMonth method
 - of LocalDate [116, 119](#)
- getDayOfWeek method
 - of LocalDate [119](#)
- getDeclaredAnnotations method
 - of AnnotatedElement [626](#)
- getDeclaredAnnotationXxx methods
 - of AnnotatedElement [624](#)
- getDeclaredConstructor method
 - of Class [430](#)
- getDeclaredConstructors method [261](#)
 - of Class [265](#)
- getDeclaredField method
 - of Class [270](#)
- getDeclaredFields method [261, 267](#)
 - of Class [264, 270](#)
- getDeclaredMethods method [261, 274](#)
 - of Class [264](#)
- getDeclaringClass method
 - of Constructor [265](#)
 - of Field [265](#)
 - of Method [265](#)
 - of StackWalker.StackFrame [367](#)
- getDefault method
 - of RandomGenerator [153](#)
- getDefaultToolkit method
 - of Toolkit [296](#)
- getDefaultUncaughtExceptionHandler method
 - of Thread [529](#)
- getElementsAnnotatedWith method [628](#)
- getEnclosedElements method [628](#)
- getEnumConstants method [430](#)
 - of Class [430](#)
- getErrorStream method [604, 605](#)
 - of Process [609](#)
- getExceptionTypes method
 - of Constructor [265](#)
 - of Field [265](#)
 - of Method [265](#)
- getField method
 - of Class [270](#)
- getFields method [261](#)
 - of Class [264, 270](#)
- getFileName method
 - of StackTraceElement [367](#)

- of StackWalker.StackFrame [366](#)
- getFilter method
 - of Handler [387](#)
- getFirst method
 - of LinkedList [461](#)
 - of SequencedCollection [469](#)
- getFormatter method
 - of Handler [387](#)
- getGenericComponentType method
 - of GenericArrayType [438](#)
- getGenericInterfaces method
 - of Class [437](#)
- getGenericParameterTypes method
 - of Method [437](#)
- getGenericReturnType method
 - of Method [437](#)
- getGenericSuperclass method
 - of Class [437](#)
- getGlobal method [389](#)
- getHead method [382](#)
 - of Formatter [388](#)
- getImage method
 - of Class [258](#)
- getInputStream method [604](#)
 - of Process [609](#)
- getInstance method [364](#)
 - of StackWalker [366](#)
- getInstant method
 - of LogRecord [388](#)
- getInteger method
 - of Integer [510](#)
- getKey method
 - of Map.Entry [478](#)
- getLast method
 - of LinkedList [461](#)
 - of SequencedCollection [469](#)
- getLength method [271](#)
 - of Array [273](#)
- getLevel method
 - of Handler [387](#)
 - of LogRecord [387](#)
- getLineNumber method
 - of StackTraceElement [367](#)
 - of StackWalker.StackFrame [366](#)
- getLogger method [378](#)
 - of System [376](#)
- getLoggerName method
 - of LogRecord [387](#)
- getLong method
 - of Long [510](#)
- getLongThreadID method
 - of LogRecord [388](#)
- getLowerBounds method
 - of WildcardType [438](#)
- getMessage method
 - of LogRecord [387](#)
 - of Throwable [355](#)
- getMethod method [274](#)
- getMethodName method
 - of StackTraceElement [367](#)
 - of StackWalker.StackFrame [367](#)
- getMethods method [261](#)
 - of Class [264](#)
- getMillis method
 - of LogRecord [388](#)
- getModifiers method
 - of Constructor [265](#)
 - of Field [265](#)
 - of java.lang.reflect.Member [260](#)
 - of Method [265](#)
- getMonth method
 - of Date [116](#)
- getMonthValue method
 - of LocalDate [116, 119](#)
- getName method
 - of Class [221, 255, 256](#)
 - of Constructor [265](#)
 - of Field [265](#)
 - of java.lang.reflect.Member [260](#)
 - of Method [265](#)
 - of RecordComponent [265](#)
 - of System.Logger [386](#)
 - of Thread [528](#)
 - of TypeVariable [437](#)
- getOrDefault method
 - of Map [473](#)
- getOutputStream method [604](#)
 - of Process [609](#)
- getOwnerType method
 - of ParameterizedType [438](#)
- getPackageName method
 - of Class [265](#)
- getParameters method
 - of LogRecord [388](#)
- getParameterTypes method
 - of Constructor [265](#)
 - of Field [265](#)
 - of Method [265](#)
- getProperties method [508](#)
 - of System [508](#)
- getProperty method
 - of Properties [507](#)
 - of System [509](#)
- getProxyClass method [345](#)
 - of Proxy [345](#)
- getQualifiedName method [628](#)
- getRawType method
 - of ParameterizedType [438](#)
- getRecordComponents method
 - of Class [265](#)
- getResource method
 - of Class [260](#)
- getResource, getResourceAsStream methods
 - of Class [258, 259](#)
- getResourceAsStream method
 - of Class [260, 653](#)
 - of Module [653](#)
- getResourceBundle method
 - of LogRecord [387](#)
- getResourceBundleName method
 - of LogRecord [387](#)
- getReturnType method
 - of Constructor [265](#)
 - of Field [265](#)
 - of Method [265](#)
- getSequenceNumber method
 - of LogRecord [388](#)
- getSimpleName method

- of Element [628](#)
- getSourceClassName method
 - of LogRecord [388](#)
- getSourceMethodName method
 - of LogRecord [388](#)
- getStackTrace method [364](#)
 - of Throwable [366](#)
- getState method
 - of SwingWorker [603](#)
 - of Thread [523](#)
- getSuperclass method
 - of Class [221](#), [430](#)
- getSuppressed method [363](#)
 - of Throwable [366](#)
- getTail method [382](#)
 - of Formatter [388](#)
- Getters/setters, generated automatically [630](#)
- getThrown method
 - of LogRecord [388](#)
- getTime method [198](#)
- getType method
 - of Field [260](#)
 - of RecordComponent [265](#)
- getTypeParameters method
 - of Class [437](#)
 - of Method [437](#)
- getUncaughtExceptionHandler method
 - of Thread [529](#)
- getUpperBounds method
 - of WildcardType [438](#)
- getValue method
 - of Map.Entry [478](#)
- getXxx method
 - of Array [273](#)
- getYear method
 - of Date [116](#)
 - of LocalDate [116](#), [119](#)
- GMT (Greenwich Mean Time) [115](#)
- Goetz, Brian [515](#), [568](#)
- Gosling, James [8](#), [9](#)
- goto keyword [72](#), [87](#), [668](#)
- Graphical User Interface
 - debugging [355](#)
 - long-running tasks in [597](#)
- Green project [8](#)
- GregorianCalendar class [117](#)
 - add method [117](#)
 - constructors for [115](#), [143](#)
- group method
 - of Thread.Builder.OfPlatform [531](#)
- GUI. See Graphical User Interface

H

- H, h conversion characters [70](#)
- handle method
 - of CompletableFuture [593](#)
- Handler class [387](#)
 - close method [387](#)
 - flush method [387](#)
 - getFilter method [387](#)
 - getFormatter method [387](#)
 - getLevel method [387](#)
 - publish method [387](#)
 - setFilter method [387](#)
 - setFormatter method [387](#)
 - setLevel method [387](#)
- Hansen, Per Brinch [567](#), [568](#)
- Hash codes [211](#), [461](#)
 - default [212](#)
 - formatting output for [70](#)
- Hash collisions [213](#), [462](#)
- Hash maps [471](#)
 - concurrent [580](#)
 - identity [481](#)
 - linked [478](#)
 - setting [471](#)
 - vs. tree maps [471](#)
 - weak [478](#)
- hash method
 - of Objects [213](#), [214](#)
- Hash sets [461](#)
 - linked [478](#)
- Hash tables [461](#), [462](#)
 - legacy [504](#)
 - load factor of [463](#)
 - rehashing [463](#)
- hashCode method [211](#)
 - equals method and [212](#), [214](#)
 - null-safe [213](#)
 - of Annotation [619](#)
 - of Arrays [213](#), [215](#)
 - of Boolean [215](#)
 - of Byte [215](#)
 - of Character [215](#)
 - of Double [215](#)
 - of Float [215](#)
 - of Integer [215](#)
 - of LocalDate [213](#)
 - of Long [215](#)
 - of Object [214](#), [465](#)
 - of Objects [213](#), [214](#)
 - of proxy classes [344](#)
 - of records [154](#), [214](#)
 - of Set [450](#)
 - of Short [215](#)
 - of String [461](#)
- HashMap class [471](#), [473](#)
 - as a concrete collection type [451](#)
 - Constructor [473](#)
 - newHashMap method [473](#)
- HashSet class [463](#), [465](#)
 - add, contains methods [463](#)
 - as a concrete collection type [451](#)
 - Constructor [465](#)
 - iterating over [443](#)
 - newHashSet method [465](#)
- Hashtable class [439](#), [504](#), [588](#)
 - as a concrete collection type [451](#)
 - synchronized methods [504](#)
- hasMoreElements method [443](#), [504](#)
 - of Enumeration [505](#)
- hasNext method
 - of Iterator [442](#), [443](#), [447](#)
 - of Scanner [69](#)
- hasNextDouble method
 - of Scanner [69](#)
- hasNextInt method

- of Scanner [69](#)
- hasPrevious method [455](#)
 - of ListIterator [460](#)
- “Has-a” relationship [110](#)
- headMap method
 - of NavigableMap [493](#)
 - of SortedMap [487, 492](#)
- headSet method
 - of NavigableSet [487, 492](#)
 - of SortedSet [487, 492](#)
- Heap [470](#)
- Helper methods [130, 291, 416](#)
- Hexadecimal numbers
 - formatting output for [70](#)
 - prefix for [34](#)
- HexFormat class [70](#)
- higher method
 - of NavigableSet [468](#)
- Hoare, Tony [567](#)
- Hold count [553](#)
- HotJava browser [9](#)
- Hotspot just-in-time compiler [16, 512](#)
- HTML [9, 11](#)
 - generating documentation in [630](#)
 - in JavaDoc comments [178](#)

I

- Identifiers [667](#)
- Identity hash maps [481](#)
- identityHashCode method [481](#)
 - of System [221, 483](#)
- IdentityHashMap class [481, 482](#)
 - as a concrete collection type [451](#)
 - Constructor [482](#)
- identityToString method
 - of Objects [221](#)
- IEEE 754 specification [35, 44](#)
- if keyword [73, 668](#)
- IllegalAccessException class [266](#)
- IllegalStateException class [444, 575](#)
- Immutable classes [130, 278](#)
- Implementations [440](#)
- implements keyword [283, 668](#)
- Implicit parameters [126](#)
 - none, in static methods [133](#)
 - state of [389](#)
- Implicitly declared class [135](#)
- import keyword [159, 668](#)
 - no annotations for [615](#)
- InaccessibleObjectException class [267](#)
- increment method
 - of LongAdder [571](#)
- Increment operators [47](#)
- Incremental linking [5](#)
- incrementAndGet method [570](#)
- incrementExact method [45](#)
- Indentation, in text blocks [66](#)
- Index class [94](#)
- indexOf method
 - of List [460](#)
 - of String [59](#)
- indexOfSubList method
 - of Collections [500](#)

