

Updated through Java 21



Core Java for the Impatient

Fourth Edition

Cay S. Horstmann



Updated through Java 21



Core Java for the Impatient

Fourth Edition

Cay S. Horstmann



About This eBook

ePUB is an open, industry-standard format for eBooks. However, support of ePUB and its many features varies across reading devices and applications. Use your device or app settings to customize the presentation to your liking. Settings that you can customize often include font, font size, single or double column, landscape or portrait mode, and figures that you can click or tap to enlarge. For additional information about the settings and features on your reading device or app, visit the device manufacturer's Web site.

Many titles include programming code or configuration examples. To optimize the presentation of these elements, view the eBook in single-column, landscape mode and adjust the font size to the smallest setting.

Core Java for the Impatient

Fourth Edition

Cay S. Horstmann

 Addison-Wesley

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

A NOTE FOR EARLY RELEASE READERS

With Early Release eBooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this title, please reach out to Pearson at PearsonITAcademics@pearson.com

Cover illustration by Morphart Creation / Shutterstock

Figures 1.1 and 1.3: Microsoft

Figure 1.2: Eclipse Foundation

Figures 1.4, 1.5, 11.1: Oracle Corporation

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.

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

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

Visit us on the Web: informit.com/aw

Copyright © 2025 Pearson Education, Inc.

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

ISBN-13: 978-0-13-540454-6

To Chi—the most patient person in my life.

Table of Contents

About This eBook

Preface

Acknowledgments

1. Fundamental Programming Structures

1.1. Our First Program

1.2. Primitive Types

1.3. Variables

1.4. Arithmetic Operations

1.5. Strings

1.6. Input and Output

1.7. Control Flow

1.8. Arrays and Array Lists

1.9. Functional Decomposition

Exercises

2. Object-Oriented Programming

2.1. Working with Objects

2.2. Implementing Classes

2.3. Object Construction

2.4. Records

2.5. Static Variables and Methods

2.6. Packages

2.7. Nested Classes

2.8. Documentation Comments

Exercises

3. Interfaces and Lambda Expressions

3.1. Interfaces

3.2. Static, Default, and Private Methods

3.3. Examples of Interfaces

3.4. Lambda Expressions

3.5. Method and Constructor References

3.6. Processing Lambda Expressions

3.7. Lambda Expressions and Variable Scope

- 3.8. Higher-Order Functions
- 3.9. Local and Anonymous Classes
- Exercises
- 4. Inheritance and Reflection
 - 4.1. Extending a Class
 - 4.2. Inheritance Hierarchies
 - 4.3. Object: The Cosmic Superclass
 - 4.4. Enumerations
 - 4.5. Pattern Matching
 - 4.6. Runtime Type Information and Resources
 - 4.7. Reflection
- Exercises
- 5. Exceptions, Assertions, and Logging
 - 5.1. Exception Handling
 - 5.2. Assertions
 - 5.3. Logging
- Exercises
- 6. Generic Programming
 - 6.1. Generic Classes
 - 6.2. Generic Methods
 - 6.3. Type Bounds
 - 6.4. Type Variance and Wildcards
 - 6.5. Generics in the Java Virtual Machine
 - 6.6. Restrictions on Generics
 - 6.7. Reflection and Generics
- Exercises
- 7. Collections
 - 7.1. An Overview of the Collections Framework
 - 7.2. Iterators
 - 7.3. Sets
 - 7.4. Maps
 - 7.5. Other Collections
 - 7.6. Views
- Exercises
- 8. Streams
 - 8.1. From Iterating to Stream Operations

- 8.2. Stream Creation
 - 8.3. The `filter`, `map`, and `flatMap` Methods
 - 8.4. Extracting Substreams and Combining Streams
 - 8.5. Other Stream Transformations
 - 8.6. Simple Reductions
 - 8.7. The Optional Type
 - 8.8. Collecting Results
 - 8.9. Collecting into Maps
 - 8.10. Grouping and Partitioning
 - 8.11. Downstream Collectors
 - 8.12. Reduction Operations
 - 8.13. Primitive Type Streams
 - 8.14. Parallel Streams
 - Exercises
- 9. Processing Input and Output
 - 9.1. Input/Output Streams, Readers, and Writers
 - 9.2. Paths, Files, and Directories
 - 9.3. HTTP Connections
 - 9.4. Regular Expressions
 - 9.5. Serialization
 - Exercises
- 10. Concurrent Programming
 - 10.1. Concurrent Tasks
 - 10.2. Thread Safety
 - 10.3. Threadsafe Data Structures
 - 10.4. Parallel Algorithms
 - 10.5. Asynchronous Computations
 - 10.6. Atomic Counters and Accumulators
 - 10.7. Locks and Conditions
 - 10.8. Threads
 - 10.9. Processes
 - Exercises
- 11. Annotations
 - 11.1. Using Annotations
 - 11.2. Defining Annotations
 - 11.3. Annotations in the Java API

- 11.4. Processing Annotations at Runtime
- 11.5. Source-Level Annotation Processing
- Exercises
- 12. The Java Platform Module System
 - 12.1. The Module Concept
 - 12.2. Naming Modules
 - 12.3. The Modular “Hello, World!” Program
 - 12.4. Requiring Modules
 - 12.5. Exporting Packages
 - 12.6. Modules and Reflective Access
 - 12.7. Modular JARs
 - 12.8. Automatic Modules
 - 12.9. The Unnamed Module
 - 12.10. Command-Line Flags for Migration
 - 12.11. Transitive and Static Requirements
 - 12.12. Qualified Exporting and Opening
 - 12.13. Service Loading
 - 12.14. Tools for Working with Modules
- Exercises

Index

Preface

Java has seen many changes since its initial release in 1996. The classic book, *Core Java*, covers, in meticulous detail, not just the language but all core libraries and a multitude of changes between versions, spanning two volumes and over 2,000 pages. However, if you just want to be productive with modern Java, there is a much faster, easier pathway for learning the language and core libraries. In this book, I don't retrace history and don't dwell on features of past versions. I show you the good parts of Java as it exists today, so you can put your knowledge to work quickly.

As with my previous "Impatient" books, I quickly cut to the chase, showing you what you need to know to solve a programming problem without lecturing about the superiority of one paradigm over another. I also present the information in small chunks, organized so that you can quickly retrieve it when needed.

Assuming you are proficient in some other programming language, such as Python, C++, JavaScript, Swift, PHP, or Ruby, with this book you will learn how to become a competent Java programmer. I cover all aspects of Java that a developer needs to know today, including the powerful concepts of lambda expressions and streams, as well as modern constructs such as records and pattern matching.

This book is fully updated to Java 21. It uses modern features and does not dwell on historical or obsolete constructs. Preview features that may make it to the language in the future are not covered either.

A key reason to use Java is to tackle concurrent programming. With parallel algorithms and threadsafe data structures readily available in the Java library, the way application programmers should handle concurrent programming has completely changed. I provide fresh coverage, showing you how to use the powerful library features instead of error-prone low-level constructs.

Traditionally, books on Java have focused on user interface programming, but nowadays, few developers produce user interfaces on desktop computers. You will be able to use this book effectively without being distracted by lengthy GUI code.

Finally, this book is written for application programmers, not for a college course and not for systems wizards. The book covers issues that application programmers need to wrestle with, such as logging and working with files, but you won't learn how to implement a linked list by hand or how to write a web server.

I hope you enjoy this rapid-fire introduction into modern Java, and I hope it will make your work with Java productive and enjoyable.

If you find errors or have suggestions for improvement, please visit <http://horstmann.com/javaimpatient>, head for the errata page, and leave a comment. Be sure to visit that site to download the runnable code examples that complement this book.

Acknowledgments

My thanks go, as always, to my editor Greg Doench, who enthusiastically supported the vision of a short book that gives a fresh introduction to Java. My special gratitude goes to the excellent team of reviewers for this and previous editions who spotted many errors and gave thoughtful suggestions for improvement. They are: Andres Almiray, Gail Anderson, Paul Anderson, Marcus Biel, Brian Goetz, Mark Lawrence, Doug Lea, Simon Ritter, Yoshiki Shibata, and Christian Ullenboom.

I wrote the book using HTML and CSS, and Prince (<https://princexml.com>) turned it into PDF—a workflow that I highly recommend.

*Cay Horstmann
Düsseldorf, Germany
August 2024*

Chapter 1 ■ Fundamental Programming Structures

In this chapter, you will learn about the basic data types and control structures of the Java language. I assume that you are an experienced programmer in some other language and that you are familiar with concepts such as variables, loops, function calls, and arrays, but perhaps with a different syntax. This chapter will get you up to speed on the Java way. I will also give you some tips on the most useful parts of the Java API for manipulating common data types.

The key points of this chapter are:

1. In Java, all methods are declared in a class. You invoke a non-static method on an object of the class to which the method belongs.
2. Static methods are not invoked on objects. Program execution starts with the static `main` method.
3. Java has eight primitive types: four signed integral types, two floating-point types, `char`, and `boolean`.
4. The Java operators and control structures are very similar to those of C or JavaScript.
5. There are four forms of `switch`: expressions and statements with and without fall-through.
6. The `Math` class provides common mathematical functions.
7. `String` objects are sequences of characters or, more precisely, Unicode code points in the UTF-16 encoding.
8. Use the “text box” syntax to declare multiline string literals.
9. With the `System.out` object, you can display output in a terminal window. A `Scanner` tied to `System.in` lets you read terminal input.
10. Arrays and collections can be used to collect elements of the same type.

1.1. Our First Program

When learning any new programming language, it is traditional to start with a program that displays the message “Hello, World!”. That is what we will do in the following sections.

1.1.1. Dissecting the “Hello, World” Program

Without further ado, here is the “Hello, World” program in Java:

```
package ch01.sec01;

// Our first Java program

public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Let’s examine this program:

- Java is an object-oriented language. In your program, you manipulate (mostly) *objects* by having them do work. Each object that you manipulate belongs to a specific *class*, and we say that the object is an *instance* of that class. A class defines what an object’s state can be and what it can do. In Java, all code is defined inside classes. We will look at objects and classes in detail in

[Chapter 2](#). This program is made up of a single class `HelloWorld`.

- `main` is a *method*, that is, a function declared inside a class. The `main` method is the first method that is called when the program runs. It is declared as `static` to indicate that the method does not operate on any objects. (When `main` gets called, there are only a handful of predefined objects, and none of them are instances of the `HelloWorld` class.) The method is declared as `void` to indicate that it does not return any value. See [Section 1.8.8](#) for the meaning of the parameter declaration `String[] args`.
- In Java, you can declare many features as `public` or `private`, and there are a couple of other visibility levels as well. Here, we declare the `HelloWorld` class and the `main` method as `public`, which is the most common arrangement for classes and methods.
- A *package* is a set of related classes. It is a good idea to place each class in a package so you can group related classes together and avoid conflicts when multiple classes have the same name. In this book, we'll use chapter and section numbers as package names. The full name of our class is `ch01.sec01.HelloWorld`.

[Chapter 2](#) has more to say about packages and package naming conventions.

- The line starting with `//` is a comment. All characters between `//` and the end of the line are ignored by the compiler and are meant for human readers only.
- Finally, we come to the body of the `main` method. In our example, it consists of a single line with a command to print a message to `System.out`, an object representing the “standard output” of the Java program.

As you can see, Java is not a scripting language that can be used to quickly dash off a few commands. It is squarely intended as a language for larger programs that benefit from being organized into classes, packages, and modules. (Modules are introduced in

Chapter 12.)

Java is also quite simple and uniform. Some languages have global variables and functions as well as variables and methods inside classes. In Java, everything is declared inside a class. This uniformity can lead to somewhat verbose code, but it makes it easy to understand the meaning of a program.



Note: You have just seen a // comment that extends to the end of the line. You can also have multiline comments between /* and */ delimiters, such as

```
/*
    This is the first sample program in Core Java for
    the Impatient.
    The program displays the traditional greeting
    "Hello, World!".
*/
```

There is a third comment style, called *documentation comment*, with /** and */ as delimiters, that you will see in the next chapter.

1.1.2. Compiling and Running a Java Program

To compile and run this program, you need to install the Java Development Kit (JDK) and, optionally, an integrated development environment (IDE). You should also download the sample code, which you will find at the companion website for this book, <http://horstmann.com/javaimpatient>. Since instructions for installing software don't make for interesting reading, I put them on the companion website as well.

Once you have installed the JDK, open a terminal window, change to the directory containing the ch01 directory, and run the commands

```
javac ch01/sec01/HelloWorld.java  
java ch01.sec01.HelloWorld
```

The familiar greeting will appear in the terminal window (see [Figure 1.1](#)).

Note that two steps were involved to execute the program. The `javac` command *compiles* the Java source code into an intermediate machine-independent representation, called *byte codes*, and saves them in *class files*. The `java` command launches a *virtual machine* that loads the class files and executes the byte codes.

Once compiled, byte codes can run on any Java virtual machine, whether on your desktop computer or on a device in a galaxy far, far away. The promise of “write once, run anywhere” was an important design criterion for Java.

```
Terminal  
~$ cd books/cji/code  
~/books/cji/code$ javac ch01/sec01/HelloWorld.java  
~/books/cji/code$ ls ch01/sec01  
HelloWorld.class HelloWorld.java MethodDemo.java  
~/books/cji/code$ java ch01.sec01.HelloWorld  
Hello, World!  
~/books/cji/code$
```

Class file

Program output

Figure 1.1: Running a Java program in a terminal window



Note: The `javac` compiler is invoked with the name of a *file*, with slashes separating the path segments, and an extension `.java`. The `java` virtual machine launcher is invoked with the name of a *class*, with dots separating the package segments, and no extension.



Note: If your program consists of a single source file, then you can skip the compilation step and run the program with the command

```
java ch01/sec01/HelloWorld.java
```

Behind the scenes, the program is compiled before it runs, but no class files are produced.



Note: On Unix-like operating systems, you can turn a Java file into an executable program by following these steps:

1. Rename the file so that it doesn't have extension `.java`:

```
mv HelloWorld.java hello
```

2. Make the file executable:

```
chmod +x hello
```

3. Add a “shebang” line at the top of the file:

```
#!/path/to/jdk/bin/java --source 21
```

Now you can run the program as

```
./hello
```



Note: The verbosity of `public static void main` is about to give way to something simpler. Java 21 has a preview feature, described in Java Enhancement Proposal (JEP) 445, that allows you to write the `main` method like this:

```
class HelloWorld {  
    void main() {  
        System.out.println("Hello, World!");  
    }  
}
```

No `public`, no `static`, no `String[] args`.

For very simple programs, JEP 445 allows you to omit the class, provided, of course, there is only one. The file `HelloWorld.java` can simply contain:

```
void main() {  
    System.out.println("Hello, World!");  
}
```

For now, you need to run the program with a command-line flag:

```
java --enable-preview --source 21 HelloWorld.java
```

Since these features are still in preview, I won't use them in this edition, but in the future, the first sample program will be a bit simpler.

To run the `HelloWorld` program in an IDE, you need to first make a project, as described in the installation instructions. Then, select the `HelloWorld` class and tell the IDE to run it. [Figure 1.2](#) shows how this looks in Eclipse. Eclipse is a popular open-source IDE, but there are many other excellent choices. As you get more comfortable with Java programming, you should try out a few and pick one that you like.

eclipse-workspace - cji3/ch01/sec01/HelloWorld.java - Eclipse IDE

File Edit Source Refactor Navigate Search Project Run Window Help

Package x HelloWorld.java x

```
1 package ch01.sec01;
2
3 // Our first Java program
4
5 public class HelloWorld {
6     public static void main(String[] args) {
7         System.out.println("Hello, World!");
8     }
9 }
10
```

Problems Javadoc Declaration Search Console x

<terminated> HelloWorld (1) [Java Application] /data/apps/dk-17/bin/java (6 apr 2022, 12:00:23 - 12:00:23)

Hello, World!

Writable Smart Insert 1:1:0

A screenshot of the Eclipse IDE interface. The title bar says "eclipse-workspace - cji3/ch01/sec01/HelloWorld.java - Eclipse IDE". The menu bar includes File, Edit, Source, Refactor, Navigate, Search, Project, Run, Window, and Help. The toolbar has various icons for file operations. On the left is the Package Explorer view showing a project structure with packages like "cji3" and "ch01.sec01" containing files like "HelloWorld.java". The central area is the "HelloWorld.java" editor window displaying the Java code. Below it is the "Console" view showing the output of the application: "Hello, World!". The bottom status bar shows "Writable", "Smart Insert", and a ratio of "1:1:0".

Figure 1.2: Running a Java program inside the Eclipse IDE

Congratulations! You have just followed the time-honored ritual of running the “Hello, World!” program in Java. Now we are ready to examine the basics of the Java language.



Tip: At <http://horstmann.com/javaimpatient>, you can download the sample code for all chapters of this book that presents the book’s code snippets in context. The code is organized so that you can make a single project that holds all the example programs. I encourage you to download, run, and study the code as you follow along with the book content.

1.1.3. Object Instances and Method Calls

Let us have a closer look at the single statement of the `main` method:

```
System.out.println("Hello, World!");
```

`System.out` is an object. It is an *instance* of a class called `PrintStream`. The `PrintStream` class has methods `println`, `print`, and so on. These methods are called *instance methods* because they operate on objects, or instances, of the class.

To invoke an instance method on an object, you use the *dot notation*

$$\text{object.methodName(arguments)}$$

In this case, there is just one argument, the string "Hello, World!".

Let’s try it with another example. Strings such as "Hello, World!" are instances of the `String` class. The `String` class has a method `length` that

returns the length of a `String` object. To call the method, you again use the dot notation:

```
"Hello, World!".length()
```

The `length` method is invoked on the object `"Hello, World!"`, and it has no arguments. Unlike the `println` method, the `length` method returns a result. One way of using that result is to print it:

```
System.out.println("Hello, World!".length());
```

Give it a try. Make a Java program with this statement and run it to see how long the string is.

In Java, you need to *construct* most objects (unlike the `System.out` and `"Hello, World!"` objects, which are already there, ready for you to use). Here is a simple example.

An object of the `Random` class can generate random numbers. You construct a `Random` object with the `new` operator:

```
new Random()
```

After the class name is the list of construction arguments, which is empty in this example.

You can call a method on the constructed object. The call

```
new Random().nextInt()
```

yields the next integer that the newly constructed random number generator has to offer.

You just saw how the `new` operator constructs an object instance. *Factory methods* provide an alternate way of producing objects. For example, here we produce another random number generator with a factory method:

```
RandomGenerator generator = RandomGenerator.getDefault();
```

In modern Java, this is the preferred way of getting a random number generator. The factory method yields a generator with better characteristics than the legacy `Random` class.

In this example, the object is stored in a variable (see [Section 1.3](#)) so that it can be accessed more than once. Here we print two random numbers:

```
System.out.println(generator.nextInt());  
System.out.println(generator.nextInt());
```



Note: The `RandomGenerator` type is declared in the `java.util.random` package. To use it in your program, add an `import` statement, like this:

```
package ch01.sec01;  
  
import java.util.random.RandomGenerator;  
  
public class MethodDemo {  
    ...  
}
```

We will look at packages and the `import` statement in more detail in

Chapter 2.

1.1.4. JShell

In [Section 1.1.2](#), you saw how to compile and run a Java program. The JShell program provides a “read-evaluate-print loop” (REPL) that allows you to experiment with Java code without compiling and running a program. When you type a Java expression, JShell evaluates your input, prints the result, and waits for your next input. To start JShell, simply type `jshell` in a terminal window ([Figure 1.3](#)).

A screenshot of a terminal window titled "Terminal ~\$". The window contains the following text:

```
~$ jshell
| Welcome to JShell -- Version 17
| For an introduction type: /help intro

jshell> "Hello, World!".length()
$1 ==> 13

jshell> new Random().nextInt()
$2 ==> -762255172

jshell> Random generator = new Random(42)
generator ==> java.util.Random@7cc355be

jshell> generator.nextInt()
$4 ==> -1170105035

jshell> generator.nextInt()
$5 ==> 234785527

jshell> generator.next
nextBoolean()      nextBytes()        nextDouble()       nextExponential()
nextFloat()        nextGaussian()     nextInt()         nextLong()
jshell> generator.next
```

The terminal window has a dark theme with white text and a light gray background. The scroll bar on the right side shows that there is more text than fits on the screen.

Figure 1.3: Running JShell

JShell starts with a greeting, followed by a prompt:

```
| Welcome to JShell -- Version 21
| For an introduction type: /help intro

jshell>
```

Now type any Java expression, such as

```
"Hello, World!".length()
```

JShell responds with the result and another prompt.

```
$1 ==> 13
```

```
jshell>
```

Note that you do *not* type `System.out.println`. JShell automatically prints the value of every expression that you enter.

The `$1` in the output indicates that the result is available in further calculations. For example, if you type

```
3 * $1 + 3
```

the response is

```
$2 ==> 42
```

If you need a variable many times, you can give it a more memorable name. You have to follow the Java syntax and specify both the type and the name (see [Section 1.3](#)). For example,

```
jshell> int answer = 42
answer ==> 42
```

You can have JShell fill in the type for you. Type an expression and instead of hitting the Enter key, hit Shift+Tab and then the V key. For example, when you type

```
new Random()
```

followed by Shift+Tab and the V key, you get

```
jshell> Random = new Random()
```

with the cursor positioned just before the = symbol. Now type a variable name and hit Enter:

```
jshell> Random generator = new Random()  
generator ==> java.util.Random@3fee9989
```

Another useful feature is tab completion. Type
generator.

followed by the Tab key. You get a list of all methods that you can invoke on the generator variable:

```
jshell> generator.  
doubles()           equals()           getClass()  
hashCode()          ints()             isDeprecated()  
longs()              nextBoolean()       nextBytes()  
nextDouble()         nextExponential()  nextFloat()  
nextGaussian()       nextInt()           nextLong()  
notify()             notifyAll()         setSeed()  
toString()           wait()
```

Now type ne and hit the Tab key again. The method name is completed to next, and you get a shorter list:

```
jshell> generator.next  
nextBoolean()        nextBytes()        nextDouble()  
nextExponential()    nextFloat()        nextGaussian()  
nextInt()            nextLong()
```

Type a D and Tab again, and now the completion nextDouble(is filled in. Hit Tab again, and you see three alternatives:

Signatures:

```
double java.util.random.RandomGenerator.nextDouble(double  
bound)  
double java.util.random.RandomGenerator.nextDouble(double
```

```
origin, double bound)
double Random.nextDouble()
```

<press tab again to see documentation>

Type a) to select the third version:

```
jshell> generator.nextDouble()
$3 ==> 0.9560346568377398
```



Note: In the autocompletion list, methods that may require an argument are only followed by a left parenthesis, such as `nextDouble()`, but methods without arguments have both parentheses, such as `nextBoolean()`.

To repeat a command, hit the ↑ key until you see the line that you want to reissue or edit. You can move the cursor in the line with the ← and → keys, and add or delete characters. Hit Enter when you are done. For example, hit ↑ and replace Double with Int, then hit Enter:

```
jshell> generator.nextInt()
$4 ==> -352355569
```

By default, JShell imports the following packages:

```
java.io
java.math
java.net
java.nio.file
java.util
java.util.concurrent
java.util.function
java.util.prefs
java.util.regex
java.util.stream
```

That's why you can use the `Random` class in JShell without any import statements. If you need to import another class, you can type the import statement at the JShell prompt. Or, more conveniently, you can have JShell search for it, by typing Shift+Tab and the I key. For example, type `Duration` followed by Shift+Tab and the I key. You get a list of potential actions:

```
jshell> Duration
0: Do nothing
1: import: java.time.Duration
2: import: javafx.util.Duration
3: import: javax.xml.datatype.Duration
Choice:
```

Type 1, and you receive a confirmation:

```
Imported: java.time.Duration
```

followed by

```
jshell> Duration
```

so that you can pick up where you left off, but with the import in place.

These commands are enough to get you started with JShell. To get help, type `/help` and Enter. To exit, type `/exit` and Enter, or simply Ctrl+D.

JShell makes it easy and fun to learn about the Java language and library, without having to launch a heavy-duty development environment, and without fussing with `public static void main`.

1.2. Primitive Types

Even though Java is an object-oriented programming language, not all Java values are objects. Instead, some values belong to *primitive types*. Four of these types are signed integer types, two are floating-point number types, one is the character type `char` that is used in the encoding for strings, and

one is the `boolean` type for truth values. We will look at these types in the following sections.

1.2.1. Signed Integer Types

The signed integer types are for numbers without fractional parts. Negative values are allowed. Java provides the four signed integer types shown in [Table 1.1](#).

Table 1.1: Java Signed 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



Note: The constants `Integer.MIN_VALUE` and `Integer.MAX_VALUE` are the smallest and largest `int` values. The `Long`, `Short`, and `Byte` classes also have `MIN_VALUE` and `MAX_VALUE` constants.

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.



Note: If the `long` type is not sufficient, use the `BigInteger` class. See [Section 1.4.6](#) for details.

In Java, the ranges of the integer types do not depend on the machine on which you will be running your program. After all, Java is designed as a “write once, run anywhere” language. In contrast, the integer types in C and C++ programs depend on the processor for which a program is compiled.

You write `long` integer literals with a suffix `L` (for example, `4000000000L`). There is no syntax for literals of type `byte` or `short`. Use the cast notation (see [Section 1.4.4](#)), for example, `(byte) 127`.

Hexadecimal literals have a prefix `0x` (for example, `0xCAFEBABE`). Binary values have a prefix `0b`. For example, `0b1001` is 9.



Caution: Octal numbers have a prefix `0`. For example, `011` is 9. This can be confusing, so it seems best to stay away from octal literals and leading zeroes.

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: 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, a `byte` value `b` represents the range from `-128` to `127`. If you want a range from `0` to `255`, you can still store it in a `byte`. Due to the nature of binary arithmetic, addition, subtraction, and multiplication will all work, provided they don't overflow. For other operations, call `Byte.toIntUnsignedInt(b)` to get an `int` value between `0` and `255`, then process the integer value, and cast the result back to `byte`. The

`Integer` and `Long` classes have methods for unsigned division and remainder.

1.2.2. Floating-Point Types

The floating-point types denote numbers with fractional parts. The two floating-point types are shown in [Table 1.2](#).

Table 1.2: Floating-Point Types

Type	Storage requirement	Range
float	4 bytes	Approximately $\pm 3.40282347E+38F$ (6–7 significant decimal digits)
double	8 bytes	Approximately $\pm 1.79769313486231570E+308$ (15 significant decimal digits)

Many years ago, when memory was a scarce resource, four-byte floating-point numbers were in common use. But seven decimal digits don't go very far, and nowadays, “double precision” numbers are the default. It only makes sense to use `float` when you need to store a large number of them.

Numbers of type `float` have a suffix `F` (for example, `3.14F`). Floating-point literals without an `F` suffix (such as `3.14`) have type `double`. You can optionally supply the `D` suffix (for example, `3.14D`).



Note: You can specify floating-point literals in hexadecimal. For example, $0.0009765625 = 2^{-10}$ can be written as `0x1.0p-10`. In hexadecimal notation, you use a `p`, not an `e`, to denote the exponent. (An `e` is a hexadecimal digit.) Note that, even though the digits are

written in hexadecimal, the exponent (that is, the power of 2) is written in decimal.

There are special floating-point values `Double.POSITIVE_INFINITY` for ∞ , `Double.NEGATIVE_INFINITY` for $-\infty$, and `Double.NaN` for “not a number.” For example, the result of computing `1.0 / 0.0` is positive infinity. Computing `0.0 / 0.0` or the square root of a negative number yields `NaN`.



Caution: All “not a number” values are considered to be distinct from each other. Therefore, you cannot use the test `if (x == Double.NaN)` to check whether `x` is a `NaN`. Instead, call `if (Double.isNaN(x))`. There are also methods `Double.isInfinite` to test for $\pm \infty$, and `Double.isFinite` to check that a floating-point number is neither infinite nor a `NaN`.

Floating-point numbers are not suitable for financial calculations in which roundoff errors cannot be tolerated. For example, the command `System.out.println(2.0 - 1.7)` prints `0.3000000000000004`, not `0.3` 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 $3/10$, just as there is no accurate representation of the fraction $1/3$ in the decimal system. If you need precise numerical computations with arbitrary precision and without roundoff errors, use the `BigDecimal` class, introduced in [Section 1.4.6](#).

1.2.3. The `char` Type

The `char` type describes “code units” in the UTF-16 character encoding used by Java. The details are rather technical—see [Section 1.5](#). You probably won’t use the `char` type very much.

Occasionally, you may encounter character literals, enclosed in single quotes. For example, `'J'` is a character literal with value 74 (or hexadecimal

4A), the code unit for denoting the Unicode character “U+004A Latin Capital Letter J.” A code unit can be expressed in hexadecimal, with the \u prefix. For example, '\u004A' is the same as 'J'. A more exotic example is '\u2620', the code unit for ☠, “U+2620 Skull and Crossbones.”

The special codes '\n', '\r', '\t', '\b' denote a line feed, carriage return, tab, and backspace.

Use a backslash to escape a single quote '\'' and a backslash '\\'.

1.2.4. The boolean Type

The boolean type has two values, `false` and `true`.

In Java, the boolean type is not a number type. There is no relationship between boolean values and the integers 0 and 1.

1.3. Variables

In the following sections, you will learn how to declare and initialize variables and constants.

1.3.1. Variable Declarations

Java is a strongly typed language. Each variable can only hold values of a specific type. When you declare a variable, you specify the type, the name, and an optional initial value. For example,

```
int total = 0;
```

You can declare multiple variables of the same type in a single statement:

```
int total = 0, count; // count is an uninitialized integer
```

Most Java programmers prefer to use separate declarations for each variable.

Consider this variable declaration:

```
Random generator = new Random();
```

Here, the name of the object's class occurs twice. The first `Random` is the type of the variable `generator`. The second `Random` is a part of the `new` expression for constructing an object of that class.

To avoid this repetition, you can declare a variable with the `var` keyword:

```
var generator = new Random();
```

Now, the type of the variable is the type of the expression with which the variable is initialized. In this example, `generator` is a variable of type `Random`.

In this book, I will use `var` whenever the type of the declared variable is completely obvious.

1.3.2. Identifiers

The name of a variable, method, or class is called an *identifier*. In Java, an identifier must begin with a letter. It can consist of any letters, digits, currency symbols, and “punctuation connectors” such as `_` and a few other underscore-like characters such as `__`. However, the `$` symbol is intended for automatically generated identifiers, and you should not use it. Finally, the `_` by itself is not a valid identifier.

Here, letters and digits can be from *any* alphabet, not just the Latin alphabet. For example, `π` and `élévation` are valid identifiers. Letter case is significant: `count` and `Count` are different identifiers.

You cannot use spaces or symbols in identifiers. Finally, you cannot use keywords such as `double`.

By convention, names of variables and methods start with a lowercase letter, and names of classes start with an uppercase letter. Java programmers like

“camel case,” where uppercase letters are used when names consist of multiple words, like `countOfInvalidInputs`.

1.3.3. Initialization

When you declare a variable in a method, you must initialize it before you can use it. For example, the following code results in a compile-time error:

```
int count;  
count++; // Error—uses an uninitialized variable
```

The compiler must be able to verify that a variable has been initialized before it has been used. For example, the following code is also an error:

```
int count;  
if (total == 0) {  
    count = 0;  
} else {  
    count++; // Error—count might not be initialized  
}
```

You are allowed to declare a variable anywhere within a method. It is considered good style to declare a variable as late as possible, just before you need it for the first time. For example,

```
var in = new Scanner(System.in); // See Section 1.6.1 for  
reading input  
System.out.println("How old are you?");  
int age = in.nextInt();
```

The variable is declared at the point at which its initial value is available.

1.3.4. Constants

The `final` keyword denotes a value that cannot be changed once it has been assigned. In other languages, one would call such a value a *constant*. For

example,

```
final int DAYS_PER_WEEK = 7;
```

By convention, uppercase letters are used for names of constants.

You can also declare a constant outside a method, using the `static` keyword:

```
public class Calendar {  
    public static final int DAYS_PER_WEEK = 7;  
    ...  
}
```

Then the constant can be used in multiple methods. Inside `Calendar`, you refer to the constant as `DAYS_PER_WEEK`. To use the constant in another class, prepend the class name: `Calendar.DAYS_PER_WEEK`.



Note: The `System` class declares a constant

```
public static final PrintStream out
```

that you can use anywhere as `System.out`. This is one of the few examples of a constant that is not written in uppercase.

It is legal to defer the initialization of a `final` variable, provided it is initialized exactly once before it is used for the first time. For example, the following is legal:

```
final int DAYS_IN_FEBRUARY;  
if (leapYear) {  
    DAYS_IN_FEBRUARY = 29;  
} else {  
    DAYS_IN_FEBRUARY = 28;  
}
```

That is the reason for calling them “final” variables. Once a value has been assigned, it is final and can never be changed.



Note: Sometimes, you need a set of related constants, such as

```
public static final int MONDAY = 0;  
public static final int TUESDAY = 1;  
...
```

In this case, you can define an *enumeration* like this:

```
enum Weekday { MONDAY, TUESDAY, WEDNESDAY, THURSDAY,  
FRIDAY,  
SATURDAY, SUNDAY };
```

Then, `Weekday` is a type with values `Weekday.MONDAY` and so on. Here is how you declare and initialize a `Weekday` variable:

```
Weekday startDay = Weekday.MONDAY;
```

We will discuss enumerations in

1.4. Arithmetic Operations

Java uses the familiar operators of any C-based language (see [Table 1.3](#)). We will review them in the following sections.

Table 1.3: Java Operators

Operators	Associativity
[] . () (method call)	Left
! ~ ++ -- + (unary) - (unary) () (cast) new	Right
* / % (modulus)	Left
+ -	Left
<< >> >>> (arithmetic shift)	Left
< > <= >= instanceof	Left
== !=	Left
& (bitwise and)	Left
^ (bitwise exclusive or)	Left
(bitwise or)	Left
&& (logical and)	Left
(logical or)	Left

Operators	Associativity
? : (conditional)	Left
= += -= *= /= %= <<= >>= >>>= &= ^= =	Right



Note: In this table, operators are listed by decreasing *precedence*. For example, since + has a higher precedence than <<, the value of $3 + 4 << 5$ is $(3 + 4) << 5$. An operator is *left-associative* when it is grouped left to right. For example, $3 - 4 - 5$ means $(3 - 4) - 5$. But -= is right-associative, and $i -= j -= k$ means $i -= (j -= k)$.

1.4.1. Assignment

The last row in [Table 1.3](#) shows the assignment operators. The statement

`x = expression;`

sets `x` to the value of the right-hand side, replacing the previous value.

Assignment is an operator with a value, namely the value that is being assigned. It is legal to use an assignment in another expression. For example,

```
(next = in.read()) != -1
```

sets `next` to the value that is returned by `in.read()`, and then evaluates to true if that value is not `-1`.

When `=` is preceded by an operator, the operator combines the left- and right-hand sides and the result is assigned. For example,

```
amount -= fee;
```

is the same as

```
amount = amount - fee;
```

1.4.2. Basic Arithmetic

Addition, subtraction, multiplication, and division are denoted by + - * /. For example, $2 * n + 1$ means to multiply 2 and n and add 1.

You need to be careful with the / operator. If both operands are integer types, it denotes integer division, discarding the remainder. For example, $17 / 5$ is 3, whereas $17.0 / 5$ is 3.4.

An integer division by zero gives rise to an exception which, if not caught, will terminate your program. (See

[Chapter 5](#) for more information on exception handling.) A floating-point division by zero yields an infinite value or NaN (see [Section 1.2.2](#)), without causing an exception.

The % operator yields the remainder. For example, $17 \% 5$ is 2, the amount that remains from 17 after subtracting 15 (the largest integer multiple of 5 that “fits” into 17). If the remainder of $a \% b$ is zero, then a is an integer multiple of b .

A common use is to test whether an integer is even. The expression $n \% 2$ is 0 if n is even. What if n is odd? Then $n \% 2$ is 1 if n is positive or -1 if n is negative. That handling of negative numbers is unfortunate in practice. Always be careful using % with potentially negative operands.

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: `(position + adjustment) % 12`. But what if `adjustment` makes the position negative? Then you might get a negative number. So you have to introduce a branch, or use `((position + adjustment) % 12 + 12) % 12`. Either way, it is a hassle.



Tip: In this case, it is easier to use the `Math.floorMod` method: `Math.floorMod(position + adjustment, 12)` always yields a value between 0 and 11.

Sadly, `floorMod` gives negative results for negative divisors, but that situation doesn't often occur in practice.

Java has increment and decrement operators:

```
n++; // Adds one to n  
n--; // Subtracts one from n
```

As in other C-based languages, there is also a prefix form of these operators. Both `n++` and `++n` increment the variable `n`, but they have different values

when they are used inside an expression. The first form yields the value before the increment, and the second the value after the increment. For example,

```
String arg = args[n++];
```

sets `arg` to `args[n]`, and *then* increments `n`. This made sense thirty years ago when compilers didn't do a good job optimizing code. Nowadays, there is no performance drawback in using two separate statements, which many programmers find easier to read.

1.4.3. Mathematical Methods

There is no operator for raising numbers to a power. Instead, call the `Math.pow` method: `Math.pow(x, y)` yields x^y . To compute the square root of `x`, call `Math.sqrt(x)`.

These are *static* methods that don't operate on objects. Like with *static* constants, you prepend the name of the class in which they are declared.

Also useful are `Math.min` and `Math.max` for computing the minimum and maximum of two values.

In addition, the `Math` class provides trigonometric and logarithmic functions as well as the constants `Math.PI`, `Math.E`, and `Math.TAU`.



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 also

methods `addExact`, `subtractExact`, `incrementExact`,
`decrementExact`, `negateExact`, all with `int` and `long` parameters.

A few mathematical methods are in other classes. For example, there are methods `compareUnsigned`, `divideUnsigned`, and `remainderUnsigned` in the `Integer` and `Long` classes to work with unsigned values.

1.4.4. Number Type Conversions

When an operator combines operands of different number types, the numbers are automatically converted to a common type before they are combined. Conversion occurs in this order:

1. If either of the operands is of type `double`, the other one is converted to `double`.
2. If either of the operands is of type `float`, the other one is converted to `float`.
3. If either of the operands is of type `long`, the other one is converted to `long`.
4. Otherwise, both operands are converted to `int`.

For example, if you compute `3.14 + 42`, the second operand is converted from an `int` to the `double` value `42.0`, and then the sum is computed, yielding the `double` value `45.14`.

If you compute `'J' + 1`, the `char` value `'J'` is converted to the `int` value `74`, and the result is the `int` value `75`. Read on to find out how to convert that value back to a `char`.

When you assign a value of a numeric type to a variable, or pass it as an argument to a method, and the types don't match, the value must be converted.

For example, in the assignment

```
double x = 42;
```

the value 42 is automatically converted from `int` to `double`.

In Java, conversions always exist if there is no loss of information:

- From `byte` to `short`, `int`, `long`, or `double`
- From `short` and `char` to `int`, `long`, or `double`
- From `int` to `long` or `double`

All integer types can be converted to floating-point types.



Caution: The following conversions may lose information:

- From `int` to `float`
- From `long` to `float` or `double`

For example, consider the assignment

```
float f = 123456789;
```

Because a `float` only has about seven significant digits, `f` is actually 1.23456792E8.

To make a conversion that is not among these automatic ones, use a *cast* operator: the name of the target type in parentheses. For example,

```
double x = 3.75;  
int n = (int) x;
```

In this case, the fractional part is discarded, and `n` is set to 3.

If you want to round to the nearest integer instead, use the `Math.round` method. That method returns a `long`. If you know the answer fits into an `int`, call

```
int n = (int) Math.round(x);
```

In our example, where `x` is 3.75, `n` is set to 4.

To convert an integer type to another one with fewer bytes, also use a cast:

```
int n = 1;  
char next = (char)('J' + n); // Converts 75 to 'K'
```

In such a cast, only the last bytes are retained.

```
int n = (int) 3000000000L; // Sets n to -1294967296
```



Note: If you worry that a cast can silently throw away important parts of a number, use the `Math.toIntExact` method instead. When it cannot convert a `long` to an `int`, an exception occurs.

1.4.5. Relational and Logical Operators

The `==` and `!=` operators test for equality. For example, `n != 0` is `true` when `n` is not zero.

There are also the usual `<` (less than), `>` (greater than), `<=` (less than or equal), and `>=` (greater than or equal) operators.

You can combine expressions of type `boolean` with the `&&` (and), `||` (or), and `!` (not) operators. For example,

$$0 \leq n \&\& n < \text{length}$$

is `true` if `n` lies between zero (inclusive) and `length` (exclusive).

If the first condition is `false`, the second condition is not evaluated. This “short circuit” evaluation is useful when the second condition could cause an error. Consider the condition

$$n \neq 0 \&\& s + (100 - s) / n < 50$$

If n is zero, then the second condition, which contains a division by n , is never evaluated, and no error occurs.

Short circuit evaluation is also used for “or” operations, but then the evaluation stops as soon as an operand is `true`. For example, the computation of

```
n == 0 || s + (100 - s) / n >= 50
```

yields `true` if n is zero, again without evaluating the second condition.

Finally, the *conditional* operator takes three operands: a condition and two values. The result is the first of the values if the condition is `true`, the second otherwise. For example,

```
time < 12 ? "am" : "pm"
```

yields the string "`am`" if `time < 12` and the string "`pm`" otherwise.



Note: There are *bitwise* operators `&` (and), `|` (or), and `^` (xor) that are related to the logical operators. They operate on the bit patterns of integers. For example, since `0xF` has binary digits `0...01111`, $n \& 0xF$ yields the lowest four bits in n , $n = n | 0xF$ sets the lowest four bits to 1, and $n = n ^ 0xF$ flips them. The analog to the `!` operator is `~`, which flips all bits of its operand: $\sim 0xF$ is `1...10000`.

There are also operators which shift a bit pattern to left or right. For example, `0xF << 2` has binary digits `0...0111100`. There are two right shift operators: `>>` extends the sign bit into the top bits, and `>>>` fills the top bits with zero. If you do bit-fiddling in your programs, you know what that means. If not, you won’t need these operators.



Caution: The right-hand side operand of the shift operators is reduced modulo 32 if the left hand side is an `int`, or modulo 64 if

the left hand side is a `long`. For example, the value of `1 << 35` is the same as `1 << 3` or `8`.



Tip: The `&` (and) and `|` (or) operators, when applied to `boolean` values, force evaluation of both operands before combining the results. This usage is very uncommon. Provided that the right hand side doesn't have a side effect, they act just like `&&` and `||`, except they are less efficient. If you really need to force evaluation of the second operand, assign it to a `boolean` variable so that the flow of execution is plainly visible.

1.4.6. Big Numbers

If the precision of the primitive integer and floating-point types is not sufficient, you can turn to the `BigInteger` and `BigDecimal` classes in the `java.math` package. Objects of these classes represent 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. Of course, computations with big numbers are far slower than with primitive numeric types.

The static `valueOf` method turns a `long` into a `BigInteger`:

```
BigInteger n = BigInteger.valueOf(876543210123456789L);
```

You can also construct a `BigInteger` from a string of digits:

```
var k = new BigInteger("9876543210123456789");
```

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

Java does not permit the use of operators with objects, so you must use method calls to work with big numbers.

```
BigInteger r = BigInteger.valueOf(5).multiply(n.add(k)); //  
r = 5 * (n + k)
```

The `parallelMultiply` method yields the same result as `multiply` but can potentially compute the result faster by using multiple processor cores.

In [Section 1.2.2](#), you saw that the result of the floating-point subtraction $2.0 - 1.7$ is 0.3000000000000004 . The `BigDecimal` class can compute the result accurately.

The call `BigDecimal.valueOf(n, e)` returns a `BigDecimal` instance with value $n \times 10^{-e}$. The result of

```
BigDecimal.valueOf(2, 0).subtract(BigDecimal.valueOf(17,  
1))
```

is exactly 0.3 .

1.5. Strings

A string is a sequence of characters. In Java, a string can contain any Unicode characters. For example, the string "Java™" or "Java\u2122" consists of the five characters J, a, v, a, and ™. The last character is "U+2122 Trade Mark Sign."

1.5.1. Concatenation

Use the `+` operator to concatenate two strings. For example,

```
String location = "Java";  
String greeting = "Hello " + location;
```

sets `greeting` to the string "Hello Java". (Note the space at the end of the first operand.)

When you concatenate a string with another value, that value is converted to a string.

```
int age = 42;  
String output = age + " years";
```

Now `output` is "42 years".



Caution: If you mix concatenation and addition, then you may get unexpected results. For example,

```
"Next year, you will be " + age + 1 // Error
```

first concatenates `age` and then `1`. The result is "Next year, you will be 421". In such cases, use parentheses:

```
"Next year, you will be " + (age + 1) // OK
```

To combine several strings, separated with a delimiter, use the `join` method:

```
String names = String.join(", ", "Peter", "Paul", "Mary");  
// Sets names to "Peter, Paul, Mary"
```

The first argument is the separator string, followed by the strings you want to join. There can be any number of them, or you can supply an array of strings. (Arrays are covered in [Section 1.8](#).)

It is somewhat inefficient to concatenate a large number of strings if all you need is the final result. In that case, use a `StringBuilder` instead:

```
var builder = new StringBuilder();  
while (more strings) {  
    builder.append(next string);
```

```
}
```

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

1.5.2. Substrings

To take strings apart, use the `substring` method. For example,

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

```
String location = greeting.substring(7, 12); // Sets
```

```
location to "World"
```

The first argument of the `substring` method is the starting position of the substring to extract. Positions start at `0`.

The second argument is the first position that should not be included in the substring. In our example, position `12` of `greeting` is the `!`, which we do not want. It may seem curious to specify an unwanted position, but there is an advantage: the difference `12 - 7` is the length of the substring.

Sometimes, you want to extract all substrings from a string that are separated by a delimiter. The `split` method carries out that task, returning an array of substrings.

```
String names = "Peter, Paul, Mary";
```

```
String[] result = names.split(", ");
```

```
// An array of three strings ["Peter", "Paul", "Mary"]
```

The separator can be any regular expression (see

[Chapter 9](#)). For example, `input.split("\\s+")` splits `input` at white space.

1.5.3. String Comparison

To check whether two strings are equal, use the `equals` method. For example,

```
location.equals("World")
```

yields `true` if `location` is in fact the string "World".



Caution: Never use the `==` operator to compare strings. The comparison

```
location == "World" // Don't do that!
```

returns `true` only if `location` and "World" are *the same object in memory*. In the virtual machine, there is only one instance of each literal string, so "World" `==` "World" will be `true`. But if `location` was computed, for example, as

```
String location = greeting.substring(7, 12);
```

then the result is placed into a separate `String` object, and the comparison `location == "World"` will return `false`!

Like any object variable, a `String` variable can be `null`, which indicates that the variable does not refer to any object at all, not even an empty string.

```
String middleName = null;
```

To test whether an object is `null`, you do use the `==` operator:

```
if (middleName == null) ...
```

Note that `null` is not the same as an empty string `""`. An empty string is a string of length zero, whereas `null` isn't any string at all.



Caution: Invoking any method on `null` causes a “null pointer exception.” Like all exceptions, it terminates your program if you don’t explicitly handle it.



Tip: When comparing a string against a literal string, it is a good idea to put the literal string first:

```
if ("World".equals(location)) ...
```

This test works correctly even when `location` is `null`.

To compare two strings without regard to case, use the `equalsIgnoreCase` method. For example,

```
"world".equalsIgnoreCase(location);
```

returns `true` if `location` is `"World"`, `"world"`, `"WORLD"`, and so on.

Sometimes, one needs to put strings in order. The `compareTo` method tells you whether one string comes before another in dictionary order. The call

```
first.compareTo(second)
```

returns a negative integer (not necessarily `-1`) if `first` comes before `second`, a positive integer (not necessarily `1`) if `first` comes after `second`, and `0` if they are equal.

The strings are compared a character at a time, until one of them runs out of characters or a mismatch is found. For example, when comparing `"word"` and `"worlD"`, the first three characters match. Since `d` has a Unicode value that is less than that of `l`, `"word"` comes first. The call

"word".compareTo("world") returns -8, the difference between the Unicode values of d and l.

This comparison can be unintuitive to humans because it depends on the Unicode values of characters. "café" comes after "caffeine" because é happens to have a higher Unicode value than f.



Tip: When sorting human-readable strings, use a `Collator` object that knows about language-specific sorting rules.

```
Collator.getInstance(Locale.GERMAN).compare("café",  
    "caffeine") // < 0
```

1.5.4. Converting Between Numbers and Strings

To turn an integer into a string, call the static `Integer.toString` method:

```
int n = 42;  
String str = Integer.toString(n); // Sets str to "42"
```

A variant of this method has a second parameter, a radix (between 2 and 36):

```
String str2 = Integer.toString(n, 2); // Sets str2 to  
    "101010"
```



Note: An even simpler way of converting an integer to a string is to concatenate with the empty string: "" + n. Some people find this ugly, and it is slightly less efficient.

Conversely, to convert a string containing an integer to the number, use the `Integer.parseInt` method:

```
String str = "101010";
int n = Integer.parseInt(str); // Sets n to 101010
```

You can also specify a radix:

```
int n2 = Integer.parseInt(str, 2); // Sets n2 to 42
```

For floating-point numbers, use `Double.toString` and `Double.parseDouble`:

```
String str = Double.toString(3.14); // Sets str to "3.14"
double x = Double.parseDouble(str); // Sets x to 3.14
```

1.5.5. The String API

As you might expect, the `String` class has a large number of methods. Some of the more useful ones are shown in [Table 1.4](#).

Table 1.4: Useful String Methods

Method	Purpose
<code>boolean startsWith(String str)</code> <code>boolean endsWith(String str)</code> <code>boolean contains(CharSequence str)</code>	Checks whether a string starts with, ends with, or contains a given string.
<code>int indexOf(String str)</code> <code>int indexOf(String str, int fromIndex)</code> <code>int indexOf(String str, int fromIndex, int toIndex)</code> <code>int lastIndexOf(String</code>	Gets the position of the first or last occurrence of <code>str</code> , searching the entire string or the substring starting at <code>fromIndex</code> . Returns -1 if no match is found.

Method	Purpose
<code>str) int lastIndexOf(String str, int fromIndex)</code>	
<code>String replace(CharSequence oldString, CharSequence newString)</code>	Returns a string that is obtained by replacing all occurrences of <code>oldString</code> with <code>newString</code> .
<code>String toUpperCase() String toLowerCase()</code>	Returns a string consisting of all characters of the original string converted to upper- or lowercase.
<code>String strip()</code>	Returns a string obtained by removing all leading and trailing white space.

Note that in Java, the `String` class is *immutable*. That is, none of the `String` methods modify the string on which they operate. For example,

```
greeting.toUpperCase()
```

returns a *new* string "HELLO, WORLD!" without changing `greeting`.

Also note that some methods have parameters of type `CharSequence`. This is a common supertype of `String`, `StringBuilder`, and other sequences of characters.

For a detailed description of each method, turn to the online Java API documentation at

<https://docs.oracle.com/en/java/javase/21/docs/api>. Type the class name into the search box and select the matching type (in this case, `java.lang.String`), as shown in [javadoc1](#).

Overview (Java SE 17 & JDK 17) – Mozilla Firefox

https://docs.oracle.com/en/java/javase/17/docs/api/

OVERVIEW MODULE PACKAGE CLASS USE TREE PREVIEW NEW DEPRECATED INDEX HELP Java SE 17 & JDK 17

SEARCH: X

Java® Platform, Standard Edition Version 17 API Specification

This document is divided into two sections:

Java SE
The Java Platform, Standard Edition
These APIs are in modules whose names end in `.SE`.

JDK
The Java Development Kit (JDK) API
implementations of the Java SE Platform.

All Modules Java SE JDK Other M

Module	Description
<code>java.base</code>	Defines the foundation of the Java SE platform.
<code>java.compiler</code>	Defines the Language compiler API.
<code>java.datatransfer</code>	Defines the API for clipboard operations.
<code>java.desktop</code>	Defines the AWT and Swing components and JavaBeans.
<code>java.instrument</code>	Defines services that support Java agent development.
<code>java.logging</code>	Defines the Java Logging API.
<code>java.management</code>	Defines the Java Management API.
<code>java.management.rmi</code>	Defines the RMI connection manager API.
<code>java.naming</code>	Defines the Java Naming and Directory Interface (JNDI) API.
<code>java.net.http</code>	Defines the HTTP Client API.
<code>java.prefs</code>	Defines the Preferences API.

Classes and Interfaces

- `java.lang.String`
- `java.lang.StringBuffer`
- `java.io.StringBufferInputStream`
- `java.lang.StringBuilder`
- `java.text.StringCharacterIterator`
- `java.lang.invoke.StringConcatException`
- `java.lang.invoke.StringConcatFactory`
- `javax.swing.text.StringContent`
- `java.lang.StringIndexOutOfBoundsException`
- `java.util.StringJoiner`
- `javax.management.monitor.StringMonitor`
- `javax.management.monitor.StringMonitorMBean`
- `java.io.StringReader`
- `javax.naming.StringRefAddr`
- `com.sun.jdi.StringReference`
- `java.awt.datatransfer.StringSelection`
- `java.util StringTokenizer`
- `javax.management.StringValueExp`
- `java.io.StringWriter`
- `com.sun.jdi.connect.Connector.StringArgument`
- `java.text.AttributedString`
- `javax.management.BadStringOperationException`
- `org.w3c.dom.DOMStringList`
- `javax.swing.table.TableStringConverter`

Members

- `java.lang.String.String()`
- `java.lang.String.String(byte[])`
- `java.lang.String.String(byte[], Charset)`
- `java.lang.String.String(byte[], int)`

<code>java.rmi</code>	Defines the Remote	<code>java.lang.String(String(byte[], int, int))</code>
<code>java.scripting</code>	Defines the Scripting	<code>java.lang.String(String(byte[], int, int, Charset))</code>
<code>java.se</code>	Defines the API of th	<code>java.lang.String(String(byte[], int, int, String))</code>
<code>java.security.jgss</code>	Defines the Java bind	<code>java.lang.String(String(byte[], String))</code>
<code>java.security.sasl</code>	Defines Java support	<code>java.lang.String(String(char[]))</code>
<code>java.smartcardio</code>	Defines the Java Sm	<code>java.lang.String(String(char[], int, int))</code>
<code>java.sql</code>	Defines the JDBC AP	<code>java.lang.String(String(int[]))</code>
<code>java.sql.rowset</code>	Defines the JDBC Ro	<code>java.lang.String(String(StringBuffer))</code>
<code>java.transaction.xa</code>	Defines an API for supporting distributed transactions in JDBC.	
<code>java.xml</code>	Defines the Java API for XML Processing (JAXP), the Streaming API for XML (StAX), the Simple API for XML (SAX), and the W3C Document Object Model (DOM) API.	

Figure 1.4: Searching the API Documentation

You then get a page that documents each method ([Figure 1.5](#)). If you happen to know the name of a method, you can type its name into the search box.

String (Java Platform SE 9 [build 167]) - Mozilla Firefox

download.java.net/java/jdk9/docs/api/index.html?overview-summary.html

Please note that the specifications and other information contained herein are not final and are subject to change. The information is being made available to you solely for purpose of evaluation.

OVERVIEW MODULE PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

SEARCH: Search

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

that sequence *rep* occurs in *subject*,
that matches the literal target sequence
with the specified literal replacement
sequence.

String	<code>replaceAll(String regex, String replacement)</code>	Replaces each substring of this string that matches the given regular expression with the given replacement.
String	<code>replaceFirst(String regex, String replacement)</code>	Replaces the first substring of this string that matches the given regular expression with the given replacement.
String[]	<code>split(String regex)</code>	Splits this string around matches of the given regular expression.
String[]	<code>split(String regex, int limit)</code>	Splits this string around matches of the given regular expression.
boolean	<code>startsWith(String prefix)</code>	Tests if this string starts with the specified prefix.
boolean	<code>startsWith(String prefix, int toffset)</code>	Tests if the substring of this string beginning at the specified index starts with the specified prefix.
CharSequence	<code>subSequence(int beginIndex, int endIndex)</code>	Returns a character sequence that is a subsequence of this sequence.
String	<code>substring(int beginIndex)</code>	Returns a string that is a substring of this string.
String	<code>substring(int beginIndex, int endIndex)</code>	Returns a string that is a substring of this string.
char[]	<code>toCharArray()</code>	Converts this string to a new character



Figure 1.5: The `String` methods in the API Documentation

In this book, I do not present the API in minute detail since it is easier to browse the API documentation. If you are not always connected to the Internet, you can download and unzip the documentation for offline browsing.

1.5.6. Code Points and Code Units

When Java was first created, it proudly embraced the Unicode standard that had been developed shortly before. Prior to Unicode, there were many incompatible character encodings. For English, there was near-universal agreement on the 7-bit ASCII standard that assigned codes between 0 and 127 to all English letters, the decimal digits, and many symbols. In Western Europe, ASCII was extended to an 8-bit code that contained accented characters such as ä and é. But in Russia, ASCII was extended to hold Cyrillic characters in the positions 128 to 255. In Japan, a variable-length encoding was used to encode English and Japanese characters. Multiple incompatible encodings of Chinese characters were in common use. Exchanging files with different encodings was a major problem.

Unicode set out to fix all that by assigning each character in all of the writing systems ever devised a unique 16-bit code between 0 and 65535. In 1991, Unicode 1.0 was released, using slightly less than half of the available 65536 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. But then something awkward happened. There turned out to be many more characters than previously estimated—mostly Chinese ideographs. This pushed Unicode well beyond a 16-bit code.

Nowadays, Unicode requires 21 bits. Each valid Unicode value is called a *code point* and written as U+, followed by four to six hexadecimal digits. For example, the code point for the letter A is U+0041, and the beer mug emoji 🍺 has code point U+1F37A.

Representing code points as 32-bit `int` values is quite wasteful. Java uses a variable-length encoding, called UTF-16. All “classic” Unicode characters are encoded with a single 16-bit value. Those beyond U+FFFF become pairs of 16-bit values taken from a special region of the code space called “surrogate characters.” In this encoding, the letter A is represented by one `char` value, written as \u0041, and the emoji 🍺 is encoded as the pair of `char` values \uD83C and \uDF7A.

In other words, a `char` is not a Unicode code point. It is a *code unit*, a 16-bit value used in the UTF-16 encoding. Some Unicode code points are made up of one `char` value, and others of two.

But it gets even more complex. Code points do not always 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 have modifiers for gender and skin tone, and with an ever-growing number of other composite symbols.

For example, the pirate flag 🏴☠️ is composed of 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 encode the symbol. A sequence of Unicode code points that yields a single perceived symbol is called a *grapheme cluster*.

When analyzing strings, you should generally stay away from processing individual code units and code points. For example, do not use the .

```
char ch = str.charAt(i);
```

It gets the *i*th code unit, but what are you going to do with it? It may well be a part of a complex symbol.

Instead, extract substrings and work with them. Of course, to extract a substring, you need to know the starting and ending index of the code units. You can get a starting index with the `indexOf` and `lastIndexOf` methods. You also get indexes when matching regular expressions (see

[Chapter 9](#)). If you want to extract until the end of the string, then the `length` method gives the last index:

```
int lastIndex = str.length();
```

In Java 1.0, when Unicode was still simple, the `length` method yielded the length of the string; that is, the number of characters. Nowadays, it just counts the code units.

Finally, you can get an array of all grapheme clusters of a string like this:

```
"Ahoy 🏴☠️".split("\\b{g}")  
    // An array with the six elements "A", "h", "o", "y", "  
    ", "🏴☠️"
```



Note: In the past, strings were internally represented in the UTF-16 encoding, as arrays of `char` values. Nowadays, `String` objects use a `byte` array of ISO-8859-1 characters when possible. A future version of Java may switch to using UTF-8 internally. Of course, this internal representation is not visible to programmers.

1.5.7. Text Blocks

The text block syntax 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  
""";
```

This string contains two line breaks: one after `Hello` and one after `World`. The line break after the *opening* """" is not included in the string literal.

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

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

In any line, you can suppress the line break with a backslash as the last character:

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

This string does not contain any line breaks.

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>  
""";
```

Note that you don't have to escape the quotation marks. There are just two situations where you need to escape a quotation mark 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 to escape all backslashes.

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

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 space into a `\s` escape. The following string ends in two spaces:

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

The story is more complex for leading white space. 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 longest sequence of leading white space 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 line.

The white space preceding the closing `""` is significant. However, entirely blank lines are not considered in the indentation removal process.



Caution: The white space prefix has to match *exactly* for all lines in the text block. If you use a mixture of tabs and spaces, you may find that less white space is subtracted than you expect.

1.6. Input and Output

To make our sample programs more interesting, they should be able to interact with the user. In the following sections, you will see how to read terminal input and how to produce formatted output.

1.6.1. Reading Input

When you call `System.out.println`, output is sent to the “standard output stream” and shows up in a terminal window. Reading from the “standard input stream” isn’t quite as simple because the corresponding `System.in` object only has methods to read individual bytes. To read strings and numbers, construct a `Scanner` that is attached to `System.in`:

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

The `nextLine` method reads a line of input.

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

Here, it makes sense to use the `nextLine` method because the input might contain spaces. To read a single word (delimited by whitespace), call

```
String firstName = in.next();
```

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

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

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

You can use the `hasNextLine`, `hasNext`, `hasNextInt`, and `hasNextDouble` methods to check that there is another line, word, integer, or floating-point number available.

```
if (in.hasNextInt()) {  
    int age = in.nextInt();  
    ...  
}
```

The `Scanner` class is located in the `java.util` package. In order to use the class, add the line

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

to the top of your program file.



Tip: To read a password, you do not want to use the `Scanner` class since the input is visible in the terminal. Instead, use the `Console` class:

```
Console terminal = System.console();  
String username = terminal.readLine("User name: ");  
char[] passwd = terminal.readPassword("Password: ");
```

The password is returned in an array of characters. This is marginally more secure than storing the password in a `String` because you can overwrite the array when you are done.



Tip: If you want to read input from a file or write output to a file, you can use the redirection syntax of your shell:

```
java mypackage.MainClass < input.txt > output.txt
```

Now `System.in` reads from `input.txt` and `System.out` writes to `output.txt`. You will see in

[Chapter 9](#) how to carry out more general file input and output.

1.6.2. Formatted Output

You have already seen the `println` method of the `System.out` object for writing a line of output. There is also a `print` method that does not start a new line. That method is often used for input prompts:

```
System.out.print("Your age: "); // Not println  
int age = in.nextInt();
```

Then the cursor rests after the prompt instead of the next line.

When you print a fractional number with `print` or `println`, all of its digits except trailing zeroes will be displayed. For example,

```
System.out.print(1000.0 / 3.0);
```

prints

333.333333333333

That is a problem if you want to display, for example, dollars and cents. To limit the number of digits, use the `printf` method:

```
System.out.printf("%8.2f", 1000.0 / 3.0);
```

The *format string* "%8.2f" indicates that a floating-point number is printed with a *field width* of 8 and 2 digits of *precision*. That is, the printout contains two leading spaces and six characters:

333.33

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

```
System.out.printf("Hello, %s. Next year, you'll be %d.\n",  
name, age);
```

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 1.5](#) shows all conversion characters.

Table 1.5: Conversion Characters for Formatted Output

Conversion character	Purpose	Example
d	Decimal integer	159
x or X	Hexadecimal integer (for more control over hexadecimal formatting, use the <code>HexFormat</code> class)	9f or 9F
o	Octal integer	237
f	Fixed floating-point	15.9
e or E	Exponential floating-point	1.59e+01 or 1.59E+01
g or G	General floating-point: e/E if the exponent is greater than the precision or < -4, f/F otherwise	15.9000 at the default precision of 6, 2e+01 at precision 1
a or A	Hexadecimal floating-point	0x1.fccdp3 or 0X1.FCCDP3

Conversion character	Purpose	Example
s or S	String	Java or JAVA
c or C	Character	j or J
b or B	boolean	false or FALSE
h or H	Hash code (see the section on hash codes)	42628b2 or 42628B2

Conversion character	Purpose	Example
	Chapter 4)	
t or T	Date and time (obsolete; use <code>DateTimeFormatter.ofPattern</code> instead)	—
%	The percent symbol	%
n	The platform-dependent line separator	—

In addition, you can specify flags to control the appearance of the formatted output. [Table 1.6](#) shows all flags. For example, the comma flag adds grouping separators, and + yields a sign for positive numbers. The statement

```
System.out.printf("%,.2f", 100000.0 / 3.0);
```

prints

+33,333.33

Table 1.6: Flags for Formatted Output

Flag	Purpose	Example
+	Prints sign for positive and negative numbers.	+3333.33
space	Adds a space before positive numbers.	_3333.33
-	Left-justifies field.	3333.33__
0	Adds leading zeroes.	003333.33

Flag	Purpose	Example
(Encloses negative values in parentheses.	(3333.33)
,	Uses group separators.	3,333.33
# (for f or e format)	Always includes a decimal point.	3333.
# (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

You can use the `formatted` method to create a formatted string without printing it:

```
String message = "Hello, %s. Next year, you'll be
%d.\n".formatted(name, age);
```

1.7. Control Flow

In the following sections, you will see how to implement branches and loops. The Java syntax for control flow statements is very similar to that of

other commonly used languages, in particular C/C++ and JavaScript.

1.7.1. Branches

The `if` statement has a condition in parentheses, followed by either one statement or a group of statements enclosed in braces.

```
if (count > 0) {  
    double average = sum / count;  
    System.out.println(average);  
}
```

You can have an `else` branch that runs if the condition is not fulfilled.

```
if (count > 0) {  
    double average = sum / count;  
    System.out.println(average);  
} else {  
    System.out.println(0);  
}
```

The statement in the `else` branch may be another `if` statement:

```
if (count > 0) {  
    double average = sum / count;  
    System.out.println(average);  
} else if (count == 0) {  
    System.out.println(0);  
} else {  
    System.out.println("Huh?");  
}
```

1.7.2. Switches

A `switch` expression compares a *selector* against multiple *cases*, producing a value for each case:

```
String seasonName = switch (seasonCode) { // switch expression
    case 0 -> "Spring";
    case 1 -> "Summer";
    case 2 -> "Fall";
    case 3 -> "Winter";
    default -> {
        System.out.printf("Unexpected seasonCode %d%n",
seasonCode);
        yield "??";
    }
};
```

Note that this `switch` is an *expression*—it has a value, namely one of the five strings "Spring", "Summer", "Fall", "Winter", and "???". The value of the `switch` expression is assigned to the `seasonName` variable.

Most often, a case is simply followed by an expression. However, you can do additional work in a brace-enclosed block, as is done in the `default` case of the preceding example. Then you need a `yield` statement inside the block to produce the value.

There is also a statement form, which looks like this:

```
switch (seasonCode) { // switch statement
    case 0 -> seasonName = "Spring";
    case 1 -> seasonName = "Summer";
    case 2 -> seasonName = "Fall";
    case 3 -> seasonName = "Winter";
    default -> {
        System.out.printf("Unexpected seasonCode %d%n",
seasonCode);
        seasonName = "??";
    }
}
```

In the preceding examples, the case labels were integers. You can use values of any of the following types:

- A constant expression of type `char`, `byte`, `short`, or `int` (or their corresponding wrapper classes `Character`, `Byte`, `Short`, and `Integer` that will be introduced in [Section 1.8.3](#))
- A string literal
- A value of an enumeration (see

Chapter 4)

A case can have multiple labels, separated by commas:

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



Note: A switch is *exhaustive* if one of the cases applies for any selector value. A switch *expression* must be exhaustive since it must yield a value, no matter what the selector value is. A switch on an `enum` is exhaustive when there is a case for each enumeration constant. You can make other switches exhaustive by adding a `default` case.

A case can throw an *exception*, as shown in the preceding example. Exceptions are covered in

Chapter 5.

If the selector value of a `switch` is `null`, a `NullPointerException` is thrown. This can happen when the selector type is `String` or an enumeration. To avoid this, use an explicit `case null`.



Caution: A `switch` is *enhanced* when it has one or more features that were not available in Java 14. One such feature is `case null`. All enhanced switches—even statements—must be exhaustive. If you add `case null` to a non-exhaustive `switch` statement, you also need to add a `default` case.

In the `switch` expressions and statements that you just saw, only one case is executed for a given selector value. There is another variant, called “fall-through,” where execution *starts* at the matching case but then continues to the next case unless it is stopped by a `yield` or `break` statement. The fall-through variant also has expression and statement forms. In the following examples, fall-through occurs when `seasonName` is "Spring".