- Inferred types [306](#)
- info method
 - of ProcessHandle [610](#)
- Information hiding. See Encapsulation
- Inheritance [110, 187](#)
 - design hints for [277](#)
 - equality testing and [208](#)
 - hierarchies of [193](#)
 - multiple [194, 290](#)
 - preventing [198](#)
 - private fields and [189](#)
 - vs. type parameters [395, 409](#)
- inheritIO method
 - of ProcessBuilder [608](#)
- initCause method
 - of Throwable [365](#)
- Initialization blocks [148](#)
 - static [149](#)
- Inlining [6, 200](#)
- Inner classes [322](#)
 - accessing object state with [323](#)
 - anonymous [330](#)
 - applicability of [327](#)
 - defined [322](#)
 - local [328](#)
 - private [324](#)
 - static [323, 334](#)
 - syntax of [326](#)
 - translated into regular classes [327](#)
 - vs. lambda expressions [308](#)
- Input, reading [66](#)
- insert method
 - of StringBuilder [64](#)
- Instance field
 - declared with prefix [147](#)
- Instance fields [108](#)
 - final [130](#)
 - initializing [148, 183](#)
 - explicit [146](#)
 - names of [154](#)
 - not present in interfaces [282, 289](#)
 - private [123, 183](#)
 - protected [277](#)
 - shadowing [124, 147](#)
 - values of [127, 128](#)
 - volatile [568](#)
 - vs. local variables [124, 127, 144](#)
- Instance method [54](#)
- instanceof keyword [52, 201, 210, 288, 668](#)
 - annotating [615](#)
 - pattern matching for [202](#)
- Instances [108](#)
 - creating on the fly [257](#)
- Instrumentation API [638](#)
- int type [33, 668](#)
 - converting to other numeric types [45](#)
 - fixed size for [5](#)
 - platform-independent [34](#)
- Integer class [215, 231, 287, 510](#)
 - compare method [287, 308](#)
 - converting from int [228](#)
 - getInteger method [510](#)
 - hashCode method [215](#)
 - intValue method [231](#)

- parseInt method [66](#), [230](#), [232](#)
- toString method [231](#)
- valueOf method [232](#)
- Integer types [33](#)
 - arithmetic computations with [42](#)
 - arrays of [217](#)
 - computations of [45](#)
 - converting from/to floating-point [200](#)
 - formatting output for [70](#)
 - no unsigned types in Java [34](#)
- Integrated Development Environment (IDE) [23](#)
- IntelliJ IDEA [23](#)
- @interface [616](#), [617](#), [618](#)
- interface keyword [282](#), [668](#)
- Interface types [441](#)
- Interface variables [288](#)
- Interfaces [281](#)
 - abstract classes and [289](#)
 - annotating [613](#), [615](#)
 - binary- vs. source-compatible [292](#)
 - callbacks and [294](#)
 - constants in [289](#)
 - declared inside a class [338](#)
 - documentation comments for [175](#)
 - evolution of [292](#)
 - extending [288](#)
 - for custom algorithms [502](#)
 - functional [307](#), [620](#), [621](#)
 - implementing [283](#), [288](#), [291](#)
 - methods in
 - clashes between [293](#)
 - nonabstract [307](#)
 - private [291](#)
 - static [291](#)
 - no instance fields in [282](#), [289](#)
 - properties of [288](#)
 - public [175](#)
 - sealed [289](#)
 - tagging [300](#), [450](#)
 - vs. implementations [440](#)
- Internal errors [349](#), [352](#), [373](#)
- Internationalization. See Localization
- Internet Explorer [8](#)
- Interpreted languages [12](#)
- Interpreter [5](#)
- interrupt method
 - of Thread [524](#), [527](#)
- interrupted method
 - of Thread [526](#), [527](#)
- InterruptedException class [516](#), [524](#), [532](#)
- Intrinsic locks [561](#), [568](#), [569](#)
- Introduction to Algorithms* (Cormen et al.) [465](#)
- intValue method
 - of Integer [231](#)
- Invocation handlers [340](#)
- InvocationHandler interface [340](#), [345](#)
 - invoke method [345](#)
 - invokeDefault method [345](#)
- InvocationTargetException class [257](#)
- invoke method
 - of InvocationHandler [340](#), [345](#)
 - of Method [273](#), [276](#)
- invokeAll method
 - of ExecutorService [542](#)
- invokeAny method
 - of ExecutorService [542](#)
- invokeAny/All methods (ExecutorService) [538](#)
- invokeDefault method
 - of InvocationHandler [345](#)
- IO [31](#)
 - print method [69](#)
 - readln method [66](#)
- IO class [66](#), [68](#)
 - print method [68](#)
 - println method [68](#)
 - readln method [68](#)
- IOException class [351](#), [353](#), [356](#), [362](#)
- isAbstract method
 - of Modifier [266](#)
- isAlive method [606](#)
 - of Process [609](#)
- isAnnotationPresent method
 - of AnnotatedElement [626](#)
- isArray method
 - of Class [273](#)
- isBound method
 - of ScopedValue [545](#)
- isCancelled method
 - of Future [533](#)
- isCancelled, isDone methods
 - of Future [532](#)
- isCancelled, isDone methods (Future) [536](#)
- isDone method
 - of Future [533](#)
- isEmpty method
 - of Collection [292](#), [446](#), [447](#)
 - of String [58](#)
- isEnum method
 - of Class [265](#)
- isFinal method [260](#)
 - of Modifier [266](#)
- isInterface method
 - of Class [265](#)
 - of Modifier [266](#)
- isInterrupted method [524](#)
 - of Thread [527](#)
- isJavaIdentifierXXX methods
 - of Character [39](#)
- isLoggable method
 - of Filter [382](#), [388](#)
 - of System.Logger [386](#)
- isNaN method [35](#)
- isNative method
 - of Modifier [266](#)
- isNativeMethod method
 - of StackTraceElement [367](#)
 - of StackWalker.StackFrame [367](#)
- ISO 8601 format [621](#)
- ISO 8859-1 [37](#), [506](#)
- isPrivate method
 - of Modifier [266](#)
- isPrivate, isProtected, isPublic methods
 - of Modifier [260](#)
- isProtected method
 - of Modifier [266](#)
- isProxyClass method [345](#)
 - of Proxy [345](#)
- isPublic method

- of Modifier [266](#)
- isRecord method
 - of Class [265](#)
- isStatic method
 - of Modifier [266](#)
- isStrict method
 - of Modifier [266](#)
- isSynchronized method
 - of Modifier [266](#)
- isVirtual method
 - of Thread [524](#)
- isVolatile method
 - of Modifier [266](#)
- “Is-a” relationship [110, 194, 277](#)
- Iterable interface [94](#)
- Iterator interface [442, 447](#)
 - “for each” loop [443](#)
 - forEachRemaining method [442, 448](#)
 - generic [445](#)
 - hasNext method [442, 443, 447](#)
 - next method [442, 445, 448](#)
 - remove method [442, 444, 445, 448](#)
- iterator method
 - of Collection [442, 447](#)
 - of ServiceLoader [340](#)
- Iterators [442](#)
 - being between elements [444](#)
 - weakly consistent [580](#)
- IzPack [172](#)

J

- J#, J++ programming languages [6](#)
- Jar [170, 649](#)
 - command-line options of [170, 172, 174](#)
- Jar Bundler [172](#)
- JAR files [166, 169](#)
 - analyzing dependencies of [663, 664](#)
 - creating [170](#)
 - executable [172](#)
 - file resources in [653](#)
 - in jre/lib/ext directory [169](#)
 - manifest of [171, 654](#)
 - META-INF/services directory [661](#)
 - modular [649, 655](#)
 - multi-release [172](#)
 - resources and [258](#)
 - scanning for deprecated elements [620](#)
- Java [19](#)
 - add-exports option [657](#)
 - add-opens option [657](#)
 - illegal-access option [657](#)
 - module, --module-path options [644](#)
 - javaagent option [638](#)
 - architecture-neutral object file format of [4](#)
 - as a programming platform [1](#)
 - available under GPL [12](#)
 - backward compatibility of [172, 203, 321, 395](#)
 - basic syntax of [29, 120](#)
 - case-sensitiveness of [29, 39, 504](#)
 - command-line options of [174, 371](#)
 - design of [2](#)
 - documentation for [18](#)
 - dynamic [6](#)

- history of [8](#)
- interpreter in [5](#)
- libraries in [3, 9, 11](#)
 - installing [18](#)
- misconceptions about [11](#)
- networking capabilities of [3](#)
- no multiple inheritance in [290](#)
- no operator overloading in [90](#)
- no unsigned types in [34](#)
- reliability of [3](#)
- security of [4, 12](#)
- simplicity of [2, 305](#)
- strongly typed [33, 285](#)
- versions of [9, 10](#)
- vs. C++ [2, 512](#)
- Java bug parade [31](#)
- Java Collections Framework [439](#)
 - algorithms in [493](#)
 - converting to/from arrays in [502](#)
 - copies and views in [483](#)
 - interfaces in [448](#)
 - vs. implementations [440](#)
 - legacy classes in [504](#)
 - operations in
 - bulk [501](#)
 - optional [489](#)
 - vs. traditional collections libraries [443](#)
- Java Concurrency in Practice* (Goetz) [515](#)
- Java Development Kit (JDK) [4, 15](#)
 - documentation in [59](#)
 - downloading [15](#)
 - installation of [15](#)
 - default [170](#)
 - obsolete features in [642](#)
 - setting up [16](#)
- Java Language Specification [31](#)
- Java Memory Model and Thread Specification [568](#)
- Java Persistence Architecture [611](#)
- Java Platform Module System [641, 665](#)
 - migration to [654, 658](#)
- Java Runtime Environment (JRE) [16](#)
- Java Virtual Machine (JVM) [5](#)
 - generics in [404, 431](#)
 - launching [19](#)
 - managing applications in [391](#)
 - method tables in [197](#)
 - thread priority levels in [529](#)
 - truncating computations in [43](#)
 - watching class loading in [391](#)
- Java Virtual Machine Specification [31, 632](#)
- java.awt package [642](#)
- java.desktop module [658, 659](#)
- java.lang.annotation package [619](#)
- java.lang.Object class [109](#)
- java.lang.reflect package [260, 270](#)
- java.logging module [658](#)
- java.se module [659](#)
- java.util.Collections class [496](#)
- java.util.concurrent package [551](#)
 - efficient collections in [580](#)
- java.util.concurrent.atomic package [570](#)
- java.util.function package [308](#)
- java.util.logging package [375, 378](#)
- java.util.Timer class [295](#)

Javac [19](#)
 -processor option [627](#)
 -XprintRounds option [630](#)
 current directory in [167](#)
 JavaDoc [175](#)
 command-line options of [182](#)
 comments in [175](#), [178](#)
 extracting [182](#)
 overview [183](#)
 redeclaring Object methods for [307](#)
 HTML markup in [178](#)
 including annotations in [621](#)
 links in [180](#)
 online documentation of [183](#)
 JavaFX [598](#)
 javafx.css.CssParser class [172](#)
 javan.log files [380](#)
 Javap [173](#), [327](#)
 JavaScript [13](#)
 for of loop in [72](#)
 javax.annotation package [619](#)
 javax.swing.Timer class [295](#)
 JAXB [652](#)
 JCommander [611](#)
 Jconsole [379](#), [391](#), [559](#), [560](#)
 Jdeprscan [620](#)
 Jdeps [663](#), [664](#)
 JEP 264 (platform logging API) [375](#)
 Jimage [665](#)
 Jlink [664](#)
 Jmod [665](#)
 JMOD files [665](#)
 Jmol applet [7](#)
 join method
 of String [59](#)
 of Thread [521](#), [522](#), [523](#)
 JOptionPane class [296](#)
 showMessageDialog method [296](#)
 JShell [5](#), [25](#)
 JShell, loading modules into [651](#)
 JSlider class
 setLabelTable method [408](#)
 JSON [242](#)
 JSON-B [652](#), [653](#)
 JUnit [611](#), [612](#)
 JUnit framework [389](#)
 Just-in-time compiler [4](#), [5](#), [12](#), [200](#), [512](#)
 JVM
 specification for [632](#)