```
int numLetters = switch (seasonName) { // switch expression
    with fall-through
        case "Spring":
            System.out.println("spring time!");
        case "Summer", "Winter":
            yield 6;
        case "Fall":
            yield 4;
        default:
            throw new IllegalArgumentException();
};

switch (seasonName) { // switch statement with fall-through
    case "Spring":
        System.out.println("spring time!");
```

```
        case "Summer", "Winter":  
            numLetters = 6;  
            break;  
        case "Fall":  
            numLetters = 4;  
            break;  
        default:  
            throw new IllegalArgumentException();  
    }  
}
```

Note that in the fall-through variant, each `case` is followed by a colon, not a `->`. Any number of statements can follow, and braces are not needed. In a `switch` expression with fall-through, you must use `yield` to produce a value.



Caution: With the fall-through variant, it is a common error to forget a `yield` or `break` at the end of a case. Avoid that variant unless you actually need the fall-through behavior.

1.7.3. Loops

The `while` loop keeps executing its body while more work needs to be done, as determined by a condition.

For example, consider the task of summing up numbers until the sum has reached a target. For the source of numbers, we will use a random number generator.

```
RandomGenerator generator = RandomGenerator.getDefault();  
while (sum < target) {  
    int next = generator.nextInt(10);  
    sum += next;  
    count++;  
}
```

This is a typical use of a `while` loop. While the sum is less than the target, the loop keeps executing.

Sometimes, you need to execute the loop body before you can evaluate the condition. Suppose you want to find out how long it takes to get a particular value. Before you can test that condition, you need to enter the loop and get the value. In this case, use a `do/while` loop:

```
int next;
do {
    next = generator.nextInt(10);
    count++;
} while (next != target);
```

The loop body is entered, and `next` is set. Then the condition is evaluated. As long as it is fulfilled, the loop body is repeated.

In the preceding examples, the number of loop iterations was not known. However, in many loops that occur in practice, the number of iterations is fixed. In those situations, it is best to use the `for` loop.

This loop computes the sum of a fixed number of random values:

```
for (int i = 1; i <= 20; i++) {
    int next = generator.nextInt(10);
    sum += next;
}
```

This loop runs 20 times, with `i` set to 1, 2, . . . , 20 in each loop iteration.

You can rewrite any `for` loop as a `while` loop. The loop above is equivalent to

```
int i = 1;
while (i <= 20) {
    int next = generator.nextInt(10);
```

```
    sum += next;  
    i++;  
}
```

However, with the `while` loop, the initialization, test, and update of the variable `i` are scattered in different places. With the `for` loop, they stay neatly together.

The initialization, test, and update can take on arbitrary forms. For example, you can double a value while it is less than the target:

```
for (int i = 1; i < target; i *= 2) {  
    System.out.println(i);  
}
```

Instead of declaring a variable in the header of the `for` loop, you can initialize an existing variable:

```
for (i = 1; i <= target; i++) // Uses existing variable i
```

You can declare multiple variables, separated by commas, provided they are of the same type. You can initialize previously declared variables of any types, separated by commas. You can provide multiple updates, also separated by commas. For example,

```
for (int i = 0, j = n - 1; i < j; i++, j--)  
for (s = "", x = 0.5; s.length() < 10; s = s + "*", x /= 2)
```

If no initialization or update is required, leave them blank. If you omit the condition, it is deemed to always be true.

```
for (;;) // An infinite loop
```

You will see in the next section how you can break out of such a loop.

1.7.4. Breaking and Continuing

If you want to exit a loop in the middle, you can use the `break` statement. For example, suppose you want to process words until the user enters the letter Q. Here is a solution that uses a boolean variable to control the loop:

```
boolean done = false;
while (!done) {
    String input = in.next();
    if ("Q".equals(input)) {
        done = true;
    } else {
        Process input
    }
}
```

This loop carries out the same task with a `break` statement:

```
while (true) {
    String input = in.next();
    if ("Q".equals(input)) break; // Exits loop
    Process input
}
// break jumps here
```

When the `break` statement is reached, the loop is exited immediately.

The `continue` statement is similar to `break`, but instead of jumping to the end of the loop, it jumps to the end of the current loop iteration. You might use it to skip unwanted inputs like this:

```
while (in.hasNextInt()) {
    int input = in.nextInt();
    if (input < 0) continue; // Jumps to test of
    in.hasNextInt()
    Process input
}
```

In a `for` loop, the `continue` statement jumps to the next update statement:

```
for (int i = 1; i <= target; i++) {  
    int input = in.nextInt();  
    if (n < 0) continue; // Jumps to i++  
    Process input  
}
```

The **break** statement only breaks out of the immediately enclosing loop or **switch**. If you want to jump to the end of another enclosing statement, use a *labeled break* statement. Label the statement that should be exited, and provide the label with the **break** like this:

```
outer:  
while (...) {  
    ...  
    while (...) {  
        ...  
        if (...) break outer;  
        ...  
    }  
    ...  
}  
// Labeled break jumps here
```

The label can be any name.



Caution: You label the top of the statement, but the **break** statement jumps to the *end*.

A regular **break** can only be used to exit a loop or **switch**, but a labeled **break** can transfer control to the end of any statement, even a block statement:

```
exit: {  
    ...  
    if (...) break exit;
```

```
...
}
// Labeled break jumps here
```

There is also a labeled `continue` statement that jumps to the next iteration of a labeled loop.



Tip: Many programmers find the `break` and `continue` statements confusing. These statements are entirely optional—you can always express the same logic without them. In this book, I never use `break` or `continue`.

1.7.5. Local Variable Scope

Now that you have seen examples of nested blocks, it is a good idea to go over the rules for variable scope. A *local variable* is any variable that is declared in a method, including the method's parameter variables. The *scope* of a variable is the part of the program where you can access the variable. The scope of a local variable extends from the point where it is declared to the end of the enclosing block.

```
while (...) {
    System.out.println(...);
    String input = in.next(); // Scope of input starts here
    ...
    // Scope of input ends here
}
```

In other words, a new copy of `input` is created for each loop iteration, and the variable does not exist outside the loop.

The scope of a parameter variable is the entire method.

```
public static void main(String[] args) { // Scope of args  
starts here  
    ...  
    // Scope of args ends here  
}
```

Here is a situation where you need to understand scope rules. This loop counts how many tries it takes to get a particular random digit:

```
int count = 0;  
int next;  
do {  
    next = generator.nextInt(10);  
    count++;  
} while (next != target);
```

The variable **next** had to be declared outside the loop so it is available in the condition. Had it been declared inside the loop, its scope would only reach to the end of the loop body.

When you declare a variable in a **for** loop, its scope extends to the end of the loop, including the test and update statements.

```
for (int i = 0; i < n; i++) { // i is in scope for the test and  
update  
    ...  
}  
// i not defined here
```

If you need the value of **i** after the loop, declare the variable outside:

```
int i;  
for (i = 0; !found && i < n; i++) {  
    ...  
}  
// i still available
```

In Java, you cannot have local variables with the same name in overlapping scopes.

```
int i = 0;
while (...) {
    String i = in.next(); // Error to declare another variable i
    ...
}
```

However, if the scopes do not overlap, you can reuse the same variable name:

```
for (int i = 0; i < n / 2; i++) { ... }
for (int i = n / 2; i < n; i++) { ... } // OK to redefine i
```

1.8. Arrays and Array Lists

Arrays are a fundamental programming construct for collecting multiple items of the same type. Java has array types built into the language, and it also supplies an *ArrayList* class for arrays that grow and shrink on demand. The *ArrayList* class is a part of a larger collections framework that is covered in

[Chapter 7](#).

1.8.1. Working with Arrays

For every type, there is a corresponding array type. An array of integers has type `int[]`, an array of `String` objects has type `String[]`, and so on. Here is a variable that can hold an array of strings:

```
String[] names;
```

The variable isn't yet initialized. Let's initialize it with a new array. For that, we need the `new` operator:

```
names = new String[100];
```

Of course, you can combine these two statements:

```
String[] names = new String[100];
```

Now `names` refers to an array with 100 elements, which you can access as `names[0]` ... `names[99]`.



Caution: If you try to access an element that does not exist, such as `names[-1]` or `names[100]`, an `ArrayIndexOutOfBoundsException` occurs.

The length of an array can be obtained as `array.length`. For example, this loop fills the array with empty strings:

```
for (int i = 0; i < names.length; i++) {  
    names[i] = "";  
}
```



Note: It is legal to use the C syntax for declaring an array variable, with the [] following the variable name:

```
int numbers[];
```

However, this syntax is unfortunate since it intertwines the name `numbers` and the type `int[]`. Few Java programmers use it.

1.8.2. Array Construction

When you construct an array with the `new` operator, it is filled with a default value.

- Arrays of numeric type (including `char`) are filled with zeroes.
 - Arrays of `boolean` are filled with `false`.
 - Arrays of objects are filled with `null` references.
-



Caution: Whenever you construct an array of objects, you need to fill it with objects. Consider this declaration:

```
BigInteger[] numbers = new BigInteger[100];
```

At this point, you do not have any `BigInteger` objects yet, just an array of 100 `null` references. You need to replace them with references to `BigInteger` objects:

```
for (int i = 0; i < 100; i++) {  
    numbers[i] = BigInteger.valueOf(i);  
}
```

You can fill an array with values by writing a loop, as you saw in the preceding section. However, sometimes you know the values that you want, and you can just list them inside braces:

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

You don't use the `new` operator, and you don't specify the array length. A trailing comma 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
};
```

Use a similar initialization syntax if you don't want to give the array a name—for example, to assign it to an existing array variable:

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



Note: It is legal to have arrays of length 0. You can construct such an array as `new int[0]` or `new int[] {}`. For example, if a method returns an array of matches, and there weren't any for a particular input, return an array of length 0. Note that this is not the same as `null`: If `a` is an array of length 0, then `a.length` is 0; if `a` is `null`, then `a.length` causes a `NullPointerException`.

1.8.3. Array Lists

When you construct an array, you need to know its length. Once constructed, the length can never change. That is inconvenient in many practical applications. A remedy is to use the `ArrayList` class in the `java.util` package. An `ArrayList` object manages an array internally. When that array becomes too small or is insufficiently utilized, another internal array is automatically created, and the elements are moved into it. This process is invisible to the programmer using the array list.

The syntax for arrays and array lists is completely different. Arrays use a special syntax—the `[]` operator for accessing elements, the `Type[]` syntax for array types, and the `new Type[n]` syntax for constructing arrays. In contrast, array lists are classes, and you use the normal syntax for constructing instances and invoking methods.

However, unlike the classes that you have seen so far, the `ArrayList` class is a *generic class*—a class with a type parameter.

[Chapter 6](#) covers generic classes in detail.

Specify the element type in angle brackets. For example, the type of an array list holding `String` objects is denoted as `ArrayList<String>`.

To declare and initialize a variable of this type, you can use any of the following three statements:

```
ArrayList<String> friends = new ArrayList<String>();  
var friends = new ArrayList<String>();  
ArrayList<String> friends = new ArrayList<>();
```

Note the empty `<>` in the last declaration. The compiler infers the type from the type of the variable. (This shortcut is called the *diamond syntax* because the empty angle brackets have the shape of a diamond.)

There are no construction arguments in this call, but it is still necessary to supply the `()` at the end.

The result is an array list of size 0. You can add elements to the end with the `add` method:

```
friends.add("Peter");  
friends.add("Paul");
```

Unfortunately, there is no initializer syntax for array lists. The best you can do is construct an array list like this:

```
var friends = new ArrayList<>(List.of("Peter", "Paul"));
```

The `List.of` method yields an unmodifiable list of the given elements which you then use to construct an `ArrayList`.

You can add and remove elements anywhere in the `ArrayList`.

```
friends.remove(1);  
friends.add(0, "Paul"); // Adds before index 0
```

To access elements, use method calls, not the [] syntax. The `get` method reads an element, and the `set` method replaces an element with another:

```
String first = friends.get(0);
friends.set(1, "Mary");
```

The `size` method yields the current size of the list. Use the following loop to traverse all elements:

```
for (int i = 0; i < friends.size(); i++) {
    System.out.println(friends.get(i));
}
```

1.8.4. Wrapper Classes for Primitive Types

There is one unfortunate limitation of generic classes: You cannot use primitive types as type parameters. For example, an `ArrayList<int>` is illegal. The remedy is to use a *wrapper class*. For each primitive type, there is a corresponding wrapper class: `Integer`, `Byte`, `Short`, `Long`, `Character`, `Float`, `Double`, and `Boolean`. To collect integers, use an `ArrayList<Integer>`:

```
var numbers = new ArrayList<Integer>();
numbers.add(42);
int first = numbers.get(0);
```

Conversion between primitive types and their corresponding wrapper types is automatic. In the call to `add`, an `Integer` object holding the value 42 was automatically constructed in a process called *autoboxing*.

In the last line of the code segment, the call to `get` returned an `Integer` object. Before assigning to the `int` variable, the object was *unboxed* to yield the `int` value inside.



Caution: Conversion between primitive types and wrappers is almost completely transparent to programmers, with one exception.

The `==` and `!=` operators compare object references, not the contents of objects. A condition `if (numbers.get(i) == numbers.get(j))` does not test whether the numbers at index `i` and `j` are the same. Just like with strings, you need to remember to call the `equals` method with wrapper objects.

1.8.5. The Enhanced for Loop

Very often, you want to visit all elements of an array. For example, here is how you compute the sum of all elements in an array of numbers:

```
int sum = 0;
for (int i = 0; i < numbers.length; i++) {
    sum += numbers[i];
}
```

As this loop is so common, there is a convenient shortcut, called the *enhanced for* loop:

```
int sum = 0;
for (int n : numbers) {
    sum += n;
}
```

The loop variable of the enhanced for loop traverses the elements of the array, not the index values. The variable `n` is assigned to `numbers[0]`, `numbers[1]`, and so on.

You can also use the enhanced for loop with array lists. If `friends` is an array list of strings, you can print them all with the loop

```
for (String name : friends) {
    System.out.println(name);
}
```

1.8.6. Copying Arrays and Array Lists

You can copy one array variable into another, but then both variables will refer to the same array, as shown in [Figure 1.6](#).

```
int[] numbers = primes;  
numbers[5] = 42; // Now primes[5] is also 42
```

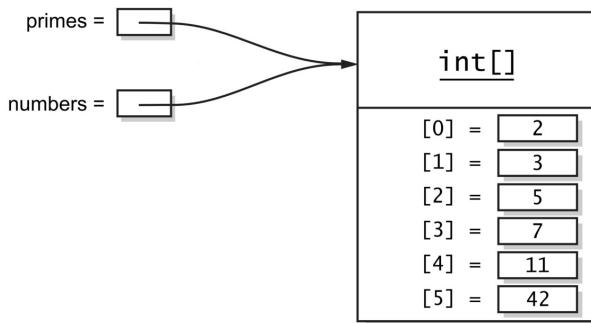


Figure 1.6: Two variables referencing the same array

If you don't want this sharing, you need to make a copy of the array. Use the static `Arrays.copyOf` method.

```
int[] copiedPrimes = Arrays.copyOf(primes, primes.length);
```

This method constructs a new array of the desired length and copies the elements of the original array into it.

Array list references work the same way:

```
ArrayList<String> people = friends;  
people.set(0, "Mary"); // Now friends.get(0) is also "Mary"
```

To copy an array list, construct a new array list from the existing one:

```
var copiedFriends = new ArrayList<>(friends);
```

That constructor can also be used to copy an array into an array list. Wrap the array into an immutable list, using the `List.of` method, and then construct an `ArrayList`:

```
String[] names = ...;
var friends = new ArrayList<>(List.of(names));
```

You can also copy an array list into an array. For depressing reasons of backward compatibility that I will explain in

[Chapter 6](#), you must supply an array of the correct type.

```
String[] names = friends.toArray(new String[0]);
```



Note: There is no easy way to convert between primitive type arrays and the corresponding array lists of wrapper classes. For example, to convert between an `int[]` and an `ArrayList<Integer>`, you need an explicit loop or an `IntStream` (see

[Chapter 8\).](#)

1.8.7. Array Algorithms

The `Arrays` and `Collections` classes provide implementations of common algorithms for arrays and array lists. Here is how to fill an array or an array list:

```
Arrays.fill(numbers, 0); // int[] array  
Collections.fill(friends, ""); // ArrayList<String>
```

To sort an array or array list, use the `sort` method:

```
Arrays.sort(names);  
Collections.sort(friends);
```



Note: For arrays (but not array lists), you can use the `parallelSort` method that distributes the work over multiple processors if the array is large.

The `Arrays.toString` method yields a string representation of an array. This is particularly useful to print an array for debugging.

```
System.out.println(Arrays.toString(primes));  
// Prints [2, 3, 5, 7, 11, 13]
```

Array lists have a `toString` method that yields the same representation.

```
String elements = friends.toString();  
// Sets elements to "[Peter, Paul, Mary]"
```

For printing, you don't even need to call it—the `println` method takes care of that.

```
System.out.println(friends);
    // Calls friends.toString() and prints the result
```

There are a couple of useful algorithms for array lists that have no counterpart for arrays.

```
Collections.reverse(names); // Reverses the elements
Collections.shuffle(names); // Randomly shuffles the elements
```

1.8.8. Command-Line Arguments

As you have already seen, the `main` method of every Java program has a parameter that is a string array:

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

When a program is executed, this parameter is set to the arguments specified on the command line.

For example, consider this program:

```
public class Greeting {
    public static void main(String[] args) {
        for (int i = 0; i < args.length; i++) {
            String arg = args[i];
            if (arg.equals("-h")) arg = "Hello";
            else if (arg.equals("-g")) arg = "Goodbye";
            System.out.println(arg);
        }
    }
}
```

If the program is called as

```
java Greeting -g cruel world
```

then `args[0]` is "-g", `args[1]` is "cruel", and `args[2]` is "world".

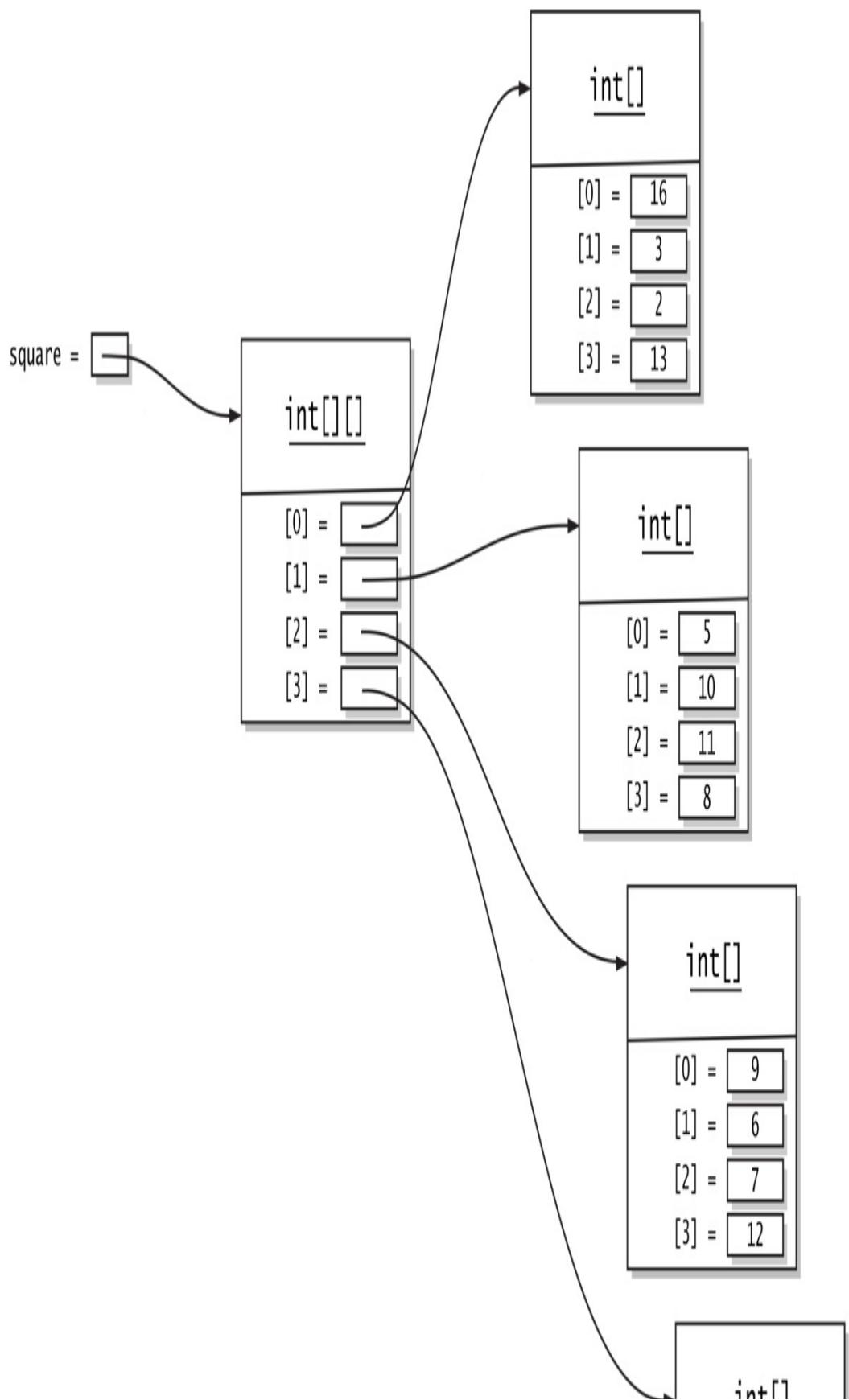
Note that neither "java" nor "Greeting" are passed to the `main` method.

1.8.9. Multidimensional Arrays

Java does not have true multidimensional arrays. They are implemented as arrays of arrays. For example, here is how you declare and implement a two-dimensional array of integers:

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

Technically, this is a one-dimensional array of `int[]` arrays—see [Figure 1.7](#).



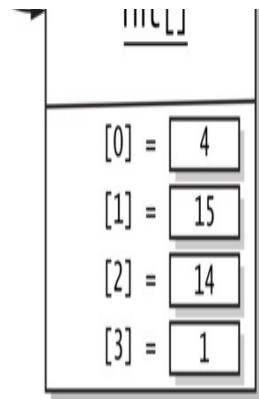


Figure 1.7: A two-dimensional array

To access an element, use two bracket pairs:

```
int element = square[1][2]; // Sets element to 11
```

The first index selects the row array `square[1]`. The second index picks the element from that row.

You can even swap rows:

```
int[] temp = square[0];
square[0] = square[1];
square[1] = temp;
```

If you do not provide an initial value, you must use the `new` operator and specify the number of rows and columns.

```
int[][] square = new int[4][4]; // First rows, then columns
```

Behind the scenes, an array of rows is filled with an array for each row.

There is no requirement that the row arrays have equal length. For example, you can store the Pascal triangle:

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
...
```

First construct an array of n rows:

```
int[][] triangle = new int[n][];
```

Then construct each row in a loop and fill it.

```
for (int i = 0; i < n; i++) {
    triangle[i] = new int[i + 1];
    triangle[i][0] = 1;
    triangle[i][i] = 1;
    for (int j = 1; j < i; j++) {
        triangle[i][j] = triangle[i - 1][j - 1] +
triangle[i - 1][j];
    }
}
```

To traverse a two-dimensional array, you need two loops, one for the rows and one for the columns:

```
for (int r = 0; r < triangle.length; r++) {
    for (int c = 0; c < triangle[r].length; c++) {
        System.out.printf("%4d", triangle[r][c]);
    }
    System.out.println();
}
```

You can also use two enhanced **for** loops:

```
for (int[] row : triangle) {  
    for (int element : row) {  
        System.out.printf("%4d", element);  
    }  
    System.out.println();  
}
```

These loops work for square arrays as well as arrays with varying row lengths.



Tip: To print out a list of the elements of a two-dimensional array for debugging, call

```
System.out.println(Arrays.deepToString(triangle));  
// Prints [[1], [1, 1], [1, 2, 1], [1, 3, 3, 1],  
[1, 4, 6, 4, 1], ...]
```



Note: There are no two-dimensional array lists, but you can declare a variable of type `ArrayList<ArrayList<Integer>>` and build up the rows yourself.

1.9. Functional Decomposition

If your `main` method gets too long, you can decompose your program into multiple classes, as you will see in

[Chapter 2](#). However, for simple programs, you can place your program's code into separate methods in the same class. For reasons that will become clear in

[Chapter 2](#), these methods must be declared with the `static` modifier, just as the `main` method itself.

1.9.1. Declaring and Calling Static Methods

When you declare a method, provide the type of the return value (or `void` if the method doesn't return anything), the method name, and the types and names of the parameters in the *method header*. Then provide the implementation in the *method body*. Use a `return` statement to return the result.

```
public static double average(double x, double y) {  
    double sum = x + y;  
    return sum / 2;  
}
```

Place the method in the same class as the `main` method. It doesn't matter if it's above or below `main`. Then, call it like this:

```
public static void main(String[] args) {  
    double a = ...;  
    double b = ...;  
    double result = average(a, b);  
    ...  
}
```

1.9.2. Array Parameters and Return Values

You can pass arrays into methods. The method simply receives a reference to the array, through which it can modify it. This method swaps two elements in an array:

```
public static void swap(int[] values, int i, int j) {  
    int temp = values[i];  
    values[i] = values[j];
```

```
    values[j] = temp;  
}
```

Methods can return arrays. This method returns an array consisting of the first and last values of a given array (which is not modified):

```
public static int[] firstLast(int[] values) {  
    if (values.length == 0) return new int[0];  
    else return new int[] { values[0], values[values.length  
- 1] };  
}
```

1.9.3. Variable Arguments

Some methods allow the caller to supply a variable number of arguments. You have already seen such a method: `printf`. For example, the calls

```
System.out.printf("%d", n);
```

and

```
System.out.printf("%d %s", n, "widgets");
```

both call the same method, even though one call has two arguments and the other has three.

Let us define an `average` method that works the same way, so we can call `average` with as many arguments as we like—for example, `average(3, 4.5, -5, 0)`. Declare a “varargs” parameter with `...` after the type:

```
public static double average(double... values)
```

The parameter is actually an array of type `double`. When the method is called, an array is created and filled with the arguments. In the method body, you use it as you would any other array.

```
public static double average(double... values) {  
    double sum = 0;  
    for (double v : values) sum += v;  
    return values.length == 0 ? 0 : sum / values.length;  
}
```

Now you can call

```
double avg = average(3, 4.5, -5, 0);
```

If you already have the arguments in an array, you don't have to unpack them. You can pass the array instead of the list of arguments:

```
double[] scores = { 3, 4.5, -5, 0 };  
double avg = average(scores);
```

The variable parameter must be the *last* parameter of the method, but you can have other parameters before it. For example, this method ensures that there is at least one argument:

```
public static double max(double first, double... rest) {  
    double result = first;  
    for (double v : rest) result = Math.max(v, result);  
    return result;  
}
```

Exercises

1. Write a program that reads an integer and prints it in binary, octal, and hexadecimal. Print the reciprocal as a hexadecimal floating-point number.
2. Write a program that reads an integer angle (which may be positive or negative) and normalizes it to a value between 0 and 359 degrees. Try it first with the % operator, then with `floorMod`.
3. Using only the conditional operator, write a program that reads three integers and prints the largest. Repeat with `Math.max`.

4. Write a program that prints the smallest and largest positive `double` values. Hint: Look up `Math.nextUp` in the Java API.
5. What happens when you cast a `double` to an `int` that is larger than the largest possible `int` value? Try it out.
6. Write a program that computes the factorial $n! = 1 \times 2 \times \dots \times n$, using `BigInteger`. Compute the factorial of 1000.
7. Write a program that reads in two integers between 0 and 4294967295, stores them in `int` variables, and computes and displays their unsigned sum, difference, product, quotient, and remainder. Do not convert them to `long` values.
8. Write a program that reads a string and prints all of its nonempty substrings.
9. [Section 1.5.3](#) has an example of two strings `s` and `t` so that `s.equals(t)` but `s != t`. Come up with a different example that doesn't use `substring`.
10. Write a program that produces a random string of letters and digits by generating a random `long` value and printing it in base 36.
11. Write a program that reads a line of text and prints all characters that are not ASCII, together with their Unicode values.
12. Write a `switch` expression that, when given a string with a compass direction "N", "S", "E", or "W", yields an array of x- and y-offsets. For example, "W" should yield `new int[] { -1, 0 }`.
13. Write a `switch` statement that, when given a string with a compass direction "N", "S", "E", or "W", adjusts the variables `x` and `y`. For example, "W" should decrement `x` by 1.
14. Can you use a `break` in a `switch` statement without fall-through? In a `switch` expression? Why or why not?
15. Come up with a useful scenario where the fall-through behavior is beneficial for a `switch` expression or statement. Most web searches will produce examples that make sense for C or C++, where execution falls from `case A` with no action to `case B`. In Java, that is not significant because you can use `case A, B`.
16. A “Quine” is a program that, without reading any input or file, prints its own source code. Write such a program in Java, using a text block.

17. The Java Development Kit includes a file `src.zip` with the source code of the Java library. Unzip and, with your favorite text search tool, find usages of the labeled `break` and `continue` sequences. Take one and rewrite it without a labeled statement.
18. Write a program that prints a lottery combination, picking six distinct numbers between 1 and 49. To pick six distinct numbers, start with an array list filled with `1...49`. Pick a random index and remove the element. Repeat six times. Print the result in sorted order.
19. Write a program that reads a two-dimensional array of integers and determines whether it is a magic square (that is, whether the sum of all rows, all columns, and the diagonals is the same). Accept lines of input that you break up into individual integers, and stop when the user enters a blank line. For example, with the input

```
16 3 2 13  
5 10 11 8  
9 6 7 12  
4 15 14 1  
(Blank line)
```

- your program should respond affirmatively.
20. Write a program that stores Pascal's triangle up to a given n in an `ArrayList<ArrayList<Integer>>`.
 21. Improve the `average` method so that it is called with at least one argument.

Chapter 2 ■ Object-Oriented Programming

In object-oriented programming, work is carried out by collaborating objects whose behavior is defined by the classes to which they belong. Java was one of the first mainstream programming languages to fully embrace object-oriented programming. As you have already seen, in Java every method is declared in a class and, except for a few primitive types, every value is an object. In this chapter, you will learn how to implement your own classes and methods.

The key points of this chapter are:

1. Mutator methods change the state of an object; accessor methods don't.
2. In Java, variables don't hold objects; they hold references to objects.
3. Instance variables and method implementations are declared inside the class declaration.
4. An instance method is invoked on an object, which is accessible through the `this` reference.
5. A constructor has the same name as the class. A class can have multiple (overloaded) constructors.
6. Static variables don't belong to any objects. Static methods are not invoked on objects.
7. A **record** is an immutable class with public accessors for all instance variables.
8. Classes are organized into packages. Use import declarations so that you don't have to use the package name in your programs.
9. Classes can be nested in other classes.
10. An inner class is a non-static nested class. Its instances have a reference to the object of the enclosing class that constructed it.
11. The `javadoc` utility processes source files, producing HTML files with declarations and programmer-supplied comments.

2.1. Working with Objects

In ancient times, before objects were invented, you wrote programs by calling *functions*. When you call a function, it returns a result that you use without worrying how it was computed. Functions have an important benefit: They allow work to be shared. You can call a function that someone else wrote without having to know how it does its task.

Objects add another dimension. Each object can have its own *state*. The state affects the results that you get from calling a method. For example, if `in` is a `Scanner` object and you call `in.next()`, the object remembers what was read before and gives you the next input token.

When you use objects that someone else implemented and invoke methods on them, you do not need to know what goes on under the hood. This principle, called *encapsulation*, is a key concept of object-oriented programming.

At some point, you may want to make your work available for other programmers by providing them with objects they can use. In Java, you provide a *class*—a mechanism for creating and using objects with the same behavior.

Consider a common task: manipulation of calendar dates. Calendars are somewhat messy, with varying month lengths and leap years, not to mention leap seconds. It makes sense to have experts who figure out those messy details and who provide implementations that other programmers can use. In this situation, objects arise naturally. A date is an object whose methods can provide information such as “on what weekday does this date fall” and “what date is tomorrow.”

In Java, experts who understand date computations provided classes for dates and other date-related concepts such as weekdays. If you want to do computations with dates, use one of those classes to create date objects and invoke methods on them, such as a method that yields the weekday or the next date.

Few of us want to ponder the details of date arithmetic, but you are probably an expert in some other area. To enable other programmers to leverage your knowledge, you can provide them with classes. And even if you are not enabling other programmers, you will find it useful in your own work to use classes so that your programs are structured in a coherent way.

Before learning how to declare your own classes, let us run through a nontrivial example of using objects.

The Unix program `cal` prints a calendar for a given month and year, in a format similar to the following:

Mon	Tue	Wed	Thu	Fri	Sat	Sun
					1	
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30						

How can you implement such a program? With the standard Java library, you use the `LocalDate` class to express a date at some unspecified location. We need an object of that class representing the first of the month. Here is how you get one:

```
LocalDate date = LocalDate.of(year, month, 1);
```

To advance the date, you call `date.plusDays(1)`. The result is a newly constructed `LocalDate` object that is one day further. In our application, we simply reassign the result to the `date` variable:

```
date = date.plusDays(1);
```

You apply methods to obtain information about a date, such as the month on which it falls. We need that information so that we can keep printing while we are still in the same month.

```
while (date.getMonthValue() == month) {  
    System.out.printf("%4d", date.getDayOfMonth());  
    date = date.plusDays(1);  
    ...  
}
```

Another method yields the weekday on which a date falls.

```
DayOfWeek weekday = date.getDayOfWeek();
```

You get back an object of another class `DayOfWeek`. In order to compute the indentation of the first day of the month in the calendar, we need know the numerical value of the weekday. There is a method for that:

```
int value = weekday.getValue();  
for (int i = 1; i < value; i++)  
    System.out.print("    ");
```

The `getValue` method follows the international convention where the weekend comes at the end of the week, returning 1 for Monday, 2 for Tuesday, and so on. Sunday has value 7.



Note: You can *chain* method calls, like this:

```
int value = date.getDayOfWeek().getValue();
```

The first method call is applied to the `date` object, and it returns a `DayOfWeek` object. The `getValue` method is then invoked on the returned object.

You will find the complete program in the book's companion code. It was easy to solve the problem of printing a calendar because the designers of the `LocalDate` class provided us with a useful set of methods. In this chapter, you will learn how to implement methods for your own classes.

2.1.1. Accessor and Mutator Methods

Consider again the method call `date.plusDays(1)`. There are two ways in which the designers of the `LocalDate` class could have implemented the `plusDays` method. They could make it change the state of the `date` object and return no result. Or they could leave `date` unchanged and return a newly constructed `LocalDate` object. As you can see, they chose to do the latter.

We say that a method is a *mutator* if it changes the object on which it was invoked. It is an *accessor* if it leaves the object unchanged. The `plusDays` method of the `LocalDate` class is an accessor.

In fact, *all* methods of the `LocalDate` class are accessors. This situation is increasingly common because mutation can be risky, particularly if two computations mutate an object simultaneously. Nowadays, most computers have multiple processing units, and safe concurrent access is a serious issue. One way to address this issue is to make objects *immutable* by providing only accessor methods.

Still, there are many situations where mutation is desirable. The `add` method of the `ArrayList` class is an example of a mutator. After calling `add`, the array list object is changed.

```
var friends = new ArrayList<String>();
    // friends is empty
friends.add("Peter");
    // friends has size 1
```

2.1.2. Object References

In some programming languages (such as C++), a variable can actually hold the object—that is, the bits that make up the object’s state. In Java, that is not the case. A variable can only hold a *reference* to an object. The actual object is elsewhere, and the reference is some implementation-dependent way of locating the object (see [Figure 2.1](#)).



Note: References behave like pointers in C and C++, except that they are perfectly safe. In C and C++, you can modify pointers and use them to overwrite arbitrary memory locations. With a Java reference, you can only access a specific object.

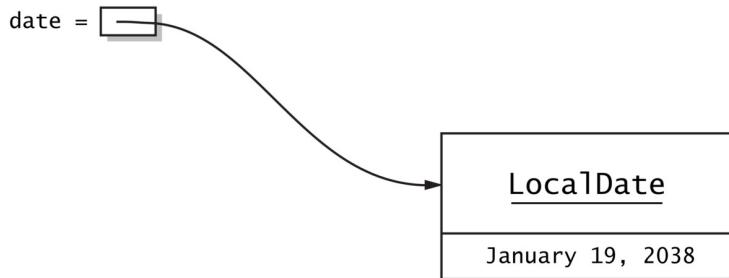


Figure 2.1: An object reference

When you assign a variable holding an object reference to another, you have two references to the same object.

```
ArrayList<String> people = friends;  
// Now people and friends refer to the same object
```

If you mutate the shared object, the mutation is observable through both references. Consider the call

```
people.add("Paul");
```

Now the array list `people` has size 2, and so does `friends` (see [Figure 2.2](#)). (Of course, it isn't technically true that `people` or `friends` "have" size 2. After all, `people` and `friends` are not objects. They are references to an object, namely an array list with size 2.)

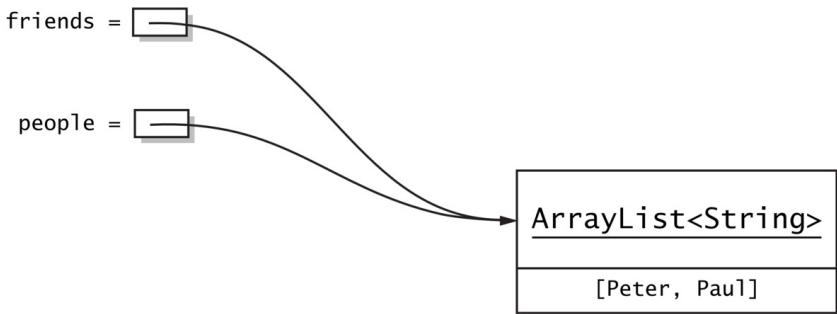


Figure 2.2: Two references to the same object

Most of the time, this sharing of objects is efficient and convenient, but you have to be aware that it is possible to mutate a shared object through any of its references.

However, if a class has no mutator methods (such as `String` or `LocalDate`), you don't have to worry. Since nobody can change such an object, you can freely give out references to it.

It is possible for an object variable to refer to no object at all, by setting it to the special value `null`.

```
LocalDate date = null; // Now date doesn't refer to any object
```

This can be useful if you don't yet have an object for `date` to refer to, or if you want to indicate a special situation, such as an unknown date.



Caution: Null values can be dangerous when they are not expected. Invoking a method on `null` causes a `NullPointerException` (which should really have been called a `NullReferenceException`). For that reason, it is not recommended to use `null` for optional values. Use the `Optional` type instead (see

[Chapter 8\).](#)

Finally, have another look at the assignments

```
date = LocalDate.of(year, month, 1);  
date = date.plusDays(1);
```

After the first assignment, `date` refers to the first day of the month. The call to `plusDays` yields a new `LocalDate` object, and after the second assignment, the `date` variable refers to the new object. What happens to the first one?

There is no reference to the first object, so it is no longer needed.

Eventually, the *garbage collector* will recycle the memory and make it available for reuse. In Java, this process is completely automatic, and programmers never need to worry about deallocated memory.

2.2. Implementing Classes

Now let us turn to implementing our own classes. To show the various language rules, I use the classic example of an `Employee` class. An employee has a name and a salary. In this example, the name can't change, but ever so often an employee can get a well-deserved raise.

2.2.1. Instance Variables

From the description of employee objects, you can see that the state of such an object is described by two values: name and salary. In Java, you use *instance variables* to describe the state of an object. They are declared in a class like this:

```
public class Employee {  
    private String name;  
    private double salary;
```

```
 }
```

```
...
```

That means that every instance of the `Employee` class has these two variables.

In Java, instance variables are usually declared as `private`. That means that only methods of the same class can access them. There are a couple of reasons why this protection is desirable: You control which parts of your program can modify the variables, and you can decide at any point to change the internal representation. For example, you might store the employees in a database and only leave the primary key in the object. As long as you reimplement the methods so they work the same as before, the users of your class won't care.

2.2.2. Method Headers

Now let's turn to implementing the methods of the `Employee` class. When you declare a method, you provide its name, the types and names of its parameters, and the return type, like this:

```
public void raiseSalary(double byPercent)
```

This method has a parameter of type `double` and doesn't return any value, as indicated by the return type `void`.

The `getName` method has a different signature:

```
public String getName()
```

The method has no parameters and returns a `String`.



Note: Most methods are declared as `public`, which means anyone can call such a method. Sometimes, a helper method is declared as `private`, which restricts it to being used only in other methods of the same class. You should do that for methods that are not relevant

to class users, particularly if they depend on implementation details. You can safely change or remove private methods if the implementation changes.

2.2.3. Method Bodies

Following the method header, you provide the body:

```
public void raiseSalary(double byPercent) {  
    double raise = salary * byPercent / 100;  
    salary += raise;  
}
```

Use the `return` keyword if the method yields a value:

```
public String getName() {  
    return name;  
}
```

Place the method declarations inside the class declaration:

```
public class Employee {  
    private String name;  
    private double salary;  
  
    public void raiseSalary(double byPercent) {  
        double raise = salary * byPercent / 100;  
        salary += raise;  
    }  
  
    public String getName() {  
        return name;  
    }  
    ...  
}
```

2.2.4. Instance Method Invocations

Consider this example of a method call on an object `fred` of class `Employee`:

```
fred.raiseSalary(5);
```

In this call, the argument 5 is used to initialize the parameter variable `byPercent`, equivalent to the assignment

```
double byPercent = 5;
```

Then the following actions occur:

```
double raise = fred.salary * byPercent / 100;  
fred.salary += raise;
```

Note that the `salary` instance variable is applied to the instance on which the method is invoked.

Unlike the static methods that you have seen at the end of the preceding chapter, a method such as `raiseSalary` operates on an instance of a class. Therefore, such a method is called an *instance method*. In Java, all methods that are not declared as `static` are instance methods.

For the `raiseSalary` method to work, it must receive two values: a reference to the object on which the method is invoked, and the argument of the call. Technically, both of these are parameters of the method, but in Java, as in other object-oriented languages, the first one takes on a special role. It is sometimes called the *receiver* of the method call.

2.2.5. The `this` Reference

When a method is called on an object, `this` is set to that object. If you like, you can use the `this` reference in the implementation:

```
public void raiseSalary(double byPercent) {  
    double raise = this.salary * byPercent / 100;  
    this.salary += raise;  
}
```

Some programmers prefer that style because it clearly distinguishes between local and instance variables—it is now obvious that `raise` is a local variable and `salary` is an instance variable.

It is very common to use the `this` reference when you don't want to come up with different names for parameter variables. For example,

```
public void setSalary(double salary) {  
    this.salary = salary;  
}
```

When an instance variable and a local variable have the same name, the unqualified name (such as `salary`) denotes the local variable, and `this.salary` is the instance variable.



Note: In some programming languages, instance variables are decorated in some way, for example `_name` and `_salary`. This is legal in Java but is not commonly done.



Note: If you like, you can even declare `this` as a parameter of any method (but not a constructor):

```
public void setSalary(Employee this, double salary) {  
    this.salary = salary;  
}
```

This does not change the way the method is implemented or invoked. The notation exists so that you can annotate the receiver of the method—see

[Chapter 11.](#)

2.2.6. Call by Value

When you pass an object to a method, the method obtains a copy of the object reference. Through this reference, it can access or mutate the parameter object. For example:

```
public class EvilManager {  
    private Random generator;  
    ...  
    public void giveRandomRaise(Employee e) {  
        double percentage = 10 * generator.nextGaussian();  
        e.raiseSalary(percentage);  
    }  
}
```

Consider the call

```
boss.giveRandomRaise(fred);
```

The reference `fred` is copied into the parameter variable `e` (see [Figure 2.3](#)). The method mutates the object that is shared by the two references.

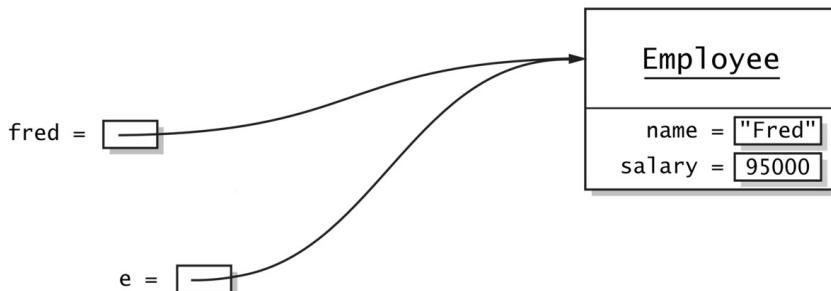


Figure 2.3: A parameter variable holding a copy of an object reference

In Java, you can never write a method that updates primitive type variables. A method that tries to increase a `double` value won't work:

```
public void increaseRandomly(double x) { // Won't update a
    passed-in variable
    double amount = x * generator.nextDouble();
    x += amount;
}
```

If you call

```
boss.increaseRandomly(sales);
```

then `sales` is copied into `x`. Then `x` is increased, but that doesn't change `sales`. The parameter variable then goes out of scope, and the increase leaves no useful effect.

For the same reason, it is not possible to write a method that changes an object reference to something different. For example, this method does not work as presumably intended:

```
public class EvilManager {
    ...
    public void replaceWithZombie(Employee e) {
        e = new Employee("", 0);
    }
}
```

In the call

```
boss.replaceWithZombie(fred);
```

the reference `fred` is copied into the variable `e` which is then set to a different reference. When the method exits, `e` goes out of scope. At no point was `fred` changed.



Note: Some people say that Java uses “call by reference” for objects. As you can see from the second example, that is not true. In a language that supports call by reference, a method can replace the contents of variables passed to it. In Java, all parameters—object references as well as primitive type values—are passed by value.

2.3. Object Construction

One step remains to complete the `Employee` class: We need to provide a constructor, as detailed in the following sections.

2.3.1. Implementing Constructors

Declaring a constructor is similar to declaring a method. However, the name of the constructor is the same as the class name, and there is *no return type*.

```
public Employee(String name, double salary) {  
    this.name = name;  
    this.salary = salary;  
}
```



Note: This constructor is public. It can also be useful to have private constructors. For example, the `LocalDate` class has no public constructors. Instead, users of the class obtain objects from “factory methods” such as `now` and `of`. These methods call a private constructor.



Caution: If you accidentally specify a return type, such as

```
public void Employee(String name, double salary)
```

then you declare a method named `Employee`, not a constructor!

A constructor executes when you use the `new` operator. For example, the expression

```
new Employee("James Bond", 500000)
```

allocates an object of the `Employee` class and invokes the constructor body, which sets the instance variables to the arguments supplied in the constructor.

The `new` operator returns a reference to the constructed object. You will normally want to save that reference in a variable:

```
var james = new Employee("James Bond", 500000);
```

or pass it to a method:

```
var staff = new ArrayList<Employee>();
staff.add(new Employee("James Bond", 500000));
```

2.3.2. Overloading

You can supply more than one version of a constructor. For example, if you want to make it easy to model nameless worker bees, supply a second constructor that only accepts a salary.

```
public Employee(double salary) {
    this.name = "";
    this.salary = salary;
}
```

Now the `Employee` class has two constructors. Which one is called depends on the arguments.

```
var james = new Employee("James Bond", 500000);
    // calls Employee(String, double) constructor
var anonymous = new Employee(40000);
    // calls Employee(double) constructor
```

In this case, we say that the constructor is *overloaded*.



Note: A method is overloaded if there are multiple versions with the same name but different parameters. For example, there are overloaded versions of the `println` method with parameters `int`, `double`, `String`, and so on. Since you have no choice how to name a constructor, it is common to overload constructors.

2.3.3. Calling One Constructor from Another

When there are multiple constructors, they usually have some work in common, and it is best not to duplicate that code. It is often possible to put that common initialization into one constructor.

You can call one constructor from another, but only as the *first statement* of the constructor body. Somewhat surprisingly, you don't use the name of the constructor for the call but the keyword `this`:

```
public Employee(double salary) {
    this("", salary); // Calls Employee(String, double)
    // Other statements can follow
}
```



Note: Here, `this` is *not* a reference to the object that is being constructed. Instead, it is a special syntax that is only used for invoking another constructor of the same class.

2.3.4. Default Initialization

If you don't set an instance variable explicitly in a constructor, it is automatically set to a default value: numbers to `0`, boolean values to `false`, and object references to `null`.

For example, you could supply a constructor for unpaid interns.

```
public Employee(String name) {  
    // salary automatically set to zero  
    this.name = name;  
}
```



Note: In this regard, instance variables are very different from local variables. Recall that you must always explicitly initialize local variables.

For numbers, the initialization with zero is often convenient. But for object references, it is a common source of errors. Suppose we didn't set the `name` variable to the empty string in the `Employee(double)` constructor:

```
public Employee(double salary) {  
    // name automatically set to null  
    this.salary = salary;  
}
```

If anyone called the `getName` method, they would get a `null` reference that they probably don't expect. A condition such as

```
if (e.getName().equals("James Bond"))
```

would then cause a null pointer exception.

2.3.5. Instance Variable Initialization

You can specify an initial value for any instance variables, like this:

```
public class Employee {  
    private String name = "";  
    ...  
}
```

This initialization occurs after the object has been allocated and before a constructor runs. Therefore, the initial value is present in all constructors. Of course, some of them may choose to overwrite it.

In addition to initializing an instance variable when you declare it, you can include arbitrary *initialization blocks* in the class declaration.

```
public class Employee() {  
    private String name = "";  
    private int id;  
    private double salary;  
  
    { // An initialization block  
        var generator = new Random();  
        id = 1 + generator.nextInt(1_000_000);  
    }  
  
    public Employee(String name, double salary) {  
        ...  
    }  
}
```

 **Note:** This is not a commonly used feature. Most programmers place lengthy initialization code into a helper method and invoke that method from the constructors.

Instance variable initializations and initialization blocks are executed in the order in which they appear in the class declaration, and before the body of

the constructor.

2.3.6. Final Instance Variables

You can declare an instance variable as final. Such a variable must be initialized by the end of every constructor. Afterwards, the variable may not be modified again. For example, the `name` variable of the `Employee` class may be declared as `final` because it never changes after the object is constructed—there is no `setName` method.

```
public class Employee {  
    private final String name;  
    ...  
}
```



Note: When used with a reference to a mutable object, the `final` modifier merely states that the reference will never change. It is perfectly legal to mutate the object.

```
public class Person {  
    private final ArrayList<Person> friends = new  
    ArrayList<>();  
    // OK to add elements to this array list  
    ...  
}
```

Methods may mutate the array list to which `friends` refers, but they can never replace it with another. In particular, it can never become `null`.

2.3.7. The Constructor with No Arguments

Many classes contain a constructor with no arguments that creates an object whose state is set to an appropriate default. For example, here is a

constructor with no arguments for the `Employee` class:

```
public Employee() {  
    name = "";  
    salary = 0;  
}
```

Just like an indigent defendant is provided with a public defender, a class with no constructors is automatically given a constructor with no arguments that does nothing at all. All instance variables stay at their default values (zero, `false`, or `null`) unless they have been explicitly initialized.

Thus, every class has at least one constructor.



Note: If a class already has a constructor, it does *not* automatically get another constructor with no arguments. If you supply a constructor and also want a no-argument constructor, you have to write it yourself.



Note: In the preceding sections, you saw what happens when an object is constructed. In some programming languages, notably C++, it is common to specify what happens when an object is destroyed. Java has a deprecated mechanism for “finalizing” an object when it is reclaimed by the garbage collector. But this happens at unpredictable times, so you should not use it. However, as you will see in

[Chapter 5](#), there is a mechanism for closing resources such as files.

2.4. Records

Sometimes, data is just data, and the data hiding that object-oriented programming provides gets in the way. Consider a class `Point` that describes a point in the plane, with x - and y -coordinates.

Sure, you can provide a class:

```
class Point {  
    private final double x;  
    private final double y;  
    public Point(double x, double y) { this.x = x; this.y =  
        y; }  
    public getX() { return x; }  
    public getY() { return y; }  
    public String toString() { return "Point[x=%d,  
y=%d]".formatted(x, y); }  
    // More methods ...  
}
```

But does it really buy us anything to hide x and y , and then make the values available through the getter methods?

Would we ever want to change the implementation of a `Point`? Sure, there are polar coordinates, but you would not use them with a graphics API. In practice, a point in the plane is completely described by its x - and y -coordinates.

To define classes such as this one more concisely, you can use a special form of classes called “records.”

2.4.1. The Record Concept

A record is a special form of a class whose state is immutable and readable by the public. To declare a record, provide the name and the instance variables that hold the object state. Here is a typical example:

```
record Point(double x, double y) { }
```

The result is a class `Point` with instance variables:

```
private final double x;  
private final double y;
```

The class has a constructor

```
Point(double x, double y)
```

and accessor methods

```
public double x()  
public double y()
```

Note that the accessors are called `x` and `y`, not `getX` and `getY`. (It is legal in Java to have an instance variable and a method with the same name.)

```
var p = new Point(3, 4);  
double slope = p.y() / p.x();
```



Note: If you have an instance of a record, you can use *pattern matching* to obtain the record components. For example:

```
double slope = switch (p) {  
    case Point(var a, var b) -> b / a;  
};
```

Pattern matching is covered in

[Chapter 4.](#)

In addition to the variable accessor methods, every record has three methods defined automatically: `toString`, `equals`, and `hashCode`. You will learn more about these methods in the next chapter.

You can add your own methods to a record:

```
record Point(double x, double y) {  
    public double distanceFromOrigin() {  
        return Math.hypot(x, y);  
    }  
}
```



Caution: You can define your own versions of the automatically provided methods, as long as they have the same parameter and return types. For example, this definition is legal:

```
record Point(double x, double y) {  
    public double x() { return y; } // BAD  
}
```

But it is surely not a good idea.

You cannot declare instance variables in the body of a record:

```
record Point(double x, double y) {  
    private double r; // Cannot declare instance variables here  
    ...  
}
```



Caution: The instance variables of a record are automatically `final`. However, they may be references to mutable objects:

```
record Point(double[] coords) { ... }
```

Then record instances are mutable:

```
Point p = ...;
p.coords[0] = 10;
```

If you intend record instances to be immutable, don't use mutable types for instance variables.

A record, like any class, can have static fields and methods:

```
record Point(double x, double y) {
    public static Point ORIGIN = new Point(0, 0);
    public static double distance(Point p, Point q) {
        return Math.hypot(p.x - q.x, p.y - q.y);
    }
    ...
}
```

2.4.2. Constructors: Canonical, Custom, and Compact

The automatically defined constructor that sets all instance variables is called the *canonical constructor*.

To validate or normalize the parameters for the canonical constructor, you use a *compact* form. You don't specify the parameter list:

```
record Range(int from, int to) {
    public Range { // Compact form, validating parameters
        if (from > to) throw new IllegalArgumentException();
    }
}
```

or

```

record Range(int from, int to) {
    public Range { // Compact form, normalizing parameters
        if (from > to) { // Swap the bounds
            int temp = from;
            from = to;
            to = temp;
        }
    }
}

```

The body of the compact form is the “prelude” to the canonical constructor. It merely modifies the parameter variables `from` and `to` before they are assigned to the instance fields `this.from` and `this.to`. You cannot read or modify the instance variables in the body of a compact constructor.

You can define additional *custom constructors*. The first statement of such a constructor must call another constructor, so that ultimately the canonical constructor is invoked. Here is an example:

```

record Point(double x, double y) {
    public Point() { this(0, 0); }
}

```

This record has two constructors: the canonical constructor and a no-argument constructor yielding the origin.

If the canonical constructor needs to do additional work, you can provide your own implementation. But this is very uncommon. For example, the following implementation is valid, but, as you have seen, the same effect can be achieved more easily with a compact constructor.

```

record Range(int from, int to) {
    public Range(int from, int to) { // legal, but more work
        than a compact constructor
        if (from <= to) {
            this.from = from;

```

```

        this.to = to;
    } else {
        this.from = to;
        this.to = from;
    }
}

```

2.5. Static Variables and Methods

In all sample programs that you have seen, the main method is tagged with the `static` modifier. In the following sections, you will learn what this modifier means.

2.5.1. Static Variables

If you declare a variable in a class as `static`, then there is only one such variable per class. In contrast, each object has its own copy of an instance variable. For example, suppose we want to give each employee a distinct ID number. Then we can share the last ID that was given out.

```

public class Employee {
    private static int lastId = 0;
    private int id;
    ...
    public Employee() {
        lastId++;
        id = lastId;
    }
}

```

Every `Employee` object has its own instance variable `id`, but there is only one `lastId` variable that belongs to the class, not to any particular instance of the class.

When a new `Employee` object is constructed, the shared `lastId` variable is incremented and the `id` instance variable is set to that value. Thus, every employee gets a distinct `id` value.



Caution: This code will not work if `Employee` objects can be constructed concurrently in multiple threads.

[Chapter 10](#) shows how to remedy that problem.



Note: You may wonder why a variable that belongs to the class, and not to individual instances, is named “static.” The term is a meaningless holdover from C++ which borrowed the keyword from an unrelated use in the C language instead of coming up with something more appropriate. A more descriptive term is “class variable.”

2.5.2. Static Constants

Mutable static variables are rare, but static constants (that is, `static final` variables) are quite common. For example, the `Math` class declares a static constant:

```
public class Math {  
    ...  
    public static final double PI = 3.14159265358979323846;  
    ...  
}
```

You can access this constant in your programs as `Math.PI`.

Without the `static` keyword, `PI` would have been an instance variable of the `Math` class. That is, you would need an object of the class to access `PI`, and every `Math` object would have its own copy of `PI`.

Here is an example of a static `final` variable that is an object, not a number. It is both wasteful and insecure to construct a new random number generator each time you want a random number. You are better off sharing a single generator among all instances of a class.

```
public class Employee {  
    private static final Random generator = new Random();  
    private int id;  
    ...  
    public Employee() {  
        id = 1 + generator.nextInt(1_000_000);  
    }  
}
```

Another example of a static constant is `System.out`. It is declared in the `System` class like this:

```
public class System {  
    public static final PrintStream out;  
    ...  
}
```



Caution: Even though `out` is declared as `final` in the `System` class, there is a method `setOut` that sets `System.out` to a different stream. This method is a “native” method, not implemented in Java, which can bypass the access control mechanisms of the Java language. This is a very unusual situation from the early days of Java, and not something you are likely to encounter elsewhere.

2.5.3. Static Initialization Blocks

In the preceding sections, static variables were initialized as they were declared. Sometimes, you need to do additional initialization work. You can put it into a *static initialization block*.

```
public class CreditCardForm {  
    private static final ArrayList<Integer> expirationYear  
    = new ArrayList<>();  
    static {
```

```
// Add this and the next twenty years to the array list
int year = LocalDate.now().getYear();
for (int i = year; i <= year + 20; i++) {
    expirationYear.add(i);
}
...
}
```

Static initialization occurs when the class is first loaded. Like instance variables, static variables are `0`, `false`, or `null` unless you explicitly set them to another value. All static variable initializations and static initialization blocks are executed in the order in which they occur in the class declaration.

2.5.4. Static Methods

Static methods are methods that do not operate on objects. For example, the `pow` method of the `Math` class is a static method. The expression

```
Math.pow(x, a)
```

computes the power x^a . It does not use any `Math` object to carry out its task.

As you have already seen in

[Chapter 1](#), a static method is declared with the `static` modifier:

```
public class Math {  
    public static double pow(double base, double exponent)  
{  
    ...  
}  
}
```

Why not make `pow` into an instance method? It can't be an instance method of `double` since, in Java, primitive types are not classes. One could make it an instance method of the `Math` class, but then you would need to construct a `Math` object in order to call it.

Another common reason for static methods is to provide added functionality to classes that you don't own. For example, wouldn't it be nice to have a method that yields a random integer in a given range? You can't add a method to the `Random` class in the standard library. But you can provide a static method:

```
public class RandomNumbers {  
    public static int nextInt(Random generator, int low,  
    int high) {  
        return low + generator.nextInt(high - low + 1);  
    }  
}
```

Call this method as

```
int dieToss = RandomNumbers.nextInt(gen, 1, 6);
```



Note: It is legal to invoke a static method on an object. For example, instead of calling `LocalDate.now()` to get today's date, you can call `date.now()` on an object `date` of the `LocalDate` class. But that does not make a lot of sense. The `now` method doesn't look

at the `date` object to compute the result. Most Java programmers would consider this poor style.

Since static methods don't operate on objects, you cannot access instance variables from a static method. However, static methods can access the static variables in their class. For example, in the `RandomNumbers.nextInt` method, we can make the random number generator into a static variable:

```
public class RandomNumbers {  
    private static final Random generator = new Random();  
    public static int nextInt(int low, int high) {  
        return low + generator.nextInt(high - low + 1);  
        // OK to access the static generator variable  
    }  
}
```

2.5.5. Factory Methods

A common use for static methods is a *factory method*, a static method that returns new instances of the class. For example, the `NumberFormat` class uses factory methods that yield formatter objects for various styles.

```
NumberFormat currencyFormatter =  
NumberFormat.getCurrencyInstance();  
NumberFormat percentFormatter =  
NumberFormat.getPercentInstance();  
double x = 0.1;  
System.out.println(currencyFormatter.format(x)); // Prints  
$0.10  
System.out.println(percentFormatter.format(x)); // Prints  
10%
```

Why not use a constructor instead? The only way to distinguish two constructors is by their parameter types. You cannot have two constructors with no arguments.

Moreover, a constructor `new NumberFormat(...)` yields a `NumberFormat`. A factory method can return an object of a subclass. In fact, these factory methods return instances of the `DecimalFormat` class. (See

[Chapter 4](#) for more information about subclasses.)

A factory method can also return a shared object, instead of unnecessarily constructing new ones. For example, the call `Collections.emptyList()` returns a shared immutable empty list.

2.6. Packages

In Java, you place related classes into a package. Packages are convenient for organizing your work and for separating it from code libraries provided by others. As you have seen, the standard Java library is distributed over a number of packages, including `java.lang`, `java.util`, `java.math`, and so on.

One reason for using packages is to guarantee the uniqueness of class names. Suppose two programmers come up with the bright idea of supplying an `Element` class. (In fact, at least five developers had that bright idea in the Java API alone.) As long as all of them place their classes into different packages, there is no conflict.

In the following sections, you will learn how to work with packages.

2.6.1. Package Declarations

A package name is a dot-separated list of identifiers such as `java.util.regex`.

To guarantee unique package names, it is a good idea to use an Internet domain name (which is known to be unique) written in reverse. For example, I own the domain name `horstmann.com`. For my projects, I use package names such as `com.horstmann.corejava`. A major exception to this rule is the standard Java library whose package names start with `java` or `javax`.



Note: In Java, packages do not nest. For example, the packages `java.util` and `java.util.regex` have nothing to do with each other. Each is its own independent collection of classes.

To place a class in a package, you add a `package` statement as the first statement of the source file:

```
package com.horstmann.corejava;

public class Employee {
    ...
}
```

Now the `Employee` class is in the `com.horstmann.corejava` package, and its *fully qualified name* is `com.horstmann.corejava.Employee`.

There is also a *default package* with no name that you can use for simple programs. To add a class to the default package, don't provide a `package` statement. However, the use of the default package is not recommended.

When class files are read from a file system, the path name needs to match the package name. For example, the file `Employee.class` must be in a subdirectory `com/horstmann/corejava`.

If you arrange the source files in the same way and compile from the directory that contains the initial package names, then the class files are automatically put in the correct place. Suppose the `EmployeeDemo` class makes use of `Employee` objects, and you compile it as

```
javac com/horstmann/corejava/EmployeeDemo.java
```

The compiler generates class files `com/horstmann/corejava/EmployeeDemo.class` and `com/horstmann/corejava/Employee.class`. You run the program by specifying the fully qualified class name:

```
java com.horstmann.corejava.EmployeeDemo
```



Caution: If a source file is not in a subdirectory that matches its package name, the `javac` compiler will not complain and will generate a class file, but you will need to put the class file in the right place. This can be quite confusing—see [Exercise 26](#).



Tip: It is a good idea to run `javac` with the `-d` option. Then the class files are generated in a separate directory, without cluttering up the source tree, and they have the correct subdirectory structure.

2.6.2. The jar Command

Instead of storing class files in the file system, you can place them into one or more archive files called JAR files. You can make such an archive with the `jar` utility that is a part of the JDK. Its command-line options are similar to those of the Unix `tar` program.