K

Key/value pairs
 in annotations [612](#), [618](#)
 keySet method
 of ConcurrentHashMap [587](#)
 of Map [476](#), [477](#)
 Keywords [667](#)
 hyphenated [245](#)
 not used [42](#)
 redundant [289](#)
 reserved [40](#)
 restricted [667](#)
 Knuth, Donald [87](#)

KOI-8 [37](#)

L

L, l suffixes (for long integers) [34](#)
 Lambda expressions [304](#)
 accessing variables in [314](#)
 annotating targets for [621](#)
 atomic updates with [570](#)
 capturing values by [315](#)
 for loggers [377](#)
 functional interfaces and [307](#)
 method references and [310](#)
 not for variables of type Object [308](#)
 parameter types of [306](#)
 processing [317](#)
 result type of [306](#)
 scope of [316](#)
 syntax of [305](#)
 this keyword in [316](#)
 vs. inner classes [308](#)
 vs. method references [313](#)
 Language model API [628](#)
 last method
 of SortedSet [468](#)
 lastIndexOf method
 of List [460](#)
 of String [59](#)
 lastIndexOfSubList method
 of Collections [500](#)
 lastKey method
 of SortedMap [474](#)
 Launch4J [172](#)
 Legacy classes [154](#)
 generics and [408](#)
 Legacy collections [504](#)
 bit sets [510](#)
 enumerations [504](#)
 hash tables [504](#)
 property maps [505](#)
 stacks [510](#)
 Length
 of arrays [94](#)
 length method
 of BitSet [511](#)
 of String [55](#), [57](#), [58](#)
 of StringBuilder [64](#)
 Line feed character
 escape sequence for [36](#)
 in output [31](#), [64](#)
 in text blocks [64](#)
 Linked hash maps/sets [478](#)
 Linked lists [453](#)
 concurrent modifications of [457](#)
 doubly linked [453](#)
 printing [459](#)
 random access in [457](#), [494](#)
 removing elements from [454](#)
 LinkedBlockingDeque class [579](#)
 Constructor [579](#)
 LinkedBlockingQueue method [579](#)
 LinkedBlockingDeque method
 of LinkedBlockingQueue [579](#)
 LinkedBlockingQueue class [576](#), [579](#)

- Constructor [579](#)
- LinkedBlockingDeque method [579](#)
- LinkedBlockingQueue method
 - of LinkedBlockingDeque [579](#)
- LinkedHashMap class [478, 482](#)
 - access vs. insertion order in [479](#)
 - as a concrete collection type [451](#)
 - Constructor [482](#)
 - removeEldestEntry method [480, 482](#)
- LinkedHashSet class [478, 481](#)
 - as a concrete collection type [451](#)
 - Constructor [481](#)
- LinkedList class [454, 457, 460, 468](#)
 - addFirst method [461](#)
 - addLast method [461](#)
 - as a concrete collection type [451](#)
 - Constructor [460, 461](#)
 - get method [457](#)
 - getFirst method [461](#)
 - getLast method [461](#)
 - listIterator method [455](#)
 - next/previousIndex methods [458](#)
 - removeAll method [458](#)
 - removeFirst method [461](#)
 - removeLast method [461](#)
- Linux
 - IDEs for [23](#)
 - JDK in [15](#)
 - no thread priorities in OpenJDK VM for [529](#)
 - paths in [167, 168](#)
 - troubleshooting Java programs in [20](#)
- List class [449](#)
 - add method [449](#)
 - copyOf method [485](#)
 - get method [449](#)
 - of method [483, 502](#)
 - remove method [449](#)
 - set method [449](#)
 - subList method [487](#)
- List interface [460, 490, 492, 497, 501](#)
 - add method [460](#)
 - addAll method [460](#)
 - copyOf method [490](#)
 - get method [460](#)
 - indexOf method [460](#)
 - lastIndexOf method [460](#)
 - listIterator method [460](#)
 - of method [490](#)
 - remove method [460](#)
 - replaceAll method [501](#)
 - set method [460](#)
 - sort method [497](#)
 - subList method [492](#)
- list method
 - of Collections [505](#)
- ListIterator interface [457, 460](#)
 - add method [454, 456, 460](#)
 - hasPrevious method [455, 460](#)
 - nextIndex method [460](#)
 - previous method [455, 460](#)
 - previousIndex method [460](#)
 - remove method [456](#)
 - set method [456, 460](#)
- listIterator method
 - of LinkedList [455](#)
 - of List [460](#)
- Lists [449](#)
 - modifiable/resizable [496](#)
 - with given elements [483](#)
- load method
 - of Properties [506, 507](#)
 - of ServiceLoader [340](#)
- Load time [638](#)
- Local inner classes [328](#)
 - accessing variables from outer methods in [329](#)
- Local variables
 - annotating [408, 613, 614](#)
 - vs. instance fields [124, 127, 144](#)
- LocalDate class [115, 119](#)
 - getDayOfMonth method [119](#)
 - getDayOfWeek method [119](#)
 - getMonthValue method [119](#)
 - getYear method [119](#)
 - hashCode method [213](#)
 - minusDays method [120](#)
 - now method [119](#)
 - now, of methods [115](#)
 - of method [119](#)
 - plusDays method [116, 120](#)
 - processing arrays of [414](#)
- Locales [71](#)
- Localization [112, 258, 259](#)
- Lock interface [554, 558, 561](#)
 - await method [555](#)
 - lock method [554](#)
 - newCondition method [555, 558](#)
 - signal method [556](#)
 - signalAll method [555](#)
 - tryLock method [521](#)
 - unlock method [552, 554](#)
 - vs. synchronization methods [563](#)
- lock method
 - of Lock [554](#)
- Locks [551](#)
 - client-side [566](#)
 - condition objects for [554](#)
 - deadlocks [556, 559](#)
 - fair [554](#)
 - hold count for [553](#)
 - in synchronized blocks [565](#)
 - intrinsic [561, 568, 569](#)
 - not with try-with-resources statement [552](#)
 - not wrapper objects for [230](#)
 - reentrant [553](#)
- Log file pattern variables [381](#)
- Log handlers [379](#)
 - filtering/formatting [382](#)
- Log messages, adding to classes [633](#)
- log method
 - of System.Logger [376, 386](#)
- log, log10 methods
 - of Math [44](#)
- Log4j [375](#)
- Logback [375](#)
- Logger class
 - getGlobal method [389](#)
- Loggers
 - filtering/formatting [382](#)

- hierarchy of [379](#)
- naming [376](#)
- Logging [375](#)
 - configuring [378, 379](#)
 - including class names in [333](#)
 - levels of [377, 379](#)
 - messages for [217](#)
 - recipe for [383](#)
- Logging proxy [389](#)
- Logical “and”, “or” [48](#)
- Logical conditions [38](#)
- LogRecord class [387](#)
 - getInstant method [388](#)
 - getLevel method [387](#)
 - getLoggerName method [387](#)
 - getLongThreadID method [388](#)
 - getMessage method [387](#)
 - getMillis method [388](#)
 - getParameters method [388](#)
 - getResourceBundle method [387](#)
 - getResourceBundleName method [387](#)
 - getSequenceNumber method [388](#)
 - getSourceClassName method [388](#)
 - getSourceMethodName method [388](#)
 - getThrown method [388](#)
- Long class [215, 510](#)
 - converting from long [228](#)
 - getLong method [510](#)
 - hashCode method [215](#)
- Long Term Support (LTS) [16](#)
- long type [33, 668](#)
 - platform-independent [34](#)
- LongAccumulator class, methods of [571](#)
- LongAdder class [571, 583](#)
 - add, increment, sum methods [571](#)
- Loops
 - break statements in [87](#)
 - continue statements in [89](#)
 - determinate (for) [80](#)
 - “for each” [94](#)
 - while [75](#)
- lower method
 - of NavigableSet [468](#)

M

- Mac OS X
 - executing JARs in [172](#)
 - IDEs for [23](#)
 - JDK in [15](#)
- main ,method
 - launching [32](#)
- main method [134](#)
 - body of [29](#)
 - not defined [150](#)
 - separate for each class [389](#)
 - String[] args parameter of [96](#)
- MANIFEST.MF [171](#)
 - editing [171](#)
 - newline characters in [172](#)
- Map interface [449, 473, 475, 477, 490](#)
 - compute method [475](#)
 - computeIfAbsent method [476](#)
 - computeIfPresent method [476](#)

- containsKey method [473](#)
- containsValue method [473](#)
- copyOf method [485, 490](#)
- entry method [484, 490](#)
- entrySet method [476, 477](#)
- forEach method [473](#)
- get method [449, 472, 473](#)
- getOrDefault method [473](#)
- keySet method [476, 477](#)
- merge method [475](#)
- of method [483, 484, 490](#)
- ofEntries method [484, 490](#)
- put method [449, 472, 473](#)
- putAll method [473](#)
- putIfAbsent method [476](#)
- remove method [472](#)
- replaceAll method [476](#)
- values method [476, 478](#)
- Map.Entry interface [476, 478](#)
 - copyOf method [478](#)
 - getKey method [478](#)
 - getValue method [478](#)
 - setValue method [478](#)
- mappingCount method [580](#)
- Maps [471](#)
 - adding/retrieving objects to/from [471](#)
 - concurrent [580](#)
 - garbage collecting [478](#)
 - hash vs. tree [471](#)
 - implementations for [471](#)
 - keys for [472](#)
 - enumerating [476](#)
 - subranges of [487](#)
 - with given key/value pairs [483](#)
- Marker interfaces [300](#)
- Math class [27, 43](#)
 - E, PI static constants [44](#)
 - floorMod method [43](#)
 - log, log10 methods [44](#)
 - pow method [43, 133](#)
 - round method [46](#)
 - sqrt method [43, 274, 275](#)
 - trigonometric functions [44](#)
 - xxxExact methods [45](#)
- max method
 - of Collections [500](#)
- MAX_PRIORITY field
 - of Thread [530](#)
- Maximum value, computing [398](#)
- merge method
 - of ConcurrentHashMap [583](#)
 - of Map [475](#)
- Merge sort algorithm [495](#)
- Meta-annotations [617, 623](#)
- META-INF [171](#)
- META-INF/versions directory [172](#)
- Method class [260, 265, 276, 437](#)
 - accessFlags method [265](#)
 - getDeclaringClass method [265](#)
 - getExceptionTypes method [265](#)
 - getGenericParameterTypes method [437](#)
 - getGenericReturnType method [437](#)
 - getModifiers method [265](#)
 - getModifiers, getName methods [260](#)

- getName method [265](#)
- getParameterTypes method [265](#)
- getReturnType method [265](#)
- getTypeParameters method [437](#)
- instance [54](#)
- invoke method [273, 276](#)
- static [54](#)
- toString method [261](#)
- Method parameters. See Parameters
- Method pointers [273, 274, 275](#)
- Method references [310](#)
 - annotating [615](#)
 - this, super parameters in [313](#)
 - vs. lambda expressions [313](#)
- Method tables [197](#)
- MethodHandles class [654](#)
- Methods [108](#)
 - abstract [234](#)
 - in functional interfaces [307](#)
 - accessor [116, 127, 128, 412](#)
 - adding logging messages to [633](#)
 - adding, in subclasses [190](#)
 - annotating [613](#)
 - applying to objects [113](#)
 - asynchronous [531](#)
 - body of [29, 31](#)
 - bridge [406, 407, 427](#)
 - calling by reference vs. by value [138](#)
 - casting [200](#)
 - chaining calls of [320](#)
 - concrete [234](#)
 - conflicts in [292](#)
 - consistent [208](#)
 - default [291](#)
 - deprecated [116, 620](#)
 - destructor [145](#)
 - documentation comments for [175, 180](#)
 - dynamic binding for [192, 196](#)
 - error checking in [128](#)
 - exception specification in [351](#)
 - factory [134](#)
 - final [196, 200, 260, 288](#)
 - generic [399, 406, 445](#)
 - getters/setters, generated automatically [630](#)
 - helper [130, 416](#)
 - inlining [6, 200](#)
 - invoking [31, 273](#)
 - mutator [116, 128, 412](#)
 - names of [154, 184](#)
 - overloading [144](#)
 - overriding [189, 211, 278, 619, 620](#)
 - exceptions and [352](#)
 - return type and [406](#)
 - package scope of [165](#)
 - passing objects to [113](#)
 - private [130, 196, 260, 291](#)
 - protected [175, 205, 277, 302](#)
 - public [175, 260, 283](#)
 - reflexive [208](#)
 - return type of [144, 196](#)
 - signature of [144, 196](#)
 - static [133, 161, 196, 424, 563](#)
 - adding to interfaces [291](#)
 - symmetric [208](#)
 - tracing [341](#)
 - transitive [208](#)
 - used for serialization [619, 621](#)
 - utility [291](#)
 - varargs [232, 420](#)
 - visibility of, in subclasses [198](#)
- Micro Edition [9](#)
- Microsoft
 - ActiveX [4](#)
 - C# [6, 9, 200](#)
 - Internet Explorer [8](#)
 - J#, J++ [6](#)
 - JDK in [15](#)
 - .NET platform [4](#)
 - Visual Basic [2, 112](#)
 - Visual Studio [18](#)
- Microsoft Windows. See Windows operating system
- min method
 - of Collections [500](#)
- MIN_PRIORITY field
 - of Thread [530](#)
- Minimum value, computing [398](#)
- minusDays method
 - of LocalDate [120](#)
- mod method
 - of BigInteger [91](#)
- Modifier class [266](#)
 - isAbstract method [266](#)
 - isFinal method [266](#)
 - isInterface method [266](#)
 - isNative method [266](#)
 - isPrivate method [266](#)
 - isProtected method [266](#)
 - isPublic method [266](#)
 - isStatic method [266](#)
 - isStrict method [266](#)
 - isSynchronized method [266](#)
 - isVolatile method [266](#)
 - isXxx methods [260](#)
 - toString method [266](#)
- Module class
 - getResourceAsStream method [653](#)
- module keyword [644, 668](#)
- Module path [169](#)
- Module-info.class [650, 654](#)
- Module-info.java [643, 654](#)
- Modules [10, 166, 641, 665](#)
 - accessing [651, 657](#)
 - automatic [654, 657](#)
 - declaration of [644](#)
 - explicit [656](#)
 - exporting packages [646](#)
 - loading into JShell [651](#)
 - migration to [654, 658](#)
 - naming [642, 654](#)
 - not passing access rights [645](#)
 - open [653](#)
 - opening packages in [653](#)
 - packages with the same names in [649](#)
 - qualified exports of [660](#)
 - reading other modules [646](#)
 - requiring [645](#)
 - service implementations and [661](#)
 - tools for [663](#)

- unnamed [267](#), [656](#)
- versioning [642](#), [644](#)
- Modulus [42](#)
- Monitor concept [567](#)
- Mosaic [8](#)
- Multi-release JARs [172](#)
- Multidimensional arrays [99](#), [103](#)
 - printing [217](#)
 - ragged [102](#)
- Multiple inheritance [290](#)
 - not supported in Java [194](#)
- Multiple selections [83](#)
- Multiplication [42](#)
- multiply method
 - of BigDecimal [92](#)
 - of BigInteger [91](#)
- multiplyExact method [45](#)
- Multitasking [515](#)
- Multithreading [6](#), [515](#)
 - deadlocks in [556](#), [559](#)
 - deferred execution in [317](#)
 - performance and [571](#), [576](#)
 - synchronization in [547](#)
 - using pools for [534](#)
- Mutator methods [116](#), [412](#)
 - error checking in [128](#)

N

- n conversion character [70](#)
- Name
 - qualified [158](#), [160](#), [339](#)
 - simple [160](#)
 - of enumeration [238](#)
- name method
 - of Enum [242](#)
 - of Thread.Builder [531](#)
- NaN [35](#)
- native keyword [668](#)
- naturalOrder method [321](#)
- Naughton, Patrick [8](#), [9](#)
- NavigableMap interface [450](#), [493](#)
 - headMap method [493](#)
 - subMap method [493](#)
 - tailMap method [493](#)
- NavigableSet interface [450](#), [467](#), [468](#), [492](#)
 - ceiling method [468](#)
 - floor method [468](#)
 - headSet method [492](#)
 - headSet, subSet, tailSet methods [487](#)
 - higher method [468](#)
 - lower method [468](#)
 - pollFirst method [468](#)
 - pollLast method [468](#)
 - subSet method [492](#)
 - tailSet method [492](#)
- nCopies method [484](#)
 - of Collections [491](#)
- negateExact method [45](#)
- Negation operator [48](#)
- Negative infinity [35](#)
- Nested classes
 - annotating [615](#)
- .NET platform [4](#)

- NetBeans [23](#), [388](#)
- Netscape [8](#)
 - LiveScript/JavaScript [13](#)
 - Navigator browser [8](#)
- Networking [3](#)
- new keyword [52](#), [112](#), [123](#), [668](#)
 - in constructor references [313](#)
 - not for interfaces [288](#)
 - return value of [114](#)
 - with arrays [92](#)
 - with generic classes [222](#)
 - with threads [521](#)
- newCachedThreadPool method [534](#)
 - of Executors [537](#)
- newCondition method [555](#)
 - of Lock [558](#)
- newFixedThreadPool method [534](#)
 - of Executors [537](#)
- newHashMap method
 - of HashMap [473](#)
- newHashSet method
 - of HashSet [465](#)
- newInstance method
 - of Array [271](#), [273](#)
 - of Class [257](#), [430](#)
 - of Constructor [257](#)
 - of ScopedValue [544](#)
- newKeySet method [587](#)
- Newline. See Line feed character
- newProxyInstance method [341](#), [345](#)
 - of Proxy [345](#)
- newSetFromMap method
 - of Collections [490](#)
- newSingleThreadExecutor method
 - of Executors [537](#)
- newSingleThreadXxx methods
 - of Executors [534](#)
- newThread method
 - of ThreadFactory [530](#)
- newThreadPerTaskExecutor method
 - of Executors [537](#)
- newVirtualThreadPerTaskExecutor method
 - of Executors [537](#)
- next method
 - of Iterator [442](#), [445](#), [448](#)
 - of Scanner [68](#)
- nextDouble method [67](#)
 - of Scanner [69](#)
- nextElement method [443](#), [504](#)
 - of Enumeration [505](#)
- nextInt method
 - of LinkedList [458](#)
 - of ListIterator [460](#)
- nextInt method
 - of RandomGenerator [151](#), [153](#)
 - of Scanner [67](#), [69](#)
- nextLine method [67](#)
 - of Scanner [68](#)
- No-argument constructors [145](#), [191](#), [339](#)
- non-sealed keyword [245](#), [668](#)
- noneOf method
 - of EnumSet [482](#)
- NORM_PRIORITY field
 - of Thread [530](#)

NoSuchElementException class [442](#)
 Notepad [20](#)
 notify method
 of Object [565](#)
 notify, notifyAll methods
 of Object [562](#)
 notifyAll method
 of Object [565](#)
 now method
 of LocalDate [115, 119](#)
 null literal [114, 668](#)
 as a reference [125](#)
 equality testing to [208](#)
 nullFirst/Last methods
 of Comparator [321](#)
 NullPointerException class [50, 125, 126, 230, 313, 350, 369, 370](#)
 Number class [228](#)
 NumberFormat class [232](#)
 factory methods [134](#)
 parse method [232](#)
 NumberFormatException class [369](#)
 Numbers
 floating-point [34, 46, 70, 81, 200](#)
 Java Virtual Machine (JVM) truncating computations in [43](#)
 generated random [574](#)
 hexadecimal [34, 70](#)
 octal [34, 70](#)
 prime [511](#)
 rounding [35, 46](#)
 unsigned [34](#)
 Numeric types
 casting [46](#)
 comparing [48, 321](#)
 converting
 to other numeric types [45, 200](#)
 to strings [230](#)
 default initialization of [144](#)
 fixed sizes for [5](#)
 precision of [69, 89](#)
 printing [69](#)

O

o conversion character [70](#)
 Oak [8, 350](#)
 Object class [109, 205, 214, 220, 465, 565](#)
 clone method [129, 298, 307](#)
 equals method [206, 220, 294, 486](#)
 getClass method [220](#)
 hashCode method [212, 214, 465](#)
 no redefining for methods of [294](#)
 notify method [565](#)
 notify, notifyAll methods [562](#)
 notifyAll method [565](#)
 toString method [215, 220, 294, 307](#)
 wait method [521, 562, 565](#)
 Object references
 as method parameters [139](#)
 converting [200](#)
 default initialization of [144](#)
 modifying [139](#)
 Object traversal algorithms [481](#)

Object variables [235](#)
 in predefined classes [112](#)
 initializing [113](#)
 setting to null [114](#)
 vs. C++ object pointers [114](#)
 vs. objects [113](#)
 Object-oriented programming (OOP) [3, 107, 187](#)
 passing objects in [294](#)
 time measurement in [115](#)
 vs. procedural [107](#)
 Object-relational mappers [652](#)
 Objects [107, 109](#)
 analyzing at runtime [266](#)
 applying methods to [113](#)
 behavior of [109](#)
 cloning [298](#)
 comparing [288](#)
 concatenating with strings [216](#)
 constructing [108, 143](#)
 default hash codes of [212](#)
 destruction of [145](#)
 equality testing for [206, 256](#)
 finalize method of [145](#)
 identity of [109](#)
 implementing an interface [288](#)
 in predefined classes [112](#)
 initializing [113](#)
 intrinsic locks of [561](#)
 passing to methods [113](#)
 references to [114](#)
 runtime type identification of [255](#)
 serializing [481](#)
 sorting [283](#)
 state of [108, 109, 323](#)
 vs. object variables [113](#)
 Objects class [138, 211, 214, 221](#)
 checkXxx methods [370](#)
 equals method [211](#)
 hash method [214](#)
 hash, hashCode methods [213](#)
 hashCode method [214](#)
 identityToString method [221](#)
 requireNonNull method [126, 138, 370](#)
 requireNonNullElse method [126, 138](#)
 requireNonNullElseGet method [138](#)
 Octal numbers
 formatting output for [70](#)
 prefix for [34](#)
 of method
 of EnumSet [482](#)
 of List [490](#)
 of List, Map, Set [483, 502](#)
 of LocalDate [115, 119](#)
 of Map [490](#)
 of Path [291](#)
 of ProcessHandle [607, 609](#)
 of RandomGenerator [153](#)
 of Set [490](#)
 ofEntries method
 of Map [484, 490](#)
 offer method
 of BlockingQueue [575, 576, 579](#)
 of Queue [469](#)
 offerFirst method