```
jar --create --verbose --file library.jar  
com/mycompany/*.class
```

or, with short options,

```
jar -c -v -f library.jar com/mycompany/*.class
```

or, with tar-style options,

```
jar cvf library.jar com/mycompany/*.class
```

JAR files are commonly used to package libraries.



Tip: You can use JAR files to package a program, not just a library. Generate the JAR file with

```
jar -c -f program.jar -e com.mycompany.MainClass  
com/mycompany/*.class
```

Then run the program as

```
java -jar program.jar
```



Caution: The options of commands in the Java development kit have traditionally used single dashes followed by multi-letter option names, such as `java -jar`. The exception was the `jar` command, which followed the classic option format of the `tar` command without dashes. Now, Java is moving towards the more common option format where multi-letter option names are preceded by double dashes, such as `--create`, with single-letter shortcuts for common options, such as `-c`.

This has created a muddle that will hopefully get cleaned up over time. Right now, `java -jar` works as always, but `java --jar` doesn't. You can combine some single-letter options but not others. For example, `jar -cvf filename` works, but `jar -cv -f filename` doesn't. Long argument options can follow a space or `=`, and short argument options can follow with or without a space. However, this is not fully implemented: `jar -c --file=filename` works, but `jar -c -ffilename` doesn't.

2.6.3. The Class Path

When you use library JAR files in a project, you need to tell the compiler and the virtual machine where these files are by specifying the *class path*. A class path can contain

- Directories containing class files (in subdirectories that match their package names)
- JAR files

- Directories containing JAR files

The `javac` and `java` programs have an option `-cp` (with a verbose version `--class-path` or, for backwards compatibility, `-classpath`). For example,

```
java -cp .:/libs/lib1.jar:/libs/lib2.jar  
com.mycompany.MainClass
```

This class path has three elements: the current directory (`.`) and two JAR files in the directory `./libs`.



Note: In Windows, use semicolons instead of colons to separate the path elements:

```
java -cp .;.\libs\lib1.jar;.\libs\lib2.jar  
com.mycompany.MainClass
```

If you have many JAR files, put them all in a directory and use a wildcard to include them all:

```
java -cp .:/libs/* com.mycompany.MainClass
```



Note: In Unix, the `*` must be escaped to prevent shell expansion.



Caution: The `javac` compiler always looks for files in the current directory, but the `java` program only looks into the current directory if the `“.”` directory is on the class path. If you have no class path set, this is not a problem—the default class path consists of the `“.”` directory. But if you have set the class path and forgot to include the `“.”` directory, your programs will compile without error but won’t run.



Caution: The wildcard option for the class path is convenient, but it only works reliably if the JAR files are well structured. It is possible (but not a good idea) to have two versions of the same class in different JAR files. In such a situation, the first encountered class wins. The wildcard syntax does not guarantee the ordering in which the JAR files are processed, and you should not use it if you require a particular ordering of the JAR files. (Such “JAR file hell” is a problem that the Java platform module system aims to prevent—see

[Chapter 12.\)](#)

Using the `-cp` option is the preferred approach for setting the class path. An alternate approach is the `CLASSPATH` environment variable. The details depend on your shell. If you use `bash`, use a command such as

```
export CLASSPATH=.:~/home/username/project/libs/*
```

In Windows, it is

```
SET CLASSPATH=. ;C:\Users\username\project\libs\*
```



Caution: You can set the `CLASSPATH` environment variable globally (for example, in `.bashrc` or the Windows control panel). However, many programmers have regretted this when they forgot the global setting and were surprised that their classes were not found.



Note: As you will see in

[Chapter 12](#), you can group packages together into *modules*. Modules provide strong encapsulation, hiding all packages except those that you make visible. You will see in

[Chapter 12](#) how to use the *module path* to specify the locations of the modules that your programs use.

2.6.4. Package Access

You have already encountered the access modifiers `public` and `private`. Features tagged as `public` can be used by any class. Private features can be used only by the class that declares them. If you don't specify either `public` or `private`, the feature (that is, the class, method, or variable) can be accessed by all methods in the same package.

Package access is useful for utility classes and methods that are needed by the methods of a package but are not of interest to the users of the package. Another common use case is for testing. You can place test classes in the same package, and then they can access internals of the classes being tested.



Note: A source file can contain multiple classes, but at most one of them can be declared `public`. If a source file has a public class, its name must match the class name.

For variables, it is unfortunate that package access is the default. It is a common mistake to forget the `private` modifier and accidentally make an instance variable accessible to the entire package. Here is an example from the `Window` class in the `java.awt` package:

```
public class Window extends Container {  
    String warningString;  
    ...  
}
```

Since the `warningString` variable is not private, the methods of all classes in the `java.awt` package can access it. Actually, no method other than those of the `Window` class itself does that, so it seems likely that the programmer simply forgot the `private` modifier.

This can be a security issue because packages are open ended. Any class can add itself to a package by providing the appropriate package statement.

If you are concerned about this openness of packages, you are not alone. A remedy is to place your package into a module—see

[Chapter 12](#). When a package is in a module, it is not possible to add classes to the package. All packages in the Java library are grouped into modules, so you cannot access the `Window.warningString` variable simply by crafting a class in the `java.awt` package.

2.6.5. Importing Classes

The `import` statement lets you use classes without the fully qualified name. For example, when you use

```
import java.util.Random;
```

then you can write `Random` instead of `java.util.Random` in your code.



Note: Import declarations are a convenience, not a necessity. You could drop all import declarations and use fully qualified class names everywhere.

```
private java.util.Random generator = new  
java.util.Random();
```

Place `import` statements above the first class declaration in the source file, but below the `package` statement.

You can import all classes from a package with a wildcard:

```
import java.util.*;
```

The wildcard can only import classes, not packages. You cannot use `import java.*;` to obtain all packages whose name starts with `java`.

When you import multiple packages, it is possible to have a name conflict. For example, the packages `java.util` and `java.sql` both contain a `Date` class. Suppose you import both packages:

```
import java.util.*;
import java.sql.*;
```

If your program doesn't use the `Date` class, this is not a problem. But if you refer to `Date`, without the package name, the compiler complains.

In that case, you can import the specific class that you want:

```
import java.util.*;
import java.sql.*;
import java.sql.Date;
```

If you really need both classes, you must use the fully qualified name for at least one of them.



Note: The `import` statement is a convenience for programmers. Inside class files, all class names are fully qualified.



Note: The `import` statement is very different from the `#include` directive in C and C++. That directive includes header files for compilation. Imports do not cause files to be recompiled. They just shorten names, like the C++ `using` statement.

2.6.6. Static Imports

A form of the `import` statement permits the importing of static methods and variables. For example, if you add the directive

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

to the top of your source file, you can use the static methods and static variables of the `Math` class without the class name prefix:

```
area = 4 * PI * pow(r, 2); // that is, Math.PI, Math.pow
```

You can also import a specific static method or variable:

```
import static java.lang.Math.PI;
import static java.lang.Math.pow;
```



Note: As you will see in Chapters 3 and 8, it is common to use static import declarations with `java.util.Comparator` and `java.util.stream.Collectors`, which provide a large number of static methods.

2.7. Nested Classes

In the preceding section, you have seen how to organize classes into packages. Alternatively, you can place a class inside another class. Such a class is called a *nested class*. This can be useful to restrict visibility, or to avoid cluttering up a package with generic names such as `Element`, `Node`, or `Item`. Java has two kinds of nested classes, with somewhat different behavior. Let us examine both in the following sections.

2.7.1. Static Nested Classes

Consider an `Invoice` class that bills for items, each of which has a description, quantity, and unit price. We can make `Item` into a nested class:

```
public class Invoice {
    private static class Item { // Item is nested inside
        Invoice
        String description;
        int quantity;
        double unitPrice;

        double price() { return quantity * unitPrice; }
    }
}
```

```
private ArrayList<Item> items = new ArrayList<>();  
...  
}
```

It won't be clear until the next section why this inner class is declared **static**. For now, just accept it.

There is nothing special about the **Item** class, except for access control. The class is private in **Invoice**, so only **Invoice** methods can access it. For that reason, I did not bother making the instance variables of the inner class private.

Here is an example of a method that constructs an object of the inner class:

```
public class Invoice {  
    ...  
    public void addItem(String description, int quantity,  
double unitPrice) {  
        var newItem = new Item();  
        newItem.description = description;  
        newItem.quantity = quantity;  
        newItem.unitPrice = unitPrice;  
        items.add(newItem);  
    }  
}
```

A class can make a nested class public. In that case, one would want to use the usual encapsulation mechanism.

```
public class Invoice {  
    public static class Item { // A public nested class  
        private String description;  
        private int quantity;  
        private double unitPrice;  
  
        public Item(String description, int quantity,  
double unitPrice) {
```

```

        this.description = description;
        this.quantity = quantity;
        this.unitPrice = unitPrice;
    }
    public double price() { return quantity *
unitPrice; }
    ...
}

private ArrayList<Item> items = new ArrayList<>();

public void add(Item item) { items.add(item); }

...
}

```

Now anyone can construct `Item` objects by using the qualified name `Invoice.Item`:

```

var newItem = new Invoice.Item("Blackwell Toaster", 2,
19.95);
myInvoice.add(newItem);

```

There is essentially no difference between this `Invoice.Item` class and a class `InvoiceItem` declared outside any other class. Nesting the class just makes it obvious that the `Item` class represents items in an invoice.

2.7.2. Inner Classes

In the preceding section, you saw a nested class that was declared as `static`. In this section, you will see what happens if you drop the `static` modifier. Such classes are called *inner classes*.

Consider a social network in which each member has friends that are also members.

```
public class Network {  
    public class Member { // Member is an inner class of Network  
        private String name;  
        private ArrayList<Member> friends;  
  
        public Member(String name) {  
            this.name = name;  
            friends = new ArrayList<>();  
        }  
        ...  
    }  
  
    private ArrayList<Member> members = new ArrayList<>();  
    ...  
}
```

With the `static` modifier dropped, there is an essential difference. A `Member` object knows to which network it belongs. Let's see how this works.

First, here is a method to add a member to the network:

```
public class Network {  
    ...  
    public Member enroll(String name) {  
        var newMember = new Member(name);  
        members.add(newMember);  
        return newMember;  
    }  
}
```

So far, nothing much seems to be happening. We can add a member and get a reference to it.

```
var myFace = new Network();  
Network.Member fred = myFace.enroll("Fred");
```

Now let's assume Fred feels this isn't the hottest social network anymore, so he wants to deactivate his membership.

```
fred.deactivate();
```

Here is the implementation of the `deactivate` method:

```
public class Network {  
    public class Member {  
        ...  
        public void deactivate() {  
            members.remove(this);  
        }  
    }  
  
    private ArrayList<Member> members;  
    ...  
}
```

As you can see, a method of an inner class can access instance variables of its outer class. In this case, they are the instance variables of the outer class object that created it, the unpopular `myFace` network.

This is what makes an inner class different from a static nested class. Each inner class object has a reference to an object of the enclosing class. For example, the method

```
members.remove(this);
```

actually means

```
outer.members.remove(this);
```

where I use `outer` to denote the hidden reference to the enclosing class.

A static nested class does not have such a reference (just like a static method does not have the `this` reference). Use a static nested class when the

instances of the nested class don't need to know to which instance of the enclosing class they belong. Use an inner class only if this information is important.

An inner class can also invoke methods of the outer class through its outer class instance. For example, suppose the outer class had a method to unenroll a member. Then the `deactivate` method can call it:

```
public class Network {  
    public class Member {  
        ...  
        public void deactivate() {  
            unenroll(this);  
        }  
    }  
  
    private ArrayList<Member> members;  
  
    public Member enroll(String name) { ... }  
    public void unenroll(Member m) { ... }  
    ...  
}
```

In this case,

```
unenroll(this);
```

actually means

```
outer.unenroll(this);
```

2.7.3. Special Syntax Rules for Inner Classes

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

OuterClass.this

denotes the outer class reference. For example, you can write the deactivate method of the Member inner class as

```
public void deactivate() {  
    Network.this.members.remove(this);  
}
```

In this case, the `Network.this` syntax was not necessary. Simply referring to `members` implicitly uses the outer class reference. But sometimes, you need the outer class reference explicitly. Here is a method to check whether a Member object belongs to a particular network:

```
public class Network {  
    public class Member {  
        ...  
        public boolean belongsTo(Network n) {  
            return Network.this == n;  
        }  
    }  
}
```

When you construct an inner class object, it remembers the enclosing class object that constructed it. In the preceding section, a new member was created by this method:

```
public class Network {  
    ...  
    Member enroll(String name) {  
        Member newMember = new Member(name);  
        ...  
    }  
}
```

That is a shortcut for

```
Member newMember = this.new Member(name);
```

You can invoke an inner class constructor on any instance of an outer class:

```
Network.Member wilma = myFace.new Member("Wilma");
```



Note: Inner classes can have static members. A static method of an inner class can access static members of its own class and of the enclosing classes.



Note: By historical accident, inner classes were added to the Java language at a time when the virtual machine specification was considered complete, so they are translated into regular classes with a hidden instance variable referring to the enclosing instance.

[Exercise 30](#) invites you to explore this translation.



Note: *Local classes* are another variant of inner classes that we will discuss in

2.8. Documentation Comments

The JDK contains a very useful tool, called `javadoc`, that generates HTML documentation from your source files. In fact, the online API documentation that we described in

[Chapter 1](#) is simply the result of running `javadoc` on the source code of the standard Java library.

If you add comments that start with the special delimiter `/**` to your source code, you too can easily produce professional-looking documentation. This is a very nice approach because it lets you keep your code and documentation in one place. In the bad old days, programmers often put their documentation into a separate file, and it was just a question of time for the code and the comments to diverge. When documentation comments are in the same file as the source code, it is an easy matter to update both and run `javadoc` again.

2.8.1. Comment Insertion

The `javadoc` utility extracts information for the following items:

- Public classes and interfaces
- Public and protected constructors and methods
- Public and protected variables
- Packages and modules

Interfaces are introduced in

[Chapter 3](#) and protected features in

Chapter 4.

You can (and should) supply a comment for each of these features. Each comment is placed immediately above the feature it describes. A comment starts with `/**` and ends with `*/`.

Each `/** ... */` documentation comment contains free-form text followed by tags. A tag starts with an `@`, such as `@author` or `@param`.

The *first sentence* of the free-form text should be a summary statement. The `javadoc` utility automatically generates summary pages that extract these sentences.

In the free-form text, you can use HTML modifiers such as `...` for emphasis, `<code>...</code>` for a monospaced “typewriter” font, `...` for boldface, and even `` to include an image. You should, however, stay away from headings `<h1>` or rules `<hr>` because they can interfere with the formatting of the documentation.



Note: If your comments contain links to other files such as images (for example, diagrams or images of user interface components), place those files into a subdirectory of the directory containing the source file, named `doc-files`. The `javadoc` utility will copy the `doc-files` directories and their contents from the source directory to the documentation directory. You need to specify the `doc-files` directory in your link, for example ``.

2.8.2. Class Comments

The class comment must be placed directly before the class declaration. You may want to document the author and version of a class with the `@author` and `@version` tags. There can be multiple authors.

Here is an example of a class comment:

```
/**  
 * An <code>Invoice</code> object represents an invoice  
 * with  
 *   * line items for each part of the order.  
 *   * @author Fred Flintstone  
 *   * @author Barney Rubble  
 *   * @version 1.1  
 */  
public class Invoice {  
    ...  
}
```



Note: There is no need to put a * in front of every line. However, most IDEs supply the asterisks automatically, and some even rearrange them when the line breaks change.

2.8.3. Method Comments

Place each method comment immediately before its method. Document the following features:

- Each parameter, with a comment `@param variable description`
- The return value, if not `void`: `@return description`
- Any thrown exceptions (see

[Chapter 5](#)): @throws *ExceptionClass description*

Here is an example of a method comment:

```
/**  
 * Raises the salary of an employee.  
 * @param byPercent the percentage by which to raise the  
 * salary (e.g., 10 means 10%)  
 * @return the amount of the raise  
 */  
public double raiseSalary(double byPercent) {  
    double raise = salary * byPercent / 100;  
    salary += raise;  
    return raise;  
}
```

2.8.4. Variable Comments

You only need to document public variables—generally that means static constants. For example:

```
/**  
 * The number of days per year on Earth (excepting leap  
 * years)  
 */  
public static final int DAYS_PER_YEAR = 365;
```

2.8.5. General Comments

In all documentation comments, you can use the @since tag to describe the version in which this feature became available:

```
@since version 1.7.1
```

The @deprecated tag adds a comment that the class, method, or variable should no longer be used. The text should suggest a replacement. For

example,

`@deprecated Use <code>setVisible(true)</code> instead`



Note: There is also a `@Deprecated` annotation that compilers use to issue warnings when deprecated items are used—see

[Chapter 11](#). The annotation does not have a mechanism for suggesting a replacement, so you should supply both the annotation and the Javadoc comment for deprecated items.

2.8.6. Links

You can add hyperlinks to other relevant parts of the javadoc documentation or to external documents with the @see and @link tags.

The tag @see *reference* adds a hyperlink in the “see also” section. It can be used with both classes and methods. Here, reference can be one of the following:

- *package.Class#feature label*
- *label*
- *"text"*

The first case is the most useful. You supply the name of a class, method, or variable, and `javadoc` inserts a hyperlink to its documentation. For example,

```
@see com.horstmann.corejava.Employee#raiseSalary(double)
```

makes a link to the `raiseSalary(double)` method in the `com.horstmann.corejava.Employee` class. You can omit the name of the package, or both the package and class name. Then, the feature will be located in the current package or class.

Note that you must use a #, not a period, to separate the class from the method or variable name. The Java compiler itself is highly skilled in guessing the various meanings of the period character as a separator between packages, subpackages, classes, inner classes, and their methods and variables. But the `javadoc` utility isn't quite as clever, so you have to help it along.

If the @see tag is followed by a < character, you're specifying a hyperlink. You can link to any URL you like. For example:

```
@see <a href="http://en.wikipedia.org/wiki/Leap_year">Leap  
years</a>
```

In each of these cases, you can specify an optional label that will appear as the link anchor. If you omit the label, the user will see the target code name or URL as the anchor.

If the @see tag is followed by a " character, the text in quotes is displayed in the “see also” section. For example:

```
@see "Core Java for the Impatient"
```

You can add multiple @see tags for one feature but you must keep them all together.

If you like, you can place hyperlinks to other classes or methods anywhere in any of your documentation comments. Insert a tag of the form

```
{@link package.class#feature label}
```

anywhere in a comment. The feature description follows the same rules as for the @see tag.

2.8.7. Package, Module, and Overview Comments

The class, method, and variable comments are placed directly into the Java source files, delimited by `/** ... */`. However, to generate package comments, you need to add a separate file in each package directory.

Supply a Java file named `package-info.java`. The file must contain an initial javadoc comment, delimited with `/**` and `*/`, followed by a package statement. It should contain no further code or comments.

To document a module, place your comments into `module-info.java`. You can include the `@moduleGraph` directive to include a module dependency graph. (See

[Chapter 12](#) about modules and the `module-info.java` file.)

You can also supply an overview comment for all source files. Place it in a file called `overview.xhtml`, located in the parent directory that contains all the source files. All text between the tags `<body>...</body>` is extracted. This comment is displayed when the user selects “Overview” from the navigation bar.

2.8.8. Comment Extraction

Here, `docDirectory` is the name of the directory where you want the HTML files to go. Follow these steps:

1. Change to the directory that contains the source files you want to document. If you have nested packages to document, such as `com.horstmann.corejava`, you must be working in the directory that contains the subdirectory `com`. (This is the directory that contains the `overview.xhtml` file, if you supplied one.)
2. Run the command

```
javadoc -d docDirectory package1 package2 ...
```

If you omit the `-d docDirectory` option, the HTML files are extracted to the current directory. That can get messy, so I don’t recommend it.

The `javadoc` program can be fine-tuned by numerous command-line options. For example, you can use the `-author` and `-version` options to include the `@author` and `@version` tags in the documentation. (By default, they are omitted.)

Another useful option is `-link` to include hyperlinks to standard classes. For example, if you run the command

```
javadoc -link http://docs.oracle.com/javase/17/docs/api  
*.java
```

all standard library classes are automatically linked to the documentation on the Oracle website.

If you use the `-linksource` option, each source file is converted to HTML, and each class and method name turns into a hyperlink to the source.

Exercises

1. Change the calendar printing program so it starts the week on a Sunday. Also make it print a newline at the end (but only one).
2. Consider the `nextInt` method of the `Scanner` class. Is it an accessor or mutator? Why? What about the `nextInt` method of the `Random` class?
3. Can you ever have a mutator method return something other than `void`? Can you ever have an accessor method return `void`? Give examples when possible.
4. Why can't you implement a Java method that swaps the contents of two `int` variables? Instead, write a method that swaps the contents of two `IntHolder` objects. (Look up this rather obscure class in the API documentation.) Can you swap the contents of two `Integer` objects?
5. Add methods `translate` and `scale` to the `Point` record. The `translate` method moves the point by a given amount in *x*- and *y*-direction. The `scale` method scales both coordinates by a given factor. Implement these methods so that they return new points with the results. For example,

```
Point p = new Point(3, 4).translate(1, 3).scale(0.5);
```

should set `p` to a point with coordinates (2, 3.5).

6. Repeat the preceding exercise, but now implement `Point` as a class and make `translate` and `scale` into mutators.
7. In the preceding exercise, providing the constructors and getter methods of the `Point` class was rather repetitive. Most IDEs have shortcuts for writing the boilerplate code. What does your IDE offer?
8. Add `javadoc` comments to both versions of the `Point` class from the preceding exercises.

9. Implement a record `TimeOfDay` with `hour` and `minute` components. Normalize them so that the hour is between 0 and 23 and the minute between 0 and 59. Provide a method `TimeOfDay plusMinutes(int minutes)` that yields a new `TimeOfDay` object, the given number of minutes away, and a method `int minutesFrom(TimeOfDay other)` that yields the minutes between this and the given `TimeOfDay` instance.
10. Implement a class `Car` that models a car traveling along the *x*-axis, consuming gas as it moves. Provide methods to drive by a given number of miles, to add a given number of gallons to the gas tank, and to get the current distance from the origin and fuel level. Specify the fuel efficiency (in miles/gallons) in the constructor. Should this be an immutable class? Why or why not?
11. In the `RandomNumbers` class, provide two static methods `randomElement` that get a random element from an array or array list of integers. (Return zero if the array or array list is empty.) Why couldn't you make these methods into instance methods of `int[]` or `ArrayList<Integer>`?
12. Rewrite the `Cal` class to use static imports for the `System` and `LocalDate` classes.
13. Make a file `HelloWorld.java` that declares a class `HelloWorld` in a package `ch01.sec01`. Put it into some directory, but *not* in a `ch01/sec01` subdirectory. From that directory, run `javac HelloWorld.java`. Do you get a class file? Where? Then run `java HelloWorld`. What happens? Why? (Hint: Run `javap HelloWorld` and study the warning message.) Finally, try `javac -d . HelloWorld.java`. Why is that better?
14. Download the JAR file for Apache Commons CSV from <https://commons.apache.org/proper/commons-csv/index.xhtml>. Write a class with a `main` method that reads a CSV file of your choice and prints some of the content. There is sample code on the Commons CSV website. You haven't yet learned to deal with exceptions. Just use the following header for the `main` method:

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

The point of this exercise is not to do anything useful with CSV files, but to practice using a library that is delivered as a JAR file.

15. Compile the `Network` class. Note that the inner class file is named `Network$Member.class`. Use the `javap` program to spy on the generated code. The command

```
javap -private Classname
```

displays the methods and instance variables. Where do you see the reference to the enclosing class? (In Linux/Mac OS, you need to put a \ before the \$ symbol when running `javap`.)

16. Fully implement the `Invoice` class in [Section 2.7.1](#). Provide a method that prints the invoice and a demo program that constructs and prints a sample invoice.
17. Implement a class `Queue`, an unbounded queue of strings. Provide methods `add`, adding at the tail, and `remove`, removing at the head of the queue. Store elements as a linked list of nodes. Make `Node` a nested class. Should it be static or not?
18. Provide an *iterator*—an object that yields the elements of the queue in turn—for the queue of the preceding class. Make `Iterator` a nested class with methods `next` and `hasNext`. Provide a method `iterator()` of the `Queue` class that yields a `Queue.Iterator`. Should `Iterator` be static or not?

Chapter 3 ■ Interfaces and Lambda Expressions

Java was designed as an object-oriented programming language in the 1990s when object-oriented programming was the principal paradigm for software development. Interfaces are a key feature of object-oriented programming: They let you specify what should be done, without having to provide an implementation.

Long before there was object-oriented programming, there were functional programming languages, such as Lisp, in which functions and not objects are the primary structuring mechanism. Recently, functional programming has risen in importance because it is well suited for concurrent and event-driven (or “reactive”) programming. Java supports function expressions that provide a convenient bridge between object-oriented and functional programming. In this chapter, you will learn about interfaces and lambda expressions.

The key points of this chapter are:

1. An interface specifies a set of methods that an implementing class must provide.
2. An interface is a supertype of any class that implements it. Therefore, one can assign instances of the class to variables of the interface type.
3. An interface can contain static methods. All variables of an interface are automatically public, static, and final.
4. An interface can contain default methods that an implementing class can inherit or override.
5. An interface can contain private methods that cannot be called or overridden by implementing classes.
6. The `Comparable` and `Comparator` interfaces are used for comparing objects.
7. Use the `instanceof` operator, preferably its “pattern-matching” form, if you need to test whether an object conforms to a subtype.
8. A functional interface is an interface with a single abstract method.
9. A lambda expression denotes a block of code that can be executed at a later point in time.
10. Lambda expressions are converted to functional interfaces.
11. Method and constructor references refer to methods or constructors without invoking them.
12. Lambda expressions and local classes can access effectively final variables from the enclosing scope.

3.1. Interfaces

An *interface* is a mechanism for spelling out a contract between two parties: the supplier of a service and the classes that want their objects to be usable with the service. In the following sections, you will see how to define and use interfaces in Java.

3.1.1. Using Interfaces

In

[Chapter 1](#), I used the `Random` class to illustrate how to invoke a constructor. When the `Random` class was first designed in Java 1.0, it reflected the state of the art at the time. Nowadays, many more algorithms for producing pseudorandom numbers are known. The `java.util.random` package provides implementations of strong algorithms with various tradeoffs. Depending on your needs, you may want to choose between `L64X128MixRandom`, `Xoroshiro128PlusPlus`, or another algorithm with a similarly catchy name. (The API documentation of the package provides useful advice which algorithm to choose.) Interestingly, the API does not expose a separate class for each algorithm. That is a good thing. It would be a bother to change your program whenever you switch from one algorithm to another. And it is unnecessary—all generators have the same set of methods: `nextInt`, `nextDouble`, and so on.

Instead, these methods belong to an *interface* `RandomGenerator`. An interface defines supported methods—here, `nextInt`, `nextDouble`, and so on. Once you have an instance of a class that *implements* this interface, you can invoke the methods:

```
RandomGenerator generator = RandomGenerator.getDefault();
int dieToss = 1 + generator.nextInt(6);
double randomPercentage = 100 * generator.nextDouble();
```

The static `RandomGenerator.getDefault` method obtains an object of some class whose exact nature is unknown to the caller. What is known is that the class has all methods of the `RandomGenerator` interface. With that knowledge, you can use the object productively.

3.1.2. Declaring an Interface

In this section, you will learn how to declare your own interface. Let's consider a different kind of number sequences that may or may not be random. These sequences only yield integers, and they may be finite or infinite.

Such sequences can take many forms. Here are some examples:

- A sequence of integers supplied by a user
- A sequence of random integers
- The sequence of prime numbers
- The sequence of elements in an integer array
- The sequence of code points in a string
- The sequence of digits in a number

We want to implement *a single mechanism* for dealing with all these kinds of sequences.

First, let us spell out what is common between integer sequences. At a minimum, one needs two methods for working with a sequence:

- Test whether there is a next element
- Get the next element

To declare an interface, you provide the method headers, like this:

```
public interface IntSequence {  
    boolean hasNext();  
    int next();  
}
```

You need not implement these methods, but you can provide default implementations if you like—see [Section 3.2.2](#). If no implementation is provided, we say that the method is *abstract*.



Note: All methods of an interface are automatically `public`. Therefore, it is not necessary to declare `hasNext` and `next` as `public`. Some programmers do it anyway for greater clarity.

Once you have an interface declaration, you can implement useful services. Here is one, reporting the average of the first n values:

```
public static double average(IntSequence seq, int n) {  
    int count = 0;  
    double sum = 0;  
    while (seq.hasNext() && count < n) {  
        count++;  
        sum += seq.next();  
    }  
    return count == 0 ? 0 : sum / count;  
}
```

Note that this method does not know, or need to know, how the `hasNext` and `next` methods do their job. However, the compiler must know that these methods exist. Otherwise, it would not compile the method.

3.1.3. Implementing an Interface

Now let's look at the other side of the coin: the classes that want to be usable with the `average` method. They need to *implement* the `IntSequence` interface. Here is such a class:

```
public class SquareSequence implements IntSequence {  
    private int i;  
  
    public boolean hasNext() {  
        return true;  
    }  
  
    public int next() {  
        i++;  
        return i * i;  
    }  
}
```

```
        return i * i;
    }
}
```

There are infinitely many squares, and an object of this class delivers them all, one at a time. (To keep the example simple, we ignore integer overflow—see [Exercise 12](#).)

The `implements` keyword indicates that the `SquareSequence` class intends to conform to the `IntSequence` interface.



Caution: The implementing class must declare the methods of the interface as `public`. Otherwise, they would default to package access. Since the interface requires public access, the compiler would report an error.

This code gets the average of the first 100 squares:

```
var squares = new SquareSequence();
double avg = average(squares, 100);
```

There are many classes that can implement the `IntSequence` interface. For example, this class yields a finite sequence, namely the digits of a positive integer starting with the least significant one:

```
public class DigitSequence implements IntSequence {
    private int number;

    public DigitSequence(int n) {
        number = n;
    }

    public boolean hasNext() {
        // This and the following method are declared in the interface
        return number != 0;
    }

    public int next() {
        int result = number % 10;
        number /= 10;
        return result;
    }

    public int rest() {
        // This method is not declared in the interface
        return number;
    }
}
```

An object new `DigitSequence(1729)` delivers the digits 9 2 7 1 before `hasNext` returns `false`.



Note: The `SquareSequence` and `DigitSequence` classes implement all methods of the `IntSequence` interface. If a class only implements some of the methods, then it must be declared with the `abstract` modifier. See

[Chapter 4](#) for more information on abstract classes.

3.1.4. Converting to an Interface Type

This code fragment computes the average of the digit sequence values:

```
IntSequence digits = new DigitSequence(1729);
double avg = average(digits, 100);
    // Will only look at the first four sequence values
```

Look at the `digits` variable. Its type is `IntSequence`, not `DigitSequence`. A variable of type `IntSequence` refers to an object of some class that implements the `IntSequence` interface. You can always assign an object to a variable whose type is an implemented interface, or pass it to a method expecting such an interface.

Here is a bit of useful terminology. A type `S` is a *supertype* of the type `T` (the *subtype*) when any value of the subtype can be assigned to a variable of the supertype without a conversion. For example, the `IntSequence` interface is a supertype of the `DigitSequence` class.



Note: Even though it is possible to declare variables of an interface type, you can never have an object whose type is an interface. All objects are instances of classes.

3.1.5. Casts and the instanceof Operator

Occasionally, you need the opposite conversion—from a supertype to a subtype. Then you use a *cast*. For example, if you happen to know that the object stored in an `IntSequence` is actually a `DigitSequence`, you can convert the type like this:

```
IntSequence sequence = ...;
DigitSequence digits = (DigitSequence) sequence;
System.out.println(digits.rest());
```

In this scenario, the cast was necessary because `rest` is a method of `DigitSequence` but not `IntSequence`.

See [Exercise 4](#) for a more compelling example.

You can only cast an object to its actual class or one of its supertypes. If you are wrong, a compile-time error or class cast exception will occur:

```
String digitString = (String) sequence;
    // Cannot possibly work—IntSequence is not a supertype of String
SquareSequence squares = (SquareSequence) sequence;
    // Could work, throws a class cast exception if not
```

To avoid the exception, you can first test whether the object is of the desired type, using the `instanceof` operator. The expression

```
object instanceof Type
```

returns `true` if `object` is an instance of a class that has `Type` as a supertype. It is a good idea to make this check before using a cast.

```
if (sequence instanceof DigitSequence) {  
    DigitSequence digits = (DigitSequence) sequence;  
    ...  
}
```



Note: The `instanceof` operator is null-safe: The expression `obj instanceof Type` is `false` if `obj` is `null`. After all, `null` cannot possibly be a reference to an object of any given type.

3.1.6. The “Pattern-Matching” Form of `instanceof`

Because the “`instanceof` and `cast`” idiom is so common, there is a convenient shortcut, called an `instanceof` pattern match:

```
if (sequence instanceof DigitSequence digits) {  
    // Here, you can use the digits variable  
    ...  
}
```

In this example, the `digits` variable contains the `sequence` reference, cast as a `DigitSequence`. However, if `sequence` is not an instance of `DigitSequence`, no variable is declared, and the `if` body is not executed.

A useless `instanceof` pattern is an error:

```
DigitSequence digits = ...;  
if (digits instanceof IntSequence seq) ... // Error—of course it's an IntSequence
```



Note: The equally useless

```
if (digits instanceof IntSequence) ...
```

is allowed, for backwards compatibility with Java 1.0.

When an `instanceof` pattern introduces a variable, you can use it right away, in the same expression:

```
IntSequence sequence = ...;
if (sequence instanceof DigitSequence digits && digits.rest() >= 100) ...
```

This works because the right-hand side of an `&&` expression is only evaluated if the left-hand side is `true`. If the right-hand side is evaluated, `digits` must have been bound to a `DigitSequence` instance.

However, the following is a compile-time error:

```
if (sequence instanceof DigitSequence digits || digits.rest() >= 100) ... // Error
```

The right-hand side of `||` is executed when the left-hand side is `false`, and then nothing is bound to the variable `digits`.

Here is another example with the conditional operator:

```
double rest = sequence instanceof DigitSequence digits ? digits.rest() : 0;
```

The variable `digits` is defined in the subexpression after the `?`, but not in the subexpression after the `:`.



Note: The variable-declaring `instanceof` form is called “pattern-matching” because it is similar to type patterns in `switch`, a “preview” feature of Java 17. I don’t discuss preview features in detail, but here is an example of the syntax:

```
String description = switch (sequence) {
    case DigitSequence digits ->
        "A digit sequence with rest " + digits.rest();
    case SquareSequence squares -> "A square sequence";
    default -> "Some other sequence";
}
```

As with an `instanceof` pattern, each `case` declares a variable.

3.1.7. Extending Interfaces

An interface can *extend* another, requiring or providing additional methods on top of the original ones. For example, `Closeable` is an interface with a single method:

```
public interface Closeable {
    void close();
}
```

As you will see in

[Chapter 5](#), this is an important interface for closing resources when an exception occurs.

The Channel interface extends this interface:

```
public interface Channel extends Closeable {  
    boolean isOpen();  
}
```

A class that implements the Channel interface must provide both methods, and its objects can be converted to both interface types.

3.1.8. Implementing Multiple Interfaces

A class can implement any number of interfaces. For example, a FileSequence class that reads integers from a file can implement the Closeable interface in addition to IntSequence:

```
public class FileSequence implements IntSequence, Closeable {  
    ...  
}
```

Then the FileSequence class has both IntSequence and Closeable as supertypes.

3.1.9. Constants

Any variable defined in an interface is automatically `public static final`.

For example, the SwingConstants interface defines constants for compass directions:

```
public interface SwingConstants {  
    int NORTH = 1;  
    int NORTH_EAST = 2;  
    int EAST = 3;  
    ...  
}
```

You can refer to them by their qualified name, `SwingConstants.NORTH`. If your class chooses to implement the `SwingConstants` interface, you can drop the `SwingConstants` qualifier and simply write `NORTH`. However, this is not a common idiom. It is far better to use enumerations for a set of constants; see



Note: You cannot have instance variables in an interface. An interface specifies behavior, not object state.

3.2. Static, Default, and Private Methods

In earlier versions of Java, all methods of an interface had to be abstract—that is, without a body. Nowadays you can add three kinds of methods with a concrete implementation: static, default, and private methods. The following sections describe these methods.

3.2.1. Static Methods

There was never a technical reason why an interface could not have static methods, but they did not fit into the view of interfaces as abstract specifications. That thinking has now evolved. In particular, factory methods make a lot of sense in interfaces. For example, the `IntSequence` interface can have a static method `digitsOf` that generates a sequence of digits of a given integer:

```
IntSequence digits = IntSequence.digitsOf(1729);
```

The method yields an instance of some class implementing the `IntSequence` interface, but the caller need not care which one it is.

```
public interface IntSequence {  
    ...  
    static IntSequence digitsOf(int n) {  
        return new DigitSequence(n);  
    }  
}
```



Note: In the past, it had been common to place static methods in a companion class. You find pairs of interfaces and utility classes, such as `Collection/Collections` or `Path/Paths`, in the Java API. This split is no longer necessary.

3.2.2. Default Methods

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

```
public interface IntSequence {  
    default boolean hasNext() { return true; }  
    // By default, sequences are infinite
```

```
    int next();
}
```

A class implementing this interface can choose to override the `hasNext` method or to inherit the default implementation.



Note: Default methods put an end to the classic pattern of providing an interface and a companion class that implements most or all of its methods, such as `Collection/AbstractCollection` or `WindowListener/WindowAdapter` in the Java API. Nowadays you should just implement the methods in the interface.

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

```
public class Bag implements Collection
```

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

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

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

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

3.2.3. Resolving Default Method Conflicts

If a class implements two interfaces, one of which has a default method and the other a method (default or not) with the same name and parameter types, then you must resolve the conflict. This doesn't happen very often, and it is usually easy to deal with the situation.

Let's look at an example. Suppose we have an interface `Person` with a `getId` method:

```
public interface Person {
    String getName();
    default int getId() { return 0; }
}
```

And suppose there is an interface `Identified`, also with such a method.

```
public interface Identified {  
    default int getId() { return Math.abs(hashCode()); }  
}
```

You will see what the hashCode method does in

[Chapter 4](#). For now, all that matters is that it returns some integer that is derived from the object.

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

```
public class Employee implements Person, Identified {  
    ...  
}
```

The class inherits two `getId` methods provided by the `Person` and `Identified` interfaces. There is no way for the Java compiler to choose one over the other. The compiler reports an error and leaves it up to you to resolve the ambiguity. Provide a `getId` method in the `Employee` class and either implement your own ID scheme, or delegate to one of the conflicting methods, like this:

```
public class Employee implements Person, Identified {  
    public int getId() { return Identified.super.getId(); }  
    ...  
}
```



Note: The `super` keyword lets you call a supertype method. In this case, we need to specify which supertype we want. The syntax may seem a bit odd, but it is consistent with the syntax for invoking a superclass method that you will see in

[Chapter 4.](#)

Now assume that the `Identified` interface does not provide a default implementation for `getId`:

```
interface Identified {  
    int getId();  
}
```

Can the `Employee` class inherit the default method from the `Person` interface? At first glance, this might seem reasonable. But how does the compiler know whether the `Person.getId` method actually does what `Identified.getId` is expected to do? After all, it might return the level of the person's Freudian id, not an ID number.

The Java designers decided in favor of safety and uniformity. It doesn't matter how two interfaces conflict; if at least one interface provides an implementation, the compiler reports an error, and it is up to the programmer to resolve the ambiguity.



Note: If neither interface provides a default for a shared method, then there is no conflict. An implementing class has two choices: implement the method, or leave it unimplemented and declare the class as `abstract`.



Note: If a class extends a superclass (see

[Chapter 4](#)) and implements an interface, inheriting the same method from both, the rules are easier. In that case, only the superclass method matters, and any default method from the interface is simply ignored. This is actually a more common case than conflicting interfaces. See

[Chapter 4](#) for the details.

3.2.4. Private Methods

Methods in an interface can be private. A private method can be `static` or an instance method, but it cannot be a `default` method since that can be overridden. As private methods can only be used in the methods of the interface itself, their use is limited to being helper methods for the other methods of the interface.

For example, suppose the `IntSequence` class provides methods

```
static IntSequence of(int a)
static IntSequence of(int a, int b)
static IntSequence of(int a, int b, int c)
```

Then each of these methods could call a helper method

```
private static IntSequence makeFiniteSequence(int... values) { ... }
```

3.3. Examples of Interfaces

At first glance, interfaces don't seem to do very much. An interface is just a set of methods that a class promises to implement. To make the importance of interfaces more tangible, the following sections show you four examples of commonly used interfaces from the Java API.

3.3.1. The Comparable Interface

Suppose you want to sort an array of objects. A sorting algorithm repeatedly compares elements and rearranges them if they are out of order. Of course, the rules for doing the comparison are different for each class, and the sorting algorithm should just call a method supplied by the class. As long as all classes can agree on what that method is called, the sorting algorithm can do its job. That is where interfaces come in.

If a class wants to enable sorting for its objects, it should implement the `Comparable` interface. There is a technical point about this interface. We want to compare strings against strings, employees against employees, and so on. For that reason, the `Comparable` interface has a type parameter.

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

For example, the `String` class implements `Comparable<String>` so that its `compareTo` method has the signature

```
int compareTo(String other)
```



Note: A type with a type parameter such as `Comparable` or `ArrayList` is a *generic* type.
You will learn all about generic types in

[Chapter 6.](#)

When calling `x.compareTo(y)`, the `compareTo` method returns an integer value to indicate whether `x` or `y` should come first. A positive return value (not necessarily 1) indicates that `x` should come after `y`. A negative integer (not necessarily -1) is returned when `x` should come before `y`. If `x` and `y` are considered equal, the returned value is 0.

Note that the return value can be any integer. That flexibility is useful because it allows you to return a difference of integers. That is handy, provided the difference cannot produce integer overflow.

```
public class Employee implements Comparable<Employee> {  
    ...  
    public int compareTo(Employee other) {  
        return getId() - other.getId(); // OK if IDs always ≥ 0  
    }  
}
```



Caution: Returning a difference of integers does not always work. The difference can overflow for large operands of opposite sign. In that case, use the `Integer.compare` method that works correctly for all integers. However, if you know that the integers are non-negative, or their absolute value is less than `Integer.MAX_VALUE / 2`, then the difference works fine.

When comparing floating-point values, you cannot just return the difference. Instead, use the static `Double.compare` method. It does the right thing, even for $\pm\infty$ and NaN.

Here is how the `Employee` class can implement the `Comparable` interface, ordering employees by salary:

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



Note: It is perfectly legal for the `compare` method to access `other.salary`. In Java, a method can access private features of *any* object of its class.

The `String` class, as well as over a hundred other classes in the Java library, implements the `Comparable` interface. You can use the `Arrays.sort` method to sort an array of `Comparable` objects:

```
String[] names = { "Peter", "Paul", "Mary" };
Arrays.sort(names); // names is now ["Mary", "Paul", "Peter"]
```



Note: Strangely, the `Arrays.sort` method does not check at compile time whether the argument is an array of `Comparable` objects. Instead, it throws an exception if it encounters an element of a class that doesn't implement the `Comparable` interface.

3.3.2. The Comparator Interface

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

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

```
int compare(T first, T second);
```

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

```
class LengthComparator implements Comparator<String> {
    public int compare(String first, String second) {
        return first.length() - second.length();
    }
}
```

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

```
Comparator<String> comp = new LengthComparator();
if (comp.compare(names[i], names[j]) > 0) ...
```

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



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

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

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

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

You will see in [Section 3.4.2](#) how to use a `Comparator` much more easily, using a lambda expression.

3.3.3. The Runnable Interface

At a time when just about every processor has multiple cores, you want to keep those cores busy. You may want to run certain tasks in a separate thread, or give them to a thread pool for execution. To define the task, you implement the `Runnable` interface. This interface has just one method.

```
class HelloTask implements Runnable {  
    public void run() {  
        for (int i = 0; i < 1000; i++) {  
            System.out.println("Hello, World!");  
        }  
    }  
}
```

If you want to execute this task in a new thread, create the thread from the `Runnable` and start it.

```
Runnable task = new HelloTask();  
Thread thread = new Thread(task);  
thread.start();
```

Now the `run` method executes in a separate thread, and the current thread can proceed with other work.



Note: In

[Chapter 10](#), you will see other ways of executing a `Runnable`.



Note: There is also a `Callable<T>` interface for tasks that return a result of type `T`.

3.3.4. User Interface Callbacks

In a graphical user interface, you have to specify actions to be carried out when the user clicks a button, selects a menu option, drags a slider, and so on. These actions are often called *callbacks* because some code gets called back when a user action occurs.

In Java-based GUI libraries, interfaces are used for callbacks. For example, in the Swing library, the following interface is used for reporting events:

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

To specify the action, implement the interface:

```
class ClickAction implements ActionListener {  
    public void actionPerformed(ActionEvent event) {  
        System.out.println("Thanks for clicking!");  
    }  
}
```

Then, make an object of that class and add it to the button:

```
var button = new JButton("Click me!");  
button.addActionListener(new ClickAction());
```



Note: Since it is a part of the JDK, I use Swing in these examples. (Don't worry—you need not know any more about Swing than the couple of statements you just saw.) The details don't matter; in every user interface toolkit, be it Swing, JavaFX, or Android, you give a button some code that you want to run when the button is clicked.

Of course, this way of defining a button action is rather tedious. In other languages, you just give the button a function to execute, without going through the detour of making a class and instantiating it. The next section shows how you can do the same in Java.

3.4. Lambda Expressions

A *lambda expression* is a block of code that you can pass around so it can be executed later, once or multiple times. In the preceding sections, you have seen many situations where it is useful to specify such a block of code:

- To pass a comparison method to `Arrays.sort`
- To run a task in a separate thread
- To specify an action that should happen when a button is clicked

However, Java is an object-oriented language where (just about) everything is an object. There are no function types in Java. Instead, functions are expressed as objects, instances of classes that implement a particular interface. Lambda expressions give you a convenient syntax for creating such instances.

3.4.1. The Syntax of Lambda Expressions

Consider again the sorting example from [Section 3.3.2](#). We pass code that checks whether one string is shorter than another. We compute

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

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

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

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

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

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



Note: Why the letter λ ? Did Church run out of letters of the alphabet? Actually, the venerable *Principia Mathematica* (see <http://plato.stanford.edu/entries/principia-mathematica>) used the \wedge accent to denote function parameters, which inspired Church to use an uppercase lambda Λ . But in the end, he switched to the lowercase version. Ever since, an expression with parameter variables has been called a lambda expression.

If the body of a lambda expression carries out a computation that doesn't fit in a single expression, write it exactly like you would have written a method: enclosed in {} and with explicit `return` statements. For example,

```
(String first, String second) -> {
    int difference = first.length() < second.length();
    if (difference < 0) return -1;
    else if (difference > 0) return 1;
    else return 0;
}
```

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

```
Runnable task = () -> { for (int i = 0; i < 1000; i++) doWork(); }
```

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

```
Comparator<String> comp
    = (first, second) -> first.length() - second.length();
    // Same as (String first, String second)
```

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

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

```
ActionListener listener = event ->
    System.out.println("Thanks for clicking!");
    // Instead of (event) -> or (ActionEvent event) ->
```

You never specify the result type of a lambda expression. However, the compiler infers it from the body and checks that it matches the expected type. For example, the expression

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

can be used in a context where a result of type `int` is expected (or a compatible type such as `Integer`, `long`, or `double`).

3.4.2. Functional Interfaces

As you already saw, there are many interfaces in Java that express actions, such as `Runnable` or `Comparator`. Lambda expressions are compatible with these interfaces.

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

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

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

Behind the scenes, the second parameter variable of the `Arrays.sort` method receives an object of some class that implements `Comparator<String>`. Invoking the `compare` method on that object executes the body of the lambda expression. The management of these objects and classes is completely implementation-dependent and highly optimized.

In most programming languages that support function literals, you can declare function types such as `(String, String) -> int`, declare variables of those types, put functions into those variables, and invoke them. In Java, there is *only one thing* you can do with a lambda expression: put it in a variable whose type is a functional interface, so that it is converted to an instance of that interface.



Note: You cannot assign a lambda expression to a variable of type `Object`, the common supertype of all classes in Java (see

[Chapter 4](#)). `Object` is a class, not a functional interface.

The Java API provides a large number of functional interfaces (see [Section 3.6.2](#)). One of them is

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

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

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

3.5. Method and Constructor References

Sometimes, there is already a method that carries out exactly the action that you'd like to pass on to some other code. There is special syntax for a *method reference* that is even shorter than a lambda expression calling the method. A similar shortcut exists for constructors. You will see both in the following sections.

3.5.1. Method References

Suppose you want to sort strings regardless of letter case. You could call

```
Arrays.sort(strings, (x, y) -> x.compareToIgnoreCase(y));
```

Instead, you can pass this method expression:

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

The expression `String::compareToIgnoreCase` is a *method reference* that is equivalent to the lambda expression `(x, y) -> x.compareToIgnoreCase(y)`.

Here is another example. The `Objects` class defines a method `isNull`. The call `Objects.isNull(x)` simply returns the value of `x == null`. It seems hardly worth having a method for this case, but it was designed to be passed as a method expression. The call

```
list.removeIf(Objects::isNull);
```

removes all `null` values from a list.

As another example, suppose you want to print all elements of a list. The `ArrayList` class has a method `forEach` that applies a function to each element. You could call

```
list.forEach(x -> System.out.println(x));
```

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

```
list.forEach(System.out::println);
```

As you can see from these examples, the `::` operator separates the method name from the name of a class or object. [Table 3.1](#) shows the three variations, with examples for one and two parameters:

Table 3.1: Method References

Parameters	Method expression	Lambda expression
<i>Class::staticMethod</i>		
1	Math::abs	x -> Math.abs(x)
2	Math::pow	(x, y) -> Math.pow(x, y)
<i>Class::instanceMethod</i>		
1	String::length	x -> x.length()
2	String::compareTo	(x, y) -> x.compareTo(y)
<i>object::instanceMethod</i>		
1	System.out::println	x -> System.out.println(x)
2	comp::compare	(x, y) -> comp.compare(x, y)

When there are multiple overloaded methods with the same name, the compiler will try to find from the context which one you mean. For example, there are multiple versions of the `println` method. When passed to the `forEach` method of an `ArrayList<String>`, the `println(String)` method is picked.

You can use the `this` parameter in a method reference. For example, `this::equals` is the same as `x -> this.equals(x)`.



Caution: Not all lambda expressions with simple method calls can be rewritten as method expressions. For example, consider

```
x -> x.substring(k)
```

Here, you are out of luck. None of the three variants matches.

3.5.2. Constructor References

Constructor references are just like method references, except that the name of the method is `new`. For example, `Employee::new` is a reference to an `Employee` constructor. If the class has more than one constructor, then it depends on the context which constructor is chosen.

Here is an example for using such a constructor reference. Suppose you have a list of strings

```
List<String> names = ...;
```

You want a list of employees, one for each name. As you will see in

[Chapter 8](#), you can use streams to do this without a loop: Turn the list into a stream, and then call the `map` method. It applies a function and collects all results.

```
Stream<Employee> stream = names.stream().map(Employee::new);
```

Since `names.stream()` contains `String` objects, the compiler knows that `Employee::new` refers to the constructor `Employee(String)`.

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

Array constructor references are useful to overcome a limitation of Java: It is not possible to construct an array of a generic type. (See

[Chapter 6](#) for details.) For that reason, methods such as `Stream.toArray` return an `Object` array, not an array of the element type:

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

But that is unsatisfactory. The user wants an array of employees, not objects. To solve this problem, another version of `toArray` accepts a constructor reference:

```
Employee[] staff = stream.toArray(Employee[]::new);
```

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

3.6. Processing Lambda Expressions

Up to now, you have seen how to produce lambda expressions and pass them to a method that expects a functional interface. In the following sections, you will see how to write your own methods that can consume lambda expressions.

3.6.1. Implementing Deferred Execution

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

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

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

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

To accept the lambda, we need to pick (or, in rare cases, provide) a functional interface. In this case, we can just use `Runnable`:

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



Note: Here is another example from the Java API. The static `Arrays.setAll` method applies an `IntFunction` to all elements of an array, setting each element with the result:

```

var squares = new int[10];
Arrays.setAll(squares, index -> index * index)
    // Sets squares to { 0, 1, 4, ..., 81 }

```

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

Now let's make this example a bit more sophisticated. We want to tell the action in which iteration it occurs. For that, we need to pick a functional interface that has a method with an `int` parameter and a `void` return. Instead of rolling your own, I strongly recommend that you use one of the standard ones described in the next section. The standard interface for processing `int` values is

```

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

```

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

```

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

```

And here is how you call it:

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

3.6.2. Choosing a Functional Interface

In most functional programming languages, function types are *structural*. To specify a function that maps two strings to an integer, you use a type that looks something like `Function2<String, String, Integer>` or `(String, String) -> int`. In Java, you instead declare the intent of the function using a functional interface such as `Comparator<String>`. In the theory of programming languages this is called *nominal* typing.

Of course, there are many situations where you want to accept “any function” without particular semantics. There are a number of generic function types for that purpose (see [Table 3.2](#)), and it’s a very good idea to use one of them when you can.

Table 3.2: Common Functional Interfaces

Functional interface	Parameter types	Return type	Abstract method name	Description	Other methods
Runnable	none	void	run	Runs an action without parameters or return value.	

Functional interface	Parameter types	Return type	Abstract method name	Description	Other methods
Supplier<T>	none	T	get	Supplies a value of type T.	
Consumer<T>	T	void	accept	Consumes a value of type T.	andThen
BiConsumer<T, U>	T, U	void	accept	Consumes values of types T and U.	andThen
Function<T, R>	T	R	apply	A function with a parameter of type T.	compose, andThen, identity
BiFunction<T, U, R>	T, U	R	apply	A function with parameters of types T and U.	andThen
UnaryOperator<T>	T	T	apply	A unary operator on the type T.	compose, andThen, identity
BinaryOperator<T>	T, T	T	apply	A binary operator on the type T.	andThen, maxBy, minBy
Predicate<T>	T	boolean	test	A boolean-valued function.	and, or, negate, isEqual
BiPredicate<T, U>	T, U	boolean	test	A boolean-valued function with two parameters.	and, or, negate

For example, suppose you write a method to process files that match a certain criterion. Should you use the descriptive `java.io.FileFilter` class or a `Predicate<File>`? I strongly recommend that

you use the standard `Predicate<File>`. The only reason not to do so would be if you already have many useful methods producing `FileFilter` instances.



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

```
Predicate isEqual(a).or(Predicate isEqual(b))
```

is the same as

```
x -> a.equals(x) || b.equals(x)
```

[Table 3.3](#) lists the 34 available specializations for primitive types `int`, `long`, and `double`. It is a good idea to use these specializations to reduce autoboxing. For that reason, I used an `IntConsumer` instead of a `Consumer<Integer>` in the example of the preceding section.

Table 3.3: Functional Interfaces for Primitive Types
 p, q is `int`, `long`, `double`; P, Q is `Int`, `Long`, `Double`

Functional interface	Parameter types	Return type	Abstract method name
<code>BooleanSupplier</code>	none	<code>boolean</code>	<code>getAsBoolean</code>
<code>PSupplier</code>	none	P	<code>getAsP</code>
<code>PConsumer</code>	P	<code>void</code>	<code>accept</code>
<code>ObjPConsumer<T></code>	T, P	<code>void</code>	<code>accept</code>
<code>PFunction<T></code>	P	T	<code>apply</code>
<code>PToQFunction</code>	P	Q	<code>applyAsQ</code>
<code>ToPFunction<T></code>	T	P	<code>applyAsP</code>
<code>ToPBiFunction<T, U></code>	T, U	P	<code>applyAsP</code>
<code>PUnaryOperator</code>	P	P	<code>applyAsP</code>
<code>PBinaryOperator</code>	P, P	P	<code>applyAsP</code>
<code>PPredicate</code>	P	<code>boolean</code>	<code>test</code>

3.6.3. Implementing Your Own Functional Interfaces

Ever so often, you will be in a situation where none of the standard functional interfaces work for you. Then you need to roll your own.

Suppose you want to fill an image with color patterns, where the user supplies a function yielding the color for each pixel. There is no standard type for a mapping `(int, int) -> Color`. You could use `BiFunction<Integer, Integer, Color>`, but that involves autoboxing.

In this case, it makes sense to define a new interface

```
@FunctionalInterface  
public interface PixelFunction {  
    Color apply(int x, int y);  
}
```



Note: It is a good idea to tag functional interfaces with the `@FunctionalInterface` annotation. This has two advantages. First, the compiler checks that the annotated entity is an interface with a single abstract method. Second, the javadoc page includes a statement that your interface is a functional interface.

Now you are ready to implement a method:

```
BufferedImage createImage(int width, int height, PixelFunction f) {  
    var image = new BufferedImage(width, height, BufferedImage.TYPE_INT_RGB);  
  
    for (int x = 0; x < width; x++)  
        for (int y = 0; y < height; y++) {  
            Color color = f.apply(x, y);  
            image.setRGB(x, y, color.getRGB());  
        }  
    return image;  
}
```

To call it, supply a lambda expression that yields a color value for two integers:

```
BufferedImage frenchFlag = createImage(150, 100,  
    (x, y) -> x < 50 ? Color.BLUE : x < 100 ? Color.WHITE : Color.RED);
```

3.7. Lambda Expressions and Variable Scope

In the following sections, you will learn how variables work inside lambda expressions. This information is somewhat technical but essential for working with lambda expressions.

3.7.1. Scope of a Lambda Expression

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

```
int first = 0;
Comparator<String> comp = (first, second) -> first.length() -
second.length();
// Error: Variable first already defined
```

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

As another consequence of the “same scope” rule, the `this` keyword in a lambda expression denotes the `this` parameter of the method that creates the lambda. For example, consider

```
public class Application() {
    public void doWork() {
        Runnable runner = () -> { ...; System.out.println(this.toString());
... };
        ...
    }
}
```

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

3.7.2. Accessing Variables from the Enclosing Scope

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

```
public static void repeatMessage(String text, int count) {
    Runnable r = () -> {
        for (int i = 0; i < count; i++) {
            System.out.println(text);
        }
    };
    new Thread(r).start();
}
```

Note that the lambda expression accesses the parameter variables defined in the enclosing scope, not in the lambda expression itself.

Consider a call

```
repeatMessage("Hello", 1000); // Prints Hello 1000 times in a separate thread
```

Now look at the variables `count` and `text` inside the lambda expression. If you think about it, something nonobvious is going on here. The code of the lambda expression may run long after the call to `repeatMessage` has returned and the parameter variables are gone. How do the `text` and `count` variables stay around when the lambda expression is ready to execute?

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

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

In our example, the lambda expression has two free variables, `text` and `count`. The data structure representing the lambda expression must store the values for these variables—in our case, "Hello" and `1000`. We say that these values have been *captured* by the lambda expression. (It's an implementation detail how that is done. For example, one can translate a lambda expression into an object with a single method, so that the values of the free variables are copied into instance variables of that object.)



Note: The technical term for a block of code together with the values of free variables is a *closure*. In Java, lambda expressions are closures.

As you have seen, a lambda expression can capture the value of a variable in the enclosing scope. To ensure that the captured value is well defined, there is an important restriction. In a lambda expression, you can only reference variables whose value doesn't change. This is sometimes described by saying that lambda expressions capture values, not variables. For example, the following is a compile-time error:

```
for (int i = 0; i < n; i++) {  
    new Thread(() -> System.out.println(i)).start();  
    // Error—cannot capture i  
}
```

The lambda expression tries to capture `i`, but this is not legal because `i` changes. There is no single value to capture. The rule is that a lambda expression can only access local variables from an enclosing scope that are *effectively final*. An effectively final variable is never modified—it either is or could be declared as `final`.



Note: The same rule applies to variables captured by local classes (see [Section 3.9](#)). In the past, the rule was more draconian and required captured variables to actually be declared `final`. This is no longer the case.



Note: The variable of an enhanced for loop is effectively final since its scope is a single iteration. The following is perfectly legal:

```
for (String arg : args) {  
    new Thread(() -> System.out.println(arg)).start();  
    // OK to capture arg  
}
```

A new variable `arg` is created in each iteration and assigned the next value from the `args` array. In contrast, the scope of the variable `i` in the preceding example was the entire loop.

As a consequence of the “effectively final” rule, a lambda expression cannot mutate any captured variables. For example,

```
public static void repeatMessage(String text, int count, int threads) {  
    Runnable r = () -> {  
        while (count > 0) {  
            count--; // Error: Can't mutate captured variable  
            System.out.println(text);  
        }  
    };  
    for (int i = 0; i < threads; i++) new Thread(r).start();  
}
```

This is actually a good thing. As you will see in

[Chapter 10](#), if two threads update `count` at the same time, its value is undefined.



Note: Don't count on the compiler to catch all concurrent access errors. The prohibition against mutation only holds for local variables. If `count` is an instance variable or static variable of an enclosing class, then no error is reported even though the result is just as undefined.



Caution: One can circumvent the check for inappropriate mutations by using an array of length 1:

```
int[] counter = new int[1];
button.addActionListener(event -> counter[0]++);
```

The `counter` variable is effectively final—it is never changed since it always refers to the same array, so you can access it in the lambda expression.

Of course, code like this is not threadsafe. Except possibly for a callback in a single-threaded UI, this is a terrible idea. You will see how to implement a threadsafe shared counter in

3.8. Higher-Order Functions

In a functional programming language, functions are first-class citizens. Just like you can pass numbers to methods and have methods that produce numbers, you can have arguments and return values that are functions. Functions that process or return functions are called *higher-order functions*. This sounds abstract, but it is very useful in practice. Java is not quite a functional language because it uses functional interfaces, but the principle is the same. In the following sections, we will look at some examples and examine the higher-order functions in the `Comparator` interface.

3.8.1. Methods that Return Functions

Suppose sometimes we want to sort an array of strings in ascending order and other times in descending order. We can make a method that produces the correct comparator:

```
public static Comparator<String> compareInDirection(int direction) {  
    return (x, y) -> direction * x.compareTo(y);  
}
```

The call `compareInDirection(1)` yields an ascending comparator, and the call `compareInDirection(-1)` a descending comparator.

The result can be passed to another method (such as `Arrays.sort`) that expects such an interface.

```
Arrays.sort(names, compareInDirection(-1));
```

In general, don't be shy to write methods that produce functions (or, technically, instances of classes that implement a functional interface). This is useful to generate custom functions that you pass to methods with functional interfaces.

3.8.2. Methods That Modify Functions

In the preceding section, you saw a method that yields an increasing or decreasing string comparator. We can generalize this idea by reversing any comparator:

```
public static Comparator<String> reverse(Comparator<String> comp) {  
    return (x, y) -> comp.compare(y, x);  
}
```

This method operates on functions. It receives a function and returns a modified function. To get case-insensitive descending order, use

```
reverse(String::compareToIgnoreCase)
```



Note: The `Comparator` interface has a default method `reversed` that produces the reverse of a given comparator in just this way.

3.8.3. Comparator Methods

The `Comparator` interface has a number of useful static methods that are higher-order functions generating comparators.

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

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

You can chain comparators with the `thenComparing` method to break ties. For example, sort an array of people by last name, then use the first name for people with the same last name:

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

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

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

Moreover, both the `comparing` and `thenComparing` methods have variants that avoid boxing of `int`, `long`, or `double` values. An easier way of sorting by name length would be

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

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

The `nullsFirst` method needs a comparator—in this case, one that compares two strings. The `naturalOrder` method makes a comparator for any class implementing `Comparable`. Here is the complete call for sorting by potentially null middle names. I use a static import of `java.util.Comparator.*` to make the expression more legible. Note that the type for `naturalOrder` is inferred.

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

The static `reverseOrder` method gives the reverse of the natural order.

3.9. Local and Anonymous Classes

Long before there were lambda expressions, Java had a mechanism for concisely defining classes that implement an interface (functional or not). For functional interfaces, you should definitely use lambda expressions, but once in a while, you may want a concise form for an interface that isn't functional. You will also encounter the classic constructs in legacy code.

3.9.1. Local Classes

You can define a class inside a method. Such a class is called a *local class*. You would do this for classes that are just tactical. This occurs often when a class implements an interface and the caller of the method only cares about the interface, not the class.

For example, consider a method

```
public static IntSequence randomInts(int low, int high)
```

that generates an infinite sequence of random integers with the given bounds.

Since `IntSequence` is an interface, the method must return an object of some class implementing that interface. The caller doesn't care about the class, so it can be declared inside the method:

```
private static RandomGenerator generator = RandomGenerator.getDefault();  
  
public static IntSequence randomInts(int low, int high) {  
    class RandomSequence implements IntSequence {  
        public int next() { return low + generator.nextInt(high - low + 1); }  
        public boolean hasNext() { return true; }  
    }  
  
    return new RandomSequence();  
}
```



Note: A local class is not declared as `public` or `private` since it is never accessible outside the method.

There are two advantages of making a class local. First, its name is hidden in the scope of the method. Second, the methods of the class can access variables from the enclosing scope, just like the variables of a lambda expression.

In our example, the `next` method captures three variables: `low`, `high`, and `generator`. If you turned `RandomInt` into a nested class, you would have to provide an explicit constructor that receives these values and stores them in instance variables (see [Exercise 32](#)).

3.9.2. Anonymous Classes

In the example of the preceding section, the name `RandomSequence` was used exactly once: to construct the return value. In this case, you can make the class *anonymous*:

```
public static IntSequence randomInts(int low, int high) {  
    return new IntSequence() {  
        public int next() { return low + generator.nextInt(high - low + 1); }  
        public boolean hasNext() { return true; }  
    }  
}
```

The expression

```
new Interface() { methods }
```

means: Define a class implementing the interface that has the given methods, and construct one object of that class.



Note: As always, the `()` in the `new` expression indicate the construction arguments. A default constructor of the anonymous class is invoked.

Before Java had lambda expressions, anonymous inner classes were the most concise syntax available for providing runnables, comparators, and other functional objects. You will often see them in legacy code.

Nowadays, anonymous inner classes are only necessary when you need to provide two or more methods, as in the preceding example. If the `IntSequence` interface has a default `hasNext` method, as in [Section 3.2.2](#), you can simply use a lambda expression:

```
public static IntSequence randomInts(int low, int high) {  
    return () -> low + generator.nextInt(high - low + 1);  
}
```

Exercises

1. Provide an interface `Measurable` with a method `double getMeasure()` that measures an object in some way. Make `Employee` implement `Measurable`. Provide a method `double average(Measurable[] objects)` that computes the average measure. Use it to compute the average salary of an array of employees.

2. Continue with the preceding exercise and provide a method `Measurable largest(Measurable[] objects)`. Use it to find the name of the employee with the largest salary. Why do you need a cast?
3. What are all the supertypes of `String`? Of `Scanner`? Of `ImageOutputStream`? Note that each type is its own supertype. A class or interface without declared supertype has supertype `Object`.
4. Implement a static `of` method of the `IntSequence` class that yields a sequence with the arguments. For example, `IntSequence.of(3, 1, 4, 1, 5, 9)` yields a sequence with six values. Extra credit if you return an instance of an anonymous inner class.
5. Add a static method with the name `constant` of the `IntSequence` class that yields an infinite constant sequence. For example, `IntSequence.constant(1)` yields values `1 1 1 ...`, ad infinitum. Extra credit if you do this with a lambda expression.
6. The `SquareSequence` class doesn't actually deliver an infinite sequence of squares due to integer overflow. Specifically, how does it behave? Fix the problem by defining a `Sequence<T>` interface and a `SquareSequence` class that implements `Sequence<BigInteger>`.
7. In this exercise, you will try out what happens when a method is added to an interface. In Java 7, which you can get from the Java archives at <https://www.oracle.com/java/technologies/downloads/archive>, implement a class `DigitSequence` that implements `Iterator<Integer>`, not `IntSequence`. Provide methods `hasNext`, `next`, and a do-nothing `remove`. Write a program that prints the elements of an instance. In Java 8, the `Iterator` class gained another method, `forEachRemaining`. Does your code still compile when you switch to Java 8? If you put your Java 7 class in a JAR file and don't recompile, does it work in Java 8? What if you call the `forEachRemaining` method? Also, the `remove` method has become a default method in Java 8, throwing an `UnsupportedOperationException`. What happens when `remove` is called on an instance of your class?
8. Implement the method `void luckySort(ArrayList<String> strings, Comparator<String> comp)` that keeps calling `Collections.shuffle` on the array list until the elements are in increasing order, as determined by the comparator.
9. Implement a class `Greeter` that implements `Runnable` and whose `run` method prints `n` copies of "Hello, " + `target`, where `n` and `target` are set in the constructor. Construct two instances with different messages and execute them concurrently in two threads.
10. Implement methods
 - `public static void runTogether(Runnable... tasks)`
 - `public static void runInOrder(Runnable... tasks)`

```
public static void runTogether(Runnable... tasks)
public static void runInOrder(Runnable... tasks)
```

The first method should run each task in a separate thread and then return. The second method should run all methods in the current thread and return when the last one has completed.

11. Using the `listFiles(FileFilter)` and `isDirectory` methods of the `java.io.File` class, write a method that returns all subdirectories of a given directory. Use a lambda expression instead of a `FileFilter` object. Repeat with a method expression and an anonymous inner class.
12. Using the `list(FilenameFilter)` method of the `java.io.File` class, write a method that returns all files in a given directory with a given extension. Use a lambda expression, not a `FilenameFilter`. Which variable from the enclosing scope does it capture?

13. Given an array of `File` objects, sort it so that directories come before files, and within each group, elements are sorted by path name. Use a lambda expression to specify the `Comparator`.
14. Write a method that takes an array of `Runnable` instances and returns a `Runnable` whose `run` method executes them in order. Return a lambda expression.
15. Write a call to `Arrays.sort` that sorts employees by salary, breaking ties by name. Use `Comparator.thenComparing`. Then do this in reverse order.
16. Implement the `RandomSequence` in [Section 3.9.1](#) as a nested class, outside the `randomInts` method.

Chapter 4 ■ Inheritance and Reflection

The preceding chapters introduced you to classes and interfaces. In this chapter, you will learn about another fundamental concept of object-oriented programming: inheritance. Inheritance is the process of creating new classes that are built on existing classes. When you inherit from an existing class, you reuse (or inherit) its methods, and you can add new methods and fields.



Note: Instance variables and static variables are collectively called *fields*. The fields, methods, and nested classes/interfaces inside a class are collectively called its *members*.

This chapter also covers reflection, the ability to find out more about classes and their members in a running program. Reflection is a powerful feature, but it is undeniably complex. Since reflection is of greater interest to tool builders than to application programmers, you can probably glance over that part of the chapter upon first reading and come back to it later.

The key points of this chapter are:

1. A subclass can inherit or override methods from the superclass, provided they are not **private**.
2. Use the **super** keyword to invoke a superclass method or constructor.
3. A **final** method cannot be overridden; a **final** class cannot be extended.
4. An **abstract** method has no implementation; an **abstract** class cannot be instantiated.
5. A **protected** member of a superclass is accessible in a subclass method, but only when applied to objects of the same subclass. It is also accessible in its package.
6. Every class is a subclass of **Object** which provides the **toString**, **equals**, **hashCode**, and **clone** methods.

7. Each enumerated type is a subclass of `Enum` which provides instance methods `toString` and `compareTo`, and a static `valueOf` method.
8. A `sealed` type names the subtypes that can be declared in other source files.
9. Pattern matching selects actions depending on the type and properties of an object. It is particularly useful with sealed type hierarchies.
10. The `Class` class provides information about a Java type, which can be a class, array, interface, primitive type, or `void`.
11. You can use a `Class` object to load resources that are placed alongside class files.
12. You can load classes from locations other than the class path by using a class loader.
13. The `ServiceLoader` class provides a mechanism for locating and selecting service implementations.
14. The reflection library enables programs to discover members of objects, access variables, and invoke methods.
15. Proxy objects dynamically implement arbitrary interfaces, routing all method invocations to a handler.

4.1. Extending a Class

Let's return to the `Employee` class that we discussed in

[Chapter 2](#). Suppose (alas) you work for a company at which managers are treated differently from other employees. Managers are, of course, just like employees in many respects. Both employees and managers are paid a salary. However, while employees are expected to complete their assigned tasks in return for their salary, managers get bonuses if they actually achieve what they are supposed to do. This is the kind of situation that can be modeled with inheritance.

4.1.1. Super- and Subclasses

Let's define a new class, `Manager`, retaining some functionality of the `Employee` class but specifying how managers are different.

```
public class Manager extends Employee {  
    added fields  
    added or overriding methods  
}
```

The keyword `extends` indicates that you are making a new class that derives from an existing class. The existing class is called the *superclass* and the new class is called the *subclass*. In our example, the `Employee` class is the superclass and the `Manager` class is the subclass. Note that the superclass is not "superior" to its subclass. The opposite is true: Subclasses have more functionality than their superclasses. The super/sub terminology comes from set theory. The set of managers is a subset of the set of employees.

4.1.2. Defining and Inheriting Subclass Methods

Our `Manager` class has a new instance variable to store the bonus and a new method to set it:

```
public class Manager extends Employee {  
    private double bonus;  
    ...  
    public void setBonus(double bonus) {
```

```
        this.bonus = bonus;  
    }  
}
```

When you have a `Manager` object, you can of course apply the `setBonus` method, as well as non-private methods from the `Employee` class. Those methods are *inherited*.

```
Manager boss = new Manager(...);  
boss.setBonus(10000); // Defined in subclass  
boss.raiseSalary(5); // Inherited from superclass
```

4.1.3. Method Overriding

Sometimes, a superclass method needs to be modified in a subclass. For example, suppose that the `getSalary` method is expected to report the total salary of an employee. Then the inherited method is not sufficient for the `Manager` class. Instead, you need to *override* the method so that it returns the sum of the base salary and the bonus.

```
public class Manager extends Employee {  
    ...  
    public double getSalary() { // Overrides superclass method  
        return super.getSalary() + bonus;  
    }  
}
```

This method invokes the superclass method, which retrieves the base salary, and adds the bonus. Note that a subclass method cannot access the private instance variables of the superclass directly. That is why the `Manager.getSalary` method calls the public `Employee.getSalary` method. The `super` keyword is used for invoking a superclass method.



Note: Unlike `this`, `super` is not a reference to an object, but a directive to bypass dynamic method lookup (see [Section 4.1.5](#)) and

invoke a specific method instead.

It is not required to call the superclass method when overriding a method, but it is common to do so.

When you override a method, you must be careful to match the parameter types exactly. For example, suppose that the `Employee` class has a method

```
public boolean worksFor(Employee supervisor)
```

If you override this method in the `Manager` class, you cannot change the parameter type, even though surely no manager would report to a mere employee. Suppose you defined a method

```
public class Manager extends Employee {  
    ...  
    public boolean worksFor(Manager supervisor) {  
        ...  
    }  
}
```

This is simply a new method, and now `Manager` has two separate `worksFor` methods. You can protect yourself against this type of error by tagging methods that are intended to override superclass methods with the `@Override` annotation:

```
@Override public boolean worksFor(Employee supervisor)
```

If you made a mistake and are defining a new method, the compiler reports an error.

You can change the return type to a subtype when overriding a method. (In technical terms, *covariant return types* are permitted.) For example, if the `Employee` class has a method

```
public Employee getSupervisor()
```

then the `Manager` class can override it with the method

```
@Override public Manager getSupervisor()
```



Caution: When you override a method, the subclass method must be *at least as accessible* as the superclass method. In particular, if the superclass method is public, then the subclass method must also be declared public. It is a common error to accidentally omit the `public` modifier for the subclass method. The compiler then complains about the weaker access privilege.

4.1.4. Subclass Construction

Let us supply a constructor for the `Manager` class. Since the `Manager` constructor cannot access the private instance variables of the `Employee` class, it must initialize them through a superclass constructor.

```
public Manager(String name, double salary) {  
    super(name, salary);  
    bonus = 0;  
}
```

Here, the keyword `super` indicates a call to the constructor of the `Employee` superclass with `name` and `salary` as arguments. The superclass constructor call must be the *first statement* in the constructor for the subclass.

If the subclass does not explicitly call any superclass constructor, the superclass must have a no-argument constructor which is implicitly called.

4.1.5. Superclass Assignments

It is legal to assign an object from a subclass to a variable whose type is a superclass, for example:

```
Manager boss = new Manager(...);  
Employee empl = boss; // OK to assign to superclass variable
```

Now consider what happens when one invokes a method on the superclass variable.

```
double salary = empl.getSalary();
```

Even though the type of `empl` is `Employee`, the `Manager.getSalary` method is invoked. When invoking a method, the virtual machine looks at the actual class of the object and locates its version of the method. This process is called *dynamic method lookup*.

Why would you want to assign a `Manager` object to an `Employee` variable? It allows you to write code that works for *all* employees, be they managers or janitors or instances of another `Employee` subclass.

```
Employee[] staff = new Employee[...];
staff[0] = new Employee(...);
staff[1] = new Manager(...); // OK to assign to superclass variable
staff[2] = new Janitor(...);

...
double sum = 0;
for (Employee empl : staff)
    sum += empl.getSalary();
```

Thanks to dynamic method lookup, the call `empl.getSalary()` invokes the `getSalary` method belonging to the object to which `empl` refers, which may be `Employee.getSalary`, `Manager.getSalary`, and so on.



Caution: In Java, superclass assignment also works for arrays: You can assign a `Manager[]` array to an `Employee[]` variable. (The technical term is that Java arrays are *covariant*.) This is convenient, but it is also *unsound*—that is, a possible cause of type errors.

Consider this example:

```
Manager[] bosses = new Manager[10];
Employee[] empls = bosses; // Legal in Java
empls[0] = new Employee(...); // Runtime error
```

The compiler accepts the last statement since it is generally legal to store an `Employee` in an `Employee[]` array. However, here `empls` and `bosses` reference the same `Manager[]` array, which cannot hold a lowly `Employee`. This mistake is only caught at runtime, when the virtual machine throws an `ArrayStoreException`.

4.1.6. Casts

In the preceding section, you saw how a variable `empl` of type `Employee` can refer to objects whose class is `Employee`, `Manager`, or another subclass of `Employee`. That is useful for code that deals with objects from multiple classes. There is just one drawback. You can only invoke methods that belong to the superclass. Consider, for example,

```
Employee empl = new Manager(...);  
empl.setBonus(10000); // Compile-time error
```

Even though this call could succeed at runtime, it is a compile-time error. The compiler checks that you only invoke methods that exist for the receiver type. Here, `empl` is of type `Employee` and that class has no `setBonus` method.

As with interfaces, you can use the `instanceof` operator and a cast to turn a superclass reference to a subclass.

```
if (empl instanceof Manager mgr) {  
    mgr.setBonus(10000);  
}
```

4.1.7. Anonymous Subclasses

Just as you can have an anonymous class that implements an interface, you can have an anonymous class that extends a superclass. This can be handy for debugging:

```
var names = new ArrayList<String>(100) {
    public void add(int index, String element) {
        super.add(index, element);
        System.out.printf("Adding %s at %d\n", element,
index);
    }
};
```

The arguments in the parentheses following the superclass name are passed to the superclass constructor. Here, we construct an anonymous subclass of `ArrayList<String>` that overrides the `add` method. The instance is constructed with an initial capacity of 100.

A trick called *double brace initialization* uses the inner class syntax in a rather bizarre way. Suppose you want to construct an array list and pass it to a method:

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

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

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

Note the double braces. The outer braces make an anonymous subclass of `ArrayList<String>`. The inner braces are an initialization block (see

[Chapter 2](#)).

I am not recommending that you use this trick outside of Java trivia contests. It is inefficient. The constructed object can behave inconsistently in equality tests, depending on how the `equals` method is implemented.

If the `invite` method parameter has type `List`, simply call it as follows:

```
invite(List.of("Harry", "Sally"));
```

4.1.8. Method Expressions with `super`

Recall from

[Chapter 3](#) that a method expression can have the form `object::instanceMethod`. It is also valid to use `super` instead of an object reference. The method expression

```
super::instanceMethod
```

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

```
public class Worker {  
    public void work() {  
        for (int i = 0; i < 100; i++)  
            System.out.println("Working");  
    }  
}  
  
public class ConcurrentWorker extends Worker {  
    public void work() {  
        var t = new Thread(super::work);  
        t.start();  
    }  
}
```

The thread is constructed with a `Runnable` whose `run` method calls the `work` method of the superclass.

4.2. Inheritance Hierarchies

You just saw how to extend a class. Clearly, you can extend the subclass again, and repeat that process, to form a hierarchy of classes. In the following sections, you will learn the constructs that the Java language provides for controlling what can be done in such an inheritance hierarchy.

4.2.1. Final Methods and Classes

When you declare a method as `final`, no subclass can override it.

```
public class Employee {  
    ...  
    public final String getName() {  
        return name;  
    }  
}
```

A good example of a `final` method in the Java API is the `getClass` method of the `Object` class that you will see in [Section 4.6.1](#). It does not seem wise to allow objects to lie about the class to which they belong, so this method can never be changed.

Some programmers believe that the `final` keyword is good for efficiency. This may have been true in the early days of Java, but it no longer is. Modern virtual machines will speculatively “inline” simple methods, such as the `getName` method above, even if they are not declared `final`. In the rare case when a subclass is loaded that overrides such a method, the inlining is undone.

Some programmers believe that most methods of a class should be declared `final`, and only methods specifically designed to be overridden should not be. Others find this too draconian since it prevents even harmless overriding, for example for logging or debugging purposes.

Occasionally, you may want to prevent someone from forming a subclass from one of your classes. Use the `final` modifier in the class definition to indicate this. For example, here is how to prevent others from subclassing the `Executive` class:

```
public final class Executive extends Manager {  
    ...  
}
```

There is a good number of final classes in the Java API, such as `String`, `LocalTime`, and `URL`.

4.2.2. Abstract Methods and Classes

A class can define a method without an implementation, forcing subclasses to implement it. Such a method, and the class containing it, are called *abstract* and must be tagged with the `abstract` modifier. This is commonly done for very general classes, for example:

```
public abstract class Person {  
    private String name;  
  
    public Person(String name) { this.name = name; }  
    public final String getName() { return name; }  
  
    public abstract int getId();  
}
```

Any class extending `Person` must either supply an implementation of the `getId` method or be itself declared as *abstract*.

Note that an abstract class can have nonabstract methods, such as the `getName` method in the preceding example.



Note: Unlike an interface, an abstract class can have instance variables and constructors.

It is not possible to construct an instance of an abstract class. For example, the call

```
Person p = new Person("Fred"); // Error
```

would be a compile-time error.

However, you can have a variable whose type is an abstract class, provided it contains a reference to an object of a concrete subclass. Suppose the class `Student` is declared as

```
public class Student extends Person {  
    private int id;  
  
    public Student(String name, int id) { super(name);  
this.id = id; }  
    public int getId() { return id; }  
}
```

Then you can construct a `Student` and assign it to a `Person` variable.

```
Person p = new Student("Fred", 1729); // OK, a concrete subclass
```

4.2.3. Protected Access

There are times when you want to restrict a method to subclasses only or, less commonly, to allow subclass methods to access an instance variable of a superclass. For that, declare a class feature as `protected`.

For example, suppose the superclass `Employee` declares the instance variable `salary` as `protected` instead of `private`.

```
package com.horstmann.employees;  
  
public class Employee {  
    protected double salary;  
    ...  
}
```

All classes in the same package as `Employee` can access this field. Now consider a subclass from a different package:

```
package com.horstmann.managers;  
  
import com.horstmann.employees.Employee;  
  
public class Manager extends Employee {  
    ...  
    public double getSalary() {
```

```
        return salary + bonus; // OK to access protected salary
variable
    }
}
```

The `Manager` class methods can peek inside the `salary` variable of `Manager` objects only, not of other `Employee` objects. This restriction is made so that you can't abuse the protected mechanism by forming subclasses just to gain access to protected features.

Of course, protected fields should be used with caution. Once provided, you cannot take them away without breaking classes that are using them.

Protected methods and constructors are more common. For example, the `clone` method of the `Object` class is protected since it is somewhat tricky to use (see [Section 4.3.4](#)).



Caution: In Java, `protected` grants package-level access, and it only protects access from other packages.

4.2.4. Sealed Types

Unless a class is declared `final`, anyone can form a subclass of it. What if you want to have more control? For example, suppose you feel the need to write your own JSON library because none of the ones out there do exactly what you need.

The JSON standard says that a JSON value is an array, number, string, Boolean value, object, or null. An obvious approach is to model this with classes `JSONArray`, `JSONNumber`, and so on that extend an abstract class `JSONValue`:

```
public abstract class JSONValue {
    // Methods that apply to all JSON values
}
```

```
public final class JSONArray extends JSONValue {  
    ...  
}  
  
public final class JSONNumber extends JSONValue {  
    ...  
}
```

By declaring the classes `JSONArray`, `JSONNumber`, and so on, as `final`, we can ensure that nobody forms a subclass. But we cannot stop anyone from forming another subclass of `JSONValue`.

Why might we want that control? Consider this code, which uses a `switch` expression with pattern matching (see [Section 4.5](#)):

```
public String type() {  
    return switch (this) {  
        case JSONArray j -> "array";  
        case JSONNumber j -> "number";  
        case JSONString j -> "string";  
        case JSONBoolean j -> "boolean";  
        case JSONObject j -> "object";  
        case JSONNull j -> "null";  
        // No default needed here  
    };  
}
```

The compiler can check that no `default` clause is needed since all direct subclasses of `JSONValue` occur as cases.



Note: The preceding `type` method doesn't look very object-oriented. It would be in the spirit of OOP to have each of the six classes provide its own `type` method, relying on polymorphism instead of a switch. For an open-ended hierarchy, that is a good approach. But when there is a fixed set of classes, it is often more convenient to have all alternatives in one method.

We would like the compiler to know all direct subclasses of `JSONValue`. This is not an open-ended hierarchy. The JSON standard won't change—or if it does, we, the library implementors, will add a seventh subclass. We don't want anyone else out there mess with the class hierarchy.

In Java, a *sealed* class controls which classes may extend it. A sealed interface controls which classes may implement it.

Here is how to declare the `JSONValue` class as sealed:

```
public abstract sealed class JSONValue
    permits JSONArray, JSONNumber, JSONString, JSONBoolean,
JSONObject, JSONNull {
    ...
}
```

It is an error to define a nonpermitted subclass:

```
public class JSONComment extends JSONValue { ... } // Error
```

That's just as well, since JSON doesn't allow for comments. As you can see, sealed types allow for accurate modeling of domain constraints.

All direct subtypes of a sealed type must be in the same package. However, if you use modules (see

[Chapter 12](#)), then they must be in the same module.



Note: A sealed type can be declared without a `permits` clause. Then all of its direct subtypes must be declared in the same file. Programmers without access to that file cannot form subtypes.

At first glance, it appears as if a subclass of a sealed class must be final. But for exhaustiveness testing, we only need to know all direct subclasses. It is not a problem if those classes have further subclasses. For example, we can reorganize our JSON hierarchies, as shown in [Figure 4.1](#).

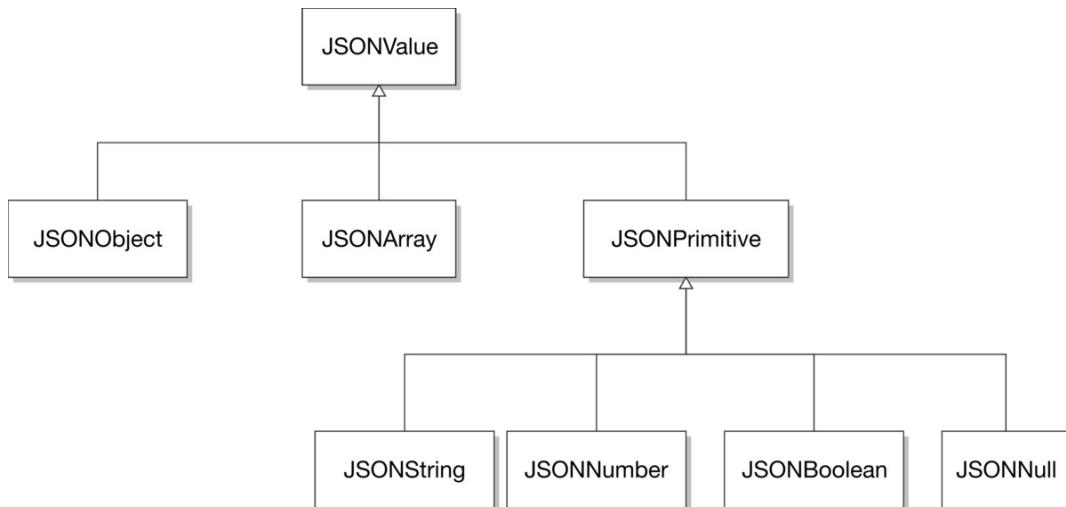


Figure 4.1: The complete hierarchy of classes for representing JSON values

In this hierarchy, `JSONValue` permits three subclasses:

```
public abstract sealed class JSONValue permits JSONObject,
JSONArray, JSONPrimitive {
    ...
}
```

The `JSONPrimitive` class is sealed as well:

```
public abstract sealed class JSONPrimitive extends JSONValue
    permits JSONString, JSONNumber, JSONBoolean,
JSONNull {
    ...
}
```

A subclass of a sealed class must specify whether it is `sealed`, `final`, or open for subclassing. In the latter case, it must be declared as `non-sealed`.



Note: Because the `sealed` keyword is much younger than the Java language, it is “contextual.” It has its meaning only when used as a type modifier. You can declare variables or methods named `sealed`:

```
int sealed = 1; // OK to use contextual keyword as identifier
```

Of course, you can’t use `non-sealed` as an identifier because of the hyphen. The only ambiguity is with subtraction:

```
int non = 0;
non = non-sealed; // Subtraction, not keyword
```

Why would you ever want a `non-sealed` subclass? Consider an XML node class with six direct subclasses:

```
public abstract sealed class Node permits Element, Text,
Comment,
    CDATASEction, EntityReference, ProcessingInstruction
{
    ...
}
```

However, the following `Element` class is not sealed and permits arbitrary subclasses:

```
public non-sealed class Element extends Node { ... }
public class HTMLDivElement extends Element { ... }
public class HTMLImgElement extends Element { ... }
...

```



Note: The example in this section shows how to use sealed classes. Interfaces can be sealed as well. Records and enumerations cannot be extended, so they cannot be sealed. However, they can extend a sealed interface.

4.2.5. Inheritance and Default Methods

Suppose a class extends a class and implements an interface, both of which happen to have a method of the same name.

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

public class Person {
    ...
    public String getName() { return name; }
}

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

In this situation, the superclass implementation always wins over the interface implementation. There is no need for the subclass to resolve the conflict.

In contrast, as you saw in

[Chapter 3](#), you must resolve a conflict when the same default method is inherited from two interfaces.

The “classes win” rule ensures compatibility. If you add default methods to an interface, it has no effect on code that worked before there were default methods.

4.3. Object: The Cosmic Superclass

Every class in Java directly or indirectly extends the class `Object`. When a class has no explicit superclass, it implicitly extends `Object`. For example,

```
public class Employee { ... }
```

is equivalent to

```
public class Employee extends Object { ... }
```

The `Object` class defines methods that are applicable to any Java object (see [Table 4.1](#)). We will examine several of these methods in detail in the following sections.



Note: Arrays are classes. Therefore, it is legal to convert an array, even a primitive type array, to a reference of type `Object`.

Table 4.1: The Methods of the `java.lang.Object` Class

Method	Description
<code>String toString()</code>	Yields a string representation of this object, by default the name of the class and the hash code.
<code>boolean equals(Object other)</code>	Returns <code>true</code> if this object should be considered equal to <code>other</code> , <code>false</code> if <code>other</code> is <code>null</code> or different from <code>other</code> . By default,

Method	Description
	two objects are equal if they are identical. Instead of <code>obj.equals(other)</code> , consider the null-safe alternative <code>Objects.equals(obj, other)</code> .
<code>int hashCode()</code>	Yields a hash code for this object. Equal objects must have the same hash code. Unless overridden, the hash code is assigned in some way by the virtual machine.
<code>Class<?> getClass()</code>	Yields the <code>Class</code> object describing the class to which this object belongs.
<code>protected Object clone()</code>	Makes a copy of this object. By default, the copy is shallow.
<code>protected void finalize()</code>	This method is called when this object is reclaimed by the garbage collector. Don't override it.
<code>wait, notify, notifyAll</code>	See

Method	Description
	Chapter 10.

4.3.1. The `toString` Method

An important method in the `Object` class is the `toString` method that returns a string description of an object. For example, the `toString` method of the `Point` class returns a string like this:

```
java.awt.Point[x=10,y=20]
```

Many `toString` methods follow this format: the name of the class, followed by the instance variables enclosed in square brackets. Here is such an implementation of the `toString` method of the `Employee` class:

```
public String toString() {
    return getClass().getName() + "[name=" + name +
",salary=" + salary + "]";
}
```

By calling `getClass().getName()` instead of hardwiring the string "`Employee`", this method does the right thing for subclasses as well.

In a subclass, call `super.toString()` and add the instance variables of the subclass, in a separate pair of brackets:

```
public class Manager extends Employee {
    ...
    public String toString() {
        return super.toString() + "[bonus=" + bonus + "]";
    }
}
```

Whenever an object is concatenated with a string, the Java compiler automatically invokes the `toString` method on the object. For example:

```
var pos = new Point(10, 20);
String message = "The current position is " + pos;
// Concatenates with pos.toString()
```



Tip: Instead of writing `x.toString()`, you can write `"" + x`. This expression even works if `x` is `null` or a primitive type value.

The `Object` class defines the `toString` method to print the class name and the hash code (see [Section 4.3.3](#)). For example, the call

```
System.out.println(System.out)
```

produces an output that looks like `java.io.PrintStream@2f6684` since the implementor of the `PrintStream` class didn't bother to override the `toString` method.



Caution: Arrays inherit the `toString` method from `Object`, with the added twist that the array type is printed in an archaic format. For example, if you have the array

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

then `primes.toString()` yields a string such as `"[I@1a46e30"`. The prefix `[I` denotes an array of integers.

The remedy is to call `Arrays.toString(primes)` instead, which yields the string `"[2, 3, 5, 7, 11, 13]"`. To correctly print multidimensional arrays (that is, arrays of arrays), use `Arrays.deepToString`.

4.3.2. The `equals` Method

The `equals` method tests whether one object is considered equal to another. The `equals` method, as implemented in the `Object` class, determines whether

two object references are identical. This is a pretty reasonable default—if two objects are identical, they should certainly be equal. For quite a few classes, nothing else is required. For example, it makes little sense to compare two `Scanner` objects for equality.

Override the `equals` method only for state-based equality testing, in which two objects are considered equal when they have the same contents. For example, the `String` class overrides `equals` to check whether two strings consist of the same characters.



Caution: Whenever you override the `equals` method, you *must* provide a compatible `hashCode` method as well—see [Section 4.3.3](#).

Suppose we want to consider two objects of a class `Item` equal if their descriptions and prices match. Here is how you can implement the `equals` method:

```
public class Item {  
    private String description;  
    private double price;  
    ...  
    public boolean equals(Object otherObject) {  
        // Must return false if the argument is null  
        if (otherObject == null) return false;  
        // Check that otherObject is an Item  
        if (getClass() != otherObject.getClass()) return  
false;  
        // Test whether the instance variables have identical values  
        var other = (Item) otherObject;  
        return Objects.equals(description,  
other.description)  
            && price == other.price;  
    }  
}
```

```
public int hashCode() { ... } // See Section 4.3.3  
}
```

There are a number of routine steps that you need to go through in an `equals` method:

1. It is common for equal objects to be identical, and that test is very inexpensive.
2. Every `equals` method is required to return `false` when comparing against `null`.
3. Since the `equals` method overrides `Object.equals`, its parameter is of type `Object`, and you need to cast it to the actual type so you can look at its instance variables. Before doing that, make a type check, either with the `getClass` method or with the `instanceof` operator.
4. Finally, compare the instance variables. Use `==` for primitive types. However, for `double` values, if you are concerned about $\pm\infty$ or `NaN`, use `Double.equals`. For objects, use `Objects.equals`, a null-safe version of the `equals` method. The call `Objects.equals(x, y)` returns `false` if `x` is `null`, whereas `x.equals(y)` would throw an exception.



Tip: If you have instance variables that are arrays, use the static `Arrays.equals` method to check that the arrays have equal length and the corresponding array elements are equal.



Caution: In the past, it was considered good practice to add a “quick test”

```
if (this == otherObject) return true;
```

to the beginning of each `equals` method. However, on modern CPUs, the branch behavior may be substantially less performant than simply comparing the instance variables.

When you define the `equals` method for a subclass, first call `equals` on the superclass. If that test doesn't pass, the objects can't be equal. If the instance variables of the superclass are equal, then you are ready to compare the instance variables of the subclass.

```
public class DiscountedItem extends Item {  
    private double discount;  
    ...  
    public boolean equals(Object otherObject) {  
        if (!super.equals(otherObject)) return false;  
        var other = (DiscountedItem) otherObject;  
        return discount == other.discount;  
    }  
  
    public int hashCode() { ... }  
}
```

Note that the `getClass` test in the superclass fails if `otherObject` is not a `DiscountedItem`.

How should the `equals` method behave when comparing values that belong to different classes? This has been an area of some controversy. In the preceding example, the `equals` method returns `false` if the classes don't match exactly. But many programmers use an `instanceof` test instead:

```
if (!(otherObject instanceof Item)) return false;
```

This leaves open the possibility that `otherObject` can belong to a subclass. For example, you can compare an `Item` with a `DiscountedItem`.

However, that kind of comparison doesn't usually work. One of the requirements of the `equals` method is that it is *symmetric*: For non-null `x` and `y`, the calls `x.equals(y)` and `y.equals(x)` need to return the same value.

Now suppose `x` is an `Item` and `y` a `DiscountedItem`. Since `x.equals(y)` doesn't consider discounts, neither can `y.equals(x)`.



Note: The Java API contains over 150 implementations of `equals` methods, with a mixture of `instanceof` tests, calling `getClass`, catching a `ClassCastException`, or doing nothing at all. Check out the documentation of the `java.sql.Timestamp` class, where the implementors note with some embarrassment that the `Timestamp` class inherits from `java.util.Date`, whose `equals` method uses an `instanceof` test, and it is therefore impossible to override `equals` to be both symmetric and accurate.

There is one situation where the `instanceof` test makes sense: if the notion of equality is fixed in the superclass and never varies in a subclass. For example, this is the case if we compare employees by ID. In that case, make an `instanceof` test and declare the `equals` method as `final`.

```
public class Employee {  
    private int id;  
    ...  
    public final boolean equals(Object otherObject) {  
        if (this == otherObject) return true;  
        if (otherObject instanceof Employee other) return id  
        == other.id;  
        return false;  
    }  
  
    public int hashCode() { ... }  
}
```

4.3.3. The `hashCode` Method

A *hash code* is an integer that is derived from an object. Hash codes should be scrambled—if `x` and `y` are two unequal objects, there should be a high probability that `x.hashCode()` and `y.hashCode()` are different. For example, "Mary".`hashCode()` is 2390779, and "Myra".`hashCode()` is 2413819.

The `String` class uses the following algorithm to compute the hash code:

```
int hash = 0;
for (int i = 0; i < length(); i++)
    hash = 31 * hash + charAt(i);
```

The `hashCode` and `equals` methods must be *compatible*: If `x.equals(y)`, then it must be the case that `x.hashCode() == y.hashCode()`. As you can see, this is the case for the `String` class since strings with equal characters produce the same hash code.

The `Object.hashCode` method derives the hash code in some implementation-dependent way. It can be derived from the object's memory location, or a number (sequential or pseudorandom) that is cached with the object, or a combination of both. Since `Object.equals` tests for identical objects, the only thing that matters is that identical objects have the same hash code.

If you redefine the `equals` method, you will also need to redefine the `hashCode` method to be compatible with `equals`. If you don't, and users of your class insert objects into a hash set or hash map, they might get lost!

In your `hashCode` method, simply combine the hash codes of the instance variables. For example, here is a `hashCode` method for the `Item` class:

```
class Item {
    ...
    public int hashCode() {
        return Objects.hash(description, price);
    }
}
```

The `Objects.hash` varargs method computes the hash codes of its arguments and combines them. The method is null-safe.

If your class has instance variables that are arrays, compute their hash codes first with the static `Arrays.hashCode` method, which computes a hash code composed of the hash codes of the array elements. Pass the result to `Objects.hash`.



Caution: In an interface, you can never make a default method that redefines one of the methods in the `Object` class. In particular, an interface can't define a default method for `toString`, `equals`, or `hashCode`. As a consequence of the “classes win” rule (see [Section 4.2.5](#)), such a method could never win against `Object.toString`, `Object.equals`, or `Object.hashCode`.

4.3.4. Cloning Objects

You have just seen the “big three” methods of the `Object` class that are commonly overridden: `toString`, `equals`, and `hashCode`. In this section, you will learn how to override the `clone` method. As you will see, this is complex, and `hashCode` it is also rarely necessary. Don’t override `clone` unless you have a good reason to do so. Less than five percent of the classes in the standard Java library implement `clone`.

The purpose of the `clone` method is to make a “clone” of an object—a distinct object with the same state of the original. If you mutate one of the objects, the other stays unchanged.

```
Employee cloneOfFred = fred.clone();
cloneOfFred.raiseSalary(10); // fred unchanged
```

The `clone` method is declared as `protected` in the `Object` class, so you must override it if you want users of your class to clone instances.

The `Object.clone` method makes a *shallow copy*. It simply copies all instance variables from the original to the cloned object. That is fine if the variables are primitive or immutable. But if they aren’t, then the original and the clone share mutable state, which can be a problem.

Consider a class for email messages that has a list of recipients.

```
public final class Message {
    private String sender;
    private ArrayList<String> recipients;
```

```

private String text;
...
public void addRecipient(String recipient) { ... };
}

```

If you make a shallow copy of a `Message` object, both the original and the clone share the `recipients` list (see [Figure 4.2](#)):

```

Message specialOffer = ...;
Message cloneOfSpecialOffer = specialOffer.clone();

```

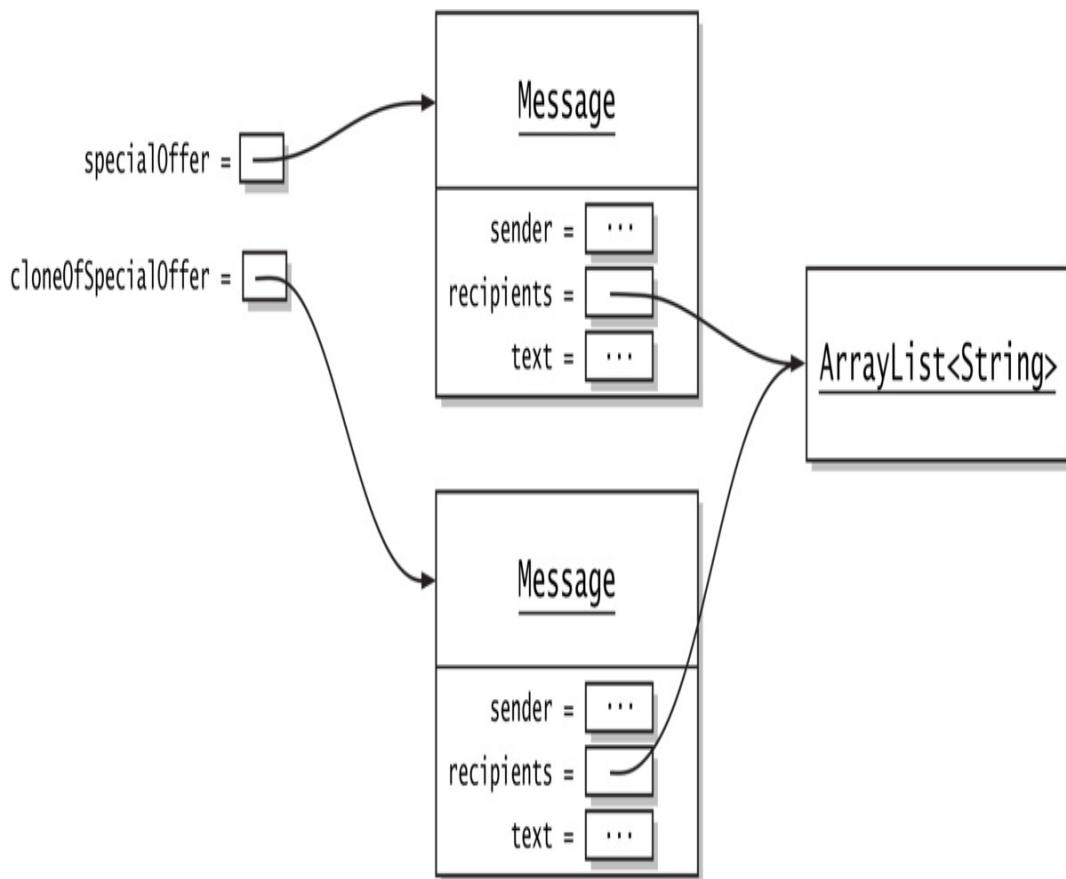


Figure 4.2: A shallow copy of an object

If either object changes the recipient list, the change is reflected in the other. Therefore, the `Message` class needs to override the `clone` method to make a

deep copy.

It may also be that cloning is impossible or not worth the trouble. For example, it would be very challenging to clone a `Scanner` object.

In general, when you implement a class, you need to decide whether

1. You do not want to provide a `clone` method, or
2. The inherited `clone` method is acceptable, or
3. The `clone` method should make a deep copy.

For the first option, simply do nothing. Your class will inherit the `clone` method, but no user of your class will be able to call it since it is `protected`.

To choose the second option, your class must implement the `Cloneable` interface. This is an interface without any methods, called a *tagging* or *marker* interface. (Recall that the `clone` method is defined in the `Object` class.) The `Object.clone` method checks that this interface is implemented before making a shallow copy, and throws a `CloneNotSupportedException` otherwise.

You will also want to raise the scope of `clone` from `protected` to `public`, and change the return type.

Finally, you need to deal with the `CloneNotSupportedException`. This is a *checked* exception, and as you will see in

[Chapter 5](#), you must either declare or catch it. If your class is `final`, you can catch it. Otherwise, declare the exception since it is possible that a subclass might again want to throw it.

```
public class Employee implements Cloneable {  
    ...  
    public Employee clone() throws  
CloneNotSupportedException {  
        return (Employee) super.clone();  
    }  
}
```

The cast (`Employee`) is necessary since the return type of `Object.clone` is `Object`.

The third option for implementing the `clone` method, in which a class needs to make a deep copy, is the most complex case. You don't need to use the `Object.clone` method at all. Here is a simple implementation of `Message.clone`:

```
public Message clone() {  
    var cloned = new Message(sender, text);  
    cloned.recipients = new ArrayList<>(recipients);  
    return cloned;  
}
```

Alternatively, you can call `clone` on the superclass and the mutable instance variables.

The `ArrayList` class implements the `clone` method, yielding a shallow copy. That is, the original and cloned lists share the element references. That is fine in our case since the elements are strings. If not, we would have had to clone each element as well.

However, for historical reasons, the `ArrayList.clone` method has return type `Object`. You need to use a cast.

```
cloned.recipients = (ArrayList<String>) recipients.clone();
// Warning
```

Unhappily, as you will see in

[Chapter 6](#), that cast cannot be fully checked at compile-time, and you will get a warning. You can suppress the warning with an annotation, but that annotation can only be attached to a declaration (see

[Chapter 11](#)). Here is the complete method implementation:

```
public Message clone() {
    try {
        var cloned = (Message) super.clone();
        @SuppressWarnings("unchecked") var clonedRecipients
            = (ArrayList<String>) recipients.clone();
        cloned.recipients = clonedRecipients;
        return cloned;
    } catch (CloneNotSupportedException ex) {
        return null; // Can't happen
    }
}
```

In this case, the `CloneNotSupportedException` cannot happen since the `Message` class is `Cloneable` and `final`, and `ArrayList.clone` does not throw the exception.



Note: Arrays have a public `clone` method whose return type is the same as the type of the array. For example, if `recipients` had been an array, not an array list, you could have cloned it as

```
cloned.recipients = recipients.clone(); // No cast required
```

4.4. Enumerations

You saw in

[Chapter 1](#) how to define enumerated types. Here is a typical example, defining a type with exactly four instances:

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

In the following sections, you will see how to work with enumerations.

4.4.1. Methods of Enumerations

Since each instance of an enumerated type is unique, you never need to use the `equals` method for values of enumerated types. Simply use `==` to compare them. (You can, if you like, call `equals` which makes the `==` test.)

You also don't need to provide a `toString` method. It is automatically provided to yield the name of the enumerated object—in our example, "SMALL", "MEDIUM", and so on.

The converse of `toString` is the static `valueOf` method that is synthesized for each enumerated type. For example, the statement

```
Size notMySize = Size.valueOf("SMALL");
```

sets `notMySize` to `Size.SMALL`. The `valueOf` method throws an exception if there is no instance with the given name.

Each enumerated type has a static `values` method that returns an array of all instances of the enumeration, in the order in which they were declared. The call

```
Size[] allValues = Size.values();
```

returns the array with elements `Size.SMALL`, `Size.MEDIUM`, and so on.



Tip: Use this method to traverse all instances of an enumerated type in an enhanced `for` loop:

```
for (Size s : Size.values()) { System.out.println(s); }
```

The `ordinal` method yields the position of an instance in the `enum` declaration, counting from zero. For example, `Size.MEDIUM.ordinal()` returns 1. Of course, you need to be careful with this method. The values shift if new constants are inserted.

Every enumerated type `E` automatically implements `Comparable<E>`, allowing comparisons only against its own objects. The comparison is by ordinal values.



Note: Technically, an enumerated type `E` extends the class `Enum<E>` from which it inherits the `compareTo` method as well as the other methods described in this section. [Table 4.2](#) shows the methods of the `Enum` class.

Table 4.2: Methods of the `java.lang.Enum<E>` Class

Method	Description
<code>String toString()</code> <code>String name()</code>	The name of this instance, as provided in the <code>enum</code> declaration. The <code>name</code> method is <code>final</code> .
<code>int ordinal()</code>	The position of this instance in the <code>enum</code> declaration.
<code>int compareTo(Enum<E> other)</code>	Compares this instance against <code>other</code> by ordinal value.
<code>static T valueOf(Class<T> type, String</code>	Returns the instance for a given name. Consider using the synthesized

Method	Description
<code>name()</code>	<code>valueOf</code> or <code>values</code> methods of the enumeration type instead.
<code>Class<E> getDeclaringClass()</code>	Gets the class in which this instance was defined. (This differs from <code>getClass()</code> if the instance has a body.)
<code>int hashCode()</code>	Calls the corresponding <code>Object</code> method.
<code>protected Object clone()</code>	Throws a <code>CloneNotSupportedException</code> .

4.4.2. Constructors, Methods, and Fields

You can, if you like, add constructors, methods, and fields to an enumerated type. Here is an example:

```
public enum Size {
    SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");

    private String abbreviation;

    Size(String abbreviation) {
        this.abbreviation = abbreviation;
    }

    public String getAbbreviation() { return abbreviation; }
}
```

Each instance of the enumeration is guaranteed to be constructed exactly once.



Note: The constructor of an enumeration is always private. You can omit the `private` modifier, as in the preceding example. It is a syntax error to declare an enum constructor as `public` or `protected`.

4.4.3. Bodies of Instances

You can add methods to each individual `enum` instance, but they have to override methods defined in the enumeration. For example, to implement a calculator, you might do this:

```
public enum Operation {  
    ADD {  
        public int eval(int arg1, int arg2) { return arg1 +  
            arg2; }  
    },  
    SUBTRACT {  
        public int eval(int arg1, int arg2) { return arg1 -  
            arg2; }  
    },  
    MULTIPLY {  
        public int eval(int arg1, int arg2) { return arg1 *  
            arg2; }  
    },  
    DIVIDE {  
        public int eval(int arg1, int arg2) { return arg1 /  
            arg2; }  
    };  
  
    public abstract int eval(int arg1, int arg2);  
}
```

In the loop of a calculator program, one would set a variable to one of these values, depending on user input, and then invoke `eval`:

```
Operation op = ...;
int result = op.eval(first, second);
```



Note: Technically, each of these constants belongs to an anonymous subclass of `Operation`. Anything that you could place into an anonymous subclass body you can also add into the body of a member.

4.4.4. Static Members

It is legal for an enumeration to have static members. However, you have to be careful with construction order. The enumerated constants are constructed *before* the static members, so you cannot refer to any static members in an enumeration constructor. For example, the following would be illegal:

```
public enum Modifier {
    PUBLIC, PRIVATE, PROTECTED, STATIC, FINAL, ABSTRACT;
    private static int maskBit = 1;
    private int mask;
    Modifier() {
        mask = maskBit; // Error—cannot access static variable in
constructor
        maskBit *= 2;
    }
    ...
}
```

The remedy is to do the initialization in a static initializer:

```
public enum Modifier {
    PUBLIC, PRIVATE, PROTECTED, STATIC, FINAL, ABSTRACT;
    private int mask;

    static {
        int maskBit = 1;
        for (Modifier m : Modifier.values()) {
```

```
    m.mask = maskBit;
    maskBit *= 2;
}
...
}
```

Once the constants have been constructed, static variable initializations and static initializers run in the usual top-to-bottom fashion.



Note: Enumerated types can be nested inside classes. Such nested enumerations are implicitly static nested classes—that is, their methods cannot reference instance variables of the enclosing class.

4.4.5. Switching on an Enumeration

You can use enumeration constants in a `switch` expression or statement.

```
enum Operation { ADD, SUBTRACT, MULTIPLY, DIVIDE };

public static int eval(Operation op, int arg1, int arg2) {
    return switch (op) {
        case ADD -> arg1 + arg2;
        case SUBTRACT -> arg1 - arg2;
        case MULTIPLY -> arg1 * arg2;
        case DIVIDE -> arg1 / arg2;
    }
}
```

No `default` case is needed if you provide cases for all instances of the enumeration.

Note that you can use `ADD` instead of `Operation.ADD` inside the `switch` statement. The enumeration is inferred from the type of the operand.



Tip: If you want to refer to the instances of an enumeration by their simple name outside a `switch`, use a static import declaration. For example, with the declaration

```
import static com.horstmann.corejava.Size.*;
```

you can use `Size.SMALL` instead of `Size.SMALL`.

4.5. Pattern Matching

You have already seen how to use the `instanceof` operator to check whether an object has a particular type. The “pattern matching” form of `instanceof` declares a variable of the matching type:

```
Employee e = ...;
String description;
if (e instanceof Executive exec)
    description = "An executive with a title of " +
exec.getTitle();
else if (e instanceof Manager mgr)
    description = "A manager with a bonus of " +
mgr.getBonus();
else
    description = "A lowly employee with a salary of " +
e.getSalary();
```

A `switch` with *type patterns* simplifies this code:

```
Employee e = ...;
String description = switch (e) {
    case Executive exec -> "An executive with a title of " +
exec.getTitle();
    case Manager mgr -> "A manager with a bonus of " +
mgr.getBonus();
```

```
    default -> "A lowly employee with a salary of " +
e.getSalary();
};
```

Note that each type pattern declares a variable of the matched type. You use those variables to access subclass methods (such as `getTitle` in our example).



Note: You must have a variable after the type name, even if you don't need it. As a preview feature in Java 21, you can use an underscore as an *unnamed variable* in this situation:

```
case Employee _
```

If you don't want to depend on a preview feature, simply use two underscores:

```
case Employee __
```

Using the expression form of `switch` is simpler than a chain of `if` statements. The `switch` expression yields a value that is assigned to the `description` variable. In contrast, when using `instanceof`, each branch of the conditional statements must set the `description` value separately.

It is possible to use the statement and fallthrough forms of `switch` with type patterns. But the expression form without fallthrough is usually the most natural choice.

As you have just seen, the basic structure of pattern matching is straightforward. The following subsections go into technical details, some of which are a bit gnarly for compatibility reasons.

4.5.1. Record Patterns

The preceding example showed a *type pattern*. If the type matches, the object is bound to a variable of that type. When the type is a `record`, you can do

better and declare variables that are bound to the components. This is called a *record pattern*:

```
record Point(double x, double y) {}
Point p = ...;
double slope = switch (p) {
    case Point(var a, var b) -> b / a;
};
```

Now **a** and **b** are bound to the **x** and **y** components of **p**. Of course, you could also invoke **p.x()** and **p.y()**, but a record pattern can be more concise.

For added clarity, you can also specify explicit types for the introduced variables:

```
case Point(double a, double b)
```

Record patterns can be nested:

```
record Circle(Point center, double radius) {}
Circle c = ...;
switch (c) {
    case Circle(Point(var a, var b), var r) -> ...
}
```

4.5.2. Guards

You can add a condition to a pattern called a *guard*, using the restricted **when** keyword.

```
case Executive exec when exec.getTitle().getLength() >= 20 -
>
    "An executive with an impressive title";
```

In the guard expression, you can use the variable that the pattern introduces.

If the guard expression is not fulfilled, the case is not selected and the next case is tested.

If an exception occurs in the `when` clause, the `switch` throws that exception.

Guards are particularly useful in record patterns:

```
Point p = ...;
String description = switch (p) {
    case Point(var x, var y) when x == 0 && y == 0 ->
"origin";
    case Point(var x, var __) when x == 0 -> "on x-axis";
    case Point(var __, var y) when y == 0 -> "on y-axis";
    default -> "not on either axis";
};
```



Caution: A record pattern invokes the component accessors. In the unlikely case that one of them throws an exception, the `switch` throws a `MatchException`. This can only happen for custom accessors.

4.5.3. Null Handling

The classic `switch` statement is null-hostile. If the selector expression is `null`, the `switch` throws a `NullPointerException`. You can avoid this by adding a `case null`:

```
String description = switch (e) {
    case Executive exec -> "An executive with a title of " +
exec.getTitle();
    case Manager mgr -> "A manager with a bonus of " +
mgr.getBonus();
    case null -> "No employee";
    default -> "A lowly employee with a salary of " +
e.getSalary();
};
```

You can combine `case null` with `default`, but not with any other case:

```
case null, default -> "No free parking";
```

4.5.4. Exhaustiveness

Any `switch expression` must be *exhaustive*: there must be a case for each possible selector value. This is an obvious requirement since the expression must always yield a value.

There are several ways for a `switch` to be exhaustive:

- Cases for every instance of an `enum`
- Unguarded type patterns that cover every subtype of a sealed type
- An unguarded type pattern of the selector type
- A `default` clause

Guarded patterns cannot be used for exhaustiveness checks since the compiler cannot evaluate them. For example,

```
case Executive exec when exec.getTitle().getLength() >= 20 -> ...
case Executive exec when exec.getTitle().getLength() < 20 -> ...
...
```

is not exhaustive. You need to rewrite the second clause as

```
case Executive exec -> ...
```

or

```
default -> ...
```



Note: When a sealed type hierarchy or an enumeration class is modified, then a previously exhaustive `switch` might become non-exhaustive. If no cases match, the `switch` throws a `MatchException`.

A classic `switch statement` need not be exhaustive. If no cases match, then no action takes place. But for greater clarity, all *enhanced* `switch` statements must be exhaustive. A `switch` statement is enhanced if it has a pattern, `case`

`null`, `case null`, or a selector type that is not `char`, `byte`, `short`, `int`, `Character`, `Byte`, `Short`, `Integer`, `String`, or an enumeration.

If you don't want to do anything when there is no match, add

```
default -> {} // No fallthrough
```

or

```
default: break; // Fallthrough
```



Caution: Testing whether the selector value is `null` is not required for exhaustiveness. You can have an exhaustive `switch` without `case null`. It throws a `NullPointerException` when invoked with a `null` selector.



Caution: The absence of `null` checking for exhaustiveness is problematic for nested records. Consider

```
record Box<T>(T contents) { }

Box<Box<String>> doubleBoxed = ...;
String unboxed = switch (doubleBoxed) {
    case Box(Box(String s)) -> s;
};
```

If `doubleBoxed` is `null`, the `switch` throws a `NullPointerException`. Conversely, if `doubleBoxed` is `new Box<>(new Box<>(null))`, the result of the expression is `null`.

However, if `doubleBoxed` is initialized with `new Box<>(null)`, then a `MatchException` occurs. There should have been a case such as

```
case Box(b) where b == null
```

but the exhaustiveness checker ignores `null`.

4.5.5. Dominance

In a classic `switch` with only constant cases, it is illegal to have the same constant in two cases. Therefore, the branches are disjoint and their order does not matter. However, when the cases are patterns, then the order matters. The cases are checked in order, and the first matching case is followed.

When the compiler finds that a case can never match because of a preceding one, a compile-time error occurs. Here is an example:

```
Employee e = ...;
String description = switch (e) {
    case Employee __ -> "A lowly employee with a salary of "
+ e.getSalary();
    case Manager mgr -> "A manager who with a bonus of" +
mgr.getBonus(); // ERROR
};
```

Since every `Manager` is an `Employee`, the second case will never match. The first case *dominates* it.

Guarded patterns are not used for dominance checking. There is no way for the compiler to reason about the guards. For example, the compiler cannot tell that

```
case Executive exec when exec.getTitle().getLength() >= 20 -
> ...
```

dominates

```
case Executive exec when exec.getTitle().getLength() >= 30 -
> ...
```

There are a couple of special rules for `default`:

- For backwards compatibility, `default` does not dominate constant cases.
 - `default` dominates `case null`, even though it does not cover it.
-



Tip: While it is possible to place `default` before constant cases, it is a poor idea. Always put it last.

4.5.6. Patterns and Constants

You cannot match against constants inside record patterns. For example,

```
case Point(0, y) // ERROR
```

is wrong. Use a guard:

```
case Point(x, y) when x == 0 // Ok
```

Numeric constant patterns are only allowed when the *selector* type is `int`, `short`, `char`, or `byte`, or their wrapper types. String constant patterns are only allowed for a selector of type `String`. For example, the following is not legal:

```
Object obj = ...;
String description = switch (obj) {
    case 0 -> "zero"; // ERROR
    case "" -> "empty string"; // ERROR
    case null -> "null";
    default -> obj.toString();
};
```

Since the selector type is `Object`, the `switch` cannot have numeric or string constant patterns. The remedy is:

```
case Integer i when i == 0
case String s when s.equals("null")
```

The rules are different for `enum` constants. The selector type must merely be a supertype of the enumerated type; that is, `Object`, `Enum`, or an interface (see

[Chapter 6](#)):

```
Object obj = ...;
String description = switch (obj) {
    case DayOfWeek.FRIDAY -> "TGIF"; // Ok
    ...
};
```

4.6. Runtime Type Information and Resources

In Java, you can find out at runtime to which class a given object belongs. This is sometimes useful, for example in the implementation of the `equals` and `toString` methods. Moreover, you can find out how the class was loaded and load its associated data, called *resources*.

4.6.1. The Class Class

Suppose you have a variable of type `Object`, filled with some object reference, and you want to know more about the object, such as to which class it belongs.

The `getClass` method yields an object of a class called, not surprisingly, `Class`.

```
Object obj = ...;
Class<?> cl = obj.getClass();
```



Note: See

[Chapter 6](#) for an explanation of the <?> suffix. For now, just ignore it. But don't omit it. If you do, not only does your IDE give you an unsightly warning, but you also turn off useful type checks in expressions involving the variable.

Once you have a `Class` object, you can find out the class name:

```
System.out.println("This object is an instance of " +  
    cl.getName());
```

Alternatively, you can get a `Class` object by using the static `Class.forName` method:

```
String className = "java.util.Scanner";  
Class<?> cl = Class.forName(className);  
    // An object describing the java.util.Scanner class
```



Caution: The `Class.forName` method, as well as many other methods used with reflection, throws checked exceptions when something goes wrong (for example, when there is no class with the given name). For now, tag the calling method with `throws ReflectiveOperationException`. You will see in

[Chapter 5](#) how to handle the exception.

The `Class.forName` method is intended for constructing `Class` objects for classes that may not be known at compile time. If you know in advance which class you want, use a *class literal* instead:

```
Class<?> cl = java.util.Scanner.class;
```

The `.class` suffix can be used to get information about other types as well:

```
Class<?> cl2 = String[].class; // Describes the array type  
String[]  
Class<?> cl3 = Runnable.class; // Describes the Runnable  
interface  
Class<?> cl4 = int.class; // Describes the int type  
Class<?> cl5 = void.class; // Describes the void type
```

Arrays are classes in Java, but interfaces, primitive types, and `void` are not. The name `Class` is a bit unfortunate—`Type` would have been more accurate.



Caution: The `getName` method returns strange names for array types:

- `String[].class.getName()` returns "[Ljava.lang.String;"
- `int[].class.getName()` returns "[I"

This notation has been used since archaic times in the virtual machine. Use `getCanonicalName` instead to get names such as "`java.lang.String[]`" and "`int[]`". You need to use the archaic notation with the `Class.forName` method if you want to generate `Class` objects for arrays.

The virtual machine manages a unique `Class` object for each type. Therefore, you can use the `==` operator to compare class objects. For example:

```
if (other.getClass() == Employee.class) ...
```

You have already seen this use of class objects in [Section 4.3.2](#).

In the following sections, you will see what you can do with `Class` objects. See [Table 4.3](#) for a summary of useful methods.

Table 4.3: Useful Methods of the `java.lang.Class<T>` Class

Method	Description
<code>static Class<?> forName(String className)</code>	Gets the <code>Class</code> object describing <code>className</code> .
<code>String getCanonicalName()</code> <code>String getSimpleName()</code> <code>String getTypeName()</code> <code>String getName()</code> <code>String toString()</code> <code>String toGenericString()</code>	Gets the name of this class, with various idiosyncrasies for arrays, inner classes, generic classes, and modifiers (see Exercise 20).
<code>Class<? super T> getSuperclass()</code> <code>Class<?>[] getInterfaces()</code> <code>Package getPackage()</code> <code>int getModifiers()</code>	Gets the superclass, the implemented interfaces, package, and modifiers of this class. Table 4.4 shows how to analyze the value returned by <code>getModifiers</code> .
<code>boolean isPrimitive()</code> <code>boolean isArray()</code> <code>boolean isEnum()</code> <code>boolean isAnnotation()</code> <code>boolean isMemberClass()</code> <code>boolean isLocalClass()</code> <code>boolean isAnonymousClass()</code> <code>boolean isRecord()</code>	Tests whether the represented type is primitive or <code>void</code> , an array, an enumeration, an annotation (see

Method	Description
<code>boolean isSealed()</code> <code>boolean isSynthetic()</code>	Chapter 11 , nested in another class, local to a method or constructor, anonymous, a record, sealed, or synthetic (see Section 4.7.7).
<code>Class<?> getComponentType()</code> <code>Class<?>[] getPermittedSubclasses()</code> <code>Class<?> getDeclaringClass()</code> <code>Class<?> getEnclosingClass()</code> <code>Constructor getEnclosingConstructor()</code> <code>Method getEnclosingMethod()</code>	Gets the component type of an array, the permitted subclasses of a sealed class, the class declaring a nested class, the class and constructor or method in which a local class is declared.
<code>boolean isAssignableFrom(Class<?> cls)</code> <code>boolean isInstance(Object obj)</code>	Tests whether the type <code>cls</code> or the class of <code>obj</code> is a subtype of this type.
<code>String getPackageName()</code>	Gets the fully qualified package name of this class or, if it is not a top-level class, its enclosing class.
<code>ClassLoader getClassLoader()</code>	Gets the class loader that loaded this class (see Section 4.6.3).
<code>InputStream getResourceAsStream(String path)</code> <code>URL getResource(String path)</code>	Loads the requested resource from the same location from which this class was loaded.

Method	Description
<pre>Field[] getFields() Method[] getMethods() Field getField(String name) Method getMethod(String name, Class<?>... parameterTypes)</pre>	Gets all public fields or methods, or the specified field or method, from this class or a superclass.
<pre>Field[] getDeclaredFields() Method[] getDeclaredMethods() Field getDeclaredField(String name) Method getDeclaredMethod(String name, Class<?>... parameterTypes)</pre>	Gets all fields or methods, or the specified field or method, from this class.
<pre>Constructor[] getConstructors() Constructor[] getDeclaredConstructors() Constructor getConstructor(Class<?>... parameterTypes) Constructor getDeclaredConstructor(Class<? >... parameterTypes)</pre>	Gets all public constructors, or all constructors, or the specified public constructor, or the specified constructor, for this class.
<pre>RecordComponent[] getRecordComponents()</pre>	Gets the component descriptors of a record.

Table 4.4: Methods of the `java.lang.reflect.Modifier` Class

Method	Description
---------------	--------------------

Method	Description
<code>static String toString(int modifiers)</code>	Returns a string with the modifiers that correspond to the bits set in <code>modifiers</code> .
<code>static boolean is(Abstract Interface Native Private Protected Public Static Strict Synchronized Volatile)(int modifiers)</code>	Tests the bit in the <code>modifiers</code> parameter that corresponds to the modifier in the method name.

4.6.2. Loading Resources

One useful service of the `Class` class is to locate resources that your program may need, such as configuration files or images. If you place a resource into the same directory as the class file, you can open an input stream to the file like this:

```
InputStream stream =
 MyClass.class.getResourceAsStream("config.txt");
 var in = new Scanner(stream);
```



Note: Some legacy methods such as `Applet.getAudioClip` and the `javax.swing.ImageIcon` constructor read data from a `URL` object. In that case, you can use the `getResource` method which returns a `URL` to the resource.

Resources can have subdirectories which can be relative or absolute. For example, `MyClass.class.getResourceAsStream("/config/menus.txt")`

locates `config/menus.txt` in the directory that contains the root of the package to which `MyClass` belongs.

If you package classes into JAR files, zip up the resources together with the class files, and they will be located as well.

4.6.3. Class Loaders

Virtual machine instructions are stored in class files. Each class file contains the instructions for a single class or interface. A class file can be located on a file system, in a JAR file, at a remote location, or it can even be dynamically constructed in memory. A *class loader* is responsible for loading the bytes and turning them into a class or interface in the virtual machine.

The virtual machine loads class files on demand, starting with the class whose `main` method is to be invoked. That class will depend on other classes, such as `java.lang.System` and `java.util.Scanner`, which will be loaded together with the classes that they depend on.

When executing a Java program, at least three class loaders are involved.

The *bootstrap class loader* loads the most fundamental Java library classes. It is a part of the virtual machine.

The *platform class loader* loads other library classes. Unlike the classes loaded with the bootstrap class loader, platform class permissions can be configured with a security policy.

The *system class loader* loads the application classes. It locates classes in the directories and JAR files on the class path and module path.



Caution: In previous releases of the Oracle JDK, the platform and system class loaders were instances of the `URLClassLoader` class.

This is no longer the case. Some programmers used the `getURLs` method of the `URLClassLoader` to find the class path. Use `System.getProperty("java.class.path")` instead.

You can load classes from a directory or JAR file that is not already on the class path, by creating your own `URLClassLoader` instance. This is commonly done to load plugins.

```
URL[] urls = {
    URI.create("file:///path/to/directory/").toURL(),
    URI.create("file:///path/to/jarfile.jar").toURL()
};
String className = "com.mycompany.plugins.Entry";
try (var loader = new URLClassLoader(urls)) {
    Class<?> cl = Class.forName(className, true, loader);
    // Now construct an instance of cl—see Section 4.7.4
    ...
}
```



Caution: The second argument in the call `Class.forName(className, true, loader)` ensures that the static initialization of the class happens after loading. You definitely want that to happen.

Do not use the `ClassLoader.loadClass` method. It does not run the static initializers.



Note: The `URLClassLoader` loads classes from the file system. If you want to load a class from somewhere else, you need to write your own class loader. The only method you need to implement is `findClass`, like this:

```
public class MyClassLoader extends ClassLoader {
    ...
    @Override public Class<?> findClass(String name)
        throws ClassNotFoundException {
        byte[] bytes = the bytes of the class file
        return defineClass(name, bytes, 0,
```

```
    bytes.length);
}
}
```

4.6.4. The Context Class Loader

Most of the time you don't have to worry about the class loading process. Classes are transparently loaded as they are required by other classes. However, if a method loads classes dynamically, and that method is called from a class that itself was loaded with another class loader, then problems can arise. Here is a specific example.

1. You provide a utility class that is loaded by the system class loader, and it has a method

```
public class Util {
    Object createInstance(String className) {
        Class<?> cl = Class.forName(className);
        ...
    }
    ...
}
```

2. You load a plugin with another class loader that reads classes from a plugin JAR.
3. The plugin calls `Util.createInstance("com.mycompany.plugins.MyClass")` to instantiate a class in the plugin JAR.

The author of the plugin expects that class to be loaded. However, `Util.createInstance` uses its own class loader to execute `Class.forName`, and that class loader won't look into the plugin JAR. This phenomenon is called *classloader inversion*.

One remedy is to pass the class loader to the utility method and then to the `forName` method.

```
public class Util {  
    public Object createInstance(String className,  
ClassLoader loader) {  
        Class<?> cl = Class.forName(className, true,  
loader);  
        ...  
    }  
    ...  
}
```

Another strategy is to use the *context class loader* of the current thread. The main thread's context class loader is the system class loader. When a new thread is created, its context class loader is set to the creating thread's context class loader. Thus, if you don't do anything, all threads will have their context class loaders set to the system class loader. However, you can set any class loader by calling

```
Thread t = Thread.currentThread();  
t.setContextClassLoader(loader);
```

The utility method can then retrieve the context class loader:

```
public class Util {  
    public Object createInstance(String className) {  
        Thread t = Thread.currentThread();  
        ClassLoader loader = t.getContextClassLoader();  
        Class<?> cl = Class.forName(className, true,  
loader);  
        ...  
    }  
    ...  
}
```

When invoking a method of a plugin class, the application should set the context class loader to the plugin class loader. Afterwards, it should restore the previous setting.



Tip: If you write a method that loads a class by name, don't simply use the class loader of the method's class. It is a good idea to offer the caller the choice between passing an explicit class loader and using the context class loader.

4.6.5. Service Loaders

Certain services need to be configurable when a program is assembled or deployed. One way to do this is to make different implementations of a service available, and have the program choose the most appropriate one among them. The `ServiceLoader` class makes it easy to load service implementations that conform to a common interface.

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

```
package com.corejava.crypt;

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

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

```
package com.corejava.crypt.impl;

public class CaesarCipher implements Cipher {
    public byte[] encrypt(byte[] source, byte[] key) {
        var result = new byte[source.length];
        for (int i = 0; i < source.length; i++)
            result[i] = (byte)(source[i] + key[0]);
```

```
        return result;
    }
    public byte[] decrypt(byte[] source, byte[] key) {
        return encrypt(source, new byte[] { (byte) -key[0]
    });
}
public int strength() { return 1; }
}
```

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

Now add the names of the provider classes to a UTF-8 encoded text file in a `META-INF/services` directory. The name of the file is the fully qualified name of the interface. In our example, the file `META-INF/services/com.corejava.crypt.Cipher` contains the line

```
com.corejava.crypt.impl.CaesarCipher
```

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

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

This should be done just once in the program.

The `iterator` method of the service loader provides an iterator through all provided implementations of the service. (See

[Chapter 7](#) for more information about iterators.) It is easiest to use an enhanced `for` loop to traverse them. In the loop, pick an appropriate object to carry out the service.

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

Alternatively, you can use streams (see

[Chapter 8](#)) to locate the desired service. The `stream` method yields a stream of `ServiceLoader.Provider` instances. That interface has methods `type` and `get` for getting the provider class and the provider instance. If you select a provider by type, then you just call `type` and no service instances are unnecessarily instantiated. In our example, we need to get the providers since we filter the stream for ciphers that have the required strength:

```
public static Optional<Cipher> getCipher2(int minStrength) {  
    return cipherLoader.stream()  
        .map(ServiceLoader.Provider::get)  
        .filter(c -> c.strength() >= minStrength)  
        .findFirst();  
}
```

If you are willing to take any implementation, simply call `findFirst`:

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

The `Optional` class is explained in

Chapter 8.

4.7. Reflection

Reflection allows a program to inspect the contents of objects at runtime and to invoke arbitrary methods on them. This capability is useful for implementing tools such as object-relational mappers or GUI builders.

Since reflection is of interest mainly to tool builders, application programmers can safely skip this section and return to it as needed.

4.7.1. Enumerating Class Members

The three classes `Field`, `Method`, and `Constructor` in the `java.lang.reflect` package describe the fields, methods, and constructors of a class. All three classes have a method called `getName` that returns the name of the member. The `Field` class has a method `getType` that returns an object, again of type `Class`, that describes the field type. The `Method` and `Constructor` classes have methods to report the types of the parameters, and the `Method` class also reports the return type.

All three of these classes also have a method called `getModifiers` that returns an integer, with various bits turned on and off, that describes the modifiers used (such as `public` or `static`). You can use static methods such as `Modifier.isPublic` and `Modifier.isStatic` to analyze the integer that `getModifiers` returns. The `Modifier.toString` returns a string of all modifiers.

The `getFields`, `getMethods`, and `getConstructors` methods of the `Class` class return arrays of the *public* fields, methods, and constructors that the class supports; this includes public inherited members. The `getDeclaredFields`, `getDeclaredMethods`, and `getDeclaredConstructors` methods return arrays consisting of all fields, methods, and constructors that are declared in the class. This includes private, package, and protected members, but not members of superclasses.

The `getParameters` method of the `Executable` class, the common superclass of `Method` and `Constructor`, returns an array of `Parameter` objects describing the method parameters.



Note: The names of the parameters are only available at runtime if the class has been compiled with the `-parameters` flag.