- of BlockingDeque [580](#)
- of Deque [469](#)
- offerLast method
 - of BlockingDeque [580](#)
 - of Deque [469](#)
- ofPlatform method
 - of Thread [531](#)
- ofVirtual method
 - of Thread [531](#)
- On-demand initialization [572](#)
- onExit method
 - of Process [609](#)
- Online documentation [58](#), [59](#), [175](#), [182](#)
- open keyword [653](#), [668](#)
- OpenJ9 just-in-time compiler [16](#)
- OpenJDK [15](#), [16](#)
- opens keyword [653](#), [660](#), [669](#)
- Operators
 - arithmetic [42](#)
 - bitwise [50](#), [52](#)
 - boolean [48](#)
 - hierarchy of [51](#)
 - increment/decrement [47](#)
 - no overloading for [90](#)
 - relational [48](#)
- Optional operations [489](#)
- or method
 - of BitSet [511](#)
- Oracle [9](#)
- Ordered collections [449](#), [454](#)
- ordinal method
 - of Enum [242](#)
- orElse method
 - of ScopedValue [544](#)
- org.omg.corba package [642](#)
- orTimeout method [593](#)
- OSGi platform [338](#)
- Out-of-bounds exceptions [370](#)
- Output
 - formatting [69](#)
 - statements in [53](#)
- Overloading resolution [143](#), [196](#)
- overview.html file [183](#)

P

- p (hexadecimal floating-point literals) [35](#)
- package keyword [159](#), [162](#), [669](#)
- Package-info.java [178](#), [614](#)
- package.html file [178](#)
- Packages [158](#), [641](#)
 - accessing [165](#)
 - adding classes into [162](#)
 - annotating [613](#), [614](#)
 - documentation comments for [175](#), [178](#)
 - exporting [646](#)
 - hidden [649](#)
 - importing [159](#)
 - names of [158](#), [255](#)
 - opening [653](#)
 - split [650](#)
 - unnamed [162](#), [165](#), [182](#), [372](#)
- Parallelism threshold [585](#)
- parallelXxx methods

- of Arrays [587](#)
- Parameter variables
 - annotating [613](#)
- Parameterized types. See Type parameters
- ParameterizedType interface [431](#), [432](#), [438](#)
 - getActualTypeArguments method [438](#)
 - getOwnerType method [438](#)
 - getRawType method [438](#)
- Parameters [138](#)
 - checking, with assertions [373](#)
 - documentation comments for [177](#)
 - explicit [126](#)
 - implicit [126](#), [133](#), [389](#)
 - modifying [139](#), [141](#)
 - names of [147](#)
 - using collection interfaces in [503](#)
 - variable number of
 - passing generic types to [420](#)
- Parent classes. See Superclasses
- parse method
 - of NumberFormat [232](#)
- parseDouble method
 - of Double [66](#)
- parseInt method [230](#)
 - of Integer [66](#), [232](#)
- Pascal [8](#)
 - compiled code in [4](#)
 - passing parameters in [140](#)
- Passwords
 - reading from console [68](#)
- Path interface, of method [291](#)
- Paths class, get method [291](#)
- Pattern matching [202](#)
- Payne, Jonathan [9](#)
- peek method
 - of BlockingQueue [575](#), [576](#)
 - of Queue [469](#)
 - of Stack [510](#)
- peekFirst method
 - of Deque [469](#)
- peekLast method
 - of Deque [469](#)
- Performance [6](#)
 - computations and [43](#), [44](#)
 - JAR files and [166](#)
 - measuring [511](#), [513](#)
 - multithreading and [571](#), [576](#)
 - of collections [449](#), [463](#), [581](#)
 - of Java vs. C++ [512](#)
 - of simple tests vs. catching exceptions [368](#)
- permits keyword [243](#), [289](#), [669](#)
- Physical limitations [348](#)
- PI
 - of Math [44](#), [132](#)
- Picocli [611](#)
- pid method
 - of ProcessHandle [610](#)
- Platform logging API [375](#), [379](#)
- plusDays method
 - of LocalDate [116](#), [120](#)
- Point class [153](#), [154](#)
- poll method
 - of BlockingQueue [575](#), [576](#), [579](#)
 - of ExecutorCompletionService [542](#)

- of Queue [469](#)
- pollFirst method
 - of BlockingDeque [580](#)
 - of Deque [469](#)
 - of NavigableSet [468](#)
- pollLast method
 - of BlockingDeque [580](#)
 - of Deque [469](#)
 - of NavigableSet [468](#)
- Polymorphism [192, 194, 244, 278](#)
- pop method
 - of Stack [510](#)
- Portability [5, 11, 43](#)
- Positive infinity [35](#)
- pow method
 - of BigInteger [91](#)
 - of Math [43, 133](#)
- powExact method [45](#)
- Precision, of numbers [69](#)
- Preconditions [374](#)
- Predefined classes [112](#)
 - mutator and accessor methods in [116](#)
 - objects, object variables in [112](#)
- Predicate interface [309, 318](#)
- Prefix
 - of instance field name [147](#)
- premain method (Instrumentation API) [638](#)
- previous method
 - of ListIterator [455, 460](#)
- previousIndex method
 - of LinkedList [458](#)
 - of ListIterator [460](#)
- Prime numbers [511](#)
- Primitive types [33](#)
 - as method parameters [139](#)
 - comparing [321](#)
 - converting to objects [228](#)
 - final fields of [130](#)
 - not for type parameters [418](#)
 - transforming hash map values to [586](#)
 - values of, not object [206](#)
- Princeton University [4](#)
- print method
 - of IO [31, 68, 69](#)
- Print statements
 - for logging [375](#)
- Printf
 - arguments of [232](#)
 - conversion characters for [70](#)
 - flags for [71](#)
- println method
 - of IO [31, 68](#)
- printStackTrace method [364, 390](#)
 - of Throwable [257](#)
- priority method
 - of Thread.Builder.OfPlatform [531](#)
- Priority queues [470](#)
- PriorityBlockingQueue class [576, 579](#)
 - Constructor [579](#)
- PriorityQueue class [471](#)
 - as a concrete collection type [451](#)
 - Constructor [471](#)
- private keyword [123, 165, 324, 669](#)
 - checking [260](#)
 - for fields, in superclasses [190](#)
 - for methods [130](#)
- Procedures [107](#)
- Process class [603, 609](#)
 - destroy method [609](#)
 - destroy, destroyForcibly methods [606](#)
 - destroyForcibly method [609](#)
 - exitValue method [606, 609](#)
 - getErrorStream method [609](#)
 - getInputStream method [609](#)
 - getOutputStream method [609](#)
 - getXxxStream methods [604, 605](#)
 - isAlive method [606, 609](#)
 - onExit method [609](#)
 - supportsNormalTermination method [609](#)
 - toHandle method [607, 609](#)
 - waitFor method [606, 609](#)
- process method
 - of SwingWorker [598, 600, 603](#)
- ProcessBuilder class [603, 608](#)
 - Constructor [608](#)
 - directory method [604, 608](#)
 - environment method [609](#)
 - inheritIO method [608](#)
 - redirectError method [608](#)
 - redirectErrorStream method [608](#)
 - redirectInput method [608](#)
 - redirectOutput method [608](#)
 - redirectXxx methods [604](#)
 - start method [605, 609](#)
 - startPipeline method [605, 609](#)
- Processes [603](#)
 - building [603](#)
 - killing [606](#)
 - running [605](#)
- ProcessHandle interface [609](#)
 - allProcesses method [607, 609](#)
 - children method [609](#)
 - children, descendants methods [607](#)
 - current method [607, 609](#)
 - descendants method [609](#)
 - info method [610](#)
 - of method [607, 609](#)
 - pid method [610](#)
- ProcessHandle.Info interface [610](#)
 - arguments method [610](#)
 - command method [610](#)
 - commandLine method [610](#)
 - startInstant method [610](#)
 - totalCpuDuration method [610](#)
 - user method [610](#)
- Processor interface [627](#)
- Producer threads [575](#)
- Programs. See Applications
- Prompt
 - readln method [66](#)
- Properties class [504, 507](#)
 - Constructor [507](#)
 - getProperty method [507](#)
 - load method [506, 507](#)
 - setProperty method [507](#)
 - store method [506, 508](#)
 - stringPropertyNames method [507](#)
- Property files

- generating [630](#)
- Property maps [505](#)
 - reading/writing [506](#)
- protected keyword [205](#), [277](#), [302](#), [669](#)
- provides keyword [663](#), [669](#)
- Proxies [340](#)
 - properties of [344](#)
 - purposes of [341](#)
- Proxy class [344](#), [345](#)
 - get/isProxyClass methods [345](#)
 - getProxyClass method [345](#)
 - isProxyClass method [345](#)
 - newProxyInstance method [341](#), [345](#)
- public keyword [122](#), [165](#), [283](#), [669](#)
 - checking [260](#)
 - for fields in interfaces [289](#)
 - for main method [31](#)
 - not specified for interfaces [282](#)
- publish method
 - of Handler [387](#)
 - of SwingWorker [598](#), [603](#)
- Pure virtual functions (C++) [235](#)
- push method
 - of Stack [510](#)
- put method
 - of BlockingQueue [575](#), [576](#), [579](#)
 - of ConcurrentHashMap [582](#)
 - of Map [449](#), [472](#), [473](#)
- putAll method
 - of Map [473](#)
- putFirst method
 - of BlockingDeque [580](#)
- putIfAbsent method
 - of ConcurrentHashMap [582](#)
 - of Map [476](#)
- putLast method
 - of BlockingDeque [580](#)

Q

- Qualified exports [660](#)
- Qualified name [158](#), [160](#), [339](#)
- Queue class [468](#)
 - implementing [440](#)
- Queue interface [469](#)
 - add method [469](#)
 - element method [469](#)
 - offer method [469](#)
 - peek method [469](#)
 - poll method [469](#)
 - remove method [469](#)
- Queues [440](#), [468](#)
 - blocking [575](#)
 - concurrent [580](#)
- QuickSort algorithm [97](#), [495](#)

R

- Race conditions [548](#), [551](#)
 - and atomic operations [570](#)
- Ragged arrays [102](#)
- Random class [153](#)
 - from method [153](#)
 - thread-safe [574](#)