For example, here is how you can print all methods of a class:

```
Class<?> cl = Class.forName(className);
while (cl != null) {
    for (Method m : cl.getDeclaredMethods()) {
        System.out.println(
            Modifier.toString(m.getModifiers()) + " " +
            m.getReturnType().getCanonicalName() + " " +
            m.getName() +
            Arrays.toString(m.getParameters()));
    }
    cl = cl.getSuperclass();
}
```

What is remarkable about this code is that it can analyze any class that the Java virtual machine can load—not just the classes that were available when the program was compiled.



Caution: As you will see in

[Chapter 12](#), the Java platform module system imposes significant restrictions on reflective access. By default, only classes in the same module can be analyzed through reflection. If you don't declare modules, all your classes belong to a single module, and they can all be accessed through reflection. However, the Java library classes belong to different modules, and reflective access to their non-public members is restricted.

4.7.2. Inspecting Objects

As you saw in the preceding section, you can get `Field` objects that describe the types and names of an object's fields. These `Field` objects can do more: They can access field values in objects that have the given field.

For example, here is how to enumerate the contents of all fields of an object:

```
Object obj = ...;
for (Field f : obj.getClass().getDeclaredFields()) {
    f.setAccessible(true);
    Object value = f.get(obj);
    System.out.println(f.getName() + ":" + value);
}
```

The key is the `get` method that reads the field value. If the field value is a primitive type value, a wrapper object is returned; in that case you can also call one of the methods `getInt`, `getDouble`, and so on.



Note: You must make private `Field` and `Method` objects “accessible” before you can use them. Calling `setAccessible(true)` “unlocks” the field or method for reflection. However, the module system or a security manager can block the request and protect objects from being accessed in this way. In that case, the `setAccessible` method throws an `InaccessibleObjectException` or `SecurityException`.

Alternatively, you can call the `trySetAccessible` method which simply returns `false` if the field or method is not accessible.



Caution: As you will see in

[Chapter 12](#), the Java platform packages are contained in modules and their classes are protected from reflection. For example, if you call

```
Field f = String.class.getDeclaredField("value");
f.setAccessible(true);
```

an `InaccessibleObjectException` is thrown.

Once a field is accessible, you can also set it. This code will give a raise to `obj`, no matter to which class it belongs, provided that it has an accessible `salary` field of type `double` or `Double`.

```
Field f = obj.getClass().getDeclaredField("salary");
f.setAccessible(true);
double value = f.getDouble(obj);
f.setDouble(obj, value * 1.1);
```

4.7.3. Invoking Methods

Just like a `Field` object can be used to read and write fields of an object, a `Method` object can invoke the given method on an object.

```
Method m = ...;
Object result = m.invoke(obj, arg1, arg2, ...);
```

If the method is static, supply `null` for the initial argument.

To obtain a method, you can search through the array returned by `getMethods` or `getDeclaredMethods` that you saw in [Section 4.7.1](#). Or you can call `getMethod` and supply the parameter types. For example, to get the `setName(String)` method on a `Person` object:

```
Person p = ...;
Method m = p.getClass().getMethod("setName", String.class);
m.invoke(obj, "*****");
```



Caution: Even though `clone` is a public method of all array types, it is not reported by `getMethod` when invoked on a `Class` object describing an array.

4.7.4. Constructing Objects

To construct an object, first find the `Constructor` object and then call its `newInstance` method. For example, suppose you know that a class has a public constructor whose parameter is an `int`. Then you can construct a new instance like this:

```
Constructor constr = cl.getConstructor(int.class);
Object obj = constr.newInstance(42);
```



Caution: The `Class` class has a `newInstance` method to construct an object of the given class with the no-argument constructor. That method is now deprecated because it has a curious flaw. If the no-argument constructor throws a checked exception, the `newInstance` method rethrows it *even though it isn't declared*, thereby completely defeating the compile-time checking of checked exceptions. Instead, you should call `cl.getConstructor().newInstance()`. Then any exception is wrapped inside an `InvocationTargetException`.

[Table 4.5](#) summarizes the most important methods for working with `Field`, `Method`, and `Constructor` objects.

Table 4.5: Useful Classes and Methods in the `java.lang.reflect` Package

Class	Method	Notes
<code>AccessibleObject</code>	<code>void setAccessible(boolean flag)</code>	<code>AccessibleObject</code> is a superclass of <code>Field</code> , <code>Method</code> , and

Class	Method	Notes
	<code>static void setAccessible(AccessibleObject[] array, boolean flag)</code>	Constructor. The methods set the accessibility of this member, or the given members.
Field	<code>String getName() int getModifiers() Object get(Object obj) p getP(Object obj) void set(Object obj, Object newValue) void setP(Object obj, p newValue)</code>	There is a <code>get</code> and <code>set</code> method for each primitive type <i>p</i> .
Method	<code>Object invoke(Object obj, Object... args)</code>	Invokes the method described by this object, passing the given arguments and returning the value that the method returns. For static methods, pass null for <code>obj</code> . Primitive type arguments and return values are wrapped.
Constructor	<code>Object newInstance(Object... args)</code>	Invokes the constructor described by this

Class	Method	Notes
		object, passing the given arguments and returning the constructed object.
Executable	String getName() int getModifiers() Parameters[] getParameters()	Executable is the superclass of Method and Constructor.
Parameter	boolean isNamePresent() String getName() Class<?> getType()	The getName method gets the name or a synthesized name such as arg0 if the name is not present.

4.7.5. JavaBeans

Many object-oriented programming languages support *properties*, mapping the expression *object.propertyName* to a call of a getter or setter method, depending on whether the property is read or written. Java does not have this syntax, but it has a convention in which properties correspond to getter/setter pairs. A *JavaBean* is a class with a no-argument constructor, getter/setter pairs, and any number of other methods.

The getters and setters must follow the specific pattern

```
public Type getProperty()
public void setProperty(Type newValue)
```

It is possible to have read-only and write-only properties by omitting the setter or getter.

The name of the property is the *decapitalized* form of the suffix after `get`/`set`. For example, a `getSalary`/`setSalary` pair gives rise to a property named `salary`. However, if the first *two* letters of the suffix are uppercase, then it is taken verbatim. For example, `getURL` yields a read-only property named `URL`.



Note: For Boolean properties, you may use either `getProperty` or `isProperty` for the getter, and the latter is preferred.

JavaBeans have their origin in GUI builders, and the JavaBeans specification has arcane rules that deal with property editors, property change events, and custom property discovery. These features are rarely used nowadays.

It is a good idea to use the standard classes for JavaBeans support whenever you need to work with arbitrary properties. Given a class, obtain a `BeanInfo` object like this:

```
Class<?> cl = ...;
BeanInfo info = Introspector.getBeanInfo(cl);
PropertyDescriptor[] props = info.getPropertyDescriptors();
```

For a given `PropertyDescriptor`, call `getName` and `getPropertyType` to get the name and type of the property. The `getReadMethod` and `getWriteMethod` yield `Method` objects for the getter and setter.

Unfortunately, there is no method to get the descriptor for a given property name, so you'll have to traverse the array of descriptors:

```
String propertyName = ...;
Object PropertyValue = null;
for (PropertyDescriptor prop : props) {
    if (prop.getName().equals(propertyName))
        PropertyValue = prop.getReadMethod().invoke(obj);
}
```

4.7.6. Working with Arrays

The `isArray` method checks whether a given `Class` object represents an array. If so, the `getComponentType` method yields the `Class` describing the type of the array elements. For further analysis, or to create arrays dynamically, use the `Array` class in the `java.lang.reflect` package. [Table 4.6](#) shows its methods.

As an exercise, let us implement the `copyOf` method in the `Arrays` class. Recall how this method can be used to grow an array that has become full.

```
var friends = new Person[100];
...
// Array is full
friends = Arrays.copyOf(friends, 2 * friends.length);
```

How can one write such a generic method? Here is a first attempt:

```
public static Object[] badCopyOf(Object[] array, int
newLength) { // Not useful
    var newArray = new Object[newLength];
    for (int i = 0; i < Math.min(array.length, newLength);
i++)
        newArray[i] = array[i];
    return newArray;
}
```

However, there is a problem with actually using the resulting array. The type of array that this method returns is `Object[]`. An array of objects cannot be cast to a `Person[]` array. The point is, as we mentioned earlier, that a Java array remembers the type of its elements—that is, the type used in the `new` expression that created it. It is legal to cast a `Person[]` array temporarily to an `Object[]` array and then cast it back, but an array that started its life as an `Object[]` array can never be cast into a `Person[]` array.

In order to make a new array of the same type as the original array, you need the `newInstance` method of the `Array` class. Supply the component type and the desired length:

```

public static Object goodCopyOf(Object array, int newLength)
{
    Class<?> cl = array.getClass();
    if (!cl.isArray()) return null;
    Class<?> componentType = cl.getComponentType();
    int length = Array.getLength(array);
    Object newArray = Array.newInstance(componentType,
newLength);
    for (int i = 0; i < Math.min(length, newLength); i++)
        Array.set(newArray, i, Array.get(array, i));
    return newArray;
}

```

Note that this `copyOf` method can be used to grow arrays of any type, not just arrays of objects.

```

int[] primes = { 2, 3, 5, 7, 11 };
primes = (int[]) goodCopyOf(primes, 10);

```

The parameter type of `goodCopyOf` is `Object`, not `Object[]`. An `int[]` is an `Object` but not an array of objects.

Table 4.6: Methods of the `java.lang.reflect.Array` Class

Method	Description
<code>static Object get(Object array, int index)</code> <code>static p getP(Object array, int index)</code> <code>static void set(Object array, int index, Object newValue)</code> <code>static void setP(Object array, int index, p newValue)</code>	Gets or sets an element of the array at the given index, where <code>p</code> is a primitive type.

Method	Description
<code>static int getLength(Object array)</code>	Gets the length of the given array.
<code>static Object newInstance(Class<?> componentType, int length)</code> <code>static Object newInstance(Class<?> componentType, int[] lengths)</code>	Returns a new array of the given component type with the given dimensions.

4.7.7. Proxies

The `Proxy` class can create, at runtime, new classes that implement a given interface or set of interfaces. Such proxies are only necessary when you don't yet know at compile time which interfaces you need to implement.

A proxy class has all methods required by the specified interfaces, and all methods defined in the `Object` class (`toString`, `equals`, and so on). However, since you cannot define new code for these methods at runtime, you supply an *invocation handler*, an object of a class that implements the `InvocationHandler` interface. That interface has a single method:

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

Whenever a method is called on the proxy object, the `invoke` method of the invocation handler gets called, with the `Method` object and arguments of the original call. The invocation handler must then figure out how to handle the call. There are many possible actions an invocation handler might take, such as routing calls to remote servers or tracing calls for debugging purposes.

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

- A class loader (see [Section 4.6.3](#)), or `null` to use the default class loader
- An array of `Class` objects, one for each interface to be implemented
- The invocation handler

To show the mechanics of proxies, here is an example where an array is populated with proxies for `Integer` objects, forwarding calls to the original objects after printing trace messages:

```
var values = new Object[1000];

for (int i = 0; i < values.length; i++) {
    var value = Integer.valueOf(i);
    values[i] = Proxy.newProxyInstance(
        null,
        value.getClass().getInterfaces(),
        // Lambda expression for invocation handler
        (Object proxy, Method m, Object[] args) -> {
            System.out.println(value + "." + m.getName() +
                Arrays.toString(args));
            return m.invoke(value, args);
        });
}
```

When calling

```
Arrays.binarySearch(values, Integer.valueOf(500));
```

the following output is produced:

```
499.compareTo[500]
749.compareTo[500]
624.compareTo[500]
561.compareTo[500]
530.compareTo[500]
514.compareTo[500]
506.compareTo[500]
502.compareTo[500]
500.compareTo[500]
```

You can see how the binary search algorithm homes in on the key by cutting the search interval in half in every step.

The point is that the `compareTo` method is invoked through the proxy, even though this was not explicitly mentioned in the code. All methods in any interfaces implemented by `Integer` are proxied.



Caution: When the invocation handler is called with a method call that has no parameters, the argument array is `null`, not an `Object[]` array of length `0`. That is utterly reprehensible and not something you should do in your own code.

Exercises

1. Define a class `Point` with a constructor `public Point(double x, double y)` and accessor methods `getX`, `getY`. Define a subclass `LabeledPoint` with a constructor `public LabeledPoint(String label, double x, double y)` and an accessor method `getLabel`.
2. Define `toString`, `equals`, and `hashCode` methods for the classes of the preceding exercise.
3. Make the instance variables `x` and `y` of the `Point` class in [Exercise 2](#) `protected`. Show that the `LabeledPoint` class can access these variables only in `LabeledPoint` instances.
4. Define an abstract class `Shape` with an instance variable of class `Point`, a constructor, a concrete method `public void moveBy(double dx, double dy)` that moves the point by the given amount, and an abstract method `public Point getCenter()`. Provide concrete subclasses `Circle`, `Rectangle`, `Line` with constructors `public Circle(Point center, double radius)`, `public Rectangle(Point topLeft, double width, double height)`, and `public Line(Point from, Point to)`.
5. Define `clone` methods for the classes of the preceding exercise.
6. Suppose that in [Section 4.3.2](#), the `Item.equals` method uses an `instanceof` test. Implement `DiscountedItem.equals` so that it compares only the superclass if `otherObject` is an `Item`, but also

includes the discount if it is a `DiscountedItem`. Show that this method preserves symmetry but fails to be *transitive*—that is, find a combination of items and discounted items so that `x.equals(y)` and `y.equals(z)`, but not `x.equals(z)`.

7. Define a method `Object add(Object first, Object second)`. If the arguments are instances of `Number`, add the values. If they are instances of `Boolean`, return `Boolean.TRUE` if either one is true. Otherwise, concatenate them as strings. Use the `instanceof` operator with pattern matching.
8. Define an enumeration type for the eight combinations of primary colors `BLACK`, `RED`, `BLUE`, `GREEN`, `CYAN`, `MAGENTA`, `YELLOW`, `WHITE` with methods `getRed`, `getGreen`, and `getBlue`.
9. Conceptually, a list is either empty, or it has a head element and a tail, which is again a list. Use sealed classes to model lists of strings. Provide a `StringList` interface with exactly two subclasses `EmptyStringList` and `NonEmptyStringList`. Using exhaustive pattern matching, implement methods `size` and `append`.
10. The `Class` class has six methods that yield a string representation of the type represented by the `Class` object. How do they differ when applied to arrays, generic types, inner classes, and primitive types?
11. Write a “universal” `toString` method that uses reflection to yield a string with all instance variables of an object. Extra credit if you can handle cyclic references.
12. Use the `MethodPrinter` program in [Section 4.7.1](#) to enumerate all methods of the `int[]` class. Extra credit if you can identify the one method (discussed in this chapter) that is wrongly described.
13. Write the “Hello, World” program, using reflection to look up the `out` field of `java.lang.System` and using `invoke` to call the `println` method.
14. Measure the performance difference between a regular method call and a method call via reflection.
15. Write a method that prints a table of values for any `Method` representing a static method with a parameter of type `double` or `Double`. Besides the `Method` object, accept a lower bound, upper bound, and step size. Demonstrate your method by printing tables for `Math.sqrt` and `Double.toHexString`. Repeat, using a `DoubleFunction<Object>`

instead of a **Method**. Contrast the safety, efficiency, and convenience of both approaches.

Chapter 5 ■ Exceptions, Assertions, and Logging

In many programs, dealing with the unexpected can be more complex than implementing the “happy day” scenarios. Like most modern programming languages, Java has a robust exception-handling mechanism for transferring control from the point of failure to a competent handler. In addition, the `assert` statement provides a structured and efficient way of expressing internal assumptions. Finally, you will see how to use the logging API to keep a record of the various events, be they routine or suspicious, in the execution of your programs.

The key points of this chapter are:

1. When you throw an exception, control is transferred to the nearest handler of the exception.
2. In Java, checked exceptions are tracked by the compiler.
3. Use the `try/catch` construct to handle exceptions.
4. The `try-with-resources` statement automatically closes resources after normal execution or when an exception occurred.
5. Use the `try/finally` construct to deal with other actions that must occur whether or not execution proceeded normally.
6. You can catch and rethrow an exception, or chain it to another exception.
7. A stack trace describes all method calls that are pending at a point of execution.
8. An assertion checks a condition, provided that assertion checking is enabled for the class, and throws an error if the condition is not fulfilled.
9. The platform logging API is a simple logging frontend that is a part of the JDK.
10. The `java.util.logging` package provides a simple logging backend.

5.1. Exception Handling

What should a method do when it encounters a situation in which it cannot fulfill its contract? The traditional answer was that the method should return some error code. But that is cumbersome for the programmer calling the method. The caller is obliged to check for errors, and if it can't handle them, return an error code to its own caller. Not unsurprisingly, programmers didn't always check and propagate return codes, and errors went undetected, causing havoc later.

Instead of having error codes bubble up the chain of method calls, Java supports *exception handling* where a method can signal a serious problem by “throwing” an exception. One of the methods in the call chain, but not necessarily the direct caller, is responsible for handling the exception by “catching” it. The fundamental advantage of exception handling is that it decouples the processes of detecting and handling errors. In the following sections, you will see how to work with exceptions in Java.

5.1.1. Throwing Exceptions

A method may find itself in a situation where it cannot carry out the task at hand. Perhaps a required resource is missing, or it was called with inconsistent arguments. In such a case, it is best

to throw an exception.

Suppose you implement a method that yields a random integer between two bounds:

```
public static int randInt(int low, int high) {  
    return low + (int) (Math.random() * (high - low + 1));  
}
```

What should happen if someone calls `randInt(10, 5)`? Trying to fix this is probably not a good idea because the caller might have been confused in more than one way. Instead, throw an appropriate exception:

```
if (low > high)  
    throw new IllegalArgumentException(  
        "low should be <= high but low is %d and high is %d".formatted(low,  
        high));
```

As you can see, the `throw` statement is used to “throw” an object of a class—here, `IllegalArgumentException`. The object is constructed with a debugging message. You will see in the next section how to pick an appropriate exception class.

When a `throw` statement executes, the normal flow of execution is interrupted immediately. The `randInt` method stops executing and does not return a value to its caller. Instead, control is transferred to a handler, as you will see in [Section 5.1.4](#).

5.1.2. The Exception Hierarchy

[Figure 5.1](#) shows the hierarchy of exceptions in Java. All exceptions are subclasses of the class `Throwable`. Subclasses of `Error` are exceptions that are thrown when something exceptional happens that the program cannot be expected to handle, such as memory exhaustion. There is not much you can do about errors other than giving a message to the user that things have gone very wrong.

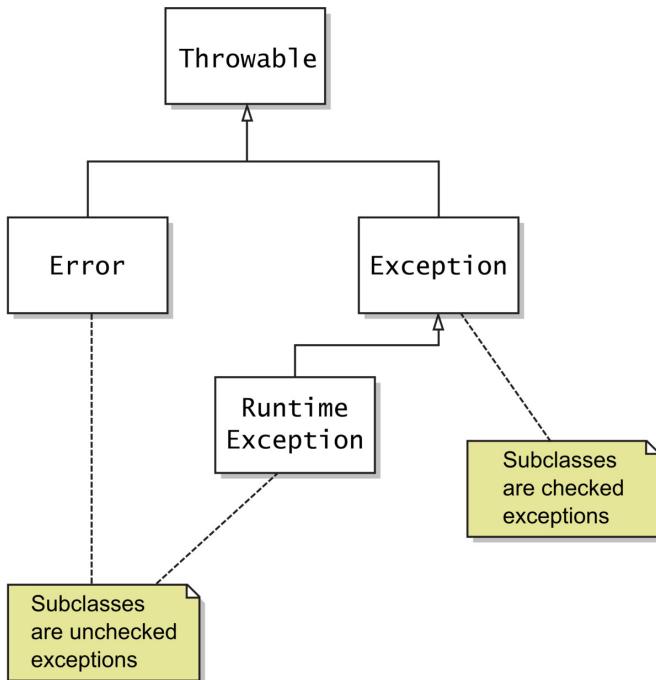


Figure 5.1: The exception hierarchy

Programmer-reported exceptions are subclasses of the class `Exception`. These exceptions fall into two categories:

- *Unchecked* exceptions are subclasses of `RuntimeException`.
- All other exceptions are *checked* exceptions.

As you will see in the next section, programmers must either catch checked exceptions or declare them in the method header. The compiler checks that these exceptions are handled properly.



Note: The name `RuntimeException` is unfortunate. Of course, all exceptions occur at runtime. However, the exceptions that are subclasses of `RuntimeException` are not checked during compilation.

Checked exceptions are used in situations where failure should be anticipated. One common reason for failure is input and output. Files may be damaged, and network connections may fail. A number of exception classes extend `IOException`, and you should use an appropriate one to report any errors that you encounter. For example, when a file that should be there turns out not be, throw a `FileNotFoundException`.

Unchecked exceptions indicate logic errors caused by programmers, not by unavoidable external risks. For example, a `NullPointerException` is not checked. Just about any method might throw

one, and programmers shouldn't spend time on catching them. Instead, they should make sure that no `null`s are dereferenced in the first place.

Sometimes, implementors need to use their judgment to make a distinction between checked and unchecked exceptions. Consider the call `Integer.parseInt(str)`. It throws an unchecked `NumberFormatException` when `str` doesn't contain a valid integer. On the other hand, `Class.forName(str)` throws a checked `ClassNotFoundException` when `str` doesn't contain a valid class name.

Why the difference? The reason is that it is possible to check whether a string is a valid integer before calling `Integer.parseInt`, but it is not possible to know whether a class can be loaded until you actually try to load it.

The Java API provides many exception classes, such as `IOException`, `IllegalArgumentException`, and so on. You should use these when appropriate. However, if none of the standard exception classes is suitable for your purpose, you can create your own by extending `Exception`, `RuntimeException`, or another existing exception class.

When you do so, it is a good idea to supply both a no-argument constructor and a constructor with a message string. For example,

```
public class FileFormatException extends IOException {  
    public FileFormatException() {}  
    public FileFormatException(String message) {  
        super(message);  
    }  
    // Also add constructors for chained exceptions—see Section 5.1.7  
}
```

5.1.3. Declaring Checked Exceptions

Any method that might give rise to a checked exception must declare it in the method header with a `throws` clause:

```
public void write(Object obj, String filename)  
    throws IOException, ReflectiveOperationException
```

List the exceptions that the method might throw, either because of a `throw` statement or because it calls another method with a `throws` clause.

In the `throws` clause, you can combine exceptions into a common superclass. Whether or not that is a good idea depends on the exceptions. For example, if a method can throw multiple subclasses of `IOException`, it makes sense to cover them all in a clause `throws IOException`. But if the exceptions are unrelated, don't combine them into `throws Exception`—that would defeat the purpose of exception checking.



Tip: Some programmers think it is shameful to admit that a method might throw an exception. Wouldn't it be better to handle it instead? Actually, the opposite is true. You should allow each exception to find its way to a competent handler. The golden rule of exceptions is, "Throw early, catch late."

When you override a method, it cannot throw more checked exceptions than those declared by the superclass method. For example, if you extend the `write` method from the beginning of this section, the overriding method can throw fewer exceptions:

```
public void write(Object obj, String filename)
    throws FileNotFoundException
```

But if the method tried to throw an unrelated checked exception, such as an `InterruptedException`, it would not compile.



Caution: If the superclass method has no `throws` clause, then no overriding method can throw a checked exception.

You can use the javadoc `@throws` tag to document when a method throws a (checked or unchecked) exception. Most programmers only do this when there is something meaningful to document. For example, there is little value in telling users that an `IOException` is thrown when there is a problem with input/output. But comments such as the following can be meaningful:

```
@throws NullPointerException if filename is null
@throws FileNotFoundException if there is no file with name filename
```



Note: You never specify the exception type of a lambda expression. However, if a lambda expression can throw a checked exception, you can only pass it to a functional interface whose method declares that exception. For example, the call

```
list.forEach(obj -> write(obj, "output.dat"));
```

is an error. The parameter type of the `forEach` method is the functional interface

```
public interface Consumer<T> {
    void accept(T t);
}
```

The `accept` method is declared not to throw any checked exception.

5.1.4. Catching Exceptions

To catch an exception, set up a `try` block. In its simplest form, it looks like this:

```
try {  
    statements  
} catch (ExceptionClass ex) {  
    handler  
}
```

If an exception of the given class occurs as the statements in the `try` block are executed, control transfers to the handler. The exception variable (`ex` in our example) refers to the exception object which the handler can inspect if desired.

There are two modifications that you can make to this basic structure. You can have multiple handlers for different exception classes:

```
try {  
    statements  
} catch (ExceptionClass1 ex) {  
    handler1  
} catch (ExceptionClass2 ex) {  
    handler2  
} catch (ExceptionClass3 ex) {  
    handler3  
}
```

The `catch` clauses are matched top to bottom, so the most specific exception classes must come first.

Alternatively, you can share one handler among multiple exception classes:

```
try {  
    statements  
} catch (ExceptionClass1 | ExceptionClass2 | ExceptionClass3 ex) {  
    handler  
}
```

In that case, the handler can only call those methods on the exception variable that belong to all exception classes.

5.1.5. The Try-with-Resources Statement

One problem with exception handling is resource management. Suppose you write to a file and close it when you are done:

```
ArrayList<String> lines = ...;
var out = new PrintWriter("output.txt");
for (String line : lines) {
    out.println(line.toLowerCase());
}
out.close();
```

This code has a hidden danger. If any method throws an exception, the call to `out.close()` never happens. That is bad. Output could be lost, or if the exception is triggered many times, the system could run out of file handles.

A special form of the `try` statement can solve this issue. You can specify *resources* in the header of the `try` statement. A resource must belong to a class implementing the `AutoCloseable` interface. You can declare variables in the `try` block header:

```
ArrayList<String> lines = ...;
try (var out = new PrintWriter("output.txt")) { // Variable declaration
    for (String line : lines)
        out.println(line.toLowerCase());
}
```

Alternatively, you can provide previously declared effectively final variables in the header:

```
var out = new PrintWriter("output.txt");
try (out) { // Effectively final variable
    for (String line : lines)
        out.println(line.toLowerCase());
}
```

The `AutoCloseable` interface has a single method

```
public void close() throws Exception
```



Note: There is also a `Closeable` interface. It is a subinterface of `AutoCloseable`, also with a single `close` method. However, that method is declared to throw an `IOException`.

When the `try` block exits, either because its end is reached normally or because an exception is thrown, the `close` methods of the resource objects are invoked. For example:

```
try (var out = new PrintWriter("output.txt")) {
    for (String line : lines) {
        out.println(line.toLowerCase());
    }
} // out.close() called here
```

You can declare multiple resources, separated by semicolons. Here is an example with two resource declarations:

```
try (var in = new Scanner(Path.of("/usr/share/dict/words"));
     var out = new PrintWriter("output.txt")) {
    while (in.hasNext())
        out.println(in.next().toLowerCase());
}
```

The resources are closed in reverse order of their initialization—that is, `out.close()` is called before `in.close()`.

Suppose that the `PrintWriter` constructor throws an exception. Now `in` is already initialized but `out` is not. The `try` statement does the right thing: calls `in.close()` and propagates the exception.

Some `close` methods can throw exceptions. If that happens when the `try` block completed normally, the exception is thrown to the caller. However, if another exception had been thrown, causing the `close` methods of the resources to be called, and one of them throws an exception, that exception is likely to be of lesser importance than the original one.

In this situation, the original exception gets rethrown, and the exceptions from calling `close` are caught and attached as “suppressed” exceptions. This is a very useful mechanism that would be tedious to implement by hand (see [Exercise 10](#)). When you catch the primary exception, you can retrieve the secondary exceptions by calling the `getSuppressed` method:

```
try {
    ...
} catch (IOException ex) {
    Throwable[] secondaryExceptions = ex.getSuppressed();
    ...
}
```

If you want to implement such a mechanism yourself in a (hopefully rare) situation when you can't use the try-with-resources statement, call `ex.addSuppressed(secondaryException)`.

A try-with-resources statement can optionally have `catch` clauses that catch any exceptions in the statement.

5.1.6. The `finally` Clause

As you have seen, the try-with-resources statement automatically closes resources whether or not an exception occurs. Sometimes, you need to clean up something that isn't an `AutoCloseable`. In that case, use the `finally` clause:

```
try {  
    Do work  
} finally {  
    Clean up  
}
```

The `finally` clause is executed when the `try` block comes to an end, either normally or due to an exception.

This pattern occurs whenever you need to acquire and release a lock, or increment and decrement a counter, or push something on a stack and pop it off when you are done. You want to make sure that these actions happen regardless of what exceptions might be thrown.

You should avoid throwing an exception in the `finally` clause. If the body of the `try` block was terminated due to an exception, it is masked by an exception in the `finally` clause. The suppression mechanism that you saw in the preceding section only works for try-with-resources statements.

Similarly, a `finally` clause should not contain a `return` statement. If the body of the `try` block also has a `return` statement, the one in the `finally` clause replaces the return value. Consider this example:

```
public static int parseInt(String s) {  
    try {  
        return Integer.parseInt(s);  
    } finally {  
        return 0; // Error  
    }  
}
```

It looks as if in the call `parseInt("42")`, the body of the `try` block returns the integer 42. However, the `finally` clause is executed before the method actually returns and causes the method to return 0, ignoring the original return value.

And it gets worse. Consider the call `parseInt("zero")`. The `Integer.parseInt` method throws a `NumberFormatException`. Then the `finally` clause is executed, and the `return` statement swallows the exception!



Tip: The body of the `finally` clause is intended for cleaning up resources. Don't put statements that change the control flow (`return`, `throw`, `break`, `continue`) inside a `finally` clause.

It is possible to form `try` statements with `catch` clauses followed by a `finally` clause. But you have to be careful with exceptions in the `finally` clause. For example, have a look at this `try` block adapted from an online tutorial:

```

BufferedReader in = null;
try {
    in = Files.newBufferedReader(path);
    Read from in
} catch (IOException ex) {
    System.err.println("Caught IOException: " + ex.getMessage());
} finally {
    if (in != null) {
        in.close(); // Caution—might throw an exception
    }
}

```

The programmer clearly thought about the case when the `Files.newBufferedReader` method throws an exception. It appears as if this code would catch and print all I/O exceptions, but it actually misses one: the one that might be thrown by `in.close()`. It is often better to rewrite a complex `try/catch/finally` statement as a try-with-resources statement or by nesting a `try/finally` inside a `try/catch` statement—see [Exercise 12](#).

5.1.7. Rethrowing and Chaining Exceptions

When an exception occurs, you may not know what to do about it, but you may want to log the failure. In that case, rethrow the exception so that a competent handler can deal with it:

```

try {
    Do work
}
catch (Exception ex) {
    logger.log(level, message, ex);
    throw ex;
}

```



Note: Something subtle is going on when this code is inside a method that may throw a checked exception. Suppose the enclosing method is declared as

```
public void read(String filename) throws IOException
```

At first glance, it looks as if one would need to change the `throws` clause to `throws Exception`. However, the Java compiler carefully tracks the flow and realizes that `ex` could only have been an exception thrown by one of the statements in the `try` block, not an arbitrary `Exception`.

Sometimes, you want to change the class of a thrown exception. For example, you may need to report a failure of a subsystem with an exception class that makes sense to the user of the subsystem. Suppose you encounter a database error in a servlet. The code that executes the servlet

may not want to know in detail what went wrong, but it definitely wants to know that the servlet is at fault. In this case, catch the original exception and chain it to a higher-level one:

```
try {
    Access the database
}
catch (SQLException ex) {
    throw new ServletException("database error", ex);
}
```

When the `ServletException` is caught, the original exception can be retrieved as follows:

```
Throwable cause = ex.getCause();
```

The `ServletException` class has a constructor that takes as a parameter the cause of the exception. Not all exception classes do that. In that case, you have to call the `initCause` method, like this:

```
try {
    Access the database
}
catch (SQLException ex) {
    var ex2 = new CruftyOldException("database error");
    ex2.initCause(ex);
    throw ex2;
}
```

If you provide your own exception class, you should provide, in addition to the two constructors described in [Section 5.1.2](#), the following constructors:

```
public class FileFormatException extends IOException {
    ...
    public FileFormatException(Throwable cause) { initCause(cause); }
    public FileFormatException(String message, Throwable cause) {
        super(message);
        initCause(cause);
    }
}
```



Tip: The chaining technique is also useful if a checked exception occurs in a method that is not allowed to throw a checked exception. You can catch the checked exception and chain it to an unchecked one.

5.1.8. Uncaught Exceptions and the Stack Trace

If an exception is not caught anywhere, a *stack trace* is displayed—a listing of all pending method calls at the point where the exception was thrown. The stack trace is sent to `System.err`, the stream for error messages.

If you want to save the exception somewhere else, perhaps for inspection by your tech support staff, set the default uncaught exception handler:

```
Thread.setDefaultUncaughtExceptionHandler((thread, ex) -> {  
    Record the exception  
});
```



Note: An uncaught exception terminates the thread in which it occurred. If your application only has one thread (which is the case for the programs that you have seen so far), the program exits after invoking the uncaught exception handler.

Sometimes, you are forced to catch an exception and don't really know what to do with it. For example, the `Class.forName` method throws a checked exception that you need to handle. Instead of ignoring the exception, at least print the stack trace:

```
try {  
    Class<?> cl = Class.forName(className);  
    ...  
} catch (ClassNotFoundException ex) {  
    ex.printStackTrace();  
}
```

If you want to store the stack trace of an exception, you can put it into a string as follows:

```
var out = new ByteArrayOutputStream();  
ex.printStackTrace(new PrintWriter(out));  
String description = out.toString();
```



Note: If you need to process the stack trace in more detail, use the `StackWalker` class. For example, the following prints all stack frames:

```
StackWalker walker = StackWalker.getInstance();  
walker.forEach(frame -> System.out.println("Frame: " + frame));
```

You can also analyze the `StackWalker.StackFrame` instances in detail. See the API documentation for details.

5.1.9. API Methods for Throwing Exceptions

The `Objects` class has a method for convenient null checks of parameters. Here is a sample usage:

```
public void process(String direction) {  
    this.direction = Objects.requireNonNull(direction);  
    ...  
}
```

If `direction` is `null`, a `NullPointerException` is thrown. At first glance, this seems a bit pointless since a `NullPointerException` would eventually be thrown when the value is used. But consider working back from a stack trace. When you see a call to `requireNonNull` as the culprit, you know right away what you did wrong.

You can also supply a message string for the exception:

```
this.direction = Objects.requireNonNull(direction, "direction must not be  
null");
```

A variant of this method allows you to supply an alternate value instead of throwing an exception:

```
this.direction = Objects.requireNonNullElse(direction, "North");
```

To compute the alternate value only when needed, use the following variant:

```
this.directory = Objects.requireNonNullElseGet(directory,  
    () -> System.getProperty("user.dir"));
```

The lambda expression is only evaluated if `directory` is `null`.

There is also a convenience method for making a range check:

```
this.index = Objects.checkIndex(index, length);
```

The method returns the index if it is between 0 and `length - 1`, and throws an `IndexOutOfBoundsException` otherwise.

5.2. Assertions

Assertions are a commonly used idiom of defensive programming. Suppose you are convinced that a particular property is fulfilled, and you rely on that property in your code. For example, you may be computing

```
double y = Math.sqrt(x);
```

You are certain that `x` is not negative. Still, you want to double-check rather than have “not a number” floating-point values creep into your computation. You could, of course, throw an exception:

```
if (x < 0) throw new IllegalStateException(x + " < 0");
```

But this condition stays in the program, even after testing is complete, slowing it down. The assertion mechanism allows you to put in checks during testing and to have them automatically removed in the production code.



Note: In Java, assertions are intended as a debugging aid for validating internal assumptions, not as a mechanism for enforcing contracts. For example, if you want to report an inappropriate argument of a public method, don't use an assertion but throw an `IllegalArgumentException`.

5.2.1. Using Assertions

There are two forms of the assertion statement in Java:

```
assert condition;  
assert condition : expression;
```

The `assert` statement evaluates the condition and throws an `AssertionError` if it is false. In the second form, the expression is turned into a string that becomes the message of the error object.



Note: If the expression is a `Throwable`, it is also set as the cause of the assertion error (see [Section 5.1.7](#)).

For example, to assert that `x` is non-negative, you can simply use the statement

```
assert x >= 0;
```

Or you can pass the actual value of `x` into the `AssertionError` object so it gets displayed later:

```
assert x >= 0 : x;
```

5.2.2. Enabling and Disabling Assertions

By default, assertions are disabled. Enable them by running the program with the `-enableassertions` or `-ea` option:

```
java -ea MainClass
```

You do not have to recompile your program because enabling or disabling assertions is handled by the class loader. When assertions are disabled, the class loader strips out the assertion code so that it won't slow execution. You can even enable assertions in specific classes or in entire packages, for example:

```
java -ea:MyClass -ea:com.mycompany.mylib... MainClass
```

This command turns on assertions for the class `MyClass` and all classes in the `com.mycompany.mylib` package *and its subpackages*. The option `-ea...` turns on assertions in all classes of the default package.

You can also disable assertions in certain classes and packages with the `-disableassertions` or `-da` option:

```
java -ea:... -da:MyClass MainClass
```

When you use the `-ea` and `-da` switches to enable or disable all assertions (and not just specific classes or packages), they do not apply to the “system classes” that are loaded without class loaders. Use the `-enablesystemassertions/-esa` switch to enable assertions in system classes.

It is also possible to programmatically control the assertion status of class loaders with the following methods:

```
void ClassLoader.setDefaultAssertionStatus(boolean enabled);  
void ClassLoader.setClassAssertionStatus(String className, boolean enabled);  
void ClassLoader.setPackageAssertionStatus(String packageName, boolean enabled);
```

As with the `-enableassertions` command-line option, the `setPackageAssertionStatus` method sets the assertion status for the given package and its subpackages.

5.3. Logging

Every Java programmer is familiar with the process of inserting `System.out.println` calls into troublesome code to gain insight into program behavior. Of course, once you have figured out the cause of trouble, you remove the print statements—only to put them back in when the next problem surfaces. Logging frameworks are designed to overcome this problem.

5.3.1. Should You Use the Java Logging Framework?

Java has a standard logging framework, usually called after its package name `java.util.logging`, and sometimes abbreviated as `j.u.l`. However, other logging frameworks have more features and are in common use, such as Log4j (<https://logging.apache.org/log4j/2.x>) and Logback (<https://logback.qos.ch>).

If you want to give users of your code the choice of logging framework, then you should use a “façade” library that sends log messages to the preferred framework. A commonly used façade with a pleasant API is SLF4J (<https://www.slf4j.org>). Another façade is the “platform logging API” (also known as JEP 246). It is very basic but a part of the JDK. The façade is sometimes called the *frontend*. It provides the API that programmers uses to log messages. The

backend is in charge of filtering and formatting the messages, and putting them somewhere. The backend needs to be configurable by deployers, usually by editing configuration files.

In the following sections, I will show you how to use the platform logging API as frontend and `java.util.logging` as a backend. This can be a reasonable choice if you find the frontend API sufficient, since you can always swap out the backend.

The `java.util.logging` backend has fewer features than its more popular alternatives, but it suffices for many use cases. Because of its simplicity, it is less susceptible to attacks. In contrast, obscure features of Log4j allowed hackers to craft program inputs that, when logged, caused malicious code execution.

Whether or not you end up using them, studying the platform logging API and `java.util.logging` backend gives you a good foundation of the capabilities of logging frameworks.

5.3.2. Logging 101

Platform loggers implement the `System.Logger` interface. Each logger has a name. The name can be arbitrary, but it is often the package name of the class whose methods generate logging messages. You get a platform logger like this:

```
System.Logger logger = System.getLogger("com.mycompany.myapp");
```

When you request a logger with a given name for the first time, it is created. Subsequent calls to the same name yield the same logger object.

Now you are ready to log:

```
logger.log(System.Logger.Level.INFO, "Opening file " + filename);
```

The record is printed like this:

```
Aug 04, 2022 09:53:34 AM com.mycompany.myapp read
INFO: Opening file data.txt
```

Note that the time and the names of the calling class and method are automatically included.

To turn off these informational messages when your program is deployed, you can configure the backend. In the case of the `java.util.logging` backend, prepare a file `logging.properties` with the following contents:

```
handlers=java.util.logging.ConsoleHandler
com.mycompany.myapp.level=WARNING
```

Then start the application like this:

```
java -Djava.util.logging.config.file=logging.properties  
com.mycompany.myapp.Main
```

Since the `INFO` level is below the `WARNING` level, the message no longer shows up.

The API for getting the logger and logging a message are part of the frontend—in this case, the platform logging API. If you use a different frontend, the API will be different.

The message destination, formatting, and filtering, as well as the mechanisms for the configuration are part of the backend—here, `java.util.logging`. If you use a different backend, follow its instructions for configuration.

5.3.3. The Platform Logging API

As you saw in the preceding section, each logged message has a level. The enumeration `System.Logger.Level` has the following values, in decreasing severity: `ERROR`, `WARNING`, `INFO`, `DEBUG`, and `TRACE`.



Tip: With the import statement

```
import static java.lang.System.Logger.Level.*;
```

you can shorten the levels:

```
logger.log(INFO, "Opening file " + filename);  
// Instead of System.Logger.Level.INFO
```

In the above example, the message `"Opening file " + filename` is created even if the message is suppressed. If you are concerned about the cost of creating the message string, you can use a lambda expression instead:

```
logger.log(INFO, () -> "Opening file " + filename);
```

Then the message is only computed when it is actually logged.

It is common to log an exception:

```
catch (IOException ex) {  
    logger.log(WARNING, "Cannot open file " + filename, ex);  
}
```

You can format the message using a pattern:

```
logger.log(WARNING, "Cannot open file {0}", filename);
```

 **Caution:** This is *not* a printf style pattern. The pattern is processed by the `MessageFormat` class that is used for localization of program messages. See Volume II for details. Here, you simply need to know that string placeholders are `{0}`, `{1}`, and so on. Braces must be escaped with single quotes: `'{', '}'`. Use two single quotes `''` to include a literal single quote in the message.

Log messages can be localized to different languages, using the resource bundle mechanism that is explained in the JavaDoc of the `ResourceBundle` class.

Supply the bundle and the key for the formatting string:

```
logger.log(WARNING, bundle, "file.bad", filename);
// Looks up file.bad in the bundle
```

Alternatively, get the logger as

```
System.Logger logger = System.getLogger("com.mycompany.myapp", bundle);
```

When calling one of the logger methods with a `String` parameter for the message or format, the argument is interpreted as a key in the bundle. The methods with `Object` or `Supplier<String>` parameters are unaffected.

Some, but not all combinations of these features (deferred message computation, adding a throwable, formatting, using a bundle), are supported. [Table 5.1](#) shows the complete API.

Table 5.1: The `System.Logger` API

Method	Description
<code>String getName()</code>	The name of this logger.
<code>boolean isLoggable(System.Logger.Level level)</code>	<code>true</code> if this logger processes logs at the given level.
<code>void log(System.Logger.Level level, String msg)</code>	Logs the given message.
<code>void log(System.Logger.Level level, Object obj)</code>	Logs the message <code>obj.toString()</code> .

Method	Description
<pre data-bbox="279 287 677 466">void log(System.Logger.Level level, String msg, Throwable thrown)</pre>	Logs the given message and throwable.
<pre data-bbox="279 519 644 682">void log(System.Logger.Level level, Supplier<String> msgSupplier)</pre>	If this logger processes logs at the given level, invokes the supplier and logs the result.
<pre data-bbox="279 745 644 929">void log(System.Logger.Level level, Supplier<String> msgSupplier, Throwable thrown)</pre>	If this logger processes logs at the given level, invokes the supplier and logs the result and throwable.
<pre data-bbox="279 992 644 1151">void log(System.Logger.Level level, String format, Object... params)</pre>	Logs the formatted message with the given parameters.
<pre data-bbox="279 1210 687 1410">void log(System.Logger.Level level, ResourceBundle bundle, String msg, Throwable thrown)</pre>	Logs the message in the bundle with key <code>msg</code> (or, if <code>bundle</code> is <code>null</code> , <code>msg</code> itself) and the throwable.
<pre data-bbox="279 1469 687 1670">void log(System.Logger.Level level, ResourceBundle bundle, String format, Object... params)</pre>	Looks up the format with key <code>format</code> in the bundle (or, if <code>bundle</code> is <code>null</code> , uses <code>format</code> itself), and logs the formatted message with the given parameters.

5.3.4. Logging Configuration

Let us now turn to the logging backend. As already mentioned, the default backend of the platform logging API is `java.util.logging`. The information in the following sections is

specific to that backend.

You can change various properties of the backend by editing a configuration file. The default configuration file is located at `conf/logging.properties` in the JDK. To use another file, set the `java.util.logging.config.file` property to the file location by starting your application with

```
java -Djava.util.logging.config.file=configFile MainClass
```



Caution: Calling `System.setProperty("java.util.logging.config.file", configFile)` in `main` has no effect because the log manager is initialized during VM startup, before `main` executes.

You can specify the logging levels for your own loggers by adding lines such as

```
com.mycompany.myapp.level=WARNING
```

That is, append the `.level` suffix to the logger name.



Note: Properties in the logging configuration are not system properties. You cannot set them with the `-D` command-line option. Instead, place the logging properties into a file that you specify with `-Djava.util.logging.config.file=configFile`.

You can also specify the root level:

```
.level=WARNING
```



Caution: For historical reasons, some of the levels have different names in the platform logging API and the `java.util.logging` framework. [Table 5.2](#) shows the correspondences.

Table 5.2: Corresponding Levels for Platform Logging and the Java Logging Framework

Platform logging	<code>java.util.logging</code>
ERROR	SEVERE
WARNING	WARNING
INFO	INFO
DEBUG	FINE

Platform logging	java.util.logging
TRACE	FINER

Similar to package names, logger names are hierarchical. In fact, they are more hierarchical than packages. There is no semantic relationship between a package and its parent, but logger parents and children share certain properties. For example, if you turn off messages to the logger "com.mycompany", then its child loggers are also deactivated.

As you will see in the next section, loggers don't actually send the messages to the console—that is the job of the handlers. Handlers also have levels. To see DEBUG/FINE messages on the console, you also need to set

```
java.util.logging.ConsoleHandler.level=FINE
```



Caution: The settings in the log manager configuration are not system properties. Starting a program with -Dcom.mycompany.myapp.level=FINE does not have any effect on the logger.

It is also possible to change logging levels in a running program by using the `jconsole` program. For details, see www.oracle.com/technetwork/articles/java/jconsole-1564139.xhtml#LoggingControl.

5.3.5. Log Handlers

The backend of the `java.util.logging` API is based on *handlers*. The simplest handler is the `ConsoleHandler` that prints log records to the `System.err` stream.

Each handler has a parent. By default, the handler simply sends each record to the parent handler. The standard logging configuration uses a `ConsoleHandler` by default. Its default logging level is `INFO`.

To send log records elsewhere, add another handler. The `java.util.logging` API provides two handlers for this purpose: a `FileHandler` and a `SocketHandler`. The `SocketHandler` sends records to a specified host and port. Of greater interest is the `FileHandler` that collects records in a file.

To add a file handler, provide this entry in the logging properties:

```
handlers=java.util.logging.ConsoleHandler,java.util.logging.FileHandler
```

The records are sent to a file `javan.log` in the user's home directory, where *n* is a number to make the file unique. By default, the records are formatted in XML. A typical log record has the form

```

<record>
  <date>2014-08-04T09:53:34</date>
  <millis>1407146014072</millis>
  <sequence>1</sequence>
  <logger>com.mycompany.myapp</logger>
  <level>INFO</level>
  <class>com.horstmann.corejava.Employee</class>
  <method>read</method>
  <thread>10</thread>
  <message>Opening file staff.txt</message>
</record>

```

You can modify the default behavior of the file handler by setting various parameters in the log manager configuration (see [Table 5.3](#)).

You probably don't want to use the default log file name. Use a pattern such as %h/myapp.log (see [Table 5.4](#) for an explanation of the pattern variables.)

If multiple applications (or multiple copies of the same application) use the same log file, you should turn the append flag on. Alternatively, use %u in the file name pattern so that each application creates a unique copy of the log.

It is also a good idea to turn file rotation on. Log files are kept in a rotation sequence, such as myapp.log.0, myapp.log.1, myapp.log.2, and so on. Whenever a file exceeds the size limit, the oldest log is deleted, the other files are renamed, and a new file with generation number 0 is created.

Table 5.3: File Handler Configuration Parameters

Configuration property	Description	Default
java.util.logging.FileHandler.level	The handler level.	Level.ALL
java.util.logging.FileHandler.append	When true, log records are appended to an existing file; otherwise, a new file is opened for each program run.	false
java.util.logging.FileHandler.limit	The approximate	0 in the FileHandler class, 50000 in the

Configuration property	Description	Default
	maximum number of bytes to write in a file before opening another ($0 =$ no limit).	default log manager configuration
java.util.logging.FileHandler.pattern	The file name pattern (see Table 5.4).	%h/java%u.log
java.util.logging.FileHandler.count	The number of logs in a rotation sequence.	1 (no rotation)
java.util.logging.FileHandler.filter	The filter for filtering log records (see Section 5.3.6).	No filtering
java.util.logging.FileHandler.encoding	The character encoding.	The platform character encoding
java.util.logging.FileHandler.formatter	The formatter for each log record.	java.util.logging.XMLFormatter

Table 5.4: Log File Pattern Variables

Variable	Description
%h	The user's home directory (the <code>user.home</code> property).
%t	The system's temporary directory.
%u	A unique number.
%g	The generation number for rotated logs. A <code>.%g</code> suffix is used if rotation is specified and the pattern doesn't contain <code>%g</code> .
%%	The percent character.

5.3.6. Filters and Formatters

Besides filtering by logging levels, each handler can have an additional filter that implements the `Filter` interface, a functional interface with a method

```
boolean isLoggable(LogRecord record)
```

To install a filter into a handler, add an entry such as the following into the logging configuration:

```
java.util.logging.ConsoleHandler.filter=com.mycompany.myapp.MyFilter
```

The `ConsoleHandler` and `FileHandler` classes emit the log records in text and XML formats. However, you can define your own formats as well. Extend the `java.util.logging.Formatter` class and override the method

```
String format(LogRecord record)
```

Format the record in any way you like and return the resulting string. In your `format` method, you can get information about the `LogRecord` by calling one of the methods in [Table 5.5](#).

In your `format` method, you may want to call the method

```
String formatMessage(LogRecord record)
```

That method formats the message part of the record, looking up the message key in a resource bundle and substituting message format parameters.

Many file formats (such as XML) require head and tail parts that surround the formatted records. To achieve this, override the methods

```
String getHead(Handler h)
String getTail(Handler h)
```

Finally, set the formatter in the logging configuration:

```
java.util.logging.FileHandler.formatter=com.mycompany.myapp.MyFormatter
```

Table 5.5: LogRecord Property Getters

Method	Property
<code>Level getLevel()</code>	The logging level of this record.
<code>String getLoggerName()</code>	The name of the logger that is logging this record.
<code>ResourceBundle getResourceBundle()</code>	The resource bundle, or its name, to be used for localizing the message, or <code>null</code> if none is provided.

Method	Property
<code>String getResourceBundleName()</code>	
<code>String getMessage()</code>	The “raw” message before localization or formatting.
<code>Object[] getParameters()</code>	The parameter objects, or <code>null</code> if none is provided.
<code>Throwable getThrown()</code>	The thrown object, or <code>null</code> if none is provided.
<code>String getSourceClassName()</code> <code>String getSourceMethodName()</code>	The location of the code that logged this record. This information may be supplied by the logging code or automatically inferred from the runtime stack. It might be inaccurate if the logging code supplied the wrong value or if the running code was optimized so that the exact location cannot be inferred.
<code>Instant getInstant()</code>	The creation time.
<code>long getSequenceNumber()</code>	The unique sequence number of this record.
<code>long getLongThreadID()</code>	The unique ID for the thread in which this record was created. These IDs are assigned by the <code>LogRecord</code> class and have no relationship to other thread IDs.

Exercises

1. Write a method `public ArrayList<Double> readValues(String filename) throws ...` that reads a file containing floating-point numbers. Throw appropriate exceptions if the file could not be opened or if some of the inputs are not floating-point numbers.
2. Write a method `public double sumOfValues(String filename) throws ...` that calls the preceding method and returns the sum of the values in the file. Propagate any exceptions to the caller.
3. Write a program that calls the preceding method and prints the result. Catch the exceptions and provide feedback to the user about any error conditions.
4. Repeat the preceding exercise, but don’t use exceptions. Instead, have `readValues` and `sumOfValues` return error codes of some kind.
5. Implement a method that contains the code with a `Scanner` and a `PrintWriter` in [Section 5.1.5](#). But don’t use the `try-with-resources` statement. Instead, just use `catch` clauses. Be sure to close both objects, provided they have been properly constructed. You need to consider the following conditions:
 - The `Scanner` constructor throws an exception.

- The `PrintWriter` constructor throws an exception.
- `hasNext`, `next`, or `println` throw an exception.
- `out.close()` throws an exception.
- `in.close()` throws an exception.

6. [Section 5.1.6](#) has an example of a broken `try` statement with `catch` and `finally` clauses.

Fix the code with (a) catching the exception in the `finally` clause, (b) a `try/catch` statement containing a `try/finally` statement, and (c) a `try-with-resources` statement with a `catch` clause.

7. Explain why

```
try (var in = new Scanner(Path.of("/usr/share/dict/words")));
    var out = new PrintWriter(outputFile)) {
    while (in.hasNext())
        out.println(in.next().toLowerCase());
}
```

is better than

```
var in = new Scanner(Path.of("/usr/share/dict/words"));
var out = new PrintWriter(outputFile);
try (in; out) {
    while (in.hasNext())
        out.println(in.next().toLowerCase());
}
```

8. For this exercise, you'll need to read through the source code of the `java.util.Scanner` class. If input fails when using a `Scanner`, the `Scanner` class catches the input exception and closes the resource from which it consumes input. What happens if closing the resource throws an exception? How does this implementation interact with the handling of suppressed exceptions in the `try-with-resources` statement?
9. Design a helper method so that one can use a `ReentrantLock` in a `try-with-resources` statement. Call `lock` and return an `AutoCloseable` whose `close` method calls `unlock` and throws no exceptions.
10. The methods of the `Scanner` and `PrintWriter` classes do not throw checked exceptions to make them easier to use for beginning programmers. How do you find out whether errors occurred during reading or writing? Note that the constructors *can* throw checked exceptions. Why does that defeat the goal of making the classes easier to use for beginners?
11. Write a recursive `factorial` method in which you print all stack frames before you return the value. Construct (but don't throw) an exception object of any kind and get its stack trace, as described in [Section 5.1.8](#).
12. Compare the use of `Objects.requireNonNull(obj)` and `assert obj != null`. Give a compelling use for each.
13. Write a method `int min(int[] values)` that, just before returning the smallest value, asserts that it is indeed \leq all values in the array. Use a private helper method or, if you already peeked into

[Chapter 8](#), Stream.allMatch. Call the method repeatedly on a large array and measure the runtime with assertions enabled, disabled, and removed.

14. Implement and test a log record filter that filters out log records containing bad words such as sex, drugs, and C++.
15. Implement and test a log record formatter that produces a JSON file.

Chapter 6 ■ Generic Programming

You often need to implement classes and methods that work with multiple types. For example, an `ArrayList<T>` stores elements of an arbitrary class `T`. We say that the `ArrayList` class is *generic*, and `T` is a *type parameter*. The basic idea is very simple and incredibly useful. The first two sections of this chapter cover the simple part.

In any programming language with generic types, the details get tricky when you restrict or vary type parameters. For example, suppose you want to sort elements. Then you must specify that `T` provides an ordering. Furthermore, if the type parameter varies, what does that mean for the generic type? For example, what should be the relationship between `ArrayList<String>` to a method that expects an `ArrayList<Object>`? [Section 6.3](#) and [Section 6.4](#) show you how Java deals with these issues.

In Java, generic programming is more complex than it perhaps should be, because generics were added when Java had been around for a while, and they were designed to be backward-compatible. As a consequence, there are a number of unfortunate restrictions, some of which affect every Java programmer. Others are only of interest to implementors of generic classes. See [Section 6.5](#) and [Section 6.6](#) for the details. The final section covers generics and reflection, and you can safely skip it if you are not using reflection in your own programs.

The key points of this chapter are:

1. A generic class is a class with one or more type parameters.
2. A generic method is a method with type parameters.
3. You can require a type parameter to be a subtype of one or more types.
4. Generic types are invariant: When `S` is a subtype of `T`, there is no relationship between `G<S>` and `G<T>`.
5. By using wildcards `G<? extends T>` or `G<? super T>`, you can specify that a method can accept an instantiation of a generic type `G`

with a subtype or supertype of T.

6. Type parameters are erased when generic classes and methods are compiled.
7. Erasure puts many restrictions on generic types. In particular, you can't instantiate generic classes or arrays, cast to a generic type, or throw an object of a generic type.
8. The `Class<T>` class is generic, which is useful because methods such as `cast` are declared to produce a value of type T.
9. Even though generic classes and methods are erased in the virtual machine, you can find out at runtime how they were declared.

6.1. Generic Classes

A *generic class* is a class with one or more *type parameters*. As a simple example, consider this class for storing key/value pairs:

```
public class Entry<K, V> {  
    private K key;  
    private V value;  
  
    public Entry(K key, V value) {  
        this.key = key;  
        this.value = value;  
    }  
  
    public K getKey() { return key; }  
    public V getValue() { return value; }  
}
```

As you can see, the type parameters K and V are specified inside angle brackets after the name of the class. In the definitions of class members, they are used as types for instance variables, method parameters, and return values.

You *instantiate* the generic class by substituting types for the type variables. For example, `Entry<String, Integer>` is an ordinary class with methods

`String getKey()` and `Integer getValue()`.



Caution: Type parameters cannot be instantiated with primitive types. For example, `Entry<String, int>` is not valid in Java.

When you *construct* an object of a generic class, you can omit the type parameters from the constructor. For example,

```
Entry<String, Integer> entry = new Entry<>("Fred", 42);  
// Same as new Entry<String, Integer>("Fred", 42)
```

Note that you still provide an empty pair of angle brackets before the construction arguments. Some people call this empty bracket pair a *diamond*. When you use the diamond syntax, the type parameters for the constructor are inferred.

6.2. Generic Methods

Just like a generic class is a class with type parameters, a *generic method* is a method with type parameters. A generic method can be a method of a regular class or a generic class. Here is an example of a generic method in a class that is not generic:

```
public class Arrays {  
    public static <T> void swap(T[] array, int i, int j) {  
        T temp = array[i];  
        array[i] = array[j];  
        array[j] = temp;  
    }  
}
```

This `swap` method can be used to swap elements in an arbitrary array, as long as the array element type is not a primitive type.

```
String[] friends = ...;  
Arrays.swap(friends, 0, 1);
```

When you declare a generic method, the type parameter is placed after the modifiers (such as `public` and `static`) and before the return type:

```
public static <T> void swap(T[] array, int i, int j)
```

When calling a generic method, you do not need to specify the type parameter. It is inferred from the method parameter and return types. For example, in the call `Arrays.swap(friends, 0, 1)`, the type of `friends` is `String[]`, and the compiler can infer that `T` should be `String`.

You can, if you like, supply the type explicitly, before the method name, like this:

```
Arrays.<String>swap(friends, 0, 1);
```

One reason why you might want to do this is to get better error messages when something goes wrong—see [Exercise 10](#).

Before plunging into the morass of technical details in the sections that follow, it is worth contemplating the examples of the `Entry` class and the `swap` method and to admire how useful and natural generic types are. With the `Entry` class, the key and value types can be arbitrary. With the `swap` method, the array type can be arbitrary. That is plainly expressed with type variables.

6.3. Type Bounds

Sometimes, the type parameters of a generic class or method need to fulfill certain requirements. You can specify a *type bound* to require that the type extends certain classes or implements certain interfaces.

Suppose, for example, you have an `ArrayList` of objects of a class that implements the `AutoCloseable` interface, and you want to close them all:

```
public static <T extends AutoCloseable> void  
closeAll(ArrayList<T> elems)  
    throws Exception {  
    for (T elem : elems) elem.close();  
}
```

The type bound `extends AutoCloseable` ensures that the element type is a subtype of `AutoCloseable`. Therefore, the call `elem.close()` is valid. You can pass an `ArrayList<PrintStream>` to this method, but not an `ArrayList<String>`. Note that the `extends` keyword in a type bound actually means “subtype”—the Java designers just used the existing `extends` keyword instead of coming up with another keyword or symbol.

[Exercise 28](#) has a more interesting variant of this method.



Note: In this example, we need a type bound because the parameter is of type `ArrayList`. If the method accepted an array, you wouldn’t need a generic method. You could simply use a regular method

```
public static void closeAll(AutoCloseable[] elems)  
throws Exception
```

This works because an array type such as `PrintStream[]` is a subtype of `AutoCloseable[]`. However, as you will see in the following section, an `ArrayList<PrintStream>` is *not* a subtype of `ArrayList<AutoCloseable>`. Using a bounded type parameter solves this problem.

A type parameter can have multiple bounds, such as

`T extends Runnable & AutoCloseable`

This syntax is similar to that for catching multiple exceptions, the only difference being that the types are combined with an “and” operator, whereas multiple exceptions are combined with an “or” operator.

You can have as many interface bounds as you like, but at most one of the bounds can be a class. If you have a class as a bound, it must be the first one in the bounds list.

6.4. Type Variance and Wildcards

Suppose you need to implement a method that processes an array of objects that are subclasses of the class `Employee`. You simply declare the parameter to have type `Employee[]`:

```
public static void process(Employee[] staff) { ... }
```

If `Manager` is a subclass of `Employee`, you can pass a `Manager[]` array to the method since `Manager[]` is a subtype of `Employee[]`. This behavior is called *covariance*. Arrays vary in the same way as the element types.

Now, suppose you want to process an array list instead. However, there is a problem: The type `ArrayList<Manager>` is *not* a subtype of `ArrayList<Employee>`.

There is a reason for this restriction. If it were legal to assign an `ArrayList<Manager>` to a variable of type `ArrayList<Employee>`, you could corrupt the array list by storing nonmanagerial employees:

```
ArrayList<Manager> bosses = new ArrayList<>();
ArrayList<Employee> empls = bosses; // Not legal, but suppose it
is...
empls.add(new Employee(...)); // A nonmanager in bosses!
```

Since conversion from `ArrayList<Manager>` to `ArrayList<Employee>` is disallowed, this error cannot occur.



Note: Can you generate the same error with arrays, where the conversion from `Manager[]` to `Employee[]` is permitted? Sure you can, as you saw in

[Chapter 4](#). Java arrays are covariant, which is convenient but unsound. When you store a mere `Employee` in a `Manager[]` array, an `ArrayStoreException` is thrown. In contrast, all generic types in Java are *invariant*.

In Java, you use *wildcards* to specify how method parameter and return types should be allowed to vary. This mechanism is sometimes called *use-site variance*. You will see the details in the following sections.

6.4.1. Subtype Wildcards

In many situations it is perfectly safe to convert between different array lists. Suppose a method never writes to the array list, so it cannot corrupt its argument. Use a wildcard to express this fact:

```
public static void printNames(ArrayList<? extends Employee>
    staff) {
    for (int i = 0; i < staff.size(); i++) {
        Employee e = staff.get(i);
        System.out.println(e.getName());
    }
}
```

The wildcard type `? extends Employee` indicates some unknown subtype of `Employee`. You can call this method with an `ArrayList<Employee>` or an array list of a subtype, such as `ArrayList<Manager>`.

The `get` method of the class `ArrayList<? extends Employee>` has return type `? extends Employee`. The statement

```
Employee e = staff.get(i);
```

is perfectly legal. Whatever type `?` denotes, it is a subtype of `Employee`, and the result of `staff.get(i)` can be assigned to the `Employee` variable `e`. (I

didn't use an enhanced for loop in this example to show exactly how the elements are fetched from the array list.)

What happens if you try to store an element into an `ArrayList<? extends Employee>`? That would not work. Consider a call

```
staff.add(x);
```

The add method has parameter type `? extends Employee`, and there is *no object* that you can pass to this method. If you pass, say, a `Manager` object, the compiler will refuse. After all, `?` could refer to *any* subclass, perhaps `Janitor`, and you can't add a `Manager` to an `ArrayList<Janitor>`.



Note: You can, of course, pass `null`, but that's not an object.

In summary, you can convert from `? extends Employee` to `Employee`, but you can never convert anything to `? extends Employee`. This explains why you can read from an `ArrayList<? extends Employee>` but cannot write to it.

6.4.2. Supertype Wildcards

The wildcard type `? extends Employee` denotes an arbitrary subtype of `Employee`. The converse is the wildcard type `? super Employee` which denotes a supertype of `Employee`. These wildcards are often useful as parameters in functional objects. Here is a typical example. The `Predicate` interface has a method for testing whether an object of type `T` has a particular property:

```
public interface Predicate<T> {
    boolean test(T arg);
    ...
}
```

This method prints the names of all employees with a given property:

```
public static void printAll(Employee[] staff,  
Predicate<Employee> filter) {  
    for (Employee e : staff)  
        if (filter.test(e))  
            System.out.println(e.getName());  
}
```

You can call this method with an object of type `Predicate<Employee>`. Since that is a functional interface, you can also pass a lambda expression:

```
printAll(employees, e -> e.getSalary() > 100000);
```

Now suppose you want to use a `Predicate<Object>` instead, for example

```
Predicate<Object> evenLength = e -> e.toString().length() %  
2 == 0;  
printAll(employees, evenLength);
```

This should not be a problem. After all, every `Employee` is an `Object` with a `toString` method. However, like all generic types, the `Predicate` interface is invariant, and there is no relationship between `Predicate<Employee>` and `Predicate<Object>`.

The remedy is to allow any `Predicate<? super Employee>`:

```
public static void printAll(Employee[] staff, Predicate<? super Employee> filter) {  
    for (Employee e : staff)  
        if (filter.test(e))  
            System.out.println(e.getName());  
}
```

Have a close look at the call `filter.test(e)`. Since the parameter of `test` has a type that is some supertype of `Employee`, it is safe to pass an `Employee` object.

This situation is typical. Functions are naturally *contravariant* in their parameter types. For example, when a function is expected that can process employees, it is OK to give one that is willing to process arbitrary objects.

In general, when you specify a generic functional interface as a method parameter, you should use a `super` wildcard.



Note: Some programmers like the “PECS” mnemonic for wildcards: producer `extends`, consumer `super`. An `ArrayList` from which you read values is a producer, so you use an `extends` wildcard. A `Predicate` to which you give values for testing is a consumer, and you use `super`.

6.4.3. Wildcards with Type Variables

Consider a generalization of the method of the preceding section that prints arbitrary elements fulfilling a condition:

```
public static <T> void printAll(T[] elements, Predicate<T>
filter) {
    for (T e : elements)
        if (filter.test(e))
            System.out.println(e.toString());
}
```

This is a generic method that works for arrays of any type. The type parameter is the type of the array that is being passed. However, it suffers from the limitation that you saw in the preceding section. The type parameter of `Predicate` must exactly match the type parameter of the method.

The solution is the same that you already saw—but this time, the bound of the wildcard is a type variable:

```
public static <T> void printAll(T[] elements, Predicate<? super T> filter)
```

This method takes a filter for elements of type `T` or any supertype of `T`.

Here is another example. The `Collection<E>` interface, which you will see in detail in the following chapter, describes a collection of elements of type `E`. It has a method

```
public boolean addAll(Collection<? extends E> c)
```

You can add all elements from another collection whose element type is also `E` or some subtype. With this method, you can add a collection of managers to a collection of employees, but not the other way around.