- RandomAccess interface [450](#), [496](#)
- RandomGenerator interface [153](#)
 - getDefault method [153](#)
 - nextInt method [151](#), [153](#)
 - of method [153](#)
- range method
 - of EnumSet [482](#)
- Raw types [404](#)
 - converting type parameters to [410](#)
 - type inquiring at runtime [418](#)
- readLine method
 - of Console [69](#)
- readLn method
 - of IO [66](#), [68](#)
- readPassword method
 - of Console [69](#)
- Receiver parameter [616](#)
- record keyword [669](#)
- RecordComponent class [265](#)
 - getAccessor method [266](#)
 - getName method [265](#)
 - getType method [265](#)
- Records [153](#), [189](#)
 - adding methods to [154](#)
 - always final [200](#)
 - declared inside a class [337](#)
 - equals method of [208](#)
 - hashCode method of [214](#)
 - implementing interfaces [289](#)
 - instance fields of [154](#), [155](#)
 - toString method of [217](#)
- Rectangle class [466](#)
- Rectangles
 - comparing [466](#)
- Recursive computations [545](#)
- RecursiveAction, RecursiveTask classes [545](#)
- Red Hat [15](#)
- Red-black trees [465](#)
- redirectError method
 - of ProcessBuilder [608](#)
- redirectErrorStream method
 - of ProcessBuilder [608](#)
- redirectInput method
 - of ProcessBuilder [608](#)
- redirectOutput method
 - of ProcessBuilder [608](#)
- redirectXxx methods
 - of ProcessBuilder [604](#)
- reduce, reduceXxx methods
 - of ConcurrentHashMap [585](#), [586](#)
- Reentrant locks [553](#)
- ReentrantLock class [551](#), [554](#)
 - Constructor [554](#)
- Reflection [187](#), [254](#)
 - accessing
 - private members [651](#), [657](#)
 - analyzing
 - classes [260](#)
 - objects, at runtime [266](#)
 - generics and [270](#), [429](#)
 - overusing [278](#)
 - processing annotations with [623](#)
- Reinhold, Mark [10](#)
- Relational operators [48](#), [52](#)

Relative resource names [258](#)
remove method
 of ArrayList [226, 227](#)
 of BlockingQueue [575, 576](#)
 of Collection [446, 447](#)
 of Iterator [442, 444, 445, 448](#)
 of List [449, 460](#)
 of ListIterator [456](#)
 of Map [472](#)
 of Queue [469](#)
 of ThreadLocal [543](#)
removeAll method
 of Collection [446, 447](#)
 of LinkedList [458](#)
removeEldestEntry method [480](#)
 of LinkedHashMap [482](#)
removeFirst method
 of LinkedList [461](#)
 of SequencedCollection [469](#)
removeIf method
 of ArrayList [309](#)
 of Collection [447, 500](#)
removeLast method
 of LinkedList [461](#)
 of SequencedCollection [469](#)
repeat method
 of String [54, 59](#)
 of StringBuilder [64](#)
REPL [25](#)
replace method
 of ConcurrentHashMap [582](#)
 of String [59](#)
replaceAll method
 of Collections [500](#)
 of List [501](#)
 of Map [476](#)
requireNonNull method [126, 370](#)
 of Objects [138](#)
requireNonNullElse method [126](#)
 of Objects [138](#)
requireNonNullElseGet method
 of Objects [138](#)
requires keyword [645, 646, 648, 649, 654, 658](#)
Reserved words. See Keywords
Resources [258](#)
 exhaustion of [349](#)
 in JAR files [653](#)
 localizing [258](#)
 names of [258](#)
Restricted views [489](#)
resultNow method
 of Future [533](#)
resume method
 of Thread [523](#)
retain method
 of Collection [446](#)
retainAll method
 of Collection [447](#)
return keyword [669](#)
 in finally blocks [362](#)
 in lambda expressions [305](#)
 not allowed in switch expressions [86](#)
Return types [196](#)
 covariant [407](#)

 documentation comments for [177](#)
 for overridden methods [406](#)
Return values [114](#)
reverse method
 of Collections [500](#)
 of StringBuilder [64](#)
reversed method
 of Comparator [497](#)
 of SequencedCollection [493](#)
 of SequencedMap [493](#)
 of SequencedSet [493](#)
reversed, reverseOrder methods
 of Comparator [495](#)
reversed, reverseOrder methods
 of Comparator [322](#)
reverseOrder method
 of Comparator [497](#)
rotate method
 of Collections [500](#)
round method
 of Math [46](#)
RoundEnvironment interface [628](#)
rt.jar file
 no longer present [665](#)
run method
 of Runnable [520](#)
 of ScopedValue.Carrier [545](#)
 of Thread [517, 520](#)
runAfterXxx methods
 of CompletableFuture [593, 594](#)
Runnable interface [318, 515, 520](#)
 lambda expressions and [307](#)
 run method [317, 520](#)
Runtime class
 analyzing objects at [266](#)
 creating classes at [340](#)
 exec method [603](#)
 setting the size of an array at [221](#)
 type identification at [201, 255, 418](#)
Runtime image file [665](#)
RuntimeException class [350, 366, 369](#)
 Constructor [366](#)

S

S, s conversion characters [70](#)
Scala programming language [292](#)
Scanner class [68](#)
 Constructor [68](#)
 hasNext method [69](#)
 hasNextDouble method [69](#)
 hasNextInt method [69](#)
 next method [68](#)
 nextDouble method [69](#)
 nextInt method [69](#)
 nextLine method [68](#)
 nextXxx methods [67](#)
ScopedValue class [544](#)
 get method [544](#)
 isBound method [545](#)
 newInstance method [544](#)
 orElse method [544](#)
 where method [545](#)
ScopedValue.Carrier class [545](#)

- ScopedValue.Carrier class, methods of [545](#)
- sealed keyword [243](#), [289](#), [669](#)
- search, searchXxx methods
 - of ConcurrentHashMap [585](#), [586](#)
- Security class [4](#), [12](#)
- SequencedCollection interface [469](#), [493](#)
 - addFirst method [469](#)
 - addLast method [469](#)
 - getFirst method [469](#)
 - getLast method [469](#)
 - removeFirst method [469](#)
 - removeLast method [469](#)
 - reversed method [493](#)
- SequencedMap interface [493](#)
 - reversed method [493](#)
- SequencedSet interface [493](#)
 - reversed method [493](#)
- Serialization [481](#)
- Service loaders [338](#), [661](#)
- ServiceLoader class [338](#), [340](#), [661](#)
 - findFirst method [340](#)
 - iterator method [340](#)
 - load method [340](#)
 - stream method [339](#), [340](#)
- ServiceLoader.Provider interface [340](#)
 - get method [340](#)
 - type method [340](#)
- ServiceLoader.Provider interface, methods of [339](#)
- Services [338](#)
- ServletException [358](#)
- Servlets [358](#)
- Set interface [490](#)
 - add, equals, hashCode, methods of [450](#)
 - copyOf method [485](#), [490](#)
 - of method [483](#), [490](#)
- set method
 - of Array [273](#)
 - of ArrayList [224](#), [227](#)
 - of BitSet [511](#)
 - of Field [270](#)
 - of List [449](#), [460](#)
 - of ListIterator [456](#), [460](#)
 - of ThreadLocal [543](#)
 - of Vector [567](#)
- setAccessible method [267](#)
 - of AccessibleObject [270](#)
- setClassAssertionStatus method
 - of ClassLoader [375](#)
- setDaemon method [527](#)
 - of Thread [527](#)
- setDefaultAssertionStatus method
 - of ClassLoader [375](#)
- setDefaultUncaughtExceptionHandler method [390](#), [528](#)
 - of Thread [529](#)
- setFilter method
 - of Handler [387](#)
- setFormatter method
 - of Handler [387](#)
- setLabelTable method [408](#)
- setLevel method
 - of Handler [387](#)
- setName method
 - of Thread [528](#)
- setOut method [132](#)
- setPackageAssertionStatus method
 - of ClassLoader [375](#)
- setPriority method
 - of Thread [530](#)
- setProperty method [378](#)
 - of Properties [507](#)
- Sets [463](#)
 - concurrent [580](#)
 - intersecting [501](#)
 - mutating elements of [464](#)
 - subranges of [487](#)
 - thread-safe [587](#)
 - with given elements [483](#)
- setTime method [198](#)
- setUncaughtExceptionHandler method
 - of Thread [529](#)
- setValue method
 - of Map.Entry [478](#)
- setXxx method
 - of Array [273](#)
- Shallow copies [298](#), [301](#)
- Shell
 - scripts for, generating [630](#)
 - scripts in [168](#)
- Shift operators [51](#)
- Short class [215](#)
 - converting from short [228](#)
 - hashCode method [215](#)
- short type [33](#), [669](#)
- showMessageDialog method
 - of JOptionPane [296](#)
- shuffle method
 - of Collections [496](#), [497](#)
- Shuffling [496](#)
- shutdown method
 - of ExecutorService [536](#), [537](#)
- shutdownNow method [536](#)
 - of ExecutorService [537](#)
- Sieve of Eratosthenes benchmark [511](#), [513](#)
- signal method
 - of Condition [556](#), [558](#), [560](#)
- signalAll method [555](#), [560](#)
 - of Condition [558](#)
- Signatures (of methods) [144](#), [196](#)
- Signatures. See Digital signatures
- Simple name [160](#)
 - of enumeration [238](#)
- sin method
 - of Math [44](#)
- singleton method
 - of Collections [492](#)
- singletonList method
 - of Collections [492](#)
- Size
 - of concurrent collections [580](#)
- size method
 - of ArrayList [223](#), [224](#)
 - of BitSet [511](#)
 - of Collection [446](#), [447](#)
- sleep method
 - of Thread [516](#), [520](#), [525](#)
- SLF4J [375](#)
- Smart cards [3](#)
- SOAP [642](#)

- SocketHandler class [380](#)
- sort method
 - of Arrays [97](#), [99](#), [283](#), [285](#), [287](#), [304](#), [308](#)
 - of Collections [495](#), [497](#)
 - of List [497](#)
- SortedMap interface [450](#), [474](#), [492](#)
 - comparator method [474](#)
 - firstKey method [474](#)
 - headMap method [492](#)
 - headMap, subMap, tailMap methods [487](#)
 - lastKey method [474](#)
 - subMap method [492](#)
 - tailMap method [492](#)
- SortedSet interface [450](#), [468](#), [492](#)
 - comparator method [468](#)
 - first method [468](#)
 - headSet method [492](#)
 - headSet, subSet, tailSet methods [487](#)
 - last method [468](#)
 - subSet method [492](#)
 - tailSet method [492](#)
- Sorting
 - algorithms for [97](#), [495](#)
 - arrays [97](#), [285](#)
 - assertions for [373](#)
 - order of [495](#)
 - people, by name [321](#)
 - strings by length [297](#), [304](#), [306](#)
- Source code, generating [620](#), [621](#), [628](#)
- Source file
 - compact [161](#)
- Source files [168](#)
 - installing [18](#)
 - running in Eclipse [25](#)
- Space. See Whitespace
- Special characters [36](#)
- Split packages [650](#)
- sqrt method
 - of BigInteger [91](#)
 - of Math [43](#), [274](#), [275](#)
- src.zip file [18](#)
- Stack class [439](#), [504](#), [510](#)
 - peek method [510](#)
 - pop method [510](#)
 - push method [510](#)
- Stack trace [364](#), [559](#)
 - no displaying to users [370](#)
- StackFrame
 - toString method [364](#)
- Stacks [510](#)
- stackSize method
 - of Thread.Builder.OfPlatform [531](#)
- StackTraceElement class [367](#)
 - getClassName method [367](#)
 - getFileName method [367](#)
 - getLineNumber method [367](#)
 - getMethodName method [367](#)
 - isNativeMethod method [367](#)
 - toString method [367](#)
- StackWalker class [364](#), [366](#)
 - forEach method [366](#)
 - getInstance method [364](#), [366](#)
 - walk method [364](#), [366](#)
- StackWalker.StackFrame interface [366](#)
- getClassName method [366](#)
- getDeclaringClass method [367](#)
- getFileName method [366](#)
- getLineNumber method [366](#)
- getMethodName method [367](#)
- isNativeMethod method [367](#)
- toString method [367](#)
- Standard Edition [9](#), [16](#)
- Standard Java library
 - companion classes in [291](#)
 - online API documentation for [58](#), [59](#), [175](#), [182](#)
- Standard Template Library (STL) [439](#), [443](#)
- start method
 - of ProcessBuilder [605](#), [609](#)
 - of Thread [517](#), [520](#), [521](#)
 - of Thread.Builder [531](#)
 - of Timer [296](#)
- startInstant method
 - of ProcessHandle.Info [610](#)
- startPipeline method [605](#)
 - of ProcessBuilder [609](#)
- startsWith method
 - of String [58](#)
- startVirtualThread method
 - of Thread [524](#)
- state method
 - of Future [533](#)
- Statements [31](#)
 - conditional [73](#)
 - in output [53](#)
- Static binding [196](#)
- Static constants [132](#)
 - documentation comments for [178](#)
- Static fields [131](#)
 - accessing, in static methods [133](#)
 - importing [161](#)
 - initializing [149](#)
 - no type variables in [424](#)
- Static imports [161](#)
- static keyword [131](#), [659](#), [669](#)
 - for fields in interfaces [289](#)
- Static method [54](#)
- Static methods [133](#)
 - accessing static fields in [133](#)
 - adding to interfaces [291](#)
 - importing [161](#)
 - no type variables in [424](#)
- Static nested classes [323](#), [334](#)
- Static variables [132](#)
- stop method
 - of Thread [523](#), [524](#)
 - of Timer [296](#)
- store method
 - of Properties [506](#), [508](#)
- Stream interface, toArray method [314](#)
- stream method
 - of BitSet [511](#)
 - of Collection [292](#)
 - of ServiceLoader [339](#), [340](#)
- strictfp keyword [669](#)
- StrictMath class [44](#)
- String
 - formatted method [232](#)
- String class [52](#), [58](#)

- charAt method [55, 58](#)
- compareTo method [58](#)
- endsWith method [58](#)
- equals method [58](#)
- equals, equalsIgnoreCase methods [57](#)
- equalsIgnoreCase method [58](#)
- hashCode method [211, 461](#)
- immutability of [56, 130, 198](#)
- implementing CharSequence [290](#)
- indexOf method [59, 144](#)
- isEmpty method [58](#)
- join method [59](#)
- lastIndexOf method [59](#)
- length method [55, 57, 58](#)
- repeat method [54, 59](#)
- replace method [59](#)
- startsWith method [58](#)
- strip method [59](#)
- substring method [56, 59, 487](#)
- toLowerCase method [59](#)
- toUpperCase method [59](#)
- transform method [320](#)
- StringBuffer class [63](#)
- StringBuilder class [61, 63](#)
 - append method [61, 64](#)
 - appendCodePoint method [64](#)
 - Constructor [63](#)
 - delete method [64](#)
 - implementing CharSequence [290](#)
 - insert method [64](#)
 - length method [64](#)
 - repeat method [64](#)
 - reverse method [64](#)
 - toString method [62, 64](#)
- stringPropertyNames method of Properties [507](#)
- Strings [52](#)
 - building [61](#)
 - code points/code units of [55](#)
 - comparing [297](#)
 - concatenating [53](#)
 - with objects [216](#)
 - converting to numbers [230](#)
 - empty [57](#)
 - equality of [57](#)
 - formatting output for [69](#)
 - immutability of [56](#)
 - length of [56, 57](#)
 - null [57](#)
 - shared, in compiler [56, 57](#)
 - sorting by length [297, 304, 306](#)
 - spanning multiple lines [64](#)
 - substrings of [56](#)
 - using ". . ." for [31](#)
- strip method of String [59](#)
- Strongly typed languages [33, 285](#)
- Subclasses [187](#)
 - adding fields/methods to [190](#)
 - anonymous [332](#)
 - cloning [302](#)
 - comparing objects from [288](#)
 - constructors for [190](#)
 - defining [188](#)
 - forbidding [243](#)
 - inheriting annotations [620](#)
 - method visibility in [198](#)
 - no access to private fields of superclass [205](#)
 - non-sealed [245](#)
 - overriding superclass methods in [190](#)
- subList method of List [487, 492](#)
- subMap method of NavigableMap [493](#)
 - of SortedMap [487, 492](#)
- submit method of ExecutorCompletionService [542](#)
 - of ExecutorService [535, 537](#)
- Subranges [487](#)
- subSet method of NavigableSet [487, 492](#)
 - of SortedSet [487, 492](#)
- Substitution principle [194](#)
- substring method of String [56, 59, 487](#)
- subtract method of BigDecimal [92](#)
 - of BigInteger [91](#)
- subtractExact method [45](#)
- Subtraction [42](#)
- sum method of LongAdder [571](#)
- Sun Microsystems [1, 4, 9, 12](#)
 - HotJava browser [9](#)
- super keyword [190, 412, 413, 669](#)
 - in method references [313](#)
 - vs. this [190, 191](#)
- Superclass wins rule [293](#)
- Superclasses [187](#)
 - accessing private fields of [190](#)
 - annotating [615](#)
 - common fields and methods in [235, 277](#)
 - overriding methods of [211](#)
 - throws specifiers in [352, 356](#)
- Supertype bounds [412](#)
- Supplier interface [318](#)
- supportsNormalTermination method of Process [609](#)
- Surrogates area (Unicode) [37](#)
- suspend method of Thread [523](#)
- swap method of Collections [500](#)
- Swing [598](#)
- SwingWorker class [598, 603](#)
 - doInBackground method [598, 599, 603](#)
 - execute method [598, 603](#)
 - getState method [603](#)
 - process method [603](#)
 - process, publish methods [598, 600](#)
 - publish method [603](#)
- switch keyword [49, 83, 669](#)
 - enumerated constants in [50](#)
 - throwing exceptions in [86](#)
 - value of [49](#)
 - with fallthrough [85](#)
 - with pattern matching [244](#)
- Synchronization [547](#)

- condition objects for [554](#)
- final fields and [569](#)
- in Vector [461](#)
- lock objects for [551](#)
- monitor concept for [567](#)
- race conditions in [548](#), [551](#), [570](#)
- volatile fields and [568](#)
- Synchronization wrappers [588](#)
- Synchronized blocks [565](#)
- synchronized keyword [551](#), [561](#), [568](#), [669](#)
- Synchronized views [489](#)
- synchronizedCollection method
 - of Collections [491](#), [589](#)
- synchronizedCollection methods (Collections) [489](#)
- synchronizedList method
 - of Collections [491](#), [589](#)
- synchronizedMap method
 - of Collections [491](#), [589](#)
- synchronizedNavigableMap method
 - of Collections [491](#)
- synchronizedNavigableSet method
 - of Collections [491](#)
- synchronizedSet method
 - of Collections [491](#), [589](#)
- synchronizedSortedMap method
 - of Collections [491](#), [589](#)
- synchronizedSortedSet method
 - of Collections [491](#), [589](#)
- System class [69](#), [221](#), [483](#), [508](#)
 - console method [69](#)
 - getLogger method [376](#), [378](#)
 - getProperties method [508](#)
 - getProperty method [508](#), [509](#)
 - identityHashCode method [221](#), [481](#), [483](#)
 - setOut method [132](#)
 - setProperty method [378](#)
- System.err [379](#), [390](#)
- System.in [66](#)
- System.Logger interface [376](#), [386](#)
 - getName method [386](#)
 - isLoggable method [386](#)
 - log method [386](#)
- System.Logger.Level enumeration [377](#)
- System.Logger.log method
 - log method of System.Logger [376](#)
- System.out [132](#)

T

- T, t conversion characters [70](#)
- Tab completion [27](#)
- Tabs, in text blocks [65](#)
- Tagging interfaces [300](#), [450](#)
- tailMap method
 - of NavigableMap [493](#)
 - of SortedMap [487](#), [492](#)
- tailSet method
 - of NavigableSet [487](#), [492](#)
 - of SortedSet [487](#), [492](#)
- take method
 - of BlockingQueue [575](#), [576](#), [579](#)
 - of ExecutorCompletionService [542](#)
- takeFirst method
 - of BlockingDeque [580](#)

- takeLast method
 - of BlockingDeque [580](#)
- tan method
 - of Math [44](#)
- tar command [170](#)
- Tasks
 - asynchronously running [531](#)
 - controlling groups of [537](#)
 - decoupling from mechanism of running [517](#)
 - long-running [597](#)
 - multiple [515](#)
 - work stealing for [546](#)
- TAU
 - of Math [44](#)
- Terminal window [20](#)
- Text blocks [64](#)
- thenAccept, thenAcceptBoth, thenCombine methods
 - of CompletableFuture [593](#)
- thenApply, thenApplyAsync methods
 - of CompletableFuture [592](#), [593](#)
- thenComparing method [321](#)
- thenCompose method [593](#)
- thenRun method [593](#)
- this keyword [127](#), [147](#), [669](#)
 - annotating [616](#)
 - in body of constructor [147](#)
 - in inner classes [326](#)
 - in lambda expressions [316](#)
 - in method references [313](#)
 - vs. super [190](#), [191](#)
- Thread class [520](#), [521](#), [522](#), [524](#), [527](#), [528](#), [529](#), [530](#), [531](#)
 - Constructor [520](#)
 - currentThread method [524](#), [527](#)
 - extending [517](#)
 - getDefaultUncaughtExceptionHandler method [529](#)
 - getName method [528](#)
 - getState method [523](#)
 - getUncaughtExceptionHandler method [529](#)
 - interrupt method [527](#)
 - interrupt, isInterrupted methods [524](#)
 - interrupted method [526](#), [527](#)
 - isInterrupted method [527](#)
 - isVirtual method [524](#)
 - join method [521](#), [522](#), [523](#)
 - MAX_PRIORITY field [530](#)
 - methods with timeout [521](#)
 - MIN_PRIORITY field [530](#)
 - NORM_PRIORITY field [530](#)
 - ofPlatform method [531](#)
 - ofVirtual method [531](#)
 - resume method [523](#)
 - run method [517](#), [520](#)
 - setDaemon method [527](#)
 - setDefaultUncaughtExceptionHandler method [390](#), [528](#), [529](#)
 - setName method [528](#)
 - setPriority method [530](#)
 - setUncaughtExceptionHandler method [529](#)
 - sleep method [516](#), [520](#), [525](#)
 - start method [517](#), [520](#), [521](#)
 - startVirtualThread method [524](#)
 - stop method [523](#), [524](#)
 - suspend method [523](#)
 - threadId method [528](#)