To see how complex type declarations can get, consider the definition of the `Collections.sort` method:

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

The `List` interface, covered in detail in the next chapter, describes a sequence of elements, such as a linked list or `ArrayList`. The `sort` method is willing to sort any `List<T>`, provided `T` is a subtype of `Comparable`. But the `Comparable` interface is again generic:

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

Its type parameter specifies the parameter type of the `compareTo` method. So, it would seem that `Collections.sort` could be declared as

```
public static <T extends Comparable<T>> void sort(List<T> list)
```

But that is too restrictive. Suppose that the `Employee` class implements `Comparable<Employee>`, comparing employees by salary. And suppose that the `Manager` class extends `Employee`. Note that it implements `Comparable<Employee>`, and *not* `Comparable<Manager>`. Therefore, `Manager` is *not* a subtype of `Comparable<Manager>`, but it is a subtype of `Comparable<? super Manager>`.



Note: In some programming languages (such as C# and Scala), you can declare type parameters to be covariant or contravariant. For example, by declaring the type parameter of `Comparable` to be contravariant, one doesn't have to use a wildcard for each `Comparable` parameter. This "declaration-site variance" is convenient, but it is less powerful than the "use-site variance" of Java wildcards.

6.4.4. Unbounded Wildcards

It is possible to have unbounded wildcards for situations where you only do very generic operations. For example, here is a method to check whether an `ArrayList` has any `null` elements:

```
public static boolean hasNulls(ArrayList<?> elements) {  
    for (Object e : elements) {  
        if (e == null) return true;  
    }  
    return false;  
}
```

Since the type parameter of the `ArrayList` doesn't matter, it makes sense to use an `ArrayList<?>`. One could equally well have made `hasNulls` into a generic method:

```
public static <T> boolean hasNulls(ArrayList<T> elements)
```

But the wildcard is easy to understand, so that's the preferred approach.

6.4.5. Wildcard Capture

Let's try to define a `swap` method using wildcards:

```
public static void swap(ArrayList<?> elements, int i, int j) {
    ? temp = elements.get(i); // Won't work
    elements.set(i, elements.get(j));
    elements.set(j, temp);
}
```

That won't work. You can use `?` as a type argument, but not as a type.

However, there is a workaround. Add a helper method, like this:

```
public static void swap(ArrayList<?> elements, int i, int j) {
    swapHelper(elements, i, j);
}

private static <T> void swapHelper(ArrayList<T> elements,
int i, int j) {
    T temp = elements.get(i);
    elements.set(i, elements.get(j));
    elements.set(j, temp);
}
```

The call to `swapHelper` is valid because of a special rule called *wildcard capture*. The compiler doesn't know what `?` is, but it stands for some type, so it is OK to call a generic method. The type parameter `T` of `swapHelper` "captures" the wildcard type. Since `swapHelper` is a generic method, not a method with wildcards in parameters, it can make use of the type variable `T` to declare variables.

What have we gained? The user of the API sees an easy-to-understand `ArrayList<?>` instead of a generic method.

6.5. Generics in the Java Virtual Machine

When generic types and methods were added to Java, the Java designers wanted the generic forms of classes to be compatible with their preexisting versions. For example, it should be possible to pass an `ArrayList<String>` to a method from pregeneric days that accepted the `ArrayList` class, which collects elements of type `Object`. The language designers decided on an implementation that “erases” the types in the virtual machine. This was very popular at the time since it enabled Java users to gradually migrate to using generics. As you can imagine, there are drawbacks to this scheme, and, as so often happens with compromises made in the interest of compatibility, the drawbacks remain long after the migration has successfully completed.

In this section, you will see what goes on in the virtual machine, and the next section examines the consequences.

6.5.1. Type Erasure

When you define a generic type, it is compiled into a *raw* type. For example, the `Entry<K, V>` class of [Section 6.1](#) turns into

```
public class Entry {  
    private Object key;  
    private Object value;  
  
    public Entry(Object key, Object value) {  
        this.key = key;  
        this.value = value;  
    }  
  
    public Object getKey() { return key; }  
    public Object getValue() { return value; }  
}
```

Every K and V is replaced by Object.

If a type variable has bounds, it is replaced with the first bound. Suppose we declare the Entry class as

```
public class Entry<K extends Comparable<? super K> &  
    Serializable,  
        V extends Serializable>
```

Then it is erased to a class

```
public class Entry {  
    private Comparable key;  
    private Serializable value;  
    ...  
}
```

6.5.2. Cast Insertion

Erasure sounds somehow dangerous, but it is actually perfectly safe. Suppose, for example, you used an Entry<String, Integer> object. When you construct the object, you must provide a key that is a String and a value that is an Integer or is converted to one. Otherwise, your program does not even compile. You are therefore guaranteed that the getKey method returns a String.

However, suppose your program compiled with “unchecked” warnings, perhaps because you used casts or mixed generic and raw Entry types. Then it is possible for an Entry<String, Integer> to have a key of a different type.

Therefore, it is also necessary to have safety checks at runtime. The compiler inserts a cast whenever one reads from an expression with erased type. Consider, for example,

```
Entry<String, Integer> entry = ...;  
String key = entry.getKey();
```

Since the erased `getKey` method returns an `Object`, the compiler generates code equivalent to

```
String key = (String) entry.getKey();
```

6.5.3. Bridge Methods

In the preceding sections, you have seen the basics of what erasure does. It is simple and safe. Well, almost simple. When erasing method parameter and return types, it is sometimes necessary for the compiler to synthesize *bridge methods*. This is an implementation detail, and you don't need to know about it unless you want to know why such a method shows up in a stack trace, or you want an explanation for one of the more obscure limitations on Java generics (see [Section 6.6.6](#)).

Consider this example:

```
public class WordList extends ArrayList<String> {  
    public boolean add(String e) {  
        return isBadWord(e) ? false : super.add(e);  
    }  
    ...  
}
```

Now consider this code fragment:

```
WordList words = ...;  
ArrayList<String> strings = words; // OK—conversion to  
superclass  
strings.add("C++");
```

The last method call invokes the (erased) `add(Object)` method of the `ArrayList` class.

One would reasonably expect dynamic method lookup to work in this case so that the `add` method of `WordList`, not the `add` method of `ArrayList`, is called when `add` is invoked on a `WordList` object.

To make this work, the compiler synthesizes a bridge method in the `WordList` class:

```
public boolean add(Object e) {  
    return add((String) e);  
}
```

In the call `strings.add("C++")`, the `add(Object)` method is called, and it calls the `add(String)` method of the `WordList` class.

Bridge methods can also be called when the return type varies. Consider this method:

```
public class WordList extends ArrayList<String> {  
    public String get(int i) {  
        return super.get(i).toLowerCase();  
    }  
    ...  
}
```

In the `WordList` class, there are two `get` methods:

```
String get(int) // Defined in WordList  
Object get(int) // Overrides the method defined in ArrayList
```

The second method is synthesized by the compiler, and it calls the first. This is again done to make dynamic method lookup work.

These methods have the same parameter types but different return types. In the Java language, you cannot implement such a pair of methods. But in the virtual machine, a method is specified by its name, the parameter types, *and* the return type, which allows the compiler to generate this method pair.



Note: Bridge methods are not only used for generic types. They are also used to implement covariant return types. For example, in

[Chapter 4](#), you saw how you should declare a `clone` method with the appropriate return type:

```
public class Employee implements Cloneable {  
    public Employee clone() throws  
        CloneNotSupportedException { ... }  
}
```

In this case, the `Employee` class has two `clone` methods:

```
Employee clone() // Defined above  
Object clone() // Synthesized bridge method
```

The bridge method, again generated to make dynamic method lookup work, calls the first method.

6.6. Restrictions on Generics

There are several restrictions when using generic types and methods in Java—some merely surprising and others genuinely inconvenient. Most of them are consequences of type erasure. The following sections show you those that you will most likely encounter in practice.

6.6.1. No Primitive Type Arguments

A type parameter can never be a primitive type. For example, you cannot form an `ArrayList<int>`. As you have seen, in the virtual machine there is only one type, the raw `ArrayList` that stores elements of type `Object`. An `int` is not an object.

When generics were first introduced, this was not considered a big deal. After all, one can form an `ArrayList<Integer>` and rely on autoboxing. Now that generics are more commonly used, however, the pain is increasing. There is a profusion of functional interfaces such as `IntFunction`, `LongFunction`, `DoubleFunction`, `ToIntFunction`,

`ToLongFunction`, `ToDoubleFunction`—and that only takes care of unary functions and three of the eight primitive types.

6.6.2. At Runtime, All Types Are Raw

In the virtual machine, there are only raw types. For example, you cannot inquire at runtime whether an `ArrayList` contains `String` objects. A condition such as

```
if (a instanceof ArrayList<String>)
```

is a compile-time error since no such check could ever be executed.

A cast to an instantiation of a generic type is equally ineffective, but it is legal.

```
Object result = ...;
ArrayList<String> list = (ArrayList<String>) result;
// Warning—this only checks whether result is a raw ArrayList
```

Such a cast is allowed because there is sometimes no way to avoid it. If `result` is the outcome of a very general process (such as calling a method through reflection, see

[Chapter 4](#)) and its exact type is not known to the compiler, the programmer must use a cast. A cast to `ArrayList` or `ArrayList<?>` would not suffice.

To make the warning go away, annotate the variable like this:

```
@SuppressWarnings("unchecked") ArrayList<String> list  
= (ArrayList<String>) result;
```



Caution: Abusing the `@SuppressWarnings` annotation can lead to *heap pollution*—objects that should belong to a particular generic type instantiation but actually belong to a different one. For example, you can assign an `ArrayList<Employee>` to an `ArrayList<String>` reference. The consequence is a `ClassCastException` when an element of the wrong type is retrieved.



Tip: The trouble with heap pollution is that the reported runtime error is far from the source of the problem—the insertion of a wrong element. If you need to debug such a problem, you can use a “checked view.” Where you constructed, say, an `ArrayList<String>`, instead use

```
List<String> strings  
= Collections.checkedList(new ArrayList<>(),  
String.class);
```

The view monitors all insertions into the list and throws an exception when an object of the wrong type is added.

The `getClass` method always returns a raw type. For example, if `list` is an `ArrayList<String>`, then `list.getClass()` returns `ArrayList.class`. In fact, there is no `ArrayList<String>.class`—such a class literal is a syntax error.

Also, you cannot have type variables in class literals. There is no `T.class`, `T[].class`, or `ArrayList<T>.class`.

6.6.3. You Cannot Instantiate Type Variables

You cannot use type variables in expressions such as `new T(...)` or `new T[...]`. These forms are outlawed because they would not do what the programmer intends when `T` is erased.

If you want to create a generic instance or array, you have to work harder. Suppose you want to provide a `repeat` method so that `Arrays.repeat(n, obj)` makes an array containing `n` copies of `obj`. Of course, you'd like the element type of the array to be the same as the type of `obj`. This attempt does not work:

```
public static <T> T[] repeat(int n, T obj) {
    T[] result = new T[n]; // Error—cannot construct an array
    new T[...]
    for (int i = 0; i < n; i++) result[i] = obj;
    return result;
}
```

To solve this problem, ask the caller to provide the array constructor as a method reference:

```
String[] greetings = Arrays.repeat(10, "Hi",
    String[]::new);
```

Here is the implementation of the method:

```
public static <T> T[] repeat(int n, T obj, IntFunction<T[]>
    constr) {
    T[] result = constr.apply(n);
    for (int i = 0; i < n; i++) result[i] = obj;
    return result;
}
```

Alternatively, you can ask the user to supply a class object, and use reflection.

```
public static <T> T[] repeat(int n, T obj, Class<T> cl) {  
    @SuppressWarnings("unchecked") T[] result  
        = (T[]) java.lang.reflect.Array.newInstance(cl, n);  
    for (int i = 0; i < n; i++) result[i] = obj;  
    return result;  
}
```

This method is called as follows:

```
String[] greetings = Arrays.repeat(10, "Hi", String.class);
```

Another option is to ask the caller to allocate the array. Usually, the caller is allowed to supply an array of any length, even zero. If the supplied array is too short, the method makes a new one, using reflection.

```
public static <T> T[] repeat(int n, T obj, T[] array) {  
    T[] result;  
    if (array.length >= n)  
        result = array;  
    else {  
        @SuppressWarnings("unchecked") T[] newArray  
            = (T[]) java.lang.reflect.Array.newInstance(  
                array.getClass().getComponentType(), n);  
        result = newArray;  
    }  
    for (int i = 0; i < n; i++) result[i] = obj;  
    return result;  
}
```



Tip: You *can* instantiate an `ArrayList` with a type variable. For example, the following is entirely legal:

```
public static <T> ArrayList<T> repeat(int n, T obj) {  
    var result = new ArrayList<T>(); // OK  
    for (int i = 0; i < n; i++) result.add(obj);  
    return result;  
}
```

This is much simpler than the workarounds you just saw, and I recommend it whenever you don't have a compelling reason for producing an array.



Note: If a generic class needs a generic array that is a private part of the implementation, you can get away with just constructing an `Object[]` array. This is what the `ArrayList` class does:

```
public class ArrayList<E> {  
    private Object[] elementData;  
  
    public E get(int index) {  
        return (E) elementData[index];  
    }  
    ...  
}
```

6.6.4. You Cannot Construct Arrays of Parameterized Types

Suppose you want to create an array of `Entry` objects:

```
var entries = new Entry<String, Integer>[100];  
// Error—cannot construct an array with generic component type
```

This is a syntax error. The construction is outlawed because, after erasure, the array constructor would create a raw `Entry` array. It would then be

possible to add `Entry` objects of any type (such as `Entry<Employee, Manager>`) without an `ArrayStoreException`.

Note that the *type* `Entry<String, Integer>[]` is perfectly legal. You can declare a variable of that type. If you really want to initialize it, you can, like this:

```
@SuppressWarnings("unchecked") Entry<String, Integer>[]  
entries  
= (Entry<String, Integer>[] ) new Entry<?, ?>[100];
```

But it is simpler to use an array list:

```
var entries = new ArrayList<Entry<String, Integer>>(100);
```

Recall that a varargs parameter is an array in disguise. If such a parameter is generic, you can bypass the restriction against generic array creation.

Consider this method:

```
public static <T> ArrayList<T> asList(T... elements) {  
    var result = new ArrayList<T>();  
    for (T e : elements) result.add(e);  
    return result;  
}
```

Now consider this call:

```
Entry<String, Integer> entry1 = ...;  
Entry<String, Integer> entry2 = ...;  
ArrayList<Entry<String, Integer>> entries =  
Lists.asList(entry1, entry2);
```

The inferred type for `T` is the generic type `Entry<String, Integer>`, and therefore `elements` is an array of type `Entry<String, Integer>`. That is just the kind of array creation that you cannot do yourself!

In this case, the compiler reports a warning, not an error. If your method only reads elements from the parameter array, it should use the `@SafeVarargs` annotation to suppress the warning:

```
@SafeVarargs public static <T> ArrayList<T> asList(T...  
elements)
```

This annotation can be applied to methods that are `static`, `final`, or `private`, or to constructors. Any other methods might be overridden and are not eligible for the annotation.

6.6.5. Class Type Variables Are Not Valid in Static Contexts

Consider a generic class with type variables, such as `Entry<K, V>`. You cannot use the type variables `K` and `V` with static variables or methods. For example, the following does not work:

```
public class Entry<K, V> {  
    private static V defaultValue;  
        // Error—V in static context  
    public static void setDefault(V value) { defaultValue =  
value; }  
        // Error—V in static context  
    ...  
}
```

After all, type erasure means there is only one such variable or method in the erased `Entry` class, and not one for each `K` and `V`.

6.6.6. Methods May Not Clash after Erasure

You may not declare methods that would cause clashes after erasure. For example, the following would be an error:

```
public interface Ordered<T> extends Comparable<T> {  
    public default boolean equals(T value) {  
        // Error—erasure clashes with Object.equals  
        return compareTo(value) == 0;  
    }  
    ...  
}
```

The `equals(T value)` method erases to `equals(Object value)`, which clashes with the same method from `Object`.

Sometimes the cause for a clash is more subtle. Here is a nasty situation:

```
public class Employee implements Comparable<Employee> {  
    ...  
    public int compareTo(Employee other) {  
        return name.compareTo(other.name);  
    }  
}  
  
public class Manager extends Employee implements  
Comparable<Manager> {  
    // Error—cannot have two instantiations of Comparable as  
    supertypes  
    ...  
    public int compareTo(Manager other) {  
        return Double.compare(salary, other.salary);  
    }  
}
```

The class `Manager` extends `Employee` and therefore picks up the supertype `Comparable<Employee>`. Naturally, managers want to compare each other by salary, not by name. And why not? There is no erasure. Just two methods

```
public int compareTo(Employee other)  
public int compareTo(Manager other)
```

The problem is that *the bridge methods clash*. Recall from [Section 6.5.3](#) that both of these methods yield a bridge method

```
public int compareTo(Object other)
```

6.6.7. Exceptions and Generics

You cannot throw or catch objects of a generic class. In fact, you cannot even form a generic subclass of `Throwable`:

```
public class Problem<T> extends Exception
    // Error—a generic class can't be a subtype of Throwable
```

You cannot use a type variable in a `catch` clause:

```
public static <T extends Throwable> void doWork(Runnable r,
Class<T> cl) {
    try {
        r.run();
    } catch (T ex) { // Error—can't catch type variable
        Logger.getGlobal().log(..., ..., ex);
    }
}
```

However, you *can* have a type variable in the `throws` declaration:

```
public static <V, T extends Throwable> V doWork(Callable<V>
c, T ex) throws T {
    try {
        return c.call();
    } catch (Throwable realEx) {
        ex.initCause(realEx);
        throw ex;
    }
}
```



Caution: You can use generics to remove the distinction between checked and unchecked exceptions. The key ingredient is this pair of methods:

```
public class Exceptions {  
    @SuppressWarnings("unchecked")  
    private static <T extends Throwable>  
        void throwAs(Throwable e) throws T {  
        throw (T) e; // The cast is erased to (Throwable)  
    }  
    public static <V> V doWork(Callable<V> c) {  
        try {  
            return c.call();  
        } catch (Throwable ex) {  
            Exceptions.<RuntimeException>throwAs(ex);  
            return null;  
        }  
    }  
}
```

Now consider this method:

```
public static String[] readWords(Path path) {  
    return doWork(() ->  
    Files.readString(path).split("\\s+"));  
}
```

Even though `Files.readString` throws a checked exception when the path is not found, that exception is neither declared nor caught in the `readWords` method!

6.7. Reflection and Generics

In the following sections, you will see what you can do with the generic classes in the reflection package and how you can find out the small amount of generic type information in the virtual machine that survives the erasure process.

6.7.1. The `Class<T>` Class

The `Class` class has a type parameter, namely the class that the `Class` object describes. Huh? Let's do this slowly.

Consider the `String` class. In the virtual machine, there is a `Class` object for this class, which you can obtain as `"Fred".getClass()` or, more directly, as the class literal `String.class`. You can use that object to find out what methods the class has, or to construct an instance.

The type parameter helps write typesafe code. Consider for example the `getConstructor` method of `Class<T>`. It is declared to return a `Constructor<T>`. And the `newInstance` method of `Constructor<T>` is declared to returns an object of type `T`. That's why `String.class` has type `Class<String>`: Its `getConstructor` method yields a `Constructor<String>`, whose `newInstance` method returns a `String`.

That information can save you a cast. Consider this method:

```
public static <T> ArrayList<T> repeat(int n, Class<T> cl)
    throws ReflectiveOperationException {
    var result = new ArrayList<T>();
    for (int i = 0; i < n; i++)
        result.add(cl.getConstructor().newInstance());
    return result;
}
```

The method compiles since `cl.getConstructor().newInstance()` returns a result of type `T`.

Suppose you call this method as `repeat(10, Employee.class)`. Then T is inferred to be the type `Employee` since `Employee.class` has type `Class<Employee>`. Therefore, the return type is `ArrayList<Employee>`.

In addition to the `getConstructor` method, there are several other methods of the `Class` class that use the type parameter. They are:

```
Class<? super T> getSuperclass()
<U> Class<? extends U> asSubclass(Class<U> clazz)
T cast(Object obj)
Constructor<T> getDeclaredConstructor(Class<?>...
parameterTypes)
T[] getEnumConstants()
```

As you have seen in

[Chapter 4](#), there are many situations where you know nothing about the class that a `Class` object describes. Then, you can simply use the wildcard type `Class<?>`.

6.7.2. Generic Type Information in the Virtual Machine

Erasure only affects instantiated type parameters. Complete information about the *declaration* of generic classes and methods is available at runtime.

For example, suppose a call `obj.getClass()` yields `ArrayList.class`. You cannot tell whether `obj` was constructed as an `ArrayList<String>` or `ArrayList<Employee>`. But you can tell that the class `ArrayList` is a generic class with a type parameter `E` that has no bounds.

Similarly, consider the method

```
static <T extends Comparable<? super T>> void sort(List<T>  
list)
```

of the `Collections` class. As you saw in

[Chapter 4](#), you can get the corresponding `Method` object as

```
Method m = Collections.class.getMethod("sort", List.class);
```

From this `Method` object, you can recover the entire method signature.

The interface `Type` in the `java.lang.reflect` package represents generic type declarations. The interface has the following subtypes:

1. The `Class` class, describing concrete types
2. The `TypeVariable` interface, describing type variables (such as `T extends Comparable<? super T>`)
3. The `WildcardType` interface, describing wildcards (such as `? super T`)
4. The `ParameterizedType` interface, describing generic class or interface types (such as `Comparable<? super T>`)
5. The `GenericArrayType` interface, describing generic arrays (such as `T[]`)

Note that the last four subtypes are interfaces—the virtual machine instantiates suitable classes that implement these interfaces.

Both classes and methods can have type variables. Technically speaking, constructors are not methods, and they are represented by a separate class in the reflection library. They too can be generic. To find out whether a `Class`, `Method`, or `Constructor` object comes from a generic declaration, call the `getTypeParameters` method. You get an array of `TypeVariable` instances, one for each type variable in the declaration, or an array of length 0 if the declaration was not generic.

The `TypeVariable<D>` interface is generic. The type parameter is `Class<T>`, `Method`, or `Constructor<T>`, depending on where the type variable was declared. For example, here is how you get the type variable of the `ArrayList` class:

```
TypeVariable<Class<ArrayList>>[] vars =
ArrayList.class.getTypeParameters();
String name = vars[0].getName(); // "E"
```

And here is the type variable of the `Collections.sort` method:

```
Method m = Collections.class.getMethod("sort", List.class);
TypeVariable<Method>[] vars = m.getTypeParameters();
String name = vars[0].getName(); // "T"
```

The latter variable has a bound, which you can process like this:

```
Type[] bounds = vars[0].getBounds();
if (bounds[0] instanceof ParameterizedType p) { //
Comparable<? super T>
    Type[] typeArguments = p.getActualTypeArguments();
    if (typeArguments[0] instanceof WildcardType t) { // ?
super T
        Type[] upper = t.getUpperBounds(); // ? extends ...
& ...
        Type[] lower = t.getLowerBounds(); // ? super ... &
...
        if (lower.length > 0) {
            String description = lower[0].getTypeName(); //
"T"
            ...
        }
    }
}
```

This gives you a flavor of how you can analyze generic declarations. I won't dwell on the details since this is not something that commonly comes up in practice. The key point is that the declarations of generic classes and methods are not erased and you have access to them through reflection.

Exercises

1. Implement a class `Stack<E>` that manages an array list of elements of type `E`. Provide methods `push`, `pop`, and `isEmpty`.
2. Reimplement the `Stack<E>` class, using an array to hold the elements. If necessary, grow the array in the `push` method. Provide two solutions, one with an `E[]` array and one with an `Object[]` array. Both solutions should compile without warnings. Which do you prefer, and why?
3. Implement a class `Table<K, V>` that manages an array list of `Entry<K, V>` elements. Supply methods to get the value associated with a key, to put a value for a key, and to remove a key.
4. In the previous exercise, make `Entry` into a nested class. Should that class be generic?
5. Consider this variant of the `swap` method where the array can be supplied with varargs:

```
public static <T> T[] swap(int i, int j, T... values) {
    T temp = values[i];
    values[i] = values[j];
    values[j] = temp;
    return values;
}
```

Now have a look at the call

```
Double[] result = Arrays.swap(0, 1, 1.5, 2, 3);
```

What error message do you get? Now call

```
Double[] result = Arrays.<Double>swap(0, 1, 1.5, 2, 3);
```

Has the error message improved? What do you do to fix the problem?

6. Implement a generic method that appends all elements from one array list to another. Use a wildcard for one of the type arguments. Provide two equivalent solutions, one with a `? extends E` wildcard and one with `? super E`.
7. Implement a class `Pair<E>` that stores a pair of elements of type `E`. Provide accessors to get the first and second element.

8. Modify the class of the preceding exercise by adding methods `max` and `min`, getting the larger or smaller of the two elements. Supply an appropriate type bound for `E`.
9. In a utility class `Arrays`, supply a method

```
public static <E> Pair<E> firstLast(ArrayList<__> a)
```

that returns a pair consisting of the first and last element of `a`. Supply an appropriate type argument.

10. Provide generic methods `min` and `max` in an `Arrays` utility class that yield the smallest and largest element in an array.
11. Continue the preceding exercise and provide a method `minMax` that yields a `Pair` with the minimum and maximum.
12. Implement the following method that stores the smallest and largest element in `elements` in the `result` list:

```
public static <T> void minmax(List<T> elements,  
        Comparator<? super T> comp, List<? super T> result)
```

Note the wildcard in the last parameter—any supertype of `T` will do to hold the result.

13. Given the method from the preceding exercise, consider this method:

```
public static <T> void maxmin(List<T> elements,  
        Comparator<? super T> comp, List<? super T>  
        result) {  
    minmax(elements, comp, result);  
    Lists.swapHelper(result, 0, 1);  
}
```

Why would this method not compile without wildcard capture? Hint: Try to supply an explicit type `Lists.<__>swapHelper(result, 0, 1)`.

14. Implement an improved version of the `closeAll` method in [Section 6.3](#). Close all elements even if some of them throw an exception. In that case, throw an exception afterwards. If two or more calls throw an exception, chain them together.

15. Implement a method `map` that receives an array list and a `Function<T, R>` object (see

[Chapter 3](#)), and that returns an array list consisting of the results of applying the function to the given elements.

16. What is the erasure of the following methods in the `Collection` class?

```
public static <T extends Comparable<? super T>>
    void sort(List<T> list)
public static <T extends Object & Comparable<? super T>>
    T max(Collection<? extends T> coll)
```

17. Define a class `Employee` that implements `Comparable<Employee>`. Using the `javap` utility, demonstrate that a bridge method has been synthesized. What does it do?

18. Consider the method

```
public static <T> T[] repeat(int n, T obj,
    IntFunction<T[]> constr)
```

in [Section 6.6.3](#). The call `Arrays.repeat(10, 42, int[]::new)` will fail. Why? How can you fix that? What do you need to do for the other primitive types?

19. Consider the method

```
public static <T> ArrayList<T> repeat(int n, T obj)
```

in [Section 6.6.3](#). This method had no trouble constructing an `ArrayList<T>` which contains an array of `T` values. Can you produce a `T[]` array from that array list without using a `Class` value or a constructor reference? If not, why not?

20. Implement the method

```
@SafeVarargs public static final <T> T[] repeat(int n,
    T... objs)
```

Return an array with `n` copies of the given objects. Note that no `Class` value or constructor reference is required since you can reflectively increase `objs`.

21. Using the `@SafeVarargs` annotation, write a method that can construct arrays of generic types. For example,

```
List<String>[] result = Arrays.  
<List<String>>construct(10);  
    // Sets result to a List<String>[] of size 10
```

22. Improve the method `public static <V, T extends Throwable> V doWork(Callable<V> c, T ex) throws T` of [Section 6.6.7](#) so that one doesn't have to pass an exception object, which may never get used. Instead, accept a constructor reference for the exception class.
23. In the cautionary note at the end of [Section 6.6.7](#), the `throwAs` helper method is used to “cast” `ex` into a `RuntimeException` and rethrow it. Why can't you use a regular cast—that is, `throw (RuntimeException) ex`?
24. Which methods can you call on a variable of type `Class<?>` without using casts?
25. Write a method `public static String genericDeclaration(Method m)` that returns the declaration of the method `m` listing the type parameters with their bounds and the types of the method parameters, including their type arguments if they are generic types.

Chapter 7 ■ Collections

Many data structures have been developed so programmers can store and retrieve values efficiently. The Java API provides implementations of common data structures and algorithms, as well as a framework to organize them. In this chapter, you will learn how to work with lists, sets, maps, and other collections.

The key points of this chapter are:

1. The `Collection` interface provides common methods for all collections, except for maps which are described by the `Map` interface.
2. A list is a sequential collection in which each element has an integer index.
3. A set is optimized for efficient containment testing. Java provides `HashSet` and `TreeSet` implementations.
4. For maps, you have the choice between `HashMap` and `TreeMap` implementations. A `LinkedHashMap` retains insertion order.
5. The `Collection` interface and `Collections` class provide many useful algorithms: set operations, searching, sorting, shuffling, and more.
6. Views provide access to data stored elsewhere using the standard collection interfaces.

7.1. An Overview of the Collections Framework

The Java collections framework provides implementations of common data structures. To make it easy to write code that is independent of the choice of data structures, the collections framework provides a number of common interfaces, shown in [Figure 7.1](#).

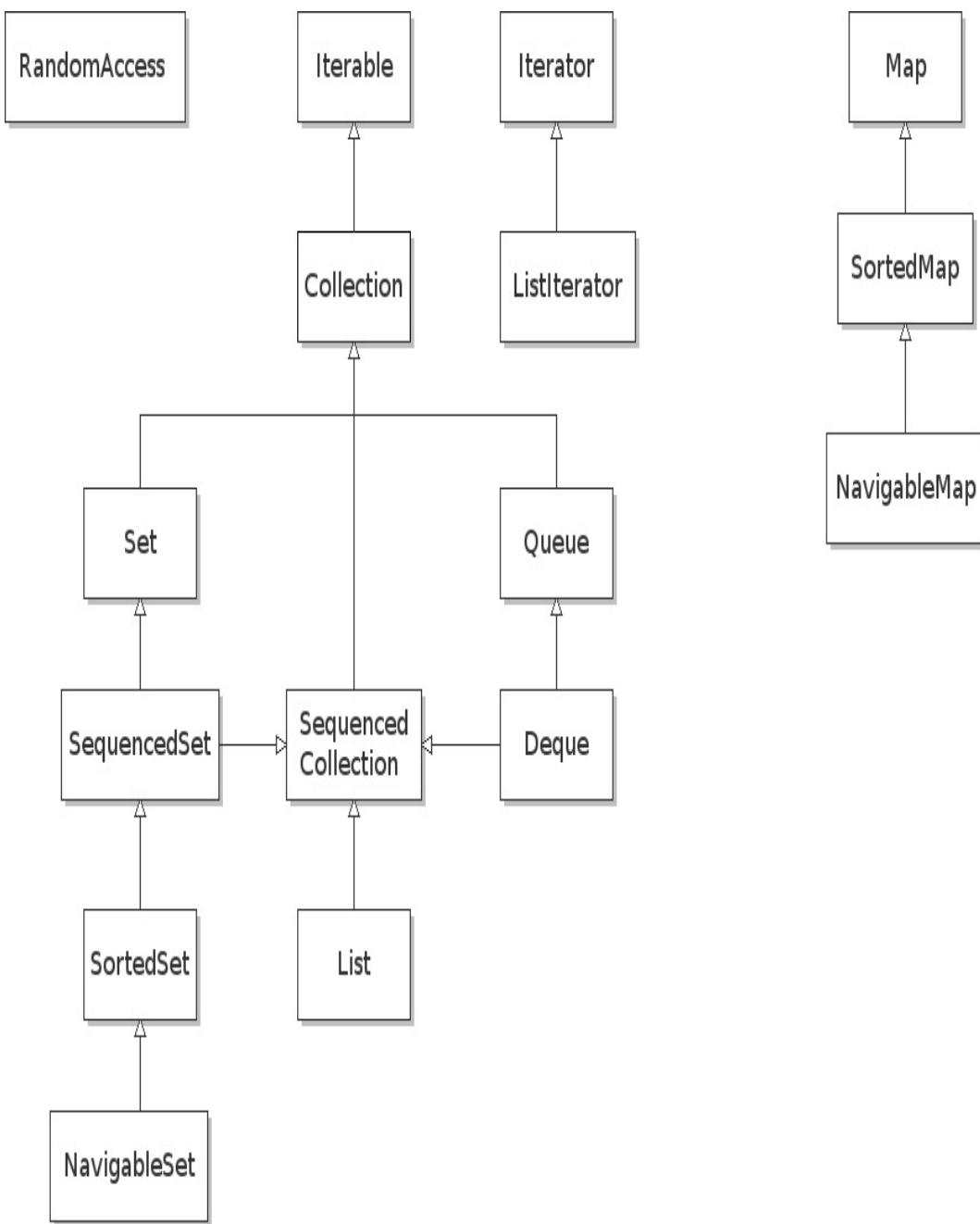


Figure 7.1: Interfaces in the Java collections framework

The fundamental interface is `Collection` whose methods are shown in [Table 7.1](#).

Table 7.1: The Methods of the Collection<E> Interface

Method	Description
<code>boolean add(E e)</code> <code>boolean addAll(Collection<? extends E> c)</code>	Adds <code>e</code> , or the elements in <code>c</code> . Returns <code>true</code> if the collection changed.
<code>boolean remove(Object o)</code> <code>boolean removeAll(Collection<?> c)</code> <code>boolean retainAll(Collection<?> c)</code> <code>boolean removeIf(Predicate<? super E> filter)</code> <code>void clear()</code>	Removes <code>o</code> , or the elements in <code>c</code> , or the elements not in <code>c</code> , or matching elements, or all elements. The first four methods return <code>true</code> if the collection changed.
<code>int size()</code>	Returns the number of elements in this collection.
<code>boolean isEmpty()</code> <code>boolean contains(Object o)</code> <code>boolean containsAll(Collection<?> c)</code>	Returns <code>true</code> if this collection is empty, or contains <code>o</code> , or contains all elements in <code>c</code> .
<code>Iterator<E> iterator()</code> <code>Stream<E> stream()</code> <code>Stream<E> parallelStream()</code> <code>Spliterator<E></code> <code>spliterator()</code>	Yields an iterator, or a stream, or a possibly parallel stream, or a spliterator for visiting the elements of this collection. See Section 7.2 for iterators and

Method	Description
	Chapter 8 for streams. Spliterators are only of interest to implementors of streams.
Object[] toArray() T[] toArray(T[] a)	Returns an array with the elements of this collection. The second method returns a if it has sufficient length.

A SequencedCollection provides access to the first and last element, and reverse traversal—see [Table 7.2](#). The SequencedSet subinterface sharpens the return type of the reversed method to SequencedSet. The SequencedMap interface has analogous methods for maps.

Table 7.2: The SequencedCollection Interface

Method	Description
E getFirst() E getLast() void addFirst(E e) void addLast(E e) E removeFirst() E removeLast()	Gets, adds, or removes the first or last element. The last two methods return the removed element.
SequencedCollection<E> reversed()	Returns a view of the collection that is iterated in reverse order (see Section 7.6).

A List is a sequential collection: Elements have position 0, 1, 2, and so on. [Table 7.3](#) shows the methods of that interface.

Table 7.3: The List Interface

Method	Description
<pre data-bbox="295 308 845 625">boolean add(int index, E e) boolean addAll(int index, Collection<? extends E> c) boolean add(E e) boolean addAll(Collection<? extends E> c)</pre>	<p>Adds <code>e</code>, or the elements in <code>c</code>, before <code>index</code> or to the end. Returns <code>true</code> if the list changed.</p>
<pre data-bbox="295 751 845 889">E get(int index) E set(int index, E element) E remove(int index)</pre>	<p>Gets, sets, or removes the element at the given index. The last two methods return the element at the index before the call.</p>
<pre data-bbox="295 1047 801 1142">int indexOf(Object o) int lastIndexOf(Object o)</pre>	<p>Returns the index of the first or last element equal to <code>o</code>, or -1 if there is no match.</p>
<pre data-bbox="295 1237 763 1417">ListIterator<E> listIterator() ListIterator<E> listIterator(int index)</pre>	<p>Yields a list iterator for all elements or the elements starting at <code>index</code>.</p>
<pre data-bbox="295 1491 845 1628">void replaceAll(UnaryOperator<E> operator)</pre>	<p>Replaces each element with the result of applying the operator to it.</p>
<pre data-bbox="295 1702 861 1797">void sort(Comparator<? super E> c)</pre>	<p>Sorts this list, using the ordering given by <code>c</code>.</p>

Method	Description
<code>static List<E> of(E... elements)</code>	Yields an unmodifiable list containing the given elements.
<code>List<E> subList(int fromIndex, int toIndex)</code>	Yields a view (Section 7.6) of the sublist starting at <code>fromIndex</code> and ending before <code>toIndex</code> .

The `List` interface is implemented both by the `ArrayList` class, which you have seen throughout this book, and the `LinkedList` class. If you took a course on data structures, you probably remember a linked list—a sequence of linked nodes, each carrying an element. Insertion in the middle of a linked list is speedy—you just splice in a node. But to get to the middle, you have to follow all the links from the beginning, which is slow. There are applications for linked lists, but most application programmers will probably stick with array lists when they need a sequential collection. Still, the `List` interface is useful. For example, the method `Collections.nCopies(n, o)` returns a `List` object with `n` copies of the object `o`. That object “cheats” in that it doesn’t actually store `n` copies but, when you ask about any one of them, returns `o`.



Caution: The `List` interface provides methods to access the *n*th element of a list, even though such an access may not be efficient. To indicate that it is, a collection class should implement the `RandomAccess` interface. This is a tagging interface without methods. For example, `ArrayList` implements `List` and `RandomAccess`, but `LinkedList` implements only the `List` interface.

In a `Set`, elements are not inserted at a particular position, and duplicate elements are not allowed. A `SortedSet` allows iteration in sort order, and a

`NavigableSet` has methods for finding neighbors of elements. You will learn more about sets in [Section 7.3](#).

A `Queue` retains insertion order, but you can only insert elements at the tail and remove them from the head (just like a queue of people). A `Deque` is a double-ended queue with insertion and removal at both ends.

All collection interfaces are generic, with a type parameter for the element type (`Collection<E>`, `List<E>`, and so on). The `Map<K, V>` interface has a type parameter `K` for the key type and `V` for the value type.

You are encouraged to use the interfaces as much as possible in your code. For example, after constructing an `ArrayList`, store the reference in a variable of type `List`:

```
List<String> words = new ArrayList<>();
```

Whenever you implement a method that processes a collection, use the least restrictive interface as parameter type. Usually, a `Collection`, `List`, or `Map` will suffice.

One advantage of a collections framework is that you don't have to reinvent the wheel when it comes to common algorithms. Some basic algorithms (such as `addAll` and `removeIf`) are methods of the `Collection` interface. The `Collections` utility class contains many additional algorithms that operate on various kinds of collections. You can sort, shuffle, rotate, and reverse lists, find the maximum or minimum, or the position of an arbitrary element in a collection, and generate collections with no elements, one element, or `n` copies of the same element. [Table 7.4](#) provides a summary.

Table 7.4: Useful Methods of the Collections Class

Method (all are static)	Description
<code>boolean disjoint(Collection<?> c1, Collection<?> c2)</code>	Returns <code>true</code> if the collections

Method (all are static)	Description
	have no elements in common.
boolean addAll(Collection<? super T> c, T... elements)	Adds all elements to c.
void copy(List<? super T> dest, List<? extends T> src)	Copies all elements from src to the same indexes in dest (which must be at least as long as src).
boolean replaceAll(List<T> list, T oldVal, T newVal)	Replaces all oldVal elements with newVal, either of which may be null. Returns true if at least one match was found.
void fill(List<? super T> list, T obj)	Sets all elements of the list to obj.
List<T> nCopies(int n, T o)	Yields an immutable list

Method (all are static)	Description
	with n copies of 0.
int frequency(Collection<?> c, Object o)	Returns the number of elements in c equal to o.
int indexOfSubList(List<?> source, List<?> target) int lastIndexOfSubList(List<?> source, List<?> target)	Returns the start of the first or last occurrence of the target list within the source list, or -1 if there is none.
int binarySearch(List<? extends Comparable<? super T>> list, T key) int binarySearch(List<? extends T> list, T key, Comparator<? super T> c)	Returns the position of the key, assuming that the list is sorted by the natural element order or c. If the key is not present, returns -i - 1 where i is the location at which the key should be inserted.

Method (all are static)	Description
<pre>sort(List<T> list) sort(List<T> list, Comparator<? super T> c)</pre>	Sorts the list, using the natural element order or c.
<pre>void swap(List<?> list, int i, int j)</pre>	Swaps the elements at the given position.
<pre>void rotate(List<?> list, int distance)</pre>	Rotates the list, moving the element with index i to $(i + distance) \% list.size()$.
<pre>void reverse(List<?> list) void shuffle(List<?> list) void shuffle(List<?> list, Random rnd)</pre>	Reverses or randomly shuffles the list.
<pre>synchronized(Collection SequencedCollection List Set SequencedSet SortedSet NavigableSet Map SequencedMap SortedMap NavigableMap) ()</pre>	Yields a synchronized view (see Section 7.6).
<pre>unmodifiable(Collection SequencedCollection List Set SequencedSet SortedSet NavigableSet </pre>	Yields an unmodifiable view (see Section 7.6).

Method (all are static)	Description
Map SequencedMap SortedMap NavigableMap) (()	
checked(Collection SequencedCollection List Set SequencedSet SortedSet NavigableSet Map SequencedMap SortedMap NavigableMap) (())	Yields a checked view (see Section 7.6).

7.2. Iterators

Each collection provides a way to iterate through its elements in some order. The `Iterable<T>` superinterface of `Collection` defines a method

```
Iterator<T> iterator()
```

It yields an iterator that you can use to visit all elements.

```
Collection<String> coll = ...;
Iterator<String> iter = coll.iterator();
while (iter.hasNext()) {
    String element = iter.next();
    Process element
}
```

In this case, you can simply use the enhanced `for` loop:

```
for (String element : coll) {
    Process element
}
```



Note: For *any* object `c` of a class that implements the `Iterable<E>` interface, the enhanced `for` loop is translated to the preceding form.

The **Iterator** interface also has a **remove** method which removes the previously visited element. This loop removes all elements that fulfill a condition:

```
while (iter.hasNext()) {  
    String element = iter.next();  
    if (element fulfills the condition)  
        iter.remove();  
}
```

However, it is easier to use the **removeIf** method:

```
coll.removeIf(e -> e fulfills the condition);
```



Caution: The **remove** method removes the last element that the iterator has returned, not the element to which the iterator points. You can't call **remove** twice without an intervening call to **next** or **previous**.

The **ListIterator** interface is a subinterface of **Iterator** with methods for adding an element before the iterator, setting the visited element to a different value, and for navigating backwards. It is mainly useful for working with linked lists.

```
var friends = new LinkedList<String>();  
ListIterator<String> iter = friends.listIterator();  
iter.add("Fred"); // Fred |  
iter.add("Wilma"); // Fred Wilma |  
iter.previous(); // Fred | Wilma  
iter.set("Barney"); // Fred | Barney
```



Caution: If an iterator traverses a data structure that is mutated between iterator method invocations, the iterator can become invalid.

An invalid iterator may throw a `ConcurrentModificationException` if you continue using it. Here is a typical example:

```
var friends = new LinkedList<String>(List.of("Fred",  
    "Wilma", "Barney"));  
for (String name : friends) {  
    if (name.length() < 5) friends.remove(name); //  
    Error—concurrent  
}  
                                //  
modification
```

The enhanced `for` loop uses an iterator to traverse the list. When the loop iterator is advanced after an element was removed, a `ConcurrentModificationException` occurs.

7.3. Sets

A set can efficiently test whether a value is an element, but it gives up something in return: It doesn't remember in which order elements were added. Sets are useful whenever the order doesn't matter. For example, if you want to disallow a set of bad words as usernames, their order doesn't matter. You just want to know whether a proposed username is in the set or not.

The `Set` interface is implemented by the `HashSet` and `TreeSet` classes. Internally, these classes use very different implementations. If you have taken a course in data structures, you may know how to implement hash tables and binary trees—but you can use these classes without knowing their internals.

Generally, hash sets are a bit more efficient, provided you have a good *hash function* for your elements. Library classes such as `String` or `Path` have good hash functions. You learned how to write hash function for your own classes in

Chapter 4.

For example, that set of bad words can be implemented simply as

```
var badWords = new HashSet<String>();
badWords.add("sex");
badWords.add("drugs");
badWords.add("c++");
if (badWords.contains(username.toLowerCase()))
    System.out.println("Please choose a different user
name");
```

You use a `TreeSet` if you want to traverse the set in sorted order. One reason you might want to do this is to present users a sorted list of choices.

The element type of the set must implement the `Comparable` interface, or you need to supply a `Comparator` in the constructor.

```
var countries = new TreeSet<String>(); // Visits added
countries in sorted order
var countries2 = new TreeSet<String>((u, v) ->
    u.equals(v) ? 0
    : u.equals("USA") ? -1
    : v.equals("USA") ? 1
    : u.compareTo(v));
// USA always comes first
```

The `TreeSet` class implements the interfaces `SortedSet` (see [Table 7.5](#)) and `NavigableSet` (see [Table 7.6](#)).

Table 7.5: `SortedSet<E>` Methods

Method	Description
<code>E first()</code> <code>E last()</code>	The first and last element in this set.

Method	Description
<pre>SortedSet<E> headSet(E toElement) SortedSet<E> subSet(E fromElement, E toElement) SortedSet<E> tailSet(E fromElement)</pre>	Returns a view of the elements starting at <code>fromElement</code> and ending before <code>toElement</code> .

Table 7.6: NavigableSet<E> Methods

Method	Description
<pre>E higher(E e) E ceiling(E e) E floor(E e) E lower(E e)</pre>	Returns the closest element $> \geq \leq < e$.
<pre>E pollFirst() E pollLast()</pre>	Removes and returns the first or last element, or returns <code>null</code> if the set is empty.
<pre>NavigableSet<E> headSet(E toElement, boolean inclusive) NavigableSet<E> subSet(E fromElement, boolean fromInclusive, E toElement, boolean toExclusive) NavigableSet<E> tailSet(E fromElement, boolean inclusive)</pre>	Returns a view of the elements from <code>fromElement</code> to <code>toElement</code> (inclusive or exclusive).

7.4. Maps

Maps store associations between keys and values. Call `put` to add a new association, or change the value of an existing key:

```
var counts = new HashMap<String, Integer>();
counts.put("Alice", 1); // Adds the key/value pair to the map
counts.put("Alice", 2); // Updates the value for the key
```

This example uses a hash map which, as for sets, is usually the better choice if you don't need to visit the keys in sorted order. If you do, use a `TreeMap` instead.

Here is how you can get the value associated with a key:

```
int count = counts.get("Alice");
```

If the key isn't present, the `get` method returns `null`. In this example, that would cause a `NullPointerException` when the value is unboxed. A better alternative is

```
int count = counts.getOrDefault("Alice", 0);
```

Then a count of `0` is returned if the key isn't present.

When you update a counter in a map, you first need to check whether the counter is present, and if so, add 1 to the existing value. The `merge` method simplifies that common operation. The call

```
counts.merge(word, 1, Integer::sum);
```

associates `word` with 1 if the key wasn't previously present, and otherwise combines the previous value and 1, using the `Integer::sum` function.

Now consider another common situation. We want to associate a set with each key. For example, in a book index, each term has a set of page numbers where the term occurs. Here is how to update the map:

```
var index = new TreeMap<String, TreeSet<Integer>>();  
...  
index.computeIfAbsent(term, k -> new TreeSet<>  
()).add(pageNumber);
```

Here, it is better to use `computeIfAbsent` because the `TreeSet` is only constructed when there was no prior set associated with the term. Conveniently, `computeIfAbsent` returns the new value, so that we can chain the call to `add`.



Caution: It is tempting to try changing `k -> new TreeSet<>()` into `TreeSet::new`, but that does not work. The function has a parameter `k` for the given key, which would be passed on to the constructor. Fortunately, there is no `TreeSet` constructor with a `String` parameter, and the code does not compile.

It could have been worse. Suppose you have a `Map<Integer, ArrayList<String>>` and call

```
map.computeIfAbsent(n, ArrayList::new)
```

There *is* an `ArrayList` constructor that takes an integer capacity. For large values of `n`, large array lists would be allocated, which was surely not intended.



Caution: The map methods are not very consistent about `null` values. Some methods treat `null` as a valid value, but others consider it in the same way as a missing key.

The `getOrDefault` method falls in the former camp, and `putIfAbsent` in the latter:

```
counts.put("C++", null);  
counts.getOrDefault("C++", -1) // Yields null without  
using the default
```

```
counts.putIfAbsent("C++", 1) // Puts 1, interpreting null as absent
```

[Table 7.7](#) summarizes the map operations.

Table 7.7: Map<K, V> Methods

Method	Description
V get(Object key) V getOrDefault(Object key, V defaultValue)	If key is associated with a non-null value v , returns v . Otherwise, returns null or defaultValue .
V put(K key, V value)	If key is associated with a non-null value v , associates key with value and returns v . Otherwise, adds entry and returns null .
V putIfAbsent(K key, V value)	If key is associated with a non-null value v , ignores value and returns v . Otherwise, adds entry and returns null .
V merge(K key, V value, BiFunction< ? super V,? super V,? extends V> remappingFunction)	If key is associated with a non-null value v , applies the function to v and value and either associates key with the result or, if the result is null , removes the key. Otherwise, associates key with value . Returns get(key) .

Method	Description
V compute(K key, BiFunction< ? super K, ? super V, ? extends V> remappingFunction)	Applies the function to key and get(key). Either associates key with the result or, if the result is null, removes the key. Returns get(key).
V computeIfPresent(K key, BiFunction< ? super K, ? super V, ? extends V> remappingFunction)	If key is associated with a non-null value v, applies the function to key and v and either associates key with the result or, if the result is null, removes the key. Returns get(key).
V computeIfAbsent(K key, Function< ? super K, ? extends V> mappingFunction)	Applies the function to key unless key is associated with a non-null value. Either associates key with the result or, if the result is null, removes the key. Returns get(key).
void putAll(Map< extends K, ? extends V> m)	Adds all entries from m.
V remove(Object key) V replace(K key, V newValue)	Removes the key and its associated value, or replaces the old value. Returns the old value, or null if none existed.
boolean remove(Object key, Object value)	Provided that key was associated with value, removes the entry or replaces

Method	Description
<code>boolean replace(K key, V value, V newValue)</code>	Replaces the value for the specified key. If the key does not exist, it does nothing and returns <code>false</code> . If the key exists and its current value is equal to the new value, it returns <code>true</code> . Otherwise, it replaces the value and returns <code>false</code> . These methods are mainly of interest when the map is accessed concurrently.
<code>int size()</code>	Returns the number of entries.
<code>boolean isEmpty()</code>	Checks if this map is empty.
<code>void clear()</code>	Removes all entries.
<code>void forEach(BiConsumer<? super K, ? super V> action)</code>	Applies the action to all entries.
<code>void replaceAll(BiFunction<? super K, ? super V,? extends V> function)</code>	Calls the function on all entries. Associates keys with non-null results and removes keys with null results.
<code>boolean containsKey(Object key) boolean containsValue(Object value)</code>	Checks whether the map contains the given key or value.
<code>Set<K> keySet() Collection<V> values() Set<Map.Entry<K, V>> entrySet()</code>	Returns views of the keys, values, and entries.

Method	Description
<pre>static Map<K, V> of() static Map<K, V> of(K k1, V v1) static Map<K, V> of(K k1, V v1, K k2, V v2)</pre>	<p>Yields an unmodifiable map containing up to ten keys and values.</p>

You can get *views* of the keys, values, and entries of a map by calling these methods:

```
Set<K> keySet()
Set<Map.Entry<K, V>> entrySet()
Collection<V> values()
```

The collections that are returned are not copies of the map data, but they are connected to the map. If you remove a key or entry from the view, then the entry is also removed from the underlying map.

To iterate through all keys and values of a map, you can iterate over the set returned by the `entrySet` method:

```
for (Map.Entry<String, Integer> entry : counts.entrySet())
{
    String k = entry.getKey();
    Integer oldValue = entry.getValue();
    int newValue = ...;
    entry.setValue(newValue);
}
```

The `Map.Entry` instances are also connected to the map. As you can see from the preceding code snippet, you can use an entry object to update a value in the underlying map. Conversely, if you update a value through other

means (for example, by calling the map's `put` method), then the entry is also updated.

If you want to pass entries to some other method, you should disassociate them from the map by calling `Map.Entry.copyOf(entry)`. You can also create unassociated `Map.Entry` instances by calling `Map.entry(key, value)`. This is handy whenever you need a pair of values.

If you don't need to mutate a map, it is simpler to traverse its entries with the `forEach` method:

```
counts.forEach((k, v) -> {  
    Process k, v  
});
```



Caution: Some map implementations (for example, `ConcurrentHashMap`) disallow `null` for keys or values. And with those that allow it (such as `HashMap`), you need to be very careful if you do use `null` values. A number of map methods interpret a `null` value as an indication that an entry is absent, or should be removed.



Tip: Sometimes, you need to present map keys in an order that is different from the sort order. For example, in the JavaServer Faces framework, you specify labels and values of a selection box with a map. Users would be surprised if the choices were sorted alphabetically (Friday, Monday, Saturday, Sunday, Thursday, Tuesday, Wednesday) or in the hash code order. In that case, use a `LinkedHashMap` that remembers the order in which entries were added and iterates through them in that order.

7.5. Other Collections

In the following sections, I briefly discuss some collection classes that you may find useful in practice.

7.5.1. Properties

The `Properties` class implements a map that can be easily saved and loaded using a plain text format. Such maps are commonly used for storing configuration options for programs. For example:

```
var settings = new Properties();
settings.put("width", "200");
settings.put("title", "Hello, World!");
try (OutputStream out = Files.newOutputStream(path)) {
    settings.store(out, "Program Properties");
}
```

The result is the following file:

```
#Program Properties
#Mon Nov 03 20:52:33 CET 2014
width=200
title>Hello, World\!
```



Note: As of Java 9, property files are encoded in UTF-8. (Previously, they were encoded in ASCII, with characters greater than '\u007e' written as Unicode escapes '\unnnn'.) Comments start with # or !. A newline in a key or value is written as \n. The characters \, #, ! are escaped as \\, \#, \!.

To load properties from a file, call

```
try (InputStream in = Files.newInputStream(path)) {
    settings.load(in);
}
```

Then use the `getProperty` method to get a value for a key. You can specify a default value used when the key isn't present:

```
String title = settings.getProperty("title", "New Document");
```



Note: For historical reasons, the `Properties` class implements `Map<Object, Object>` even though the values are always strings. Therefore, don't use the `get` method—it returns the value as an `Object`.

The `System.getProperties` method yields a `Properties` object with system properties. [Table 7.8](#) describes the most useful ones.

Table 7.8: Useful System Properties

Property key	Description
<code>user.dir</code>	The “current working directory” of this virtual machine.
<code>user.home</code>	The user’s home directory.
<code>user.name</code>	The user’s account name.
<code>java.version</code>	The Java runtime version of this virtual machine.
<code>java.home</code>	The home directory of the Java installation.
<code>java.class.path</code>	The class path with which this VM was launched.
<code>java.io.tmpdir</code>	A directory suitable for temporary files

Property key	Description
	(such as <code>/tmp</code>).
<code>os.name</code>	The name of the operating system (such as Linux).
<code>os.arch</code>	The architecture of the operating system (such as <code>amd64</code>).
<code>os.version</code>	The version of the operating system (such as <code>3.13.0-34-generic</code>).
<code>file.separator</code>	The file separator (<code>/</code> on Unix, <code>\</code> on Windows).
<code>path.separator</code>	The path separator (<code>:</code> on Unix, <code>;</code> on Windows).
<code>line.separator</code>	The newline separator (<code>\n</code> on Unix, <code>\r\n</code> on Windows).

7.5.2. Bit Sets

The `BitSet` class stores a sequence of bits. A bit set packs bits into an array of long values, so it is more efficient to use a bit set than an array of boolean values. Bit sets are useful for sequences of flag bits or to represent sets of non-negative integers, where the i th bit is 1 to indicate that i is contained in the set.

The `BitSet` class gives you convenient methods for getting and setting individual bits. This is much simpler than the bit-fiddling necessary to store bits in `int` or `long` variables. There are also methods that operate on all bits together for set operations, such as union and intersection. See [Table 7.9](#) for a complete list. Note that the `BitSet` class is not a collection class—it does not implement `Collection<Integer>`.

Table 7.9: Methods of the BitSet Class

Method	Description
<code>BitSet()</code> <code>BitSet(int nbits)</code>	Constructs a bit set that can initially hold 64, or <code>nbits</code> , bits.
<code>void set(int bitIndex)</code> <code>void set(int fromIndex, int toIndex)</code> <code>void set(int bitIndex, boolean value)</code> <code>void set(int fromIndex, int toIndex, boolean value)</code>	Sets the bit at the given index, or from <code>fromIndex</code> (inclusive) to <code>toIndex</code> (exclusive), to 1 or to the given value.
<code>void clear(int bitIndex)</code> <code>void clear(int fromIndex, int toIndex)</code> <code>void clear()</code>	Sets the bit at the given index, or from <code>fromIndex</code> (inclusive) to <code>toIndex</code> (exclusive), or all bits to 0.
<code>void flip(int bitIndex)</code> <code>void flip(int fromIndex, int toIndex)</code>	Flips the bit at the given index, or from <code>fromIndex</code> (inclusive) to <code>toIndex</code> (exclusive).
<code>boolean get(int bitIndex)</code> <code>BitSet get(int</code>	Gets the bit at the given index, or from <code>fromIndex</code> (inclusive) to <code>toIndex</code> (exclusive).

Method	Description
<code>fromIndex, int toIndex)</code>	
<code>int nextSetBit(int fromIndex) int previousSetBit(int fromIndex) int nextClearBit(int fromIndex) int previousClearBit(int fromIndex)</code>	Returns the index of the next/previous 1/0 bit, or -1 if none exists.
<code>void and(BitSet set) void andNot(BitSet set) void or(BitSet set) void xor(BitSet set)</code>	Forms the intersection difference union symmetric difference with <code>set</code> .
<code>int cardinality()</code>	Returns the number of 1 bits in this bit set. Caution: The <code>size</code> method returns the current size of the bit vector, not the size of the set.
<code>byte[] toByteArray[] long[] toLongArray[]</code>	Packs the bits of this bit set into an array.
<code>IntStream stream() String toString()</code>	Returns a stream or string of the integers (that is, indexes of 1 bits) in this bit set.
<code>static BitSet valueOf(byte[] bytes)</code>	Yields a bit set containing the supplied bits.

Method	Description
<pre data-bbox="295 297 670 713"> static BitSet valueOf(long[] longs) static BitSet valueOf(ByteBuffer bb) static BitSet valueOf(LongBuffer lb)</pre>	
<pre data-bbox="295 749 654 946"> boolean isEmpty() boolean intersects(BitSet set)</pre>	<p>Checks whether this bit set is empty, or has an element in common with <code>set</code>.</p>

7.5.3. Enumeration Sets and Maps

If you collect sets of enumerated values, use the `EnumSet` class instead of `BitSet`. The `EnumSet` class has no public constructors. Use a static factory method to construct the set:

```

enum Weekday { MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
FRIDAY, SATURDAY, SUNDAY } ;
Set<Weekday> always = EnumSet.allOf(Weekday.class);
Set<Weekday> never = EnumSet.noneOf(Weekday.class);
Set<Weekday> workday = EnumSet.range(Weekday.MONDAY,
Weekday.FRIDAY) ;
Set<Weekday> mwf = EnumSet.of(Weekday.MONDAY,
Weekday.WEDNESDAY, Weekday.FRIDAY) ;
```

You can use the methods of the `Set` interface to work with an `EnumSet`.

An `EnumMap` is a map with keys that belong to an enumerated type. It is implemented as an array of values. You specify the key type in the

constructor:

```
var personInCharge = new EnumMap<Weekday, String>
(Weekday.class);
personInCharge.put(Weekday.MONDAY, "Fred");
```

7.5.4. Stacks, Queues, Deques, and Priority Queues

A stack is a data structure for adding and removing elements at one end (the “top” of the stack). A queue lets you efficiently add elements at one end (the “tail”) and remove them from the other end (the “head”). A double-ended queue, or deque, supports insertion and removal at both ends. With all these data structures, adding elements in the middle is not supported.

The Queue and Deque interfaces define the methods for these data structures. There is no Stack interface in the Java collections framework, just a legacy Stack class from the earliest days of Java that you should avoid. If you need a stack, queue, or deque and are not concerned about thread safety, use an ArrayDeque.

With a stack, use the push and pop methods.

```
var stack = new ArrayDeque<String>();
stack.push("Peter");
stack.push("Paul");
stack.push("Mary");
while (!stack.isEmpty())
    System.out.println(stack.pop());
```

With a queue, use add and remove.

```
Queue<String> queue = new ArrayDeque<>();
queue.add("Peter");
queue.add("Paul");
queue.add("Mary");
while (!queue.isEmpty())
    System.out.println(queue.remove());
```

Threading safe queues are commonly used in concurrent programs. You will find more information about them in

Chapter 10.

A *priority queue* retrieves elements in sorted order after they were inserted in arbitrary order. That is, whenever you call the `remove` method, you get the smallest element currently in the priority queue.

A typical use for a priority queue is job scheduling. Each job has a priority. Jobs are added in random order. Whenever a new job can be started, the highest priority job is removed from the queue. (Since it is traditional for priority 1 to be the “highest” priority, the `remove` operation yields the minimum element.)

```
public record Job(int priority, String description)
    implements Comparable<Job> { ... }

...
var jobs = new PriorityQueue<Job>();
jobs.add(new Job(4, "Collect garbage"));
jobs.add(new Job(9, "Match braces"));
jobs.add(new Job(1, "Fix memory leak"));
...
while (jobs.size() > 0) {
    Job job = jobs.remove(); // The most urgent jobs are removed
    first
    execute(job);
}
```

Just like a `TreeSet`, a priority queue can hold elements of a class that implements the `Comparable` interface, or you can supply a `Comparator` in the constructor. However, unlike a `TreeSet`, iterating over the elements does not necessarily yield them in sorted order. The priority queue uses algorithms for adding and removing elements that cause the smallest element to gravitate to the root, without wasting time on sorting all elements.

7.5.5. Weak Hash Maps

The `WeakHashMap` class was designed to solve an interesting problem. What happens with a value whose key is no longer used anywhere in your

program? If the last reference to a key has gone away, there is no longer any way to refer to the value object so it should be removed by the garbage collector.

It isn't quite so simple. The garbage collector traces live objects. As long as the map object is live, all entries in it are live and won't be reclaimed—and neither will be the values that are referenced by the entries.

This is the problem that the `WeakHashMap` class solves. This data structure cooperates with the garbage collector to remove key/value pairs when the only reference to the key is the one from the hash table entry.

Technically, the `WeakHashMap` uses *weak references* to hold keys. A `WeakReference` object holds a reference to another object—in our case, a hash table key. Objects of this type are treated in a special way by the garbage collector. If an object is reachable *only* by a weak reference, the garbage collector reclaims the object and places the weak reference into a queue associated with the `WeakReference` object. Whenever a method is invoked on it, a `WeakHashMap` checks its queue of weak references for new arrivals and removes the associated entries.

7.6. Views

A collection *view* is a lightweight object that implements a collection interface, but doesn't store elements. For example, the `keySet` and `values` methods of a map yield views into the map.

In the following sections, you will see some views that are provided by the Java collections framework.

7.6.1. Small Collections

The `List`, `Set`, and `Map` interfaces provide static methods yielding a set or list with given elements, and a map with given key/value pairs.

For example,

```
List<String> names = List.of("Peter", "Paul", "Mary");  
Set<Integer> numbers = Set.of(2, 3, 5);
```

yield a list and a set with three elements. For a map, specify the keys and values like this:

```
Map<String, Integer> scores = Map.of("Peter", 2, "Paul", 3,  
"Mary", 5);
```

The elements, keys, or values may not be null.

The `List` and `Set` interfaces have 11 of methods with zero to ten arguments, and an of method with a variable number of arguments. The specializations are provided for efficiency.

For the `Map` interface, it is not possible to provide a version with variable arguments since the argument types alternate between the key and value types. There is a static method `ofEntries` that accepts an arbitrary number of `Map.Entry<K, V>` objects, which you can create with the static `entry` method. For example,

```
import static java.util.Map.*;  
...  
Map<String, Integer> scores = ofEntries(  
    entry("Peter", 2),  
    entry("Paul", 3),  
    entry("Mary", 5));
```

The of and ofEntries methods produce objects of classes that have an instance variable for each element, or that are backed by an array.

These collection objects are *unmodifiable*. Any attempt to change their contents results in an `UnsupportedOperationException`.

If you want a mutable collection, you can pass the unmodifiable collection to the constructor:

```
var names = new ArrayList<String>(List.of("Peter", "Paul",  
"Mary"));
```



Note: There is also a static `Arrays.asList` method that is similar to `List.of`. It returns a mutable list that is not resizable. That is, you can call `set` but not `add` or `remove` on the list.

7.6.2. Ranges

You can form a sublist view of a list. For example,

```
List<String> sentence = ...;  
List<String> nextFive = sentence.subList(5, 10);
```

This view accesses the elements with index 5 through 9. Any mutations of the sublist (such as setting, adding, or removing elements) affect the original.

For sorted sets and maps, you specify a range by the lower and upper bound:

```
TreeSet<String> words = ...;  
SortedSet<String> asOnly = words.subSet("a", "b");
```

As with `subList`, the first bound is inclusive, and the second exclusive.

The `headSet` and `tailSet` methods yield a subrange with no lower or upper bound.

```
NavigableSet<String> nAndBeyond = words.tailSet("n");
```

With the `NavigableSet` interface, you can choose for each bound whether it should be inclusive or exclusive—see [Table 7.6](#).

For a sorted map, there are equivalent methods `subMap`, `headMap`, and `tailMap`.

7.6.3. Unmodifiable Views

Sometimes, you want to share the contents of a collection but you don't want it to be modified. Of course, you could copy the values into a new collection, but that is potentially expensive. An unmodifiable view is a better choice. Here is a typical situation. A `Person` object maintains a list of friends. If the `getFriends` gave out a reference to that list, a caller could mutate it. But it is safe to provide an unmodifiable list view:

```
public class Person {  
    private ArrayList<Person> friends;  
  
    public List<Person> getFriends() {  
        return Collections.unmodifiableList(friends);  
    }  
    ...  
}
```

All mutator methods throw an exception when they are invoked on an unmodifiable view.

As you can see from [Table 7.4](#), you can get unmodifiable views as collections, lists, sets, sorted sets, navigable sets, maps, sorted maps, and navigable maps.



Note: In

[Chapter 6](#), you saw how it is possible to smuggle the wrong kind of elements into a generic collection (a phenomenon called “heap pollution”), and that a runtime error is reported when the inappropriate element is retrieved, not when it is inserted. If you need to debug such a problem, use a *checked view*. Where you constructed, say, an `ArrayList<String>`, instead use

```
List<String> strings  
    = Collections.checkedList(new ArrayList<>(),  
    String.class);
```

The view monitors all insertions into the list and throws an exception when an object of the wrong type is added.



Note: The `Collections` class produces *synchronized* views that ensure safe concurrent access to data structures. In practice, these views are not as useful as the data structures in the `java.util.concurrent` package that were explicitly designed for concurrent access. I suggest you use those classes and stay away from synchronized views.

7.6.4. Reversed Views

The `reversed` methods of the `SequencedCollection` and `SequencedSet` interfaces yield views that view the elements in reverse order. For example, here is how to traverse a list or tree set of strings in reverse:

```
for (String element : collection.reversed()) {  
    do something with element  
}
```

With `SequencedMap`, the `reversed` method views the map with the keys in reverse order.