- yield method [521](#)
- Thread dump [560](#)
- Thread groups [529](#)
- Thread pools [534](#)
- Thread-safe collections [574](#)
 - callable and futures [531](#)
 - concurrent [580](#)
 - copy on write arrays [587](#)
 - synchronization wrappers [588](#)
- Thread.Builder interface [531](#)
 - factory method [531](#)
 - name method [531](#)
 - start method [531](#)
 - uncaughtExceptionHandler method [531](#)
 - unstarted method [531](#)
- Thread.Builder.OfPlatform interface [531](#)
 - daemon method [531](#)
 - group method [531](#)
 - priority method [531](#)
 - stackSize method [531](#)
- Thread.UncaughtExceptionHandler interface [528, 529](#)
 - uncaughtException method [529](#)
- ThreadFactory interface [530](#)
 - newThread method [530](#)
- ThreadGroup class [529](#)
 - uncaughtException method [529](#)
- threadId method
 - of Thread [528](#)
- ThreadLocal class [543, 574](#)
 - get method [543](#)
 - remove method [543](#)
 - set method [543](#)
 - withInitial method [574](#)
- ThreadLocalRandom class [574](#)
 - current method [574](#)
- ThreadPoolExecutor class [534](#)
- Threads
 - accessing collections from [489, 574](#)
 - blocked [521, 525](#)
 - condition objects for [554](#)
 - daemon [527](#)
 - executing code in [317](#)
 - idle [545](#)
 - interrupting [524](#)
 - listing all [560](#)
 - locking [565](#)
 - new [521](#)
 - priorities of [529](#)
 - producer/customer [575](#)
 - runnable [521](#)
 - states of [520](#)
 - synchronizing [547](#)
 - terminated [516, 522, 524](#)
 - thread-local variables in [573](#)
 - timed waiting [521](#)
 - unblocking [556](#)
 - uncaught exceptions in [528](#)
 - waiting [521, 555](#)
 - work stealing for [546](#)
 - worker [597](#)
- throw keyword [353, 669](#)
- Throwable class [257, 349, 354, 365, 369](#)
 - add/getSuppressed methods [363](#)
 - addSuppressed method [366](#)
 - Constructor [354, 355, 365](#)
 - getCause method [365](#)
 - getMessage method [355](#)
 - getStackTrace method [364, 366](#)
 - getSuppressed method [366](#)
 - initCause method [365](#)
 - printStackTrace method [257, 364, 390](#)
 - toString method [354](#)
- throws keyword [258, 351, 670](#)
- Time measurement vs. calendars [115](#)
- Timed waiting threads [521](#)
- TimeoutException class [532, 593](#)
- Timer class [294, 296, 304](#)
 - Constructor [296](#)
 - start method [296](#)
 - stop method [296](#)
- to keyword [670](#)
- toArray method
 - of ArrayList [424](#)
 - of Collection [225, 446, 447, 502](#)
 - of Stream [314](#)
- toHandle method [607](#)
 - of Process [609](#)
- toLowerCase method
 - of String [59](#)
- Toolkit class [296](#)
 - beep method [296](#)
 - getDefaultToolkit method [296](#)
- toString method
 - adding to all classes [217](#)
 - Formattable and [70](#)
 - of Annotation [619](#)
 - of Arrays [95, 99](#)
 - of Date [113](#)
 - of Enum [239](#)
 - of Integer [231](#)
 - of Modifier [261, 266](#)
 - of Object [215, 220, 294](#)
 - of proxy classes [344](#)
 - of records [154, 217](#)
 - of StackFrame [364](#)
 - of StackTraceElement [367](#)
 - of StackWalker.StackFrame [367](#)
 - of StringBuilder [62, 64](#)
 - of Throwable [354](#)
 - redeclaring [307](#)
 - working with any class [267, 268](#)
- Total ordering [466](#)
- totalCpuDuration method
 - of ProcessHandle.Info [610](#)
- toUnsignedInt method [34](#)
- toUpperCase method
 - of String [59](#)
- TraceHandler [341](#)
- transfer method
 - of TransferQueue [580](#)
- TransferQueue interface [576, 580](#)
 - transfer method [580](#)
 - tryTransfer method [580](#)
- Transform [320](#)
- transform method
 - of String [320](#)
- transient keyword [670](#)
- transitive keyword [658, 659, 670](#)

- Tree maps [471](#)
- Tree sets [465](#)
 - red-black [465](#)
 - total ordering of [466](#)
 - vs. priority queues [470](#)
- TreeMap class [450, 471, 474](#)
 - as a concrete collection type [451](#)
 - Constructor [474](#)
 - vs. HashMap [471](#)
- TreeSet class [450, 465, 467](#)
 - as a concrete collection type [451](#)
 - Constructor [467](#)
- Trigonometric functions [44](#)
- trimToSize method [223](#)
 - of ArrayList [224](#)
- Troubleshooting. See Debugging
- true literal [670](#)
- Truncated computations [43](#)
- try keyword [670](#)
- try-with-resources statement [362](#)
 - effectively final variables in [363](#)
 - no locks with [552](#)
- try/catch [355, 360](#)
 - generics and [403](#)
 - wrapping entire task in try block [368](#)
- try/finally [360](#)
- tryLock method [521](#)
- trySetAccessible method
 - of AccessibleObject [270](#)
- tryTransfer method
 - of TransferQueue [580](#)
- Two-dimensional arrays [99, 103](#)
- Type bounds
 - annotating [615](#)
- Type erasure [404, 418](#)
 - clashes after [426](#)
- Type interface [431, 432](#)
- type method
 - of ServiceLoader.Provider [339, 340](#)
- Type parameters [221](#)
 - annotating [613](#)
 - converting to raw types [410](#)
 - not for arrays [409, 419](#)
 - not instantiated with primitive types [418](#)
 - vs. inheritance [395](#)
- Type variables
 - bounds for [401](#)
 - common names of [398](#)
 - in exceptions [403](#)
 - in static fields or methods [424](#)
 - matching in generic methods [430](#)
 - no instantiating for [422](#)
 - replacing with bound types [404](#)
- TypeElement interface [628](#)
- Types. See Data types
- TypeVariable interface [431, 432, 437](#)
 - getBounds method [437](#)
 - getName method [437](#)
- uncaughtException method
 - of Thread.UncaughtExceptionHandler [529](#)
 - of ThreadGroup [529](#)
- uncaughtExceptionHandler method
 - of Thread.Builder [531](#)
- Unchecked exceptions [258, 350, 352](#)
 - applicability of [369](#)
- Unequality operator [48](#)
- Unicode [5, 36, 38, 52](#)
- Unit tests [611](#)
- University of Illinois [8](#)
- UNIX [167, 168](#)
- unlock method
 - of Lock [552, 554](#)
- Unmodifiable copies [485](#)
- Unmodifiable views [485](#)
- unmodifiableCollection method
 - of Collections [491](#)
- unmodifiableCollection methods (Collections) [485, 486](#)
- unmodifiableList method
 - of Collections [491](#)
- unmodifiableMap method
 - of Collections [491](#)
- unmodifiableNavigableMap method
 - of Collections [491](#)
- unmodifiableNavigableSet method
 - of Collections [491](#)
- unmodifiableSequencedCollection method
 - of Collections [491](#)
- unmodifiableSequencedMap method
 - of Collections [491](#)
- unmodifiableSequencedSet method
 - of Collections [491](#)
- unmodifiableSet method
 - of Collections [491](#)
- unmodifiableSortedMap method
 - of Collections [491](#)
- unmodifiableSortedSet method
 - of Collections [491](#)
- Unnamed modules [267](#)
- Unnamed packages [162, 165, 182, 372](#)
- unstarted method
 - of Thread.Builder [531](#)
- UnsupportedOperationException class [477, 484, 486, 489](#)
- updateAndGet method
 - of AtomicXxx [571](#)
- User input [348](#)
- User Interface. See Graphical User Interface
- user method
 - of ProcessHandle.Info [610](#)
- User-defined types [230](#)
- uses keyword [662, 663, 670](#)
- “Uses-a” relationship [110](#)
- UTC (Coordinated Universal Time) [115](#)
- Utility classes/methods [291, 292](#)

V

- V> method
 - of ConcurrentHashMap [581](#)
 - of ConcurrentSkipListMap [582](#)
- valueOf method
 - of BigInteger [89, 91](#)
 - of Enum [239, 242](#)

U

- UCSD Pascal system [4](#)
- UML (Unified Modeling Language) notation [111](#)
- UnaryOperator interface [318](#)

- of Integer [232](#)
- values method
 - of Map [476](#), [478](#)
- var keyword [124](#), [306](#), [332](#), [670](#)
 - diamond syntax and [222](#)
- Varargs methods [232](#)
 - passing generic types to [420](#)
- Varargs parameters
 - safety of [620](#), [621](#)
- VarHandle class [267](#), [653](#)
- Variable handles [267](#), [653](#)
- VariableElement interface [628](#)
- Variables [39](#)
 - accessing
 - from outer methods [329](#)
 - in lambda expressions [314](#)
 - annotating [408](#)
 - copying [298](#)
 - declarations of [39](#), [202](#)
 - deprecated [620](#)
 - effectively final [316](#), [363](#)
 - initializing [40](#), [183](#)
 - local [124](#), [204](#), [408](#)
 - mutating in lambda expressions [315](#)
 - names of [39](#)
 - package scope of [165](#)
 - printing/logging values of [389](#)
 - static [132](#)
 - thread-local [573](#)
- Vector class [439](#), [504](#), [566](#), [567](#), [588](#)
 - for dynamic arrays [222](#)
 - get, set methods [567](#)
 - synchronization in [461](#)
- Views [483](#)
 - bulk operations for [501](#)
 - checked [488](#)
 - restricted [489](#)
 - subranges of [487](#)
 - synchronized [489](#)
 - unmodifiable [485](#)
- Visual Basic
 - built-in date type in [112](#)
 - syntax of [2](#)
- Visual Studio [18](#)
- void keyword [670](#)
- Volatile fields [568](#)
- volatile keyword [569](#), [570](#), [670](#)
- Von der Ahé, Peter [400](#)

W

- wait method
 - of Object [521](#), [562](#), [565](#)
- Wait sets [555](#)
- waitFor method [606](#)
 - of Process [609](#)
- walk method
 - of StackWalker [364](#), [366](#)
- Warning messages [620](#)
- Warning messages, suppressing [621](#)
- Warnings
 - fallthrough behavior and [85](#)
 - generic [228](#), [408](#), [420](#), [425](#)
 - suppressing [420](#), [425](#)

- Weak hash maps [478](#)
- Weak references [478](#)
- WeakHashMap class [478](#), [481](#)
 - as a concrete collection type [451](#)
 - Constructor [481](#)
- Weakly consistent iterators [580](#)
- WeakReference class [478](#)
- Web pages
 - dynamic [7](#)
 - extracting links from [592](#)
 - reading [597](#)
- whenComplete method
 - of CompletableFuture [593](#)
- where method
 - of ScopedValue [545](#)
- while keyword [75](#), [670](#)
- Whitespace
 - escape sequence for [36](#), [65](#)
 - in text blocks [65](#)
 - irrelevant to compiler [30](#)
 - leading/trailing [65](#)
- Wildcard types [397](#), [410](#)
 - annotating [615](#)
 - arrays of [419](#)
 - capturing [415](#)
 - supertype bounds for [412](#)
 - unbounded [415](#)
- WildcardType interface [431](#), [432](#), [438](#)
 - getLowerBounds method [438](#)
 - getUpperBounds method [438](#)
- Windows
 - changing warning string in [166](#)
- Windows operating system
 - executing JARs in [172](#)
 - IDEs for [23](#)
 - JDK in [15](#)
 - paths in [167](#), [169](#)
 - thread priority levels in [529](#)
- Wirth, Niklaus [4](#), [8](#), [107](#)
- with keyword [670](#)
- withInitial method
 - of ThreadLocal [574](#)
- Work stealing [546](#)
- Worker threads [597](#)
- Working directory, for a process [604](#)
- Wrappers [228](#)
 - class constructors for [230](#)
 - equality testing for [229](#)
 - immutability of [228](#)
 - locks and [230](#), [566](#)

X

- X, x conversion characters [70](#)
- XML [9](#), [11](#)
- XML descriptors, generating [630](#)
- XML/JSON binding [653](#)
- xor method
 - of BitSet [511](#)

Y

- Yasson [653](#)
- yield keyword [86](#), [670](#)

yield method
of Thread [521](#)

for JMOD files [665](#)
ZIP format [166](#), [169](#)

Z

ZIP archives