Exercises

1. Implement the “Sieve of Erathostenes” algorithm to determine all prime numbers $\leq n$. Add all numbers from 2 to n to a set. Then repeatedly find the smallest element s in the set, and remove $s^2, s \cdot (s + 1), s \cdot (s + 2)$, and so on. You are done when $s^2 > n$. Do this with both a `HashSet<Integer>` and a `BitSet`.
2. In an array list of strings, make each string uppercase. Do this with (a) an iterator, (b) a loop over the index values, and (c) the `replaceAll` method.
3. How do you compute the union, intersection, and difference of two sets, using just the methods of the `Set` interface and without using loops?
4. Produce a situation that yields a `ConcurrentModificationException`. What can you do to avoid it?
5. Implement a method `public static void swap(List<?> list, int i, int j)` that swaps elements in the usual way when the type of `list` implements the `RandomAccess` interface, and that minimizes the cost of visiting the positions at index `i` and `j` if it is not.
6. I encouraged you to use interfaces instead of concrete data structures—for example, a `Map` instead of a `TreeMap`. Unfortunately, that advice goes only so far. Suppose you have a method parameter of type `Map<String, Set<Integer>>`, and someone calls your method with a `HashMap<String, HashSet<Integer>>`. What happens? What parameter type can you use instead?
7. Write a program that reads all words in a file and prints out how often each word occurred. Use a `TreeMap<String, Integer>`.
8. Write a program that reads all words in a file and prints out on which line(s) each of them occurred. Use a map from strings to sets.
9. You can update the counter in a map of counters as

```
counts.merge(word, 1, Integer::sum);
```

Do the same without the `merge` method, (a) by using `contains`, (b) by using `get` and a null check, (c) by using `getOrDefault`, (d) by using `putIfAbsent`.

Chapter 8 ■ Streams

Streams provide a view of data that lets you specify computations at a higher conceptual level than with collections. With a stream, you specify what you want to have done, not how to do it. You leave the scheduling of operations to the implementation. For example, suppose you want to compute the average of a certain property. You specify the source of data and the property, and the stream library can then optimize the computation, for example by using multiple threads for computing sums and counts and combining the results.

The key points of this chapter are:

1. Iterators imply a specific traversal strategy and prohibit efficient concurrent execution.
2. You can create streams from collections, arrays, generators, or iterators.
3. Use `filter` to select elements and `map` to transform elements.
4. Other operations for transforming streams include `limit`, `distinct`, and `sorted`.
5. To obtain a result from a stream, use a reduction operator such as `count`, `max`, `min`, `findFirst`, or `findAny`. Some of these methods return an `Optional` value.
6. The `Optional` type is intended as a safe alternative to working with `null` values. To use it safely, take advantage of the `ifPresent` and `orElse` methods.
7. You can collect stream results in collections, arrays, strings, or maps.
8. The `groupingBy` and `partitioningBy` methods of the `Collectors` class allow you to split the contents of a stream into groups, and to obtain a result for each group.
9. There are specialized streams for the primitive types `int`, `long`, and `double`.
10. Parallel streams automatically parallelize stream operations.

8.1. From Iterating to Stream Operations

When you process a collection, you usually iterate over its elements and do some work with each of them. For example, suppose we want to count all long words in a book. First, let's put them into a list:

```
String contents = Files.readString(Path.of("alice.txt"));
List<String> words = List.of(contents.split("\\PL+"));
// Split into words; nonletters are delimiters
```

Now we are ready to iterate:

```
int count = 0;
for (String w : words) {
    if (w.length() > 12) count++;
}
```

With streams, the same operation looks like this:

```
long count = words.stream()
    .filter(w -> w.length() > 12)
    .count();
```

Now you don't have to scan the loop for evidence of filtering and counting. The method names tell you right away what the code intends to do. Moreover, where the loop prescribes the order of operations in complete detail, a stream is able to schedule the operations any way it wants, as long as the result is correct.

Simply changing `stream` into `parallelStream` allows the stream library to do the filtering and counting in parallel.

```
long count = words.parallelStream()
    .filter(w -> w.length() > 12)
    .count();
```

Streams follow the “what, not how” principle. In our stream example, we describe what needs to be done: get the long words and count them. We don’t specify in which order, or in which thread, this should happen. In contrast, the loop at the beginning of this section specifies exactly how the computation should work, and thereby forgoes any chances of optimization.

A stream seems superficially similar to a collection, allowing you to transform and retrieve data. But there are significant differences:

1. A stream does not store its elements. They may be stored in an underlying collection or generated on demand.
2. Stream operations don’t mutate their source. For example, the `filter` method does not remove elements from a stream, but it yields a new stream in which they are not present.
3. Stream operations are *lazy* when possible. This means they are not executed until their result is needed. For example, if you only ask for the first five long words instead of all, the `filter` method will stop filtering after the fifth match. As a consequence, you can even have infinite streams!

Let us have another look at the example. The `stream` and `parallelStream` methods yield a *stream* for the `words` list. The `filter` method returns another stream that contains only the words of length greater than 12. The `count` method reduces that stream to a result.

This workflow is typical when you work with streams. You set up a pipeline of operations in three stages:

1. Create a stream.
2. Specify *intermediate operations* for transforming the initial stream into others, possibly in multiple steps.
3. Apply a *terminal operation* to produce a result. This operation forces the execution of the lazy operations that precede it. Afterwards, the stream can no longer be used.

In our example, the stream was created with the `stream` or `parallelStream` method. The `filter` method transformed it, and `count` was the terminal operation.

In the next section, you will see how to create a stream. The subsequent three sections deal with stream transformations. They are followed by five sections on terminal operations.

8.2. Stream Creation

You have already seen that you can turn any collection into a stream with the `stream` method of the `Collection` interface. If you have an array, use the static `Stream.of` method instead.

```
Stream<String> words = Stream.of(contents.split("\\PL+"));
// split returns a String[] array
```

The `of` method has a varargs parameter, so you can construct a stream from any number of arguments:

```
Stream<String> song = Stream.of("gently", "down", "the",
"stream");
```

Use `Arrays.stream(array, from, to)` to make a stream from a part of an array.

To make a stream with no elements, use the static `Stream.empty` method:

```
Stream<String> silence = Stream.empty();
// Generic type <String> is inferred; same as Stream.<String>empty()
```

The `Stream` interface has two static methods for making infinite streams. The `generate` method takes a function with no parameters (or, technically, an object of the `Supplier<T>` interface—see

[Chapter 3](#)). Whenever a stream value is needed, that function is called to produce a value. You can get a stream of constant values as

```
Stream<String> echos = Stream.generate(() -> "Echo");
```

or a stream of random numbers as

```
Stream<Double> randoms = Stream.generate(Math::random);
```

To produce sequences such as $0 \ 1 \ 2 \ 3 \ \dots$, use the `iterate` method instead. It takes a “seed” value and a function (technically, a `UnaryOperator<T>`) and repeatedly applies the function to the previous result. For example,

```
Stream<BigInteger> integers
    = Stream.iterate(BigInteger.ZERO, n ->
n.add(BigInteger.ONE));
```

The first element in the sequence is the seed `BigInteger.ZERO`. The second element is $f(\text{seed})$, or 1 (as a big integer). The next element is $f(f(\text{seed}))$, or 2, and so on.

To produce a finite stream instead, add a predicate that specifies when the iteration should finish:

```
var limit = new BigInteger("10000000");
Stream<BigInteger> integers
    = Stream.iterate(BigInteger.ZERO,
        n -> n.compareTo(limit) < 0,
        n -> n.add(BigInteger.ONE));
```

As soon as the predicate rejects an iteratively generated value, the stream ends.

Finally, the `Stream.ofNullable` method makes a really short stream from an object. The stream has length 0 if the object is `null` or length 1

otherwise, containing just the object. This is mostly useful in conjunction with `flatMap`—see [Section 8.7.7](#) for an example.



Note: A number of methods in the Java API yield streams. For example, the `String` class has a `lines` method that yields a stream of the lines contained in the string:

```
Stream<String> greetings = "Hello\nGuten  
Tag\nBonjour".lines();
```

The `Pattern` class has a method `splitAsStream` that splits a `CharSequence` by a regular expression. You can use the following statement to split a string into words:

```
Stream<String> words =  
Pattern.compile("\\PL+").splitAsStream(contents);
```

The `Scanner.tokens` method yields a stream of tokens of a scanner. Another way to get a stream of words from a string is

```
Stream<String> words = new Scanner(contents).tokens();
```

The static `Files.lines` method returns a `Stream` of all lines in a file:

```
try (Stream<String> lines = Files.lines(path)) {  
    Process lines  
}
```

Note that the `try-with-resources` block is necessary to close the file.

It is becoming more common to offer data as streams in the Java API. For example, Java 21 added a static `availableLocales` method that yields a stream of `Locale` objects.



Note: To produce a finite stream, you can always add the elements to a list which you then turn into a stream. The `Stream.Builder` interface is slightly more efficient for that purpose:

```
Stream<Integer> digits(int n)
{
    Stream.Builder<Integer> builder =
    Stream.builder();
    while (n != 0)
    {
        builder.add(n % 10);
        n = n / 10;
    }
    return builder.build();
}
```

To view the contents of one of the streams introduced in this section, use the `toList` method, which collects the stream's elements in a list. Like `count`, `toList` is a terminal operation. If the stream is infinite, first truncate it with the `limit` method:

```
System.out.println(Stream.generate(Math::random).limit(10).
    toList());
```



Note: In order to turn an `Iterator`, or a general `Iterable` that is not a collection, into a stream, you need to use helper methods for *spliterators*. A spliterator is similar to an iterator, but it can be split so that each resulting spliterator yields a part of the elements. This splitting is used for processing elements in parallel—see [Section 8.14](#).

If you have an `Iterator` and want a stream of its results, use

```
StreamSupport.stream(Spliterators.spliteratorUnknownSize(  
    iterator, Spliterator.ORDERED), false);
```

If you have an `Iterable` that is not a collection, you can turn it into a stream by calling

```
StreamSupport.stream(iterable.spliterator(), false);
```



Caution: It is very important that you don't modify the collection backing a stream while carrying out a stream operation. Remember that streams don't collect their data—the data is always in the separate collection. If you modify that collection, the outcome of the stream operations becomes undefined. The JDK documentation refers to this requirement as *noninterference*.

To be exact, since intermediate stream operations are lazy, it is possible to mutate the collection up to the point where the terminal operation executes. For example, the following, while certainly not recommended, will work:

```
List<String> wordList = ...;  
Stream<String> words = wordList.stream();  
wordList.add("END");  
long n = words.distinct().count();
```

But this code is wrong:

```
Stream<String> words = wordList.stream();  
words.forEach(s -> if (s.length() < 12)  
    wordList.remove(s));  
    // Error—interference
```

8.3. The `filter`, `map`, and `flatMap` Methods

A stream transformation produces a stream whose elements are derived from those of another stream. You have already seen the `filter` transformation that yields a new stream with those elements that match a certain condition. Here, we transform a stream of strings into another stream containing only long words:

```
List<String> words = ...;
Stream<String> longWords = words.stream().filter(w ->
w.length() > 12);
```

The parameter of `filter` is a `Predicate<T>`—that is, a function from `T` to `boolean`.

Often, you want to transform the values in a stream in some way. Use the `map` method and pass the function that carries out the transformation. For example, you can transform all words to lowercase like this:

```
Stream<String> lowercaseWords =
words.stream().map(String::toLowerCase);
```

Here, we used `map` with a method reference. Often, you will use a lambda expression instead:

```
Stream<Character> firstCodeUnits = words.stream().map(s ->
s.charAt(0));
```

The resulting stream contains the first code unit of each word.

When you use `map`, a function is applied to each element, and the result is a new stream with the results. Occasionally, you are faced with a mapping function that generates a stream of results. As an example, consider a method `graphemeClusters` that yields all grapheme clusters of a string. For example, `graphemeClusters("Ahoy 🇺🇸")` is a stream of strings "A", "h", "o", "y", " ", "🇺". (Note that the flag consists of five `char` values, so the `graphemeClusters` method has to do some heavy lifting to make this happen. We will look at the implementation momentarily.)

Now let's map such a `graphemeClusters` method on a stream of strings:

```
List<String> words = List.of(..., "Ahoy", "🏴☠", ...);  
Stream<Stream<String>> result = words.stream().map(w ->  
graphemeClusters(w));
```

You will get a stream of streams, like [... ["A", "h", "o", "y"], ["🏴☠"], ...]. To flatten it out to a single stream [... "A", "h", "o", "y", "🏴☠", ...], use the `flatMap` method instead of `map`:

```
Stream<String> flatResult = words.stream().flatMap(w ->  
graphemeClusters(w));  
// Calls graphemeClusters on each word and flattens the results
```

Now that you have seen how to use the `graphemeClusters` method, how do you write it? One way is to use a regular expression that breaks along grapheme cluster boundaries:

```
public static Stream<String> graphemeClusters(String s)  
{  
    return new Scanner(s).useDelimiter("\\b{g}").tokens();  
}
```

Here we were lucky that it was easy to write a method that produces a new stream for every stream element. Sometimes that is not so easy. It can also be a little inefficient to create so many streams. The `mapMulti` method offers an alternative. Instead of producing a stream of results, you generate the results and pass them to a collector—an object of a class implementing the functional interface `Consumer`. For each result, invoke the collector's `accept` method.

Let's do this with an example. The following loop produces the grapheme clusters of a string `s`:

```
BreakIterator iter = BreakIterator.getCharacterInstance();  
...  
iter.setText(s);
```

```

int start = iter.first();
int end = iter.next();
while (end != BreakIterator.DONE) {
    String gc = s.substring(start, end);
    start = end;
    end = iter.next();
    // Do something with gc;
}

```

When calling `mapMulti`, you provide a function that is invoked with the stream element and the collector. In your function, pass your results to the collector.

```

Stream<String> result = words.stream().mapMulti((s,
collector) ->
{
    iter.setText(s);
    int start = iter.first();
    int end = iter.next();
    while (end != BreakIterator.DONE)
    {
        String gc = s.substring(start, end);
        start = end;
        end = iter.next();
        collector.accept(gc);
    }
});

```

8.4. Extracting Substreams and Combining Streams

The call `stream.limit(n)` returns a new stream that ends after `n` elements (or when the original stream ends if it is shorter). This method is particularly useful for cutting infinite streams down to size. For example,

```
Stream<Double> randoms =  
Stream.generate(Math::random).limit(100);
```

yields a stream with 100 random numbers.

The call *stream.skip(n)* does the exact opposite. It discards the first n elements. This is handy in our book reading example where, due to the way the `split` method works, the first element is an unwanted empty string. We can make it go away by calling `skip`:

```
Stream<String> words =  
Stream.of(contents.split("\\PL+")).skip(1);
```

The *stream.takeWhile(predicate)* call takes all elements from the stream while the predicate is true, and then stops.

For example, suppose we use the `graphemeClusters` method of the preceding section to split a string into characters, and we want to collect all initial digits. The `takeWhile` method can do this:

```
Stream<String> initialDigits =  
graphemeClusters(str).takeWhile(  
    s -> "0123456789".contains(s));
```

The `dropWhile` method does the opposite, dropping elements while a condition is true and yielding a stream of all elements starting with the first one for which the condition was false. For example,

```
Stream<String> withoutInitialBlanks =  
graphemeClusters(str).dropWhile(String::isBlank);
```

You can concatenate two streams with the static `concat` method of the `Stream` class:

```
Stream<String> combined = Stream.concat(  
    graphemeClusters("Hello"), graphemeClusters("World"));  
// Yields the stream ["H", "e", "l", "l", "o", "W", "o",
```

```
"r", "l", "d"]
```

Of course, the first stream should not be infinite—otherwise the second wouldn't ever get a chance.

8.5. Other Stream Transformations

The `distinct` method returns a stream that yields elements from the original stream, in the same order, except that duplicates are suppressed. The duplicates need not be adjacent.

```
Stream<String> uniqueWords
    = Stream.of("merrily", "merrily", "merrily",
"gently").distinct();
    // Only one "merrily" is retained
```

For sorting a stream, there are several variations of the `sorted` method. One works for streams of `Comparable` elements, and another accepts a `Comparator`. Here, we sort strings so that the longest ones come first:

```
Stream<String> longestFirst
    =
words.stream().sorted(Comparator.comparing(String::length).
reversed());
```

As with all stream transformations, the `sorted` method yields a new stream whose elements are the elements of the original stream in sorted order.

Of course, you can sort a collection without using streams. The `sorted` method is useful when the sorting process is part of a stream pipeline.

Finally, the `peek` method yields another stream with the same elements as the original, but a function is invoked every time an element is retrieved. That is handy for debugging:

```
Object[] powers = Stream.iterate(1.0, p -> p * 2)
    .peek(e -> System.out.println("Fetching " + e))
    .limit(20).toArray();
```

When an element is actually accessed, a message is printed. This way you can verify that the infinite stream returned by `iterate` is processed lazily.



Tip: When you use a debugger to debug a stream computation, you can set a breakpoint in a method that is called from one of the transformations. With most IDEs, you can also set breakpoints in lambda expressions. If you just want to know what happens at a particular point in the stream pipeline, add

```
.peek(x -> {
    return; })
```

and set a breakpoint on the second line.

8.6. Simple Reductions

Now that you have seen how to create and transform streams, we will finally get to the most important point—getting answers from the stream data. The methods that we cover in this section are called *reductions*. Reductions are *terminal operations*. They reduce the stream to a nonstream value that can be used in your program.

You have already seen a simple reduction: the `count` method that returns the number of elements of a stream.

Other simple reductions are `max` and `min` that return the largest or smallest value. There is a twist—these methods return an `Optional<T>` value that either wraps the answer or indicates that there is none (because the stream happened to be empty). In the olden days, it was common to return `null` in such a situation. But that can lead to null pointer exceptions when it

happens in an incompletely tested program. The `Optional` type is a better way of indicating a missing return value. We discuss the `Optional` type in detail in the next section. Here is how you can get the maximum of a stream:

```
Optional<String> largest =
words.max(String::compareToIgnoreCase);
System.out.println("largest: " + largest.orElse("));
```

The `findFirst` returns the first value in a nonempty collection. It is often useful when combined with `filter`. For example, here we find the first word that starts with the letter Q, if it exists:

```
Optional<String> startsWithQ
= words.filter(s -> s.startsWith("Q")).findFirst();
```

If you are OK with any match, not just the first one, use the `findAny` method. This is effective when you parallelize the stream, since the stream can report any match that it finds instead of being constrained to the first one.

```
Optional<String> startsWithQ
= words.parallel().filter(s ->
s.startsWith("Q")).findAny();
```

If you just want to know if there is a match, use `anyMatch`. That method takes a predicate argument, so you won't need to use `filter`.

```
boolean aWordStartsWithQ
= words.parallel().anyMatch(s -> s.startsWith("Q"));
```

There are methods `allMatch` and `noneMatch` that return `true` if all or no elements match a predicate. These methods also benefit from being run in parallel.

8.7. The Optional Type

An `Optional<T>` object is a wrapper for either an object of type `T` or no object. In the former case, we say that the value is *present*. The `Optional<T>` type is intended as a safer alternative for a reference of type `T` that either refers to an object or is `null`. But it is only safer if you use it right. The next three sections show you how.

8.7.1. Producing an Alternative

The key to using `Optional` effectively is to either produce an alternative if the value is not present, or consume the value only if it is present. There are methods for both strategies.

In this section, we look at the first strategy. Often, there is a default that you want to use when there was no match, perhaps the empty string:

```
String result = optionalString.orElse("");
    // The wrapped string, or "" if none
```

You can also invoke code to compute the default:

```
String result = optionalString.orElseGet(() ->
System.getProperty("myapp.default"));
    // The function is only called when needed
```

Or you can throw an exception if there is no value:

```
String result =
optionalString.orElseThrow(IllegalStateException::new);
    // Supply a method that yields an exception object
```

8.7.2. Consuming the Value If Present

In the preceding section, you saw how to produce an alternative if no value is present. The other strategy for working with optional values is to consume the value only if it is present.

The `ifPresent` method accepts a function. If the optional value exists, it is passed to that function. Otherwise, nothing happens.

For example, if you want to add the value to a set if it exists, call

```
optionalValue.ifPresent(v -> results.add(v));
```

or simply

```
optionalValue.ifPresent(results::add);
```

If you want to take one action if the `Optional` has a value and another action if it doesn't, use `ifPresentOrElse`:

```
optionalValue.ifPresentOrElse(
    v -> System.out.println("Found " + v),
    () -> logger.warning("No match"));
```

8.7.3. Pipelining Optional Values

In the preceding sections, you saw how to get a value out of an `Optional` object. Another useful strategy is to keep the `Optional` intact. You can transform the value inside an `Optional` by using the `map` method:

```
Optional<Path> transformed = optionalString.map(Path::of);
```

If `optionalString` is empty, then `transformed` is also empty.



Note: This `map` method is the analog of the `map` method of the `Stream` interface that you have seen in [Section 8.3](#). Simply imagine an optional value as a stream of size zero or one. The result again has size zero or one, and in the latter case, the function has been applied.

Similarly, you can use the `filter` method to only consider `Optional` values that fulfill a certain property before or after transforming it. If the property is not fulfilled, the pipeline yields an empty result:

```
Optional<Path> transformed = optionalString  
    .filter(s -> s.endsWith(".txt"))  
    .map(Path::of);
```

You can substitute an alternative `Optional` for an empty `Optional` with the `or` method. The alternative is computed lazily.

```
Optional<String> result = optionalString.or(() -> // Supply  
    an Optional  
    alternatives.stream().findFirst());
```

If `optionalString` has a value, then `result` is `optionalString`. If not, the lambda expression is evaluated, and its result is used.

8.7.4. How Not to Work with Optional Values

If you don't use `Optional` values correctly, you have no benefit over the "something or null" approach of the past.

The `get` method gets the wrapped element of an `Optional` value if it exists, or throws a `NoSuchElementException` if it doesn't. Therefore,

```
Optional<T> optionalValue = ...;  
optionalValue.get().someMethod()
```

is no safer than

```
T value = ...;  
value.someMethod();
```

The `isPresent` and `isEmpty` methods report whether or not an `Optional<T>` object has a value. But

```
if (optionalValue.isPresent())
optionalValue.get().someMethod();
```

is no easier than

```
if (value != null) value.someMethod();
```



Tip: If you find yourself using the `get` method, rethink your approach and try one of the strategies of the preceding two sections.



Note: The `orElseThrow` is a scarier-sounding synonym for the `get` method. When you call `optionalValue.orElseThrow().someMethod()`, it becomes explicit that an exception will occur if the `optionalValue` is empty. The hope is that programmers will only use `orElseThrow` when it is absolutely clear that this cannot happen.

Here are a few more tips for the proper use of the `Optional` type:

- A variable of type `Optional` should *never* be `null`.
- Don't use fields of type `Optional`. The cost is an additional object. Inside a class, use `null` for an absent field. To discourage `Optional` fields, the class is not serializable.
- Method parameters of type `Optional` are questionable. They make the call unpleasant in the common case where the requested value is present. Instead, consider two overloaded versions of the method: with and without the parameter. On the other hand, returning an `Optional` is fine. It is the proper way to indicate that a function may not have a result.
- Don't put `Optional` objects in a set, and don't use them as keys for a map. Collect the values instead.

8.7.5. Creating Optional Values

So far, we have discussed how to consume an `Optional` object someone else created. If you want to write a method that creates an `Optional` object, there are several static methods for that purpose, including `Optional.of(result)` and `Optional.empty()`. For example,

```
public static Optional<Double> inverse(Double x) {  
    return x == 0 ? Optional.empty() : Optional.of(1 / x);  
}
```

The `ofNullable` method is intended as a bridge from possibly `null` values to optional values. `Optional.ofNullable(obj)` returns `Optional.of(obj)` if `obj` is not `null` and `Optional.empty()` otherwise.

8.7.6. Composing Optional Value Functions with flatMap

Suppose you have a method `f` yielding an `Optional<T>`, and the target type `T` has a method `g` yielding an `Optional<U>`. If they were normal methods, you could compose them by calling `s.f().g()`. But that composition doesn't work since `s.f()` has type `Optional<T>`, not `T`. Instead, call

```
Optional<U> result = s.f().flatMap(T::g);
```

If `s.f()` is present, then `g` is applied to it. Otherwise, an empty `Optional<U>` is returned.

Clearly, you can repeat that process if you have more methods or lambdas that yield `Optional` values. You can then build a pipeline of steps, simply by chaining calls to `flatMap`, that will succeed only when all parts do.

For example, consider the safe `inverse` method of the preceding section. Suppose we also have a safe square root:

```
public static Optional<Double> squareRoot(Double x) {  
    return x < 0 ? Optional.empty() :  
        Optional.of(Math.sqrt(x));  
}
```

Then you can compute the square root of the `inverse` as

```
Optional<Double> result = inverse(x).flatMap(x ->  
    squareRoot(x));
```

We can make such a chain look a little more regular by starting out with an `Optional`:

```
Optional<Double> result  
    = Optional.of(-4.0).flatMap(x -> inverse(x)).flatMap(x -  
    > squareRoot(x));
```

It looks even neater with method expressions:

```
Optional<Double> result  
    =  
Optional.of(-4.0).flatMap(MyMath::inverse).flatMap(MyMath::  
squareRoot);
```

If either the `inverse` method or the `squareRoot` returns `Optional.empty()`, the result is empty.



Note: You have already seen a `flatMap` method in the `Stream` interface (see [Section 8.3](#)). That method was used to compose two methods that yield streams, by flattening out the resulting stream of streams. The `Optional.flatMap` method works in the same way if you interpret an optional value as having zero or one elements.

8.7.7. Turning an `Optional` into a `Stream`

The `stream` method turns an `Optional<T>` into a `Stream<T>` with zero or one elements. Sure, why not, but why would you ever want that?

This becomes useful with methods that return an `Optional` result. Suppose you have a stream of user IDs and a method

```
Optional<User> lookup(String id)
```

How do you get a stream of users, skipping those IDs that are invalid?

Of course, you can filter out the invalid IDs and then apply `get` to the remaining ones:

```
Stream<String> ids = ...;
Stream<User> users = ids.map(Users::lookup)
    .filter(Optional::isPresent)
    .map(Optional::get);
```

But that uses the `isPresent` and `get` methods that we warned about. It is more elegant to call

```
Stream<User> users = ids.map(Users::lookup)
    .flatMap(Optional::stream);
```

Each call to `stream` returns a stream that is empty or has one element. The `flatMap` method combines them all. That means the nonexistent users are simply dropped.



Note: In this section, we consider the happy scenario in which we have a method that returns an `Optional` value. These days, many methods return `null` when there is no valid result. Suppose `Users.classicLookup(id)` returns a `User` object or `null`, not an `Optional<User>`. Then you can of course filter out the `null` values:

```
Stream<User> users = ids.map(Users::classicLookup)
    .filter(Objects::nonNull);
```

But if you prefer the `flatMap` approach, you can use

```
Stream<User> users = ids.flatMap(
    id ->
    Stream.ofNullable(Users.classicLookup(id)));
```

or

```
Stream<User> users = ids.map(Users::classicLookup)
    .flatMap(Stream::ofNullable);
```

The call `Stream.ofNullable(obj)` yields an empty stream if `obj` is `null` or a stream just containing `obj` otherwise.

8.8. Collecting Results

When you are done with a stream, you will often want to look at the results. You can call the `iterator` method, which yields an old-fashioned iterator that you can use to visit the elements.

Alternatively, you can call the `forEach` method to apply a function to each element:

```
stream.forEach(System.out::println);
```

On a parallel stream, the `forEach` method traverses elements in arbitrary order. If you want to process them in stream order, call `forEachOrdered` instead. Of course, you might then give up some or all of the benefits of parallelism.

But more often than not, you will want to collect the result in a data structure. You can call `toArray` and get an array of the stream elements.

Since it is not possible to create a generic array at runtime, the expression `stream.toArray()` returns an `Object[]` array. If you want an array of the correct type, pass in the array constructor:

```
String[] result = stream.toArray(String[]::new);
// stream.toArray() has type Object[]
```

For collecting stream elements to another target, there is a convenient `collect` method that takes an instance of the `Collector` interface. A

collector is an object that accumulates elements and produces a result. The `Collectors` class provides a large number of factory methods for common collectors. To collect a stream into a set, simply call

```
Set<String> result = stream.collect(Collectors.toSet());
```

If you want to control which kind of set you get, use the following call instead:

```
TreeSet<String> result =
    stream.collect(Collectors.toCollection(TreeSet::new));
```

Suppose you want to collect all strings in a stream by concatenating them. You can call

```
String result = stream.collect(Collectors.joining());
```

If you want a delimiter between elements, pass it to the `joining` method:

```
String result = stream.collect(Collectors.joining(", "));
```

If your stream contains objects other than strings, you need to first convert them to strings, like this:

```
String result =
    stream.map(Object::toString).collect(Collectors.joining(", "));
```

If you want to reduce the stream results to a sum, count, average, maximum, or minimum, use one of the `summarizing(Int|Long|Double)` methods.

These methods take a function that maps the stream objects to numbers and yield a result of type `(Int|Long|Double)SummaryStatistics`, simultaneously computing the sum, count, average, maximum, and minimum.

```
IntSummaryStatistics summary = stream.collect(  
    Collectors.summarizingInt(String::length));  
double averageWordLength = summary.getAverage();  
double maxWordLength = summary.getMax();
```

8.9. Collecting into Maps

Suppose you have a `Stream<Person>` and want to collect the elements into a map so that later you can look up people by their ID. Call `Collectors.toMap` with two functions that produce the map's keys and values. For example,

```
public record Person(int id, String name) {}  
...  
Map<Integer, String> idToName = people.collect(  
    Collectors.toMap(Person::id, Person::name));
```

In the common case when the values should be the actual elements, use `Function.identity()` for the second function.

```
Map<Integer, Person> idToPerson = people.collect(  
    Collectors.toMap(Person::id, Function.identity()));
```

If there is more than one element with the same key, there is a conflict, and the collector will throw an `IllegalStateException`. You can override that behavior by supplying a third function argument that resolves the conflict and determines the value for the key, given the existing and the new value. Your function could return the existing value, the new value, or a combination of them.

Here, we construct a map that contains, for each language in the available locales, as key its name in your default locale (such as "German"), and as value its localized name (such as "Deutsch").

```
Map<String, String> languageNames =  
Locale.availableLocales().collect(  
    Collectors.toMap(  
        Locale::getDisplayLanguage,  
        loc -> loc.getDisplayLanguage(loc),  
        (existingValue, newValue) -> existingValue));
```

We don't care that the same language might occur twice (for example, German in Germany and in Switzerland), so we just keep the first entry.



Note: In this chapter, I use the `Locale` class as a source of an interesting data set. You don't need to know anything about working with locales when following these examples.

Now suppose we want to know all languages in a given country. Then we need a `Map<String, Set<String>>`. For example, the value for "Switzerland" is the set [French, German, Italian]. At first, we store a singleton set for each language. Whenever a new language is found for a given country, we form the union of the existing and the new set.

```
Map<String, Set<String>> countryLanguageSets =  
Locale.availableLocales().collect(  
    Collectors.toMap(  
        Locale::getDisplayCountry,  
        l -> Collections.singleton(l.getDisplayLanguage()),  
        (a, b) -> { // Union of a and b  
            var union = new HashSet<String>(a);  
            union.addAll(b);  
            return union; }));
```

You will see a simpler way of obtaining this map in the next section.

If you want a `TreeMap`, supply the constructor as the fourth argument. You must provide a merge function. Here is one of the examples from the beginning of the section, now yielding a `TreeMap`:

```
Map<Integer, Person> idToPerson = people.collect(  
    Collectors.toMap(  
        Person::id,  
        Function.identity(),  
        (existingValue, newValue) -> { throw new  
IllegalStateException(); },  
        TreeMap::new));
```



Note: For each of the `toMap` methods, there is an equivalent `toConcurrentMap` method that yields a concurrent map. A single concurrent map is used in the parallel collection process. When used with a parallel stream, a shared map is more efficient than merging maps. Note that elements are no longer collected in stream order, but that doesn't usually make a difference.

8.10. Grouping and Partitioning

In the preceding section, you saw how to collect all languages in a given country. But the process was a bit tedious. You had to generate a singleton set for each map value and then specify how to merge the existing and new values. Forming groups of values with the same characteristic is very common, and the `groupingBy` method supports it directly.

Let's look at the problem of grouping locales by country. First, form this map:

```
Map<String, List<Locale>> countryToLocales =  
    Locale.availableLocales().collect(  
        Collectors.groupingBy(Locale::getCountry));
```

The function `Locale::getCountry` is the *classifier function* of the grouping. You can now look up all locales for a given country code, for example

```
List<Locale> swissLocales = countryToLocales.get("CH");  
// Yields locales [it_CH, de_CH, fr_CH]
```



Note: A quick refresher on locales: Each locale has a language code (such as `en` for English) and a country code (such as `US` for the United States). The locale `en_US` describes English in the United States, and `en_IE` is English in Ireland. Some countries have multiple locales. For example, `ga_IE` is Gaelic in Ireland, and, as the preceding example shows, my JVM knows three locales in Switzerland.

When the classifier function is a predicate function (that is, a function returning a `boolean` value), the stream elements are partitioned into two lists: those where the function returns `true` and the complement. In this case, it is more efficient to use `partitioningBy` instead of `groupingBy`. For example, here we split all locales into those that use English and all others:

```
Map<Boolean, List<Locale>> englishAndOtherLocales =  
Locale.availableLocales().collect(  
    Collectors.partitioningBy(l ->  
        l.getLanguage().equals("en")));  
List<Locale> englishLocales =  
englishAndOtherLocales.get(true);
```



Note: If you call the `groupingByConcurrent` method, you get a concurrent map that, when used with a parallel stream, is concurrently populated. This is entirely analogous to the `toConcurrentMap` method.

8.11. Downstream Collectors

The `groupingBy` method yields a map whose values are lists. If you want to process those lists in some way, supply a *downstream collector*. For example, if you want sets instead of lists, you can use the `Collectors.toSet` collector that you saw in the preceding section:

```
Map<String, Set<Locale>> countryToLocaleSet =  
    Locale.availableLocales().collect(  
        groupingBy(Locale::getCountry, toSet()));
```



Note: In this example, as well as the remaining examples of this section, I assume a static import of `java.util.stream.Collectors.*` to make the expressions easier to read.

You can also apply `groupingBy` twice:

```
Map<String, Map<String, List<Locale>>>  
countryAndLanguageToLocale =  
    Locale.availableLocales().collect(  
        groupingBy(Locale::getCountry,  
        groupingBy(Locale::getLanguage))));
```

Then `countryAndLanguageToLocale.get("IN").get("hi")` is a list of the Hindi locales in India. (There are several variants.)

Several collectors are provided for reducing grouped elements to numbers:

- `counting` produces a count of the collected elements. For example,

```
Map<String, Long> countryToLocaleCounts =  
    Locale.availableLocales().collect(  
        groupingBy(Locale::getCountry, counting()));
```

counts how many locales there are for each country.

- `summing(Int|Long|Double)` and `averaging(Int|Long|Double)` apply a provided function to the downstream elements and produce the sum or average of the function's results. For example,

```
Map<String, Double> stateToCityPopulation =
  cities.collect(
    groupingBy(City::state,
    summingInt(City::population)));
```

computes the averages of populations per state in a stream of cities.

- `maxBy` and `minBy` take a comparator and produce maximum and minimum of the downstream elements. For example,

```
Map<String, Optional<City>> stateToLargestCity =
  cities.collect(
    groupingBy(City::state,
    maxBy(Comparator.comparing(City::population))));
```

produces the most populous city per state.

The `collectingAndThen` collector adds a final processing step behind a collector. For example, if you want to know how many distinct results there are, collect them into a set and then compute the size:

```
Map<Character, Integer> stringCountsByStartingLetter =
  strings.collect(
    groupingBy(s -> s.charAt(0),
    collectingAndThen(toSet(), Set::size)));
```

The `mapping` collector does the opposite. It applies a function to each collected element and passes the results to a downstream collector.

```
Map<Character, Set<Integer>> stringLengthsByStartingLetter
= strings.collect(
  groupingBy(s -> s.charAt(0),
  mapping(String::length, toSet())));
```

Here, we group strings by their first character. Within each group, we produce the lengths and collect them in a set.

The `mapping` method also yields a nicer solution to a problem from the preceding section—gathering a set of all languages in a country.

```
Map<String, Set<String>> countryToLanguages =  
    Locale.availableLocales().collect(  
        groupingBy(Locale::getDisplayCountry,  
                  mapping(Locale::getDisplayLanguage,  
                        toSet())));
```

There is a `flatMap` method as well, for use with functions that return streams (see [Exercise 16](#)).

In the preceding section, I used `toMap` instead of `groupingBy`. In this form, you don't need to worry about combining the individual sets.

If the grouping or mapping function has return type `int`, `long`, or `double`, you can collect elements into a summary statistics object, as discussed in [Section 8.8](#). For example,

```
Map<String, IntSummaryStatistics>  
stateToCityPopulationSummary = cities.collect(  
    groupingBy(City::state,  
               summarizingInt(City::population)));
```

Then you can get the sum, count, average, minimum, and maximum of the function values from the summary statistics objects of each group.

The `filtering` collector applies a filter to each group, for example:

```
Map<String, Set<City>> largeCitiesByState  
    = cities.collect(  
        groupingBy(City::state,
```

```
filtering(c -> c.population() > 500000,  
          toSet()))); // States without large cities have  
empty sets
```

Finally, you can use the `teeing` collector to branch into two downstream collections. This is useful whenever you need to compute more than one result from a stream. Suppose you want to collect city names and also compute their average populations. You can't read a stream twice, but `teeing` lets you carry out two computations. Specify two downstream collectors and a function that combines the results.

```
record Pair<S, T>(S first, T second) {}  
Pair<List<String>, Double> result = cities.filter(c ->  
c.state().equals("NV"))  
    .collect(teeing(  
        mapping(City::name, toList()), // First downstream  
        collector  
        averagingDouble(City::population), // Second  
        downstream collector  
        (list, avg) -> new Pair<>(list, avg))); //  
Combining function
```



Note: There are also three versions of a `reducing` method that apply general reductions, as described in the next section.

Composing collectors is powerful, but it can also lead to very convoluted expressions. The best use is with `groupingBy` or `partitioningBy` to process the “downstream” map values. Otherwise, simply apply methods such as `map`, `reduce`, `count`, `max`, or `min` directly on streams.

8.12. Reduction Operations

The `reduce` method is a general mechanism for computing a value from a stream. The simplest form takes a binary function and keeps applying it,

starting with the first two elements. It's easy to explain this if the function is the sum:

```
List<Integer> values = ...;  
Optional<Integer> sum = values.stream().reduce((x, y) -> x  
+ y);
```

In this case, the `reduce` method computes $v_0 + v_1 + v_2 + \dots$, where the v_i are the stream elements. The method returns an `Optional` because there is no valid result if the stream is empty.



Note: In this case, you can write `reduce(Integer::sum)` instead of `reduce((x, y) -> x + y)`.

More generally, you can use any operation that combines a partial result x with the next value y to yield a new partial result.

Here is another way of looking at reductions. Given a reduction operation op , the reduction yields $v_0 op v_1 op v_2 op \dots$, where $v_i op v_{i+1}$ denotes the function call $op(v_i, v_{i+1})$. There are many operations that might be useful in practice, such as sum, product, string concatenation, maximum and minimum, set union and intersection.

If you want to use reduction with parallel streams, the operation must be *associative*: It shouldn't matter in which order you combine the elements. In math notation, $(x op y) op z$ must be equal to $x op (y op z)$. An example of an operation that is not associative is subtraction. For example, $(6 - 3) - 2 \neq 6 - (3 - 2)$.

Often, there is an *identity* e such that $e op x = x$, and you can use that element as the start of the computation. For example, 0 is the identity for addition. Then call the second form of `reduce`:

```
List<Integer> values = ...;
Integer sum = values.stream().reduce(0, (x, y) -> x + y)
    // Computes 0 + v0 + v1 + v2 + ...
```

The identity value is returned if the stream is empty, and you no longer need to deal with the `Optional` class.

Now suppose you have a stream of objects and want to form the sum of some property, such as all lengths in a stream of strings. You can't use the simple form of `reduce`. It requires a function $(T, T) \rightarrow T$, with the same types for the parameters and the result. But in this situation, you have two types: The stream elements have type `String`, and the accumulated result is an integer. There is a form of `reduce` that can deal with this situation.

First, you supply an “accumulator” function $(total, word) \rightarrow total + word.length()$. That function is called repeatedly, forming the cumulative total. But when the computation is parallelized, there will be multiple computations of this kind, and you need to combine their results. You supply a second function for that purpose. The complete call is

```
int result = words.reduce(0,
    (total, word) -> total + word.length(),
    (total1, total2) -> total1 + total2);
```



Note: In practice, you probably won't use the `reduce` method a lot. It is usually easier to map to a stream of numbers and use one of its methods to compute sum, max, or min. (We discuss streams of numbers in [Section 8.13](#).) In this particular example, you could have called `words.mapToInt(String::length).sum()`, which is both simpler and more efficient since it doesn't involve boxing.



Note: There are times when `reduce` is not general enough. For example, suppose you want to collect the results in a `BitSet`. If the collection is parallelized, you can't put the elements directly into a

single `BitSet` because a `BitSet` object is not threadsafe. For that reason, you can't use `reduce`. Each segment needs to start out with its own empty set, and `reduce` only lets you supply one identity value. Instead, use `collect`. You provide three functions:

1. A *supplier* to construct new instances of the target object
2. An *accumulator* that adds an element to the target
3. A *combiner* that merges two target objects into one

Here is how the `collect` method works for a bit set:

```
BitSet result = stream.collect(BitSet::new,
    BitSet::set, BitSet::or);
```

8.13. Primitive Type Streams

So far, we have collected integers in a `Stream<Integer>`, even though it is clearly inefficient to wrap each integer into a wrapper object. The same is true for the other primitive types `double`, `float`, `long`, `short`, `char`, `byte`, and `boolean`. The stream library has specialized types `IntStream`, `LongStream`, and `DoubleStream` that store primitive values directly, without using wrappers. If you want to store `short`, `char`, `byte`, and `boolean`, use an `IntStream`, and for `float`, use a `DoubleStream`.

To create an `IntStream`, call the `IntStream.of` and `Arrays.stream` methods:

```
IntStream stream = IntStream.of(1, 1, 2, 3, 5);
stream = Arrays.stream(values, from, to); // values is an
int[] array
```

As with object streams, you can also use the static `generate` and `iterate` methods. In addition, `IntStream` and `LongStream` have static methods `range` and `rangeClosed` that generate integer ranges with step size one:

```
IntStream zeroToNinetyNine = IntStream.range(0, 100); //  
Upper bound is excluded  
IntStream zeroToHundred = IntStream.rangeClosed(0, 100); //  
Upper bound is included
```

The `CharSequence` interface has methods `codePoints` and `chars` that yield an `IntStream` of the Unicode codes of the characters or of the code units in the UTF-16 encoding. (See

[Chapter 1](#) for the sordid details.)

```
String greeting = "Ahoy 🏴☠️";
IntStream codes = greeting.codePoints();
    // The stream with values 65, 104, 111, 121, 32, 127988, 8205,
9760, 65039
```

The `RandomGenerator` interface has methods `ints`, `longs`, and `doubles` that return primitive type streams of random numbers.

```
IntStream randomIntegers =
RandomGenerator.getDefault().ints();
```

When you have a stream of objects, you can transform it to a primitive type stream with the `mapToInt`, `mapToLong`, or `mapToDouble` methods. For example, if you have a stream of strings and want to process their lengths as integers, you might as well do it in an `IntStream`:

```
Stream<String> words = ...;
IntStream lengths = words.mapToInt(String::length);
```

To convert a primitive type stream to an object stream, use the `boxed` method:

```
Stream<Integer> integers = IntStream.range(0, 100).boxed();
```

Generally, the methods on primitive type streams are analogous to those on object streams. Here are the most notable differences:

- The `toArray` methods return primitive type arrays.
- Methods that yield an optional result return an `OptionalInt`, `OptionalLong`, or `OptionalDouble`. These classes are analogous to the `Optional` class, but they have methods `getAsInt`, `getAsLong`, and `getAsDouble` instead of the `get` method.
- There are methods `sum`, `average`, `max`, and `min` that return the sum, average, maximum, and minimum. These methods are not defined for

object streams.

- The `summaryStatistics` method yields an object of type `IntSummaryStatistics`, `LongSummaryStatistics`, or `DoubleSummaryStatistics` that can simultaneously report the sum, count, average, maximum, and minimum of the stream.

8.14. Parallel Streams

Streams make it easy to parallelize bulk operations. The process is mostly automatic, but you need to follow a few rules. First of all, you must have a parallel stream. You can get a parallel stream from any collection with the `Collection.parallelStream()` method:

```
Stream<String> parallelWords = words.parallelStream();
```

Moreover, the `parallel` method converts any sequential stream into a parallel one.

```
Stream<String> parallelWords =  
Stream.of(wordArray).parallel();
```

As long as the stream is in parallel mode when the terminal method executes, all intermediate stream operations will be parallelized.

When stream operations run in parallel, the intent is that the same result is returned as if they had run serially. It is important that the operations are *stateless* and can be executed in an arbitrary order.

Here is an example of something you cannot do. Suppose you want to count all short words in a stream of strings:

```
var shortWords = new int[12];  
words.parallelStream().forEach(  
    s -> { if (s.length() < 12) shortWords[s.length()]++;
```

```
});  
        // Error—race condition!  
System.out.println(Arrays.toString(shortWords));
```

This is very, very bad code. The function passed to `forEach` runs concurrently in multiple threads, each updating a shared array. As you will see in

[Chapter 10](#), that's a classic *race condition*. If you run this program multiple times, you are quite likely to get a different sequence of counts in each run—each of them wrong.

It is your responsibility to ensure that any functions you pass to parallel stream operations are safe to execute in parallel. The best way to do that is to stay away from mutable state. In this example, you can safely parallelize the computation if you group strings by length and count them.

```
Map<Integer, Long> shortWordCounts
  = words.parallelStream()
    .filter(s -> s.length() < 12)
    .collect(groupingBy(
      String::length,
      counting()));
```

By default, streams that arise from ordered collections (arrays and lists), from ranges, generators, and iterators, or from calling `Stream.sorted`, are *ordered*. Results are accumulated in the order of the original elements, and are entirely predictable. If you run the same operations twice, you will get exactly the same results.

Ordering does not preclude efficient parallelization. For example, when computing `stream.map(fun)`, the stream can be partitioned into n segments, each of which is concurrently processed. Then the results are reassembled in order.

Some operations can be more effectively parallelized when the ordering requirement is dropped. By calling the `Stream.unordered` method, you indicate that you are not interested in ordering. One operation that can benefit from this is `Stream.distinct`. On an ordered stream, `distinct` retains the first of all equal elements. That impedes parallelization—the thread processing a segment can't know which elements to discard until the preceding segment has been processed. If it is acceptable to retain *any* of the unique elements, all segments can be processed concurrently (using a shared set to track duplicates).

You can also speed up the `limit` method by dropping ordering. If you just want any `n` elements from a stream and you don't care which ones you get, call

```
Stream<String> sample =  
words.parallelStream().unordered().limit(n);
```

As discussed in [Section 8.9](#), merging maps is expensive. For that reason, the `Collectors.groupingByConcurrent` method uses a shared concurrent map. To benefit from parallelism, the order of the map values will not be the same as the stream order.

```
Map<Integer, List<String>> result =  
words.parallelStream().collect(  
    Collectors.groupingByConcurrent(String::length));  
    // Values aren't collected in stream order
```

Of course, you won't care if you use a downstream collector that is independent of the ordering, such as

```
Map<Integer, Long> wordCounts  
= words.parallelStream()  
.collect(  
    groupingByConcurrent(  
        String::length,  
        counting()));
```

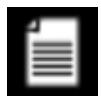
Don't turn all your streams into parallel streams in the hope of speeding up operations. Keep these issues in mind:

- There is a substantial overhead to parallelization that will only pay off for very large data sets.
- Parallelizing a stream is only a win if the underlying data source can be effectively split into multiple parts.
- The thread pool that is used by parallel streams can be starved by blocking operations such as file I/O or network access.

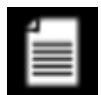
Parallel streams work best with huge in-memory collections of data and computationally intensive processing.



Tip: Generally, there is no benefit to parallelizing file input. However, the stream returned by the `Files.lines` method uses a memory-mapped file, and splitting is effective. If you process the lines of a huge file, parallelizing the stream may improve performance.



Note: If you need random numbers in parallel streams, obtain a generator from the `RandomGeneratorSplittableGenerator.of` factory method.



Note: By default, parallel streams use the global fork-join pool returned by `ForkJoinPool.commonPool`. That is fine if your operations don't block and you don't share the pool with other tasks. There is a trick to substitute a different fork-join pool. Execute the parallel stream operations in a thread from that pool:

```
ForkJoinPool customPool = ...;
result = customPool.submit(() ->
    stream.parallel().map(...).collect(...)).get();
```

Or, asynchronously:

```
CompletableFuture.supplyAsync(() ->
    stream.parallel().map(...).collect(...), customPool)
    .thenAccept(result -> ...);
```

This only works with a fork-join pool, not with arbitrary executors.

Exercises

1. Verify that asking for the first five long words does not call the `filter` method once the fifth long word has been found. Simply log each method call.
2. Measure the difference when counting long words with a `parallelStream` instead of a `stream`. Call `System.currentTimeMillis` before and after the call and print the difference. Switch to a larger document (such as *War and Peace*) if you have a fast computer.
3. Suppose you have an array `int[] values = { 1, 4, 9, 16 }`. What is `Stream.of(values)`? How do you get a stream of `int` instead?
4. Using `Stream.iterate`, make an infinite stream of random numbers—not by calling `Math.random` but by directly implementing a *linear congruential generator*. In such a generator, you start with $x_0 = \text{seed}$ and then produce $x_{n+1} = (a x_n + c) \% m$, for appropriate values of a , c , and m . You should implement a method with parameters `a`, `c`, `m`, and `seed` that yields a `Stream<Long>`. Try out $a = 25214903917$, $c = 11$, and $m = 2^{48}$.
5. The `codePoints` method in [Section 8.3](#) was a bit clumsy, first filling an array list and then turning it into a stream. Write a stream-based version instead, using the `IntStream.iterate` method to construct a finite stream of offsets, then extract the substrings.
6. Use the `String.codePoints` method to implement a method that tests whether a string is a word, consisting only of letters. (Hint: `Character.isAlphabetic`.) Using the same approach, implement a method that tests whether a string is a valid Java identifier.
7. Turning a file into a stream of tokens, list the first 100 tokens that are words in the sense of the preceding exercise. Read the file again and list the 10 most frequent words, ignoring letter case.
8. Find a realistic use for the `Collectors.flatMapting` method. Consider some class with a method yielding an `Optional`. Then group by some characteristic and, for each group, collect the nonempty optional values by using `flatMapting` and `Optional.stream`.
9. You can think of a stream as a “pull” mechanism from which values are lazily pulled when needed. The converse is a “push” mechanism

that eagerly produces values as they become available, passing them to a consumer. Examples are the `Iterable.forEach` and `Iterator.forEachRemaining` methods. These are methods with a parameter `Consumer<? super T>`.

Write a static method yielding a stream from such a method. For example,

```
Iterator<String> iter = ...;
Stream<String> stream =
fromPusher(iter::forEachRemaining);
```

The method has signature

```
public static <T> Stream<T>
fromPusher(Consumer<Consumer<? super T>> pusher)
```

Hint: Use `mapMulti` on a dummy `Stream.of((T) null)`.

10. Read the words from `/usr/share/dict/words` (or a similar word list) into a stream and produce an array of all words containing five distinct vowels.
11. Given a finite stream of strings, find the average string length.
12. Given a finite stream of strings, find all strings of maximum length.
13. Your manager asks you to write a method `public static <T> boolean isFinite(Stream<T> stream)`. Why isn't that such a good idea? Go ahead and write it anyway.
14. Write a method `public static <T> Stream<T> zip(Stream<T> first, Stream<T> second)` that alternates elements from the streams `first` and `second` (or `null` if the stream whose turn it is runs out of elements).
15. Join all elements in a `Stream<ArrayList<T>>` to one `ArrayList<T>`. Show how to do this with each of the three forms of `reduce`.
16. Write a call to `reduce` that can be used to compute the average of a `Stream<Double>`. Why can't you simply compute the sum and divide by `count()`?
17. Find 500 prime numbers with 50 decimal digits, using a parallel stream of `BigInteger` and the `BigInteger.isProbablePrime` method.

Is it any faster than using a serial stream?

18. Find the 500 longest strings in *War and Peace* with a parallel stream. Is it any faster than using a serial stream?
19. How can you eliminate adjacent duplicates from a stream? Would your method work if the stream was parallel?

Chapter 9 ■ Processing Input and Output

In this chapter, you will learn how to work with files, directories, and web pages, and how to read and write data in binary and text format. You will also find a discussion of regular expressions, which can be useful for processing input. (I couldn't think of a better place to handle that topic, and apparently neither could the Java developers—when the regular expression API specification was proposed, it was attached to the specification request for “new I/O” features.) Finally, this chapter shows you the object serialization mechanism that lets you store objects as easily as you can store text or numeric data.

The key points of this chapter are:

1. An `InputStream` is a source of bytes, and an `OutputStream` is a destination for bytes.
2. A `Reader` reads characters, and a `Writer` writes them. Be sure to specify a character encoding.
3. The `Files` class has convenience methods for reading all bytes or lines of a file.
4. The `DataInput` and `DataOutput` interfaces have methods for writing numbers in binary format.
5. Use a `RandomAccessFile` or a memory-mapped file for random access.
6. A `Path` is an absolute or relative sequence of path components in a file system. Paths can be combined (or “resolved”).
7. Use the methods of the `Files` class to copy, move, or delete files and to recursively walk through a directory tree.
8. To read or update a ZIP file, use a ZIP file system.
9. You can read the contents of a web page with the `URL` class. To read metadata or write data, use the `URLConnection` class.
10. With the `Pattern` and `Matcher` classes, you can find all matches of a regular expression in a string, as well as the captured groups for each match.
11. The serialization mechanism can save and restore any object implementing the `Serializable` interface, provided its instance variables

are also serializable.

9.1. Input/Output Streams, Readers, and Writers

In the Java API, a source from which one can read bytes is called an *input stream*. The bytes can come from a file, a network connection, or an array in memory. (These streams are unrelated to the streams of

[Chapter 8](#).) Similarly, a destination for bytes is an *output stream*. In contrast, *readers* and *writers* consume and produce sequences of *characters*. In the following sections, you will learn how to read and write bytes and characters.

9.1.1. Obtaining Streams

The easiest way to obtain a stream from a file is with the static methods

```
InputStream in = Files.newInputStream(path);
OutputStream out = Files.newOutputStream(path);
```

Here, *path* is an instance of the *Path* class that is covered in [Section 9.2.1](#). It describes a path in a file system.

If you have a URL, you can read its contents from the input stream returned by the *openStream* method of the *URL* class:

```
var url =
URI.create("https://horstmann.com/index.xhtml").toURL();
InputStream in = url.openStream();
```

[Section 9.3](#) shows how to send data to a web server.

The *ByteArrayInputStream* class lets you read from an array of bytes.

```
byte[] bytes = ...;
var in = new ByteArrayInputStream(bytes);
Read from in
```

Conversely, to send output to a byte array, use a *ByteArrayOutputStream*:

```
var out = new ByteArrayOutputStream();
Write to out
byte[] bytes = out.toByteArray();
```

9.1.2. Reading Bytes

The *InputStream* class has a method to read a single byte:

```
InputStream in = ...;
int b = in.read();
```

This method either returns the byte as an integer between 0 and 255, or returns -1 if the end of input has been reached.



Caution: The Java `byte` type has values between -128 and 127. You can cast the returned value into a `byte` *after* you have checked that it is not -1.

More commonly, you will want to read the bytes in bulk. The most convenient method is the `readAllBytes` method that simply reads all bytes from the stream into a byte array:

```
byte[] bytes = in.readAllBytes();
```



Tip: If you want to read all bytes from a file, call the convenience method

```
byte[] bytes = Files.readAllBytes(path);
```

If you want to read some, but not all bytes, provide a byte array and call the `readNBytes` method:

```
var bytes = new byte[len];
int bytesRead = in.readNBytes(bytes, offset, n);
```

The method reads until either `n` bytes are read or no further input is available, and returns the actual number of bytes read. If no input was available at all, the methods return -1.



Note: There is also a `read(byte[], int, int)` method whose description seems exactly like `readNBytes`. The difference is that the `read` method only attempts to read the bytes and returns immediately

with a lower count if it fails. The `readNBytes` method keeps calling `read` until all requested bytes have been obtained or `read` returns -1.

Finally, you can skip bytes:

```
long bytesToSkip = ...;
in.skipNBytes(bytesToSkip);
```

9.1.3. Writing Bytes

The `write` methods of an `OutputStream` can write individual bytes and byte arrays.

```
OutputStream out = ...;
int b = ...;
out.write(b);
byte[] bytes = ...;
out.write(bytes);
out.write(bytes, start, length);
```

When you are done writing a stream, you must *close* it in order to commit any buffered output. This is best done with a try-with-resources statement:

```
try (OutputStream out = ...) {
    out.write(bytes);
}
```

If you need to copy an input stream to an output stream, use the `InputStream.transferTo` method:

```
try (InputStream in = ...; OutputStream out = ...) {
    in.transferTo(out);
}
```

Both streams need to be closed after the call to `transferTo`. It is best to use a try-with-resources statement, as in the code example.

To write a file to an `OutputStream`, call

```
Files.copy(path, out);
```

Conversely, to save an `InputStream` to a file, call

```
Files.copy(in, path, StandardCopyOption.REPLACE_EXISTING);
```

9.1.4. Character Encodings

Input and output streams are for sequences of bytes, but in many cases you will work with text—that, is, sequences of characters. It then matters how characters are encoded into bytes.

Java uses the Unicode standard for characters. Each character or “code point” has a 21-bit integer number. There are different *character encodings*—methods for packaging those 21-bit numbers into bytes.

The most common encoding is UTF-8, which encodes each Unicode code point into a sequence of one to four bytes (see [Table 9.1](#)). UTF-8 has the advantage that the characters of the traditional ASCII character set, which contains all characters used in English, only take up one byte each.

Table 9.1: UTF-8 Encoding

Character range	Encoding
0...7F	$0a_6a_5a_4a_3a_2a_1a_0$
80...7FF	$110a_{10}a_9a_8a_7a_6 \ 10a_5a_4a_3a_2a_1a_0$
800...FFFF	$1110a_{15}a_{14}a_{13}a_{12} \ 10a_{11}a_{10}a_9a_8a_7a_6$ $10a_5a_4a_3a_2a_1a_0$
10000...10FFFF	$11110a_{20}a_{19}a_{18} \ 10a_{17}a_{16}a_{15}a_{14}a_{13}a_{12}$ $10a_{11}a_{10}a_9a_8a_7a_6 \ 10a_5a_4a_3a_2a_1a_0$

A less common encoding is UTF-16, which encodes each Unicode code point into one or two 16-bit values (see [Table 9.2](#)). This is the encoding used in Java strings. Actually, there are two forms of UTF-16, called “big-endian” and “little-endian.” Consider the 16-bit value `0x2122`. In big-endian format, the more significant byte comes first: `0x21` followed by `0x22`. In little-endian format, it is the other way around: `0x22 0x21`. To indicate which of the two is used, a file can start with the “byte order mark,” the 16-bit quantity `0xFEFF`. A reader can use this value to determine the byte order and discard it.

Table 9.2: UTF-16 Encoding

Character range	Encoding
<code>0...FFFF</code>	$a_{15}a_{14}a_{13}a_{12}a_{11}a_{10}a_9a_8a_7a_6a_5a_4a_3a_2a_1a_0$
<code>10000...10FFFF</code>	$110110b_{19}b_{18}b_{17}b_{16}a_{15}a_{14}a_{13}a_{12}a_{11}a_{10}$ $110111a_9a_8a_7a_6a_5a_4a_3a_2a_1a_0$ where $b_{19}b_{18}b_{17}b_{16} = a_{20}a_{19}a_{18}a_{17}a_{16} - 1$



Caution: Some programs, including Microsoft Notepad, add a byte order mark at the beginning of UTF-8 encoded files. Clearly, this is unnecessary since there are no byte ordering issues in UTF-8. But the Unicode standard allows it, and even suggests that it’s a pretty good idea since it leaves little doubt about the encoding. It is supposed to be removed when reading a UTF-8 encoded file. Sadly, Java does not do that, and bug reports against this issue are closed as “will not fix.” Your best bet is to strip out any leading `\uFEFF` that you find in your input.

In addition to the UTF encodings, there are partial encodings that cover a character range suitable for a given user population. For example, ISO 8859-1 is a one-byte code that includes accented characters used in Western European languages. Shift_JIS is a variable-length code for Japanese characters. A large number of these encodings are still in widespread use.

Because UTF-8 is so common, it has become the default encoding since Java 18. Previously, the default encoding was the *native encoding*—the character encoding that is preferred by the operating system of the computer running your program. On Windows, that is generally not UTF-8. If you are using an older version of Java, or if you are working with text in an encoding other than UTF-8, you need to explicitly specify the encoding.



Note: The native encoding is returned by the static method `Charset.defaultCharset`. The static method `Charset.availableCharsets` returns all available `Charset` instances, as a map from canonical names to `Charset` objects.

The `StandardCharsets` class has static variables of type `Charset` for the character encodings that every Java virtual machine must support:

```
StandardCharsets.UTF_8  
StandardCharsets.UTF_16  
StandardCharsets.UTF_16BE  
StandardCharsets.UTF_16LE  
StandardCharsets.ISO_8859_1  
StandardCharsets.US_ASCII
```

To obtain the `Charset` for another encoding, use the static `forName` method:

```
Charset shiftJIS = Charset.forName("Shift_JIS");
```

You use the `Charset` object to specify a character encoding. For example, you can turn an array of bytes into a string as

```
var contents = new String(bytes,  
StandardCharsets.ISO_8859_1);
```

9.1.5. Text Input

To read text input, use a `Reader`. You can obtain a `Reader` from any input stream with the `InputStreamReader` adapter:

```
InputStream inStream = ...;
var in = new InputStreamReader(inStream, charset);
```

If you want to process the input one UTF-16 code unit at a time, you can call the `read` method:

```
int ch = in.read();
```

The method returns a code unit between 0 and 65536, or -1 at the end of input.

That is not very convenient. Here are several alternatives.

With a short text file, you can read it into a string like this:

```
String content = Files.readString(path, charset);
```

But if you want the file as a sequence of lines, call

```
List<String> lines = Files.readAllLines(path, charset);
```

If the file is large, process them lazily as a `Stream<String>`:

```
try (Stream<String> lines = Files.lines(path, charset)) {
    ...
}
```



Note: If an `IOException` occurs as the stream fetches the lines, that exception is wrapped into an `UncheckedIOException` which is thrown out of the stream operation. This subterfuge is necessary because stream operations are not declared to throw any checked exceptions.

To read numbers or words from a file, use a `Scanner`, as you have seen in

[Chapter 1](#). For example,

```
var in = new Scanner(path);
while (in.hasNextDouble()) {
    double value = in.nextDouble();
    ...
}
```



Tip: To read alphabetic words, set the scanner's delimiter to a regular expression that is the complement of what you want to accept as a token. For example, after calling

```
in.useDelimiter("\\PL+");
```

the scanner reads in letters since any sequence of nonletters is a delimiter. See [Section 9.4.1](#) for the regular expression syntax.

You can then obtain a stream of all words as

```
Stream<String> words = in.tokens();
```

If your input does not come from a file, wrap the `InputStream` into a `BufferedReader`:

```
try (var reader = new BufferedReader(new
InputStreamReader(url.openStream()))) {
    Stream<String> lines = reader.lines();
    ...
}
```

A `BufferedReader` reads input in chunks for efficiency. (Oddly, this is not an option for basic readers.) It has methods `readLine` to read a single line and `lines` to yield a stream of lines.

If a method asks for a `Reader` and you want it to read from a file, call `Files.newBufferedReader(path, charset)`.

9.1.6. Text Output

To write text, use a `Writer`. With the `write` method, you can write strings. You can turn any output stream into a `Writer`:

```
OutputStream outStream = ...;
var out = new OutputStreamWriter(outStream, charset);
out.write(str);
```

To get a writer for a file, use

```
Writer out = Files.newBufferedWriter(path, charset);
```

It is more convenient to use a `PrintWriter`, which has the `print`, `println`, and `printf` that you have always used with `System.out`. Using those methods, you can print numbers and use formatted output.

If you write to a file, construct a `PrintWriter` like this:

```
var out = new PrintWriter(Files.newBufferedWriter(path,
charset));
```

If you write to another stream, use

```
var out = new PrintWriter(new OutputStreamWriter(outStream,
charset));
```



Note: `System.out` is an instance of `PrintStream`, not `PrintWriter`. This is a relic from the earliest days of Java. However, the `print`, `println`, and `printf` methods work the same way for the `PrintStream` and `PrintWriter` classes, using a character encoding for turning characters into bytes.

If you already have the text to write in a string, call

```
String content = ...;
Files.writeString(path, content, charset);
```

or

```
Files.write(path, lines, charset);
```

Here, `lines` can be a `Collection<String>`, or even more generally, an `Iterable<? extends CharSequence>`.

To append to a file, use

```
Files.writeString(path, charset, StandardOpenOption.APPEND);
Files.write(path, lines, charset, StandardOpenOption.APPEND);
```



Caution: When writing text with a partial character set such as ISO 8859-1, any unmappable characters are silently changed to a “replacement”—in most cases, either the ? character or the Unicode replacement character U+FFFD.

Sometimes, a library method wants a `Writer` to write output. If you want to capture that output in a string, hand it a `StringWriter`. Or, if it wants a `PrintWriter`, wrap the `StringWriter` like this:

```
var writer = new StringWriter();
throwable.printStackTrace(new PrintWriter(writer));
String stackTrace = writer.toString();
```

9.1.7. Reading Character Input

If you read a file with a structured format such as JSON or XML, you will use a parser that someone wrote who understands the fiddly details of that format. Such a parser typically reads a character at a time.

In the uncommon case that you need to write such a parser, use a `BufferedReader` for efficiency. Keep calling its `read` method, which yields a `char` value or -1 at the end of input. The reader converts the encoding of the input stream into UTF-16. You need to handle the UTF-16 encoding:

```
int ch = reader.read();
if (ch != -1)
{
    int codePoint;
    if (Character.isHighSurrogate((char) ch))
    {
        int ch2 = reader.read();
        if (Character.isLowSurrogate((char) ch2))
            codePoint = Character.toCodePoint(ch, ch2);
        else
            throw new MalformedInputException(2);
    }
    else
        codePoint = ch;
}
```

The `Character` class contains methods to tell whether a particular code point has a given property. For example,

`Character.isLetter(codePoint)`

returns `true` if `codePoint` is a letter in some language. Here are some other classification methods:

- `isUpperCase`
- `isLowerCase`
- `isDigit`
- `isSpaceChar`
- `isEmoji`

These methods use the rules of the Unicode standard. Others refer to the rules of the Java language:

- `isJavaIdentifierStart`
- `isJavaIdentifierPart`
- `isWhitespace`

After analyzing the code points, you often need to store them in strings, converting them back to UTF-16. The `appendCodePoint` method of the `StringBuilder` class turns a code point into one or two `char` values which are appended to the builder.

9.1.8. Reading and Writing Binary Data

The `DataInput` interface declares the following methods for reading a number, a character, a `boolean` value, or a string in binary format:

```
byte readByte()
int readUnsignedByte()
char readChar()
short readShort()
int readUnsignedShort()
int readInt()
long readLong()
float readFloat()
double readDouble()
void readFully(byte[] b)
```

The `DataOutput` interface declares corresponding `write` methods.



Note: These methods read and write numbers in big-endian format.



Caution: There are also `readUTF/writeUTF` methods that use a “modified UTF-8” format. These methods are *not* compatible with regular UTF-8, and are only useful for JVM internals.

The advantage of binary I/O is that it is fixed width and efficient. For example, `.writeInt` always writes an integer as a big-endian 4-byte binary quantity regardless of the number of digits. The space needed is the same for each value of a given type, which speeds up random access. Also, reading binary data is

faster than parsing text. The main drawback is that the resulting files cannot be easily inspected in a text editor.

You can use the `DataInputStream` and `DataOutputStream` adapters with any stream. For example,

```
DataInput in = new  
DataInputStream(Files.newInputStream(path));  
DataOutput out = new  
DataOutputStream(Files.newOutputStream(path));
```

9.1.9. Random-Access Files

The `RandomAccessFile` class lets you read or write data anywhere in a file. You can open a random-access file either for reading only or for both reading and writing; specify the option by using the string "r" (for read access) or "rw" (for read/write access) as the second argument in the constructor. For example,

```
var file = new RandomAccessFile(path.toString(), "rw");
```

A random-access file has a *file pointer* that indicates the position of the next byte to be read or written. The `seek` method sets the file pointer to an arbitrary byte position within the file. The argument to `seek` is a long integer between zero and the length of the file (which you can obtain with the `length` method). The `getFilePointer` method returns the current position of the file pointer.

The `RandomAccessFile` class implements both the `DataInput` and `DataOutput` interfaces. To read and write numbers from a random-access file, use methods such as `readInt/writeInt` that you saw in the preceding section. For example,

```
int value = file.readInt();  
file.seek(file.getFilePointer() - 4);  
file.writeInt(value + 1);
```

9.1.10. Memory-Mapped Files

Memory-mapped files provide another, very efficient approach for random access that works well for very large files. However, the API for data access is completely different from that of input/output streams. First, get a *channel* to the file:

```
FileChannel channel = FileChannel.open(path,  
    StandardOpenOption.READ, StandardOpenOption.WRITE)
```

Then, map an area of the file (or, if it is not too large, the entire file) into memory:

```
ByteBuffer buffer =  
    channel.map(FileChannel.MapMode.READ_WRITE,  
    0, channel.size());
```

Use methods `get`, `getInt`, `getDouble`, and so on to read values, and the equivalent `put` methods to write values.

```
int offset = ...;  
int value = buffer.getInt(offset);  
buffer.put(offset, value + 1);
```

At some point, and certainly when the channel is closed, these changes are written back to the file.



Note: By default, the methods for reading and writing numbers use big-endian byte order. You can change the byte order with the command

```
buffer.order(ByteOrder.LITTLE_ENDIAN);
```

9.1.11. File Locking

When multiple simultaneously executing programs modify the same file, they need to communicate in some way, or the file can easily become damaged. File locks can solve this problem.

Suppose your application saves a configuration file with user preferences. If a user invokes two instances of the application, it could happen that both of them want to write the configuration file at the same time. In that situation, the first instance should lock the file. When the second instance finds the file locked, it can decide to wait until the file is unlocked or simply skip the writing process. To lock a file, call either the `lock` or `tryLock` methods of the `FileChannel` class.

```
FileChannel channel = FileChannel.open(path,  
StandardOpenOption.WRITE);  
FileLock lock = channel.lock();
```

or

```
FileLock lock = channel.tryLock();
```

The first call blocks until the lock becomes available. The second call returns immediately, either with the lock or with `null` if the lock is not available. The file remains locked until the lock or the channel is closed. It is best to use a `try-with-resources` statement:

```
try (FileLock lock = channel.lock()) {  
    ...  
}
```

9.2. Paths, Files, and Directories

You have already seen `Path` objects for specifying file paths. In the following sections, you will see how to manipulate these objects and how to work with files and directories.

9.2.1. Paths

A `Path` is a sequence of directory names, optionally followed by a file name. The first component of a path may be a root component, such as `/` or `C:\`. The permissible root components depend on the file system. A path that starts with a root component is *absolute*. Otherwise, it is *relative*. For example, here we

construct an absolute and a relative path. For the absolute path, we assume we are running on a Unix-like file system.

```
Path absolute = Path.of("/", "home", "cay");
Path relative = Path.of("myapp", "conf", "user.properties");
```

The static `Path.of` method receives one or more strings, which it joins with the path separator of the default file system (/ for a Unix-like file system, \ for Windows). It then parses the result, throwing an `InvalidPathException` if the result is not a valid path in the given file system. The result is a `Path` object.

You can also provide a string with separators to the `Path.of` method:

```
Path homeDirectory = Path.of("/home/cay");
```



Note: A `Path` object does not have to correspond to a file that actually exists. It is merely an abstract sequence of names. To create a file, first make a path, then call a method to create the corresponding file—see [Section 9.2.2](#).

It is very common to combine or “resolve” paths. The call `p.resolve(q)` returns a path according to these rules:

- If `q` is absolute, then the result is `q`.
- if `q` does not have a root, then the result is obtained by joining `p` and `q`.
- Otherwise, the result depends on the rules of the file system.

For example, suppose your application needs to find its configuration file relative to the home directory. Here is how you can combine the paths:

```
Path workPath = homeDirectory.resolve("myapp/work");
// Same as homeDirectory.resolve(Path.of("myapp/work"));
```

There is a convenience method `resolveSibling` that resolves against a path’s parent, yielding a sibling path. For example, if `workPath` is `/home/cay/myapp/work`, the call

```
Path tempPath = workPath.resolveSibling("temp");  
yields /home/cay/myapp/temp.
```

The opposite of `resolve` is `relativize`. The call `p.relativize(r)` yields the path `q` which, when resolved with `p`, yields `r`. For example,

```
Path.of("/home/cay").relativize(Path.of("/home/fred/myapp"))
```

yields `../fred/myapp`, assuming we have a file system that uses `..` to denote the parent directory.

The `normalize` method removes any redundant `.` and `..` components (or whatever the file system may deem redundant). For example, normalizing the path `/home/cay/..../fred/./myapp` yields `/home/fred/myapp`.

The `toAbsolutePath` method yields the absolute path of a given path. If the path is not already absolute, it is resolved against the *working directory*—that is, the directory of the process in which the JVM was invoked. For example, if you launched a Java program from `/home/cay/myapp`, then `Path.of("config").toAbsolutePath()` returns `/home/cay/myapp/config`.



Note: You can obtain the working directory by a call to `System.getProperty("user.dir")`.

The `Path` interface has methods for taking paths apart and combining them with other paths. This code sample shows some of the most useful ones:

```
Path p = Path.of("/home", "cay", "myapp.properties");  
Path parent = p.getParent(); // The path /home/cay  
Path file = p.getFileName(); // The last element,  
myapp.properties  
Path root = p.getRoot(); // The initial segment / (null for a relative  
path)  
Path first = p.getName(0); // The first element  
Path dir = p.subpath(1, p.getNameCount());  
// All but the first element, cay/myapp.properties
```

The Path interface extends the Iterable<Path> element, so you can iterate over the name components of a Path with an enhanced for loop:

```
for (Path component : path) {  
    ...  
}
```



Note: Occasionally, you may need to interoperate with legacy APIs that use the File class instead of the Path interface. The Path interface has a toFile method, and the File class has a toPath method.

9.2.2. Creating Files and Directories

To create a new directory, call

```
Files.createDirectory(path);
```

All but the last component in the path must already exist. To create intermediate directories as well, use

```
Files.createDirectories(path);
```

You can create an empty file with

```
Files.createFile(path);
```

The call throws an exception if the file already exists. The checks for existence and the creation are atomic. If the file doesn't exist, it is created before anyone else has a chance to do the same.

The call `Files.exists(path)` checks whether the given file or directory exists. To test whether it is a directory or a “regular” file (that is, with data in it, not something like a directory or symbolic link), call the static methods `isDirectory` and `isRegularFile` of the `Files` class.

There are convenience methods for creating a temporary file or directory in a given or system-specific location.

```
Path tempFile = Files.createTempFile(dir, prefix, suffix);
Path tempFile = Files.createTempFile(prefix, suffix);
Path tempDir = Files.createTempDirectory(dir, prefix);
Path tempDir = Files.createTempDirectory(prefix);
```

Here, `dir` is a `Path`, and `prefix/suffix` are strings which may be null. For example, the call `Files.createTempFile(null, ".txt")` might return a path such as `/tmp/1234405522364837194.txt`.

9.2.3. Copying, Moving, and Deleting Files

To copy a file from one location to another, simply call

```
Files.copy(fromPath, toPath);
```

To move the file instead, call

```
Files.move(fromPath, toPath);
```

You can also use this command to move an empty directory.

The copy or move will fail if the target exists. If you want to overwrite an existing target, use the `REPLACE_EXISTING` option. If you want to copy all file attributes, use the `COPY_ATTRIBUTES` option. You can supply both like this:

```
Files.copy(fromPath, toPath,
StandardCopyOption.REPLACE_EXISTING,
StandardCopyOption.COPY_ATTRIBUTES);
```

You can specify that a move should be atomic. Then you are assured that either the move completed successfully, or the source continues to be present. Use the `ATOMIC_MOVE` option:

```
Files.move(fromPath, toPath, StandardCopyOption.ATOMIC_MOVE);
```

See [Table 9.3](#) for a summary of the options that are available for file operations.

Finally, to delete a file, simply call

```
Files.delete(path);
```

This method throws an exception if the file doesn't exist, so instead you may want to use

```
boolean deleted = Files.deleteIfExists(path);
```

The deletion methods can also be used to remove an empty directory.

Table 9.3: Standard Options for File Operations

Option	Description
StandardOpenOption; use with newBufferedWriter, newInputStream, newOutputStream, write	
READ	Open for reading.
WRITE	Open for writing.
APPEND	If opened for writing, append to the end of the file.
TRUNCATE_EXISTING	If opened for writing, remove existing contents.
CREATE_NEW	Create a new file and fail if it exists.
CREATE	Atomically create a new file if it doesn't exist.
DELETE_ON_CLOSE	Make a "best effort" to delete the file when it is closed.
SPARSE	A hint to the file system that this file will be sparse.
DSYNC SYNC	Requires that each update to the file data data and metadata be written

Option	Description
	synchronously to the storage device.
StandardCopyOption; use with <code>copy</code>, <code>move</code>	
ATOMIC_MOVE	Move the file atomically.
COPY_ATTRIBUTES	Copy the file attributes.
REPLACE_EXISTING	Replace the target if it exists.
LinkOption; use with all of the above methods and <code>exists</code>, <code>isDirectory</code>, <code>isRegularFile</code>	
NOFOLLOW_LINKS	Do not follow symbolic links.
FileVisitOption; use with <code>find</code>, <code>walk</code>, <code>walkFileTree</code>	
FOLLOW_LINKS	Follow symbolic links.

9.2.4. Visiting Directory Entries

The static `Files.list` method returns a `Stream<Path>` that reads the entries of a directory. The directory is read lazily, making it possible to efficiently process directories with huge numbers of entries.

Since reading a directory involves a system resource that needs to be closed, you should use a try-with-resources block:

```
try (Stream<Path> entries = Files.list(pathToDirectory)) {
    ...
}
```

The `list` method does not enter subdirectories. To process all descendants of a directory, use the `Files.walk` method instead.

```
try (Stream<Path> entries = Files.walk(pathToRoot)) {  
    // Contains all descendants, visited in depth-first order  
}
```

Here is a sample traversal of the unzipped `src.zip` tree:

```
java  
java/nio  
java/nio/DirectCharBufferU.java  
java/nio/ByteBufferAsShortBufferRL.java  
java/nio/MappedByteBuffer.java  
...  
java/nio/ByteBufferAsDoubleBufferB.java  
java/nio/charset  
java/nio/charset/CoderMalfunctionError.java  
java/nio/charset/CharsetDecoder.java  
java/nio/charset/UnsupportedCharsetException.java  
java/nio/charset/spi  
java/nio/charset/spi/CharsetProvider.java  
java/nio/charset/StandardCharsets.java  
java/nio/charset/Charset.java  
...  
java/nio/charset/CoderResult.java  
java/nio/HeapFloatBufferR.java  
...
```

As you can see, whenever the traversal yields a directory, it is entered before continuing with its siblings.

You can limit the depth of the tree that you want to visit by calling `Files.walk(pathToRoot, depth)`. Both `walk` methods have a varargs parameter of type `FileVisitOption...`, but there is only one option you can supply: `FOLLOW_LINKS` to follow symbolic links.



Note: If you filter the paths returned by `walk` and your filter criterion involves the file attributes stored with a directory, such as size, creation time, or type (file, directory, symbolic link), then use the `find` method

instead of `walk`. Call that method with a predicate function that accepts a path and a `BasicFileAttributes` object. The only advantage is efficiency. Since the directory is being read anyway, the attributes are readily available.

This code fragment uses the `Files.walk` method to copy one directory to another:

```
Files.walk(source).forEach(p -> {
    try {
        Path q = target.resolve(source.relativize(p));
        if (Files.isDirectory(p))
            Files.createDirectory(q);
        else
            Files.copy(p, q);
    } catch (IOException ex) {
        throw new UncheckedIOException(ex);
    }
});
```

Unfortunately, you cannot easily use the `Files.walk` method to delete a tree of directories since you need to first visit the children before deleting the parent. In that case, use the `walkFileTree` method. It requires an instance of the `FileVisitor` interface. Here is when the file visitor gets notified:

1. Before a directory is processed:

```
FileVisitResult preVisitDirectory(T dir, IOException ex)
```

2. When a file is encountered:

```
FileVisitResult visitFile(T path, BasicFileAttributes
                           attrs)
```

3. When an exception occurs in the `visitFile` method:

```
FileVisitResult visitFileFailed(T path, IOException ex)
```

4. After a directory is processed:

```
FileVisitResult postVisitDirectory(T dir, IOException ex)
```

In each case, the notification method returns one of the following results:

- Continue visiting the next file: `FileVisitResult.CONTINUE`
- Continue the walk, but without visiting the entries in this directory: `FileVisitResult.SKIP_SUBTREE`
- Continue the walk, but without visiting the siblings of this file: `FileVisitResult.SKIP_SIBLINGS`
- Terminate the walk: `FileVisitResult.TERMINATE`

If any of the methods throws an exception, the walk is also terminated, and that exception is thrown from the `walkFileTree` method.

The `SimpleFileVisitor` class implements this interface, continuing the iteration at each point and rethrowing any exceptions.

Here is how you can delete a directory tree:

```
Files.walkFileTree(root, new SimpleFileVisitor<Path>() {  
    public FileVisitResult visitFile(Path file,  
        BasicFileAttributes attrs) throws IOException {  
        Files.delete(file);  
        return FileVisitResult.CONTINUE;  
    }  
    public FileVisitResult postVisitDirectory(Path dir,  
        IOException ex) throws IOException {  
        if (ex != null) throw ex;  
        Files.delete(dir);  
        return FileVisitResult.CONTINUE;  
    }  
});
```



Caution: The `Files.walk` method throws an exception if any of the subdirectories are not readable. If you only want to visit readable

directories, use the `walkFileTree` method.

9.2.5. ZIP File Systems

The `Paths` class looks up paths in the default file system—the files on the user’s local disk. You can have other file systems. One of the more useful ones is a ZIP file system. If `zipname` is the name of a ZIP file, then the call

```
FileSystem zipfs =  
FileSystems.newFileSystem(Path.of(zipname));
```

establishes a file system that contains all files in the ZIP archive. It’s an easy matter to copy a file out of that archive if you know its name:

```
Files.copy(zipfs.getPath(sourceName), targetPath);
```

Here, `zipfs.getPath` is the analog of `Path.of` for an arbitrary file system.

To list all files in a ZIP archive, walk the file tree:

```
Files.walk(zipfs.getPath("/")).forEach(p -> {  
    Process p  
});
```

You have to work a bit harder to create a new ZIP file. Here is the magic incantation:

```
Path zipPath = Path.of("myfile.zip");  
var uri = new URI("jar", zipPath.toUri().toString(), null);  
    // Constructs the URI jar:file://myfile.zip  
try (FileSystem zipfs = FileSystems.newFileSystem(uri,  
        Collections.singletonMap("create", "true"))) {  
    // To add files, copy them into the ZIP file system  
    Files.copy(sourcePath,  
    zipfs.getPath("/").resolve(targetPath));  
}
```



Note: There is an older API for working with ZIP archives, with classes `ZipInputStream` and `ZipOutputStream`, but it's not as easy to use as the one described in this section.

9.3. HTTP Connections

You can read from a URL by using the input stream returned from `URL.getInputStream` method. However, if you want additional information about a web resource, or if you want to write data, you need more control over the process than the `URLConnection` class provides. The `URLConnection` class was designed before HTTP was the universal protocol of the Web. It provides support for a number of protocols, but its HTTP support is somewhat cumbersome. When the decision was made to support HTTP/2, it became clear that it would be best to provide a modern client interface instead of reworking the existing API. The `HttpClient` provides a more convenient API and HTTP/2 support.

In the following sections, I provide a cookbook for using the `HttpURLConnection` class, and then give an overview of the API.

9.3.1. The `URLConnection` and `HttpURLConnection` Classes

To use the `URLConnection` class, follow these steps:

1. Get an `URLConnection` object:

```
URLConnection connection = url.openConnection();
```

For an HTTP URL, the returned object is actually an instance of `HttpURLConnection`.

2. If desired, set request properties:

```
connection.setRequestProperty("Accept-Charset", "UTF-8,  
ISO-8859-1");
```

If a key has multiple values, separate them by commas.

3. To send data to the server, call

```
connection.setDoOutput(true);
try (OutputStream out = connection.getOutputStream()) {
    // Write to out
}
```

4. If you want to read the response headers and you haven't called `getOutputStream`, call

```
connection.connect();
```

Then query the header information:

```
Map<String, List<String>> headers =
connection.getHeaderFields();
```

For each key, you get a list of values since there may be multiple header fields with the same key.

5. Read the response:

```
try (InputStream in = connection.getInputStream()) {
    // Read from in
}
```

A common use case is to post form data. The `URLConnection` class automatically sets the content type to `application/x-www-form-urlencoded` when writing data to a HTTP URL, but you need to encode the name/value pairs:

```
URL url = ...;
URLConnection connection = url.openConnection();
connection.setDoOutput(true);
try (var out = new OutputStreamWriter(
        connection.getOutputStream())) {
    Map<String, String> postData = ...;
    boolean first = true;
    for (Map.Entry<String, String> entry :
```

```
postData.entrySet()) {
    if (first) first = false;
    else out.write("&");
    out.write(URLEncoder.encode(entry.getKey(), "UTF-
8"));
    out.write("=");
    out.write(URLEncoder.encode(entry.getValue(), "UTF-
8"));
}
try (InputStream in = connection.getInputStream()) {
    ...
}
```

9.3.2. The HTTP Client API

The HTTP client API provides another mechanism for connecting to a web server which is simpler than the `URLConnection` class with its rather fussy set of stages. More importantly, the implementation supports HTTP/2.

An `HttpClient` can issue requests and receive responses. You get a client by calling

```
HttpClient client = HttpClient.newHttpClient();
```

Alternatively, if you need to configure the client, use a builder API like this:

```
HttpClient client = HttpClient.newBuilder()
    .followRedirects(HttpClient.Redirect.ALWAYS)
    .build();
```

That is, you get a builder, call methods to customize the item that is going to be built, and then call the `build` method to finalize the building process. This is a common pattern for constructing immutable objects.

Follow the same pattern for formulating requests. Here is a GET request:

```
HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("https://horstmann.com"))
    .GET()
    .build();
```

The URI is the “uniform resource identifier” which is, when using HTTP, the same as a URL. However, in Java, the URL class has methods for actually opening a connection to a URL, whereas the URI class is only concerned with the syntax (scheme, host, port, path, query, fragment, and so on).

When sending the request, you have to tell the client how to handle the response. If you just want the body as a string, send the request with a `HttpResponse.BodyHandlers.ofString()`, like this:

```
HttpResponse<String> response
    = client.send(request,
        HttpResponse.BodyHandlers.ofString());
```

The `HttpResponse` class is a template whose type denotes the type of the body. You get the response body string simply as

```
String bodyString = response.body();
```

There are other response body handlers that get the response as a byte array or a file. One can hope that eventually the JDK will support JSON and provide a JSON handler.

With a POST request, you similarly need a “body publisher” that turns the request data into the data that is being posted. There are body publishers for strings, byte arrays, and files. Again, one can hope that the library designers will wake up to the reality that most POST requests involve form data, file uploads, or JSON objects, and provide appropriate publishers.

Nowadays, the most common POST request body contains JSON, which you need to convert to a string. Then you can form the following request:

```
HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create(urlString))
    .header("Content-Type", "application/json")
```

```
.POST(HttpRequest.BodyPublishers.ofString(jsonString))  
.build();
```

The book's companion code has examples for posting form data and file uploads.

The `HttpRequest.Builder` class also has build methods for the less common `PUT`, `DELETE`, and `HEAD` requests.

Java 16 adds a builder for filtering the headers of an existing `HttpRequest`. You provide the request and a function that receives the header names and values, returning `true` for those that should be retained. For example, here we modify the content type:

```
HttpRequest request2 = HttpRequest.newBuilder(request,  
        (name, value) -> !name.equalsIgnoreCase("Content-  
        Type")) // Remove old content type  
        .header("Content-Type", "application/xml") // Add new  
        content type  
        .build();
```

The `HttpResponse` object also yields the status code and the response headers.

```
int status = response.statusCode();  
HttpHeaders responseHeaders = response.headers();
```

You can turn the `HttpHeaders` object into a map:

```
Map<String, List<String>> headerMap = responseHeaders.map();
```

The map values are lists since in HTTP, each key can have multiple values.

If you just want the value of a particular key, and you know that there won't be multiple values, call the `firstValue` method:

```
Optional<String> lastModified = headerMap.firstValue("Last-  
Modified");
```

You get the response value or an empty optional if none was supplied.

The `HttpClient` is autocloseable, so you can declare it in a `try-with-resources` statement. Its `close` method waits for the completion of submitted requests and then closes its connection pool.



Tip: To enable logging for the `HttpClient`, add this line to `net.properties` in your JDK:

```
jdk.httpclient.HttpClient.log=all
```

Instead of `all`, you can specify a comma-separated list of `headers`, `requests`, `content`, `errors`, `ssl`, `trace`, and `frames`, optionally followed by `:control`, `:data`, `:window`, or `:all`. Don't use any spaces.

Then set the logging level for the logger named `jdk.httpclient.HttpClient` to `INFO`, for example by adding this line to the `logging.properties` file in your JDK:

```
jdk.httpclient.HttpClient.level=INFO
```

9.4. Regular Expressions

Regular expressions specify string patterns. Use them whenever you need to locate strings that match a particular pattern. For example, suppose you want to find hyperlinks in an HTML file. You need to look for strings of the pattern `>`. But wait—there may be extra spaces, or the URL may be enclosed in single quotes. Regular expressions give you a precise syntax for specifying what sequences of characters are legal matches.

In the following sections, you will see the regular expression syntax used by the Java API, and how to put regular expressions to work.

9.4.1. The Regular Expression Syntax

In a regular expression, a character denotes itself unless it is one of the reserved characters

. * + ? { | () [\ ^ \$

For example, the regular expression Java only matches the string Java.

The symbol . matches any single character. For example, .a.a matches Java and data.

The * symbol indicates that the preceding constructs may be repeated 0 or more times; for a +, it is 1 or more times. A suffix of ? indicates that a construct is optional (0 or 1 times). For example, be+s? matches be, bee, and bees. You can specify other multiplicities with { } (see [Table 9.4](#)).

A | denotes an alternative: .(oo|ee)f matches beef or woof. Note the parentheses—without them, .oo|eef would be the alternative between .oo and eef. Parentheses are also used for grouping—see [Section 9.4.4](#).

A *character class* is a set of character alternatives enclosed in brackets, such as [Jj], [0-9], [A-Za-z], or [^0-9]. Inside a character class, the - denotes a range (all characters whose Unicode values fall between the two bounds). However, a - that is the first or last character in a character class denotes itself. A ^ as the first character in a character class denotes the complement (all characters except those specified).

[Table 9.4](#) contains a number of *predefined character classes* such as \d (digits). There are many more with the \p prefix, such as \p{Sc} (Unicode currency symbols)—see [Table 9.5](#).

The characters ^ and \$ match the beginning and end of input.

If you need to have a literal . * + ? { | () [\ ^ \$, precede it by a backslash. Inside a character class, you only need to escape [and \, provided you are careful about the positions of] - ^-. For example, []^~-] is a class containing all three of them.



Caution: If the regular expression is in a string literal, each backslash needs to be escaped with another backslash. If you forget that second backslash, you usually get an error because sequences such as \\$ or \.

are not valid in string literals. But if you want to match a word boundary and accidentally use `\b` instead of `\b`, then you have a problem: `\b` is a valid escape sequence, indicating a backspace.

Instead of using backslashes, you can surround a string with `\Q` and `\E`. For example, `\($0\.99\)` and `\Q($0.99)\E` both match the string `($0.99)`.



Tip: If you have a string that may contain some of the many special characters in the regular expression syntax, you can escape them all by calling `Pattern.quote(str)`. This simply surrounds the string with `\Q` and `\E`, but it takes care of the special case where `str` may contain `\E`.

Table 9.4: Regular Expression Syntax

Expression	Description	Example
Characters		
<code>c</code> , not one of . * + ? { () [\ ^ \$	The character <code>c</code> .	J
.	Any character except line terminators, or any character if the DOTALL flag is set.	
<code>\X</code>	Any Unicode “extended grapheme cluster”, which is perceived as a character or symbol	
<code>\x{p}</code>	The Unicode code point with hex code <code>p</code> .	<code>\x{1D546}</code>
<code>\uhhhh,</code> <code>\xhh, \0o,</code>	The UTF-16 code unit with the given hex or octal value.	<code>\uFEFF</code>

Expression	Description	Example
\0oo, \0ooo		
\a, \e, \f, \n, \r, \t	Alert (\x{7}), escape (\x{1B}), form feed (\x{B}), newline (\x{A}), carriage return (\x{D}), tab (\x{9}).	\n
\cc, where c is in [A-Z] or one of @ [\] ^ _ ?	The control character corresponding to the character c.	\cH is a backspace (\x{8}).
\c, where c is not in [A-Za-zA-Z0-9]	The character c.	\\
\Q ... \E	Everything between the start and the end of the quotation.	\Q(...)\\E matches the string (...).
Character Classes		
[C ₁ C ₂ ...], where C _i are characters, ranges c-d, or character classes	Any of the characters represented by C ₁ , C ₂ ...	[0-9+-]
[^...]	Complement of a character class.	[^\d\s]
[...&&...]	Intersection of character classes.	[\p{L} && [^A-Za-z]]

Expression	Description	Example
\p{...},\P{...}	A predefined character class (see Table 9.5); its complement.	\p{L} matches a Unicode letter, and so does \p{L}—you can omit braces around a single letter.
\d, \D	Digits ([0-9], or \p{Digit} when the UNICODE_CHARACTER_CLASS flag is set); the complement.	\d+ is a sequence of digits.
\w, \W	Word characters ([a-zA-Z0-9_], or Unicode word characters when the UNICODE_CHARACTER_CLASS flag is set); the complement.	
\s, \S	Spaces ([\n\r\t\f\x{B}], or \p{IsWhite_Space} when the UNICODE_CHARACTER_CLASS flag is set); the complement.	\s*, \s* is a comma surrounded by optional white space.
\h, \v, \H, \V	Horizontal whitespace, vertical whitespace, their complements.	
Sequences and Alternatives		
<i>XY</i>	Any string from <i>X</i> , followed by any string from <i>Y</i> .	[1-9][0-9]* is a positive number without leading zero.
<i>X Y</i>	Any string from <i>X</i> or <i>Y</i> .	http ftp

Expression	Description	Example
Grouping		
(X)	Captures the match of X.	'([^\"]*)' captures the quoted text.
\n	The <i>n</i> th group.	(["\"]).*\1 matches 'Fred' or "Fred" but not "Fred".
(?<name>X)	Captures the match of X with the given name.	'(?<id>[A-Za-z0-9]+)' captures the match with name id.
\k<name>	The group with the given name.	\k<id> matches the group with name id.
(?:X)	Use parentheses without capturing X.	In (?:http ftp)://(.*), the match after // is \1.
(? <i>f₁f₂...:X>), (?<i>f₁...-f_k:X>), with <i>f_i</i> in [dimsuUx])</i></i>	Matches, but does not capture, X with the given flags on or off (after -).	(?i:jpe?g) is a case-insensitive match.
Other (?...)	See the Pattern API documentation.	
Quantifiers		
X?	Optional X.	\+? is an optional + sign.

Expression	Description	Example
X^*, X^+	0 or more X , 1 or more X .	$[1-9][0-9]^+$ is an integer ≥ 10 .
$X\{n\}, X\{n,\}, X\{m,n\}$	n times X , at least n times X , between m and n times X .	$[0-7]\{1,3\}$ are one to three octal digits.
$Q?$, where Q is a quantified expression	Reluctant quantifier, attempting the shortest match before trying longer matches.	$.^*(<.+?>).^*$ captures the shortest sequence enclosed in angle brackets.
Q^+ , where Q is a quantified expression	Possessive quantifier, taking the longest match without backtracking.	'[^']*+' matches strings enclosed in single quotes and fails quickly on strings without a closing quote.
Boundary Matches		
$^$ $$$	Beginning, end of input (or beginning, end of line in multiline mode).	Java$ matches the input or line Java.
\A \Z \z	Beginning of input, end of input, absolute end of input (unchanged in multiline mode).	
\b \B	Word boundary, nonword boundary.	$\bJava\b$ matches the word Java.
$\b\{g\}$	Grapheme cluster boundary	Useful with <code>split</code> to decompose a string into grapheme clusters
\R	A Unicode line break.	

Expression	Description	Example
\G	The end of the previous match.	

Table 9.5: Predefined Character Classes \p{...}

Name	Description
<i>posixClass</i>	<i>posixClass</i> is one of Lower, Upper, Alpha, Digit, Alnum, Punct, Graph, Print, Cntrl, XDigit, Space, Blank, ASCII, interpreted as POSIX or Unicode class, depending on the UNICODE_CHARACTER_CLASS flag.
IsScript, sc=Script, script=Script	A script accepted by Character.UnicodeScript.forName.
InBlock, blk=Block, block=Block	A block accepted by Character.UnicodeBlock.forName.
Category, InCategory, gc=Category, general_category=Category	A one- or two-letter name for a Unicode general category.
IsProperty	<i>Property</i> is one of Alphabetic, Ideographic, Letter, Lowercase, Uppercase, Titlecase, Punctuation, Control, White_Space, Digit, Hex_Digit, Join_Control, Noncharacter_Code_Point, Assigned.
javaMethod	Invokes the method Character.isMethod (must not be

Name	Description
	deprecated).

9.4.2. Testing a Match

Generally, there are two ways to use a regular expression: Either you want to test whether a string conforms to the expression, or you want to find all matches of the expressions in a string.

In the first case, simply use the static `matches` method:

```
String regex = "[+-]?\d+";
CharSequence input = ...;
if (Pattern.matches(regex, input)) {
    ...
}
```

If you need to use the same regular expression many times, it is more efficient to compile it. Then, create a `Matcher` for each input:

```
Pattern pattern = Pattern.compile(regex);
Matcher matcher = pattern.matcher(input);
if (matcher.matches()) ...
```

If the match succeeds, you can retrieve the location of matched groups—see [Section 9.4.4](#).

If you want to test whether the input *contains* a match, use the `find` method instead:

```
if (matcher.find()) ...
```

The `find` and `match` methods mutate the state of the `Matcher` object. If you just want to find out whether a given `Matcher` has already found a match, call the `hasMatch` method instead.

You can turn the pattern into a predicate:

```
Pattern digits = Pattern.compile("[0-9]+");
List<String> strings = List.of("December", "31st", "1999");
List<String> matchingStrings = strings.stream()
    .filter(digits.asMatchPredicate())
    .toList(); // ["1999"]
```

The result contains all strings that match the regular expression.

Use the `asPredicate` method to test whether a string contains a match:

```
List<String> stringsContainingMatch = strings.stream()
    .filter(digits.asPredicate())
    .toList(); // ["31st", "1999"]
```

9.4.3. Finding All Matches

In this section, we consider the other common use case for regular expressions —finding all matches in an input. Use this loop:

```
String input = ...;
Matcher matcher = pattern.matcher(input);
while (matcher.find()) {
    String match = matcher.group();
    int matchStart = matcher.start();
    int matchEnd = matcher.end();
    ...
}
```

In this way, you can process each match in turn. As shown in the code fragment, you can get the matched string as well as its position in the input string.

More elegantly, you can call the `results` method to get a `Stream<MatchResult>`. The `MatchResult` interface has methods `group`, `start`, and `end`, just like `Matcher`. (In fact, the `Matcher` class implements this interface.) Here is how you get a list of all matches:

```
List<String> matches = pattern.matcher(input)
    .results()
    .map(MatchResult::group)
    .toList();
```

If you have the data in a file, then you can use the `Scanner.findAll` method to get a `Stream<MatchResult>`, without first having to read the contents into a string. You can pass a `Pattern` or a pattern string:

```
var in = new Scanner(path);
Stream<String> words = in.findAll("\p{L+}")
    .map(MatchResult::group);
```

9.4.4. Groups

It is common to use groups for extracting components of a match. For example, suppose you have a line item in the invoice with item name, quantity, and unit price such as

Blackwell Toaster USD29.95

Here is a regular expression with groups for each component:

```
(\p{Alnum}+(\s+\p{Alnum}+)*)\s+([A-Z]{3})([0-9.]*)
```

After matching, you can extract the `n`th group from the matcher as

```
String contents = matcher.group(n);
```

Groups are ordered by their opening parenthesis, starting at 1. (Group 0 is the entire input.) In this example, here is how to take the input apart:

```
Matcher matcher = pattern.matcher(input);
if (matcher.matches()) {
    item = matcher.group(1);
    currency = matcher.group(3);
    price = matcher.group(4);
}
```

We aren't interested in group 2; it only arose from the parentheses that were required for the repetition. For greater clarity, you can use a noncapturing group:

```
(\p{Alnum}+(:\s+\p{Alnum}+)*)\s+([A-Z]{3})([0-9.]*)
```

Or, even better, use named groups:

```
(?<item>\p{Alnum}+(\s+\p{Alnum}+)*)\s+(?<currency>[A-Z]{3})(?<price>[0-9.]* )
```

Then, you can retrieve the items by name:

```
item = matcher.group("item");
```

With the `start` and `end` methods of the `Matcher` and `MatchResult` classes, you can get the group positions in the input:

```
int itemStart = matcher.start("item");
int itemEnd = matcher.end("item");
```

The `namedGroups` method yields a `Map<String, Integer>` from group names to numbers.



Note: When you have a group inside a repetition, such as `(\s+\p{Alnum}+)*` in the example above, it is not possible to get all of its matches. The `group` method only yields the last match, which is rarely useful. You need to capture the entire expression with another group.

9.4.5. Splitting along Delimiters

Sometimes, you want to break an input along matched delimiters and keep everything else. The `Pattern.split` method automates this task. You obtain an array of strings, with the delimiters removed:

```
String input = ...;
Pattern commas = Pattern.compile("\\s*,\\s*");
String[] tokens = commas.split(input);
// "1, 2, 3" turns into ["1", "2", "3"]
```

If there are many tokens, you can fetch them lazily:

```
Stream<String> tokens = commas.splitAsStream(input);
```

To also capture the delimiters, use the `splitWithDelimiters` method:

```
tokens = commas.splitWithDelimiters(input, -1); // ["1", ",",
", "2", ", ", "3", ", ", "4"]
```

If the second argument is a positive number n , the separator pattern is applied at most $n - 1$ times. and the last element is the remaining string. Otherwise, the pattern is applied as often as possible. With a limit of zero, trailing empty strings are discarded.

If you don't care about precompiling the pattern or lazy fetching, you can just use the `split` and `splitWithDelimiter` methods of the `String` class:

```
tokens = input.split("\\s*,\\s*");
```



Caution: It is easy to forget that the argument of `split` is a regular expression. For example,

```
"com.horstmann.corejava".split(".")
```

does not split along the dots. Instead, every character is a separator, and the result is an empty array!

You need to escape the dot with a backslash in the regular expression, and therefore with two backslashes in the string literal:

```
"com.horstmann.corejava".split("\\\\.")
```

Alternatively, use the `Pattern.quote` method:

```
"com.horstmann.corejava".split(Pattern.quote("."));
```

If the input is in a file, use a scanner:

```
var in = new Scanner(path);
in.useDelimiter("\\s*,\\s*");
Stream<String> tokens = in.tokens();
```

9.4.6. Replacing Matches

If you want to replace all matches of a regular expression with a string, call `replaceAll` on the matcher:

```
Matcher matcher = commas.matcher(input);
String result = matcher.replaceAll(",");
// Normalizes the commas
```

Or, if you don't care about precompiling, use the `replaceAll` method of the `String` class.

```
String result = input.replaceAll("\\s*,\\s*", ",");
```

The replacement string can contain group numbers `$n` or names `${name}`. They are replaced with the contents of the corresponding captured group.

```
String result = "3:45".replaceAll(
    "(\\d{1,2}):(?<minutes>\\d{2})",
    "$1 hours and ${minutes} minutes");
// Sets result to "3 hours and 45 minutes"
```

You can use `\` to escape `$` and `\` in the replacement string, or you can call the `Matcher.quoteReplacement` convenience method:

```
matcher.replaceAll(Matcher.quoteReplacement(str))
```

If you want to carry out a more complex operation than splicing in group matches, then you can provide a replacement function instead of a replacement

string. The function accepts a `MatchResult` and yields a string. For example, here we replace all words with at least four letters with their uppercase version:

```
String result = Pattern.compile("\\pL{4,}")
    .matcher("Mary had a little lamb")
    .replaceAll(m -> m.group().toUpperCase());
// Yields "MARY had a LITTLE LAMB"
```

The `replaceFirst` method replaces only the first occurrence of the pattern.

9.4.7. Flags

Several *flags* change the behavior of regular expressions. You can specify them when you compile the pattern:

```
Pattern pattern = Pattern.compile(regex,
    Pattern.CASE_INSENSITIVE |
    Pattern.UNICODE_CHARACTER_CLASS);
```

Or you can specify them inside the pattern:

```
String regex = "(?iU:expression)";
```

Here are the flags:

- `Pattern.CASE_INSENSITIVE` or `i`: Match characters independently of the letter case. By default, this flag takes only US ASCII characters into account.
- `Pattern.UNICODE_CASE` or `u`: When used in combination with `CASE_INSENSITIVE`, use Unicode letter case for matching.
- `Pattern.UNICODE_CHARACTER_CLASS` or `U`: Select Unicode character classes instead of POSIX. Implies `UNICODE_CASE`.
- `Pattern.MULTILINE` or `m`: Make `^` and `$` match the beginning and end of a line, not the entire input.
- `Pattern.UNIX_LINES` or `d`: Only '`\n`' is a line terminator when matching `^` and `$` in multiline mode.
- `Pattern.DOTALL` or `s`: Make the `.` symbol match all characters, including line terminators.

- `Pattern.COMMENTS` or `x`: Whitespace and comments (from # to the end of a line) are ignored.
- `Pattern.LITERAL`: The pattern is taking literally and must be matched exactly, except possibly for letter case.
- `Pattern.CANON_EQ`: Take canonical equivalence of Unicode characters into account. For example, u followed by “ (diaeresis) matches ü.

The last two flags cannot be specified inside a regular expression.

9.5. Serialization

In the following sections, you will learn about object serialization—a mechanism for turning an object into a bunch of bytes that can be shipped somewhere else or stored on disk, and for reconstituting the object from those bytes.

Serialization is an essential tool for distributed processing, where objects are shipped from one virtual machine to another. It is also used for fail-over and load balancing, when serialized objects can be moved to another server. If you work with server-side software, you will often need to enable serialization for classes. The following sections tell you how to do that.

9.5.1. The Serializable Interface

In order for an object to be serialized—that is, turned into a bunch of bytes—it must be an instance of a class that implements the `Serializable` interface. This is a marker interface with no methods, similar to the `Cloneable` interface that you saw in

Chapter 4.

For example, to make `Employee` objects serializable, the class needs to be declared as

```
public class Employee implements Serializable {  
    private String name;  
    private double salary;  
    ...  
}
```

It is appropriate for a class to implement the `Serializable` interface if all instance variables have primitive or `enum` type, or contain references to serializable objects. Many classes in the standard library are serializable. Arrays and the collection classes that you saw in

[Chapter 7](#) are serializable provided their elements are.

In the case of the `Employee` class, and indeed with most classes, there is no problem. In the following sections, you will see what to do when a little extra help is needed.

To serialize objects, you need an `ObjectOutputStream`, which is constructed with another `OutputStream` that receives the actual bytes.

```
var out = new  
ObjectOutputStream(Files.newOutputStream(path));
```

Now call the `writeObject` method:

```
var peter = new Employee("Peter", 90000);  
var paul = new Manager("Paul", 180000);  
out.writeObject(peter);  
out.writeObject(paul);
```

To read the objects back in, construct an `ObjectInputStream`:

```
var in = new ObjectInputStream(Files.newInputStream(path));
```

Retrieve the objects in the same order in which they were written, using the `readObject` method.

```
var e1 = (Employee) in.readObject();  
var e2 = (Employee) in.readObject();
```

When an object is written, the name of the class and the names and values of all instance variables are saved. If the value of an instance variable belongs to a primitive type, it is saved as binary data. If it is an object, it is again written with the `writeObject` method.

When an object is read in, the process is reversed. The class name and the names and values of the instance variables are read, and the object is reconstituted.

There is just one catch. Suppose there were two references to the same object. Let's say each employee has a reference to their boss:

```
var peter = new Employee("Peter", 90000);
var paul = new Manager("Barney", 105000);
var mary = new Manager("Mary", 180000);
peter.setBoss(mary);
paul.setBoss(mary);
out.writeObject(peter);
out.writeObject(paul);
```

When reading these two objects back in, both of them need to have the *same* boss, not two references to identical but distinct objects.

In order to achieve this, each object gets a *serial number* when it is saved. When you pass an object reference to `writeObject`, the `ObjectOutputStream` checks if the object reference was previously written. In that case, it just writes out the serial number and does not duplicate the contents of the object.

In the same way, an `ObjectInputStream` remembers all objects it has encountered. When reading in a reference to a repeated object, it simply yields a reference to the previously read object.



Note: If the superclass of a serializable class is not serializable, it must have an accessible no-argument constructor. Consider this example:

```
class Person // Not serializable
class Employee extends Person implements Serializable
```

When an `Employee` object is deserialized, its instance variables are read from the object input stream, but the `Person` instance variables are set by the `Person` constructor.

9.5.2. Transient Instance Variables

Certain instance variables should not be serialized—for example, database connections that are meaningless when an object is reconstituted. Also, when an object keeps a cache of values, it might be better to drop the cache and recompute it instead of storing it.

To prevent an instance variable from being serialized, simply tag it with the `transient` modifier. Always mark instance variables as transient if they hold instances of nonserializable classes. Transient instance variables are skipped when objects are serialized.

9.5.3. The `readObject` and `writeObject` Methods

In rare cases, you need to tweak the serialization mechanism. A serializable class can add any desired action to the default read and write behavior, by defining methods with the signature

```
@Serial private void readObject(ObjectInputStream in)
    throws IOException, ClassNotFoundException
@Serial private void writeObject(ObjectOutputStream out)
    throws IOException
```

Then, the object headers continue to be written as usual, but the instance variables fields are no longer automatically serialized. Instead, these methods are called.

Note the `@Serial` annotation. The methods for tweaking serialization don't belong to interfaces. Therefore, you can't use the `@Override` annotation to have the compiler check the method declarations. The `@Serial` annotation is meant to enable the same checking for serialization methods. Up to Java 17, the `javac` compiler doesn't do that checking, but it might happen in the future. Some IDEs check the annotation.

A number of classes in the `java.awt.geom` package, such as `Point2D.Double`, are not serializable. Now, suppose you want to serialize a class `LabeledPoint` that stores a `String` and a `Point2D.Double`. First, you need to mark the `Point2D.Double` field as `transient` to avoid a `NotSerializableException`.

```
public class LabeledPoint implements Serializable {  
    private String label;  
    private transient Point2D.Double point;  
    ...  
}
```

In the `writeObject` method, first write the object descriptor and the `String` field, `label`, by calling the `defaultWriteObject` method. This is a special method of the `ObjectOutputStream` class that can only be called from within a `writeObject` method of a serializable class. Then we write the point coordinates, using the standard `DataOutput` calls.

```
@Serial before private void writeObject(ObjectOutputStream out) throws IOException {  
    out.defaultWriteObject();  
    out.writeDouble(point.getX());  
    out.writeDouble(point.getY());  
}
```

In the `readObject` method, we reverse the process:

```
@Serial before private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {  
    in.defaultReadObject();  
    double x = in.readDouble();  
    double y = in.readDouble();  
    point = new Point2D.Double(x, y);  
}
```

Another example is the `HashSet` class that supplies its own `readObject` and `writeObject` methods. Instead of saving the internal structure of the hash table, the `writeObject` method simply saves the capacity, load factor, size, and elements. The `readObject` method reads back the capacity and load factor, constructs a new table, and inserts the elements.

The `readObject` and `writeObject` methods only need to save and load their data. They do not concern themselves with superclass data or any other class information.

The `Date` class uses this approach. Its `writeObject` method saves the milliseconds since the “epoch” (January 1, 1970). The data structure that caches calendar data is not saved.



Caution: Just like a constructor, the `readObject` method operates on partially initialized objects. If you call a non-final method inside `readObject` that is overridden in a subclass, it may access uninitialized data.



Note: If a serializable class defines a field

```
@Serial private static final ObjectStreamField[]  
serialPersistentFields
```

then serialization uses those field descriptors instead of the non-transient non-static fields. There is also an API for setting the field values before serialization or reading them after deserialization. This is useful for preserving a legacy layout after a class has evolved. For example, the `BigDecimal` class uses this mechanism to serialize its instances in a format that no longer reflects the instance fields.

9.5.4. The `readExternal` and `writeExternal` Methods

Instead of letting the serialization mechanism save and restore object data, a class can define its own mechanism. For example, you can encrypt the data or use a format that is more efficient than the serialization format.

To do this, a class must implement the `Externalizable` interface. This, in turn, requires it to define two methods:

```
public void readExternal(ObjectInputStream in)  
    throws IOException  
public void writeExternal(ObjectOutputStream out)  
    throws IOException
```

Unlike the `readObject` and `writeObject` methods, these methods are fully responsible for saving and restoring the entire object, *including the superclass data*. When writing an object, the serialization mechanism merely records the class of the object in the output stream. When reading an externalizable object, the object input stream creates an object with the no-argument constructor and then calls the `readExternal` method.

In this example, the `LabeledPixel` class extends the serializable `Point` class, but it takes over the serialization of the class and superclass. The fields of the object are not stored in the standard serialization format. Instead, the data are placed in an opaque block.

```
public class LabeledPixel extends Point implements
Externalizable {
    private String label;

    public LabeledPixel() {} // required for externalizable
class

    @Override public void writeExternal(ObjectOutput out)
        throws IOException {
        out.writeInt((int) getX());
        out.writeInt((int) getY());
        out.writeUTF(label);
    }

    @Override public void readExternal(ObjectInput in)
        throws IOException, ClassNotFoundException {
        int x = in.readInt();
        int y = in.readInt();
        setLocation(x, y);
        label = in.readUTF();
    }

    ...
}
```



Note: The `readExternal` and `writeExternal` methods should not be annotated with `@Serial`. Since they are defined in the `Externalizable` interface, you can simply annotate them with `@Override`.



Caution: Unlike the `readObject` and `writeObject` methods, which are private and can only be called by the serialization mechanism, the `readExternal` and `writeExternal` methods are public. In particular, `readExternal` potentially permits modification of the state of an existing object.



Note: You cannot customize the serialization of enumerations and records. If you define `readObject/writeObject` or `readExternal/writeExternal` methods, they are not used for serialization.

9.5.5. The `readResolve` and `writeReplace` Methods

We take it for granted that objects can only be constructed with the constructor. However, a deserialized object is *not constructed*. Its instance variables are simply restored from an object stream.

This is a problem if the constructor enforces some condition. For example, a singleton object may be implemented so that the constructor can only be called once. As another example, database entities can be constructed so that they always come from a pool of managed instances.

You shouldn't implement your own mechanism for singletons. If you need a singleton, make an enumerated type with one instance that is, by convention, called `INSTANCE`.

```
public enum PersonDatabase {  
    INSTANCE;
```

```
public Person findById(int id) { ... }  
...  
}
```

This works because `enum` are guaranteed to be deserialized properly.

Now let's suppose that you are in the rare situation where you want to control the identity of each deserialized instance. As an example, suppose a `Person` class wants to restore its instances from a database when deserializing. Then you should not serialize the object itself. Instead request that a proxy instance is saved. When restored, that proxy locates and constructs the desired object. Your class needs to provide a `writeReplace` method that returns the proxy object:

```
public class Person implements Serializable {  
    private int id;  
    // Other instance variables  
    ...  
    @Serial private Object writeReplace() {  
        return new PersonProxy(id);  
    }  
}
```

When a `Person` object is serialized, none of its instance variables are saved. Instead, the `writeReplace` method is called and *its return value* is serialized and written to the stream.

The proxy class needs to implement a `readResolve` method that yields a `Person` instance:

```
class PersonProxy implements Serializable {  
    private int id;  
  
    public PersonProxy(int id) {  
        this.id = id;  
    }  
  
    @Serial private Object readResolve() {
```

```
        return PersonDatabase.INSTANCE.findById(id);  
    }  
}
```

When the `readObject` method finds a `PersonProxy` in an `ObjectInputStream`, it deserializes the proxy, calls its `readResolve` method, and returns the result.



Note: Unlike the `readObject` and `writeObject` methods, the `readResolve` and `writeReplace` methods need not be private.



Note: With enumerations and records, `readObject/writeObject` or `readExternal/writeExternal` methods are not used for serialization. With records, but not with enumerations, the `writeReplace` method will be used.

9.5.6. Versioning

Serialization was intended for sending objects from one virtual machine to another, or for short-term persistence of state. If you use serialization for long-term persistence, or in any situation where classes can change between serialization and deserialization, you will need to consider what happens when your classes evolve. Can version 2 read the old data? Can the users who still use version 1 read the files produced by the new version?

The serialization mechanism supports a simple versioning scheme. When an object is serialized, both the name of the class and its `serialVersionUID` are written to the object stream. That unique identifier is assigned by the implementor, by defining an instance variable

```
@Serial private static final long serialVersionUID = 1L; //  
Version 1
```

When the class evolves in an incompatible way, the implementor should change the UID. Whenever a deserialized object has a nonmatching UID, the

`readObject` method throws an `InvalidClassException`.

If the `serialVersionUID` matches, deserialization proceeds even if the implementation has changed. Each non-transient instance variable of the object to be read is set to the value in the serialized state, provided that the name and type match. All other instance variables are set to the default: `null` for object references, zero for numbers, and `false` for boolean values. Anything in the serialized state that doesn't exist in the object to be read is ignored.

Is that process safe? Only the implementor of the class can tell. If it is, then the implementor should give the new version of the class the same `serialVersionUID` as the old version.

If you don't assign a `serialVersionUID`, one is automatically generated by hashing a canonical description of the instance variables, methods, and supertypes. You can see the hash code with the `serialver` utility. The command

```
serialver ch09.sec05.Employee
```

displays

```
private static final long serialVersionUID =  
-4932578720821218323L;
```

When the class implementation changes, there is a very high probability that the hash code changes as well.

If you need to be able to read old version instances, and you are certain that is safe to do so, run `serialver` on the old version of your class and add the result to the new version.



Note: If you want to implement a more sophisticated versioning scheme, override the `readObject` method and call the `readFields` method instead of the `defaultReadObject` method. You get a description of all fields found in the stream, and you can do with them what you want.



Note: Enumerations and records ignore the `serialVersionUID` field.

An enumeration always has a `serialVersionUID` of `0L`. You can declare the `serialVersionUID` of a record, but the IDs don't have to match for deserialization.



Note: In this section, you saw what happens when the reader's version of a class has instance variables that aren't present in the object stream. It is also possible during class evolution for a superclass to be added. Then a reader using the new version may read an object stream in which the instance variables of the superclass are not set. By default, those instance fields are set to their `0/false/null` default. That may leave the superclass in an unsafe state. The superclass can defend against that problem by defining an initialization method

```
@Serial private void readObjectNoData() throws  
ObjectStreamException
```

The method should either set the same state as the no-argument constructor or throw an `InvalidObjectException`. It is only called in the unusual circumstance where an object stream is read that contains an instance of a subclass with missing superclass data.

9.5.7. Deserialization and Security

During deserialization of a serializable class, objects are created without invoking any constructor of the class. Even if the class has a no-argument constructor, it is not used. The field values are set directly from the values of the object input stream.



Note: For serializable *records*, deserialization calls the canonical constructor, passing it the values of the components from the object input stream. (As a consequence, cyclic references in records are not restored.)

Bypassing construction is a security risk. An attacker can craft bytes describing an invalid object that could have never been constructed. Suppose, for example, that the `Employee` constructor throws an exception when called with a negative salary. We would like to think that no `Employee` object can have a negative salary as a result. But it is not difficult to inspect the bytes for a serialized object and modify some of them. This way, one can craft bytes for an employee with a negative salary and then deserialize them.

A serializable class can optionally implement the `ObjectInputValidation` interface and define a `validateObject` method to check whether its objects are properly deserialized. For example, the `Employee` class can check that salaries are not negative:

```
public void validateObject() throws InvalidObjectException {  
    System.out.println("validateObject");  
    if (salary < 0)  
        throw new InvalidObjectException("salary < 0");  
}
```

Unfortunately, the method is not invoked automatically. To invoke it, you also must provide the following method:

```
@Serial private void readObject(ObjectInputStream in)  
    throws IOException, ClassNotFoundException {  
    in.registerValidation(this, 0);  
    in.defaultReadObject();  
}
```

The object is then scheduled for validation, and the `validateObject` method is called when this object and all dependent objects have been loaded. The second parameter lets you specify a priority. Validation requests with higher priorities are done first.

There are other security risks. Adversaries can create data structures that consume enough resources to crash a virtual machine. More insidiously, any class on the class path can be deserialized. Hackers have been devious about piecing together “gadget chains”—sequences of operations in various utility

classes that use reflection and culminate in calling methods such as `Runtime.exec` with a string of their choice.

Any application that receives serialized data from untrusted sources over a network connection is vulnerable to such attacks. For example, some servers serialize session data and deserialize whatever data are returned in the HTTP session cookie.

You should avoid situations in which arbitrary data from untrusted sources are deserialized. In the example of session data, the server should sign the data, and only deserialize data with a valid signature.

A *serialization filter* mechanism can harden applications from such attacks. The filters see the names of serialized classes and several metrics (stream size, array sizes, total number of references, longest chain of references). Based on those data, the deserialization can be aborted.

In its simplest form, you provide a pattern describing the valid and invalid classes. For example, if you start our sample serialization demo as

```
java -Djdk.serialFilter='serial.*;java.**;!*'  
serial.ObjectStreamTest
```

then the objects will be loaded. The filter allows all classes in the `serial` package and all classes whose package name starts with `java`, but no others. If you don't allow `java.**`, or at least `java.util.Date`, deserialization fails.

You can place the filter pattern into a configuration file and specify multiple filters for different purposes. You can also implement your own filters. See <https://docs.oracle.com/en/java/javase/17/core/serialization-filtering1.xhtml> for details.

Exercises

1. Write a utility method for copying all of an `InputStream` to an `OutputStream`, without using any temporary files. Provide another solution, without a loop, using operations from the `Files` class, using a temporary file.

2. Write a program that reads a text file and produces a file with the same name but extension `.toc`, containing an alphabetized list of all words in the input file together with a list of line numbers in which each word occurs. Assume that the file's encoding is UTF-8.
3. Write a program that reads a file containing text and, assuming that most words are English, guesses whether the encoding is ASCII, ISO 8859-1, UTF-8, or UTF-16, and if the latter, which byte ordering is used.
4. Using a `Scanner` is convenient, but it is a bit slower than using a `BufferedReader`. Read in a long file a line at a time, counting the number of input lines, with (a) a `Scanner` and `hasNextLine/nextLine`, (b) a `BufferedReader` and `readLine`, (c) a `BufferedReader` and `lines`. Which is the fastest? The most convenient?
5. When an encoder of a `Charset` with partial Unicode coverage can't encode a character, it replaces it with a default—usually, but not always, the encoding of "?". Find all replacements of all available character sets that support encoding. Use the `newEncoder` method to get an encoder, and call its `replacement` method to get the replacement. For each unique result, report the canonical names of the charsets that use it.
6. The BMP file format for uncompressed image files is well documented and simple. Using random access, write a program that reflects each row of pixels in place, without writing a new file.
7. Look up the API documentation for the `MessageDigest` class and write a program that computes the SHA-512 digest of a file. Feed blocks of bytes to the `MessageDigest` object with the `update` method, then display the result of calling `digest`. Verify that your program produces the same result as the `sha512sum` utility.
8. Write a utility method for producing a ZIP file containing all files from a directory and its descendants.
9. Using the `URLConnection` class, read data from a password-protected web page with "basic" authentication. Concatenate the user name, a colon, and the password, and compute the Base64 encoding:

```
String input = username + ":" + password;
String encoding =
Base64.getEncoder().encodeToString(input.getBytes());
```

Set the HTTP header `Authorization` to the value "Basic " + encoding. Then read and print the page contents.

10. Using a regular expression, extract all decimal integers (including negative ones) from a string into an `ArrayList<Integer>` (a) using `find`, and (b) using `split`. Note that a + or - that is not followed by a digit is a delimiter.
11. Using regular expressions, extract the directory path names (as an array of strings), the file name, and the file extension from an absolute or relative path such as `/home/cay/myfile.txt`.
12. Come up with a realistic use case for using group references in `Matcher.replaceAll` and implement it.
13. Implement a method that can produce a clone of any serializable object by serializing it into a byte array and deserializing it.
14. Implement a serializable class `Point` with instance variables for `x` and `y`.
Write a program that serializes an array of `Point` objects to a file, and another that reads the file.
15. Continue the preceding exercise, but change the data representation of `Point` so that it stores the coordinates in an array. What happens when the new version tries to read a file generated by the old version? What happens when you fix up the `serialVersionUID`? Suppose your life depended upon making the new version compatible with the old. What could you do?
16. Which classes in the standard Java library implement `Externalizable`? Which of them use `writeReplace/readResolve`?
17. Unzip the API source and investigate how the `LocalDate` class is serialized. Why does the class define `writeExternal` and `readExternal` methods even though it doesn't implement `Externalizable`? (Hint: Look at the `Ser` class. Why does the class define a `readObject` method? How could it be invoked?)

Chapter 10 ■ Concurrent Programming

Java was one of the first mainstream programming languages with built-in support for concurrent programming. Early Java programmers were enthusiastic about how easy it was to load images in background threads or implement a web server that serves multiple requests concurrently. At the time, the focus was on keeping a processor busy while some tasks spend their time waiting for the network. Nowadays, most computers have multiple processors or cores, and programmers worry about keeping them all busy.

In this chapter, you will learn how to divide computations into concurrent tasks and how to execute them safely. My focus is on the needs of application programmers, not system programmers who write web servers or middleware.

For that reason, I arranged the information in this chapter so that I can, as much as possible, first show you the tools that you should be using in your work. I cover the low-level constructs later in the chapter. It is useful to know about these low-level details so that you get a feel for the costs of certain operations. But it is best to leave low-level thread programming to the experts. If you want to become one of them, I highly recommend the excellent book *Java Concurrency in Practice* by Brian Goetz et al. [Addison-Wesley, 2006].

The key points of this chapter are:

1. Your tasks can run on platform threads or lighter-weight virtual threads. Use virtual threads for tasks that mostly block.
2. A `Runnable` describes a task that can be executed asynchronously but does not return a result.
3. An `ExecutorService` schedules tasks instances for execution.
4. A `Callable` describes a task that can be executed asynchronously and yields a result.
5. You can submit one or more `Callable` instances to an `ExecutorService` and combine the results when they are available.
6. Make your tasks interruptible by propagating the `InterruptedException` or monitoring the interrupted status.
7. When multiple threads operate on shared data without synchronization, the result is unpredictable.

8. A lock ensures that only one thread at a time executes a critical section.
9. Prefer using parallel algorithms and threadsafe data structures over programming with locks.
10. A `ConcurrentHashMap` is a threadsafe hash table that allows atomic update of entries.
11. A blocking queue can safely transfer data from producers to consumers, and throttle overly zealous producers.
12. Parallel streams and array operations automatically and safely parallelize computations.
13. The `Process` class lets you execute a command in a separate process and interact with the input, output, and error streams.

10.1. Concurrent Tasks

When you design a concurrent program, you need to think about the tasks that can be run together. In the following sections, you will see how to execute tasks concurrently.

10.1.1. Running Tasks

In Java, the `Runnable` interface describes a task you want to run, perhaps concurrently with others.

```
public interface Runnable {  
    void run();  
}
```

Like all methods, the `run` method is executed in a *thread*. A thread is a mechanism for executing a sequence of instructions. The Java virtual machine supports two kinds of threads. *Platform threads* are provided by the operating system. Multiple platform threads run concurrently, by using separate processors or different time slices on the same processor. *Virtual threads* are run on platform threads. When a virtual is *blocked*, waiting for something to happen, another virtual thread can run on the underlying platform thread. Virtual threads consume far fewer resources than platform threads, but they are only suitable for tasks that are frequently blocked. You typically use platform threads for CPU-intensive operations, and virtual threads for operations that connect to external services, such as databases or web servers, and are blocked until the results arrive.

Platform threads use *preemptive scheduling*. Each running thread is given a slice of time to perform its task. When that slice of time is exhausted, the operating system *preempts* the thread and gives another thread an opportunity to work.

Virtual threads use *cooperative scheduling*. A virtual thread loses control only when it calls the `yield` method, or when it is blocked.

If you want to execute a `Runnable` in a separate thread, you could spawn a thread just for this `Runnable`, and you will see how to do that in [Section 10.8.1](#). For platform threads, it doesn't usually make sense to have a one-to-one relationship between tasks and threads. Platform threads are expensive to create, so it makes sense to reuse them. But with virtual threads, you simply want to use a thread per task. You do not want to think of these issues when you design tasks, and therefore, it is best to separate tasks and task scheduling.

In the Java concurrency library, an *executor service* schedules tasks, choosing the threads on which to run them.

```
Runnable task = () -> { ... };
ExecutorService executor = ...;
executor.execute(task);
```

The `Executors` class has factory methods for executor services with different scheduling policies.

The call

```
ExecutorService executor =
Executors.newFixedThreadPool(nthreads);
```

yields a pool with a fixed number of platform threads. When you submit a task, it is queued up until a thread becomes available. This is a good choice to use for computationally intensive tasks, or to limit the resource consumption of a service. You can derive the number of threads from the number of available processors, which you obtain as

```
int processors = Runtime.getRuntime().availableProcessors();
```

To schedule tasks on virtual threads, use:

```
ExecutorService executor =
Executors.newVirtualThreadPerTaskExecutor();
```

The `Executors` class has other factory methods for executors. For example, the `newCachedThreadPool` method yields an executor service optimized for short-lived tasks. Each task is executed on an idle platform thread if possible, but a new platform thread is allocated if all threads are busy. There is no bound on the number of concurrent threads. Threads in the pool that are idle for an extended time are terminated.

Now go ahead and run the concurrency demo program in the book's companion code. It runs two tasks concurrently.

```
public static void main(String[] args) {
    Runnable hellos = () -> {
        for (int i = 1; i <= 1000; i++)
            System.out.println("Hello " + i);
    };
    Runnable goodbyes = () -> {
        for (int i = 1; i <= 1000; i++)
            System.out.println("Goodbye " + i);
    };
    ExecutorService executor = Executors.newCachedThreadPool();
    executor.execute(hellos);
    executor.execute(goodbyes);
}
```

Run the program a few times to see how the outputs are interleaved.

```
Goodbye 1
...
Goodbye 871
Goodbye 872
Hello 806
Goodbye 873
Goodbye 874
Goodbye 875
Goodbye 876
```

Goodbye 877

Goodbye 878

Goodbye 879

Goodbye 880

Goodbye 881

Hello 807

Goodbye 882

...

Hello 1000



Note: You may note that the program waits a bit after the last printout. It terminates when the pooled threads have been idle for a while and the executor service terminates them.



Caution: If concurrent tasks try to read or update a shared value, the result may be unpredictable. We will discuss this issue in detail in [Section 10.2](#). For now, we will assume that tasks do not share mutable data.

10.1.2. Futures

The `Runnable` interface is used with tasks that don't yield values. If you have a task that computes a result, use the `Callable<V>` interface instead. Its `call` method, unlike the `run` method of the `Runnable` interface, returns a value of type `V`:

```
public interface Callable<V> {
    V call() throws Exception;
}
```

As a bonus, the `call` method can throw arbitrary exceptions which can be relayed to the code that obtains the result.

To execute a `Callable`, submit it to an executor service:

```
ExecutorService executor = ...;
Callable<V> task = ...;
Future<V> f = executor.submit(task);
```

When you submit the task, you get a *future*—an object that represents a computation whose result will be available at some future time. The Future interface has the following methods:

```
V get() throws InterruptedException, ExecutionException
V get(long timeout, TimeUnit unit)
    throws InterruptedException, ExecutionException,
TimeoutException
boolean cancel(boolean mayInterruptIfRunning)
boolean isCancelled()
boolean isDone()
```

The `get` method *blocks* until the result is available or until the timeout has been reached. That is, the thread containing the call does not progress until the method returns normally or throws an exception. If the `call` method yields a value, the `get` method returns that value. If the `call` method throws an exception, the `get` method throws an `ExecutionException` wrapping the thrown exception. If the timeout has been reached, the `get` method throws a `TimeoutException`.

The `cancel` method attempts to cancel the task. If the task isn't already running, it won't be scheduled. Otherwise, if `mayInterruptIfRunning` is `true`, the thread running the task is interrupted.



Note: A task that wants to be interruptible must periodically check for interruption requests. This is required for any tasks that you'd like to cancel when some other subtask has succeeded. See [Section 10.1.3](#) for more details on interruption.

A task may need to wait for the result of multiple subtasks. Instead of submitting each subtask separately, you can use the `invokeAll` method, passing a Collection of `Callable` instances.

For example, suppose you want to count how often a word occurs in a set of files. For each file, make a `Callable<Integer>` that returns the count for that file. Then submit them all to the executor. When all tasks have completed, you get a list of the futures (all of which are done), and you can total up the answers.

```
String word = ...;
Set<Path> paths = ...;
var tasks = new ArrayList<Callable<Long>>();
for (Path p : paths)
    tasks.add(() -> { return number of occurrences of word in p });
List<Future<Long>> futures = executor.invokeAll(tasks);
    // This call blocks until all tasks have completed
long total = 0;
for (Future<Long> f : futures) total += f.get();
```

There is also a variant of `invokeAll` with a timeout, which cancels all tasks that have not completed when the timeout is reached.



Note: If it bothers you that the calling task blocks until all subtasks are done, you can use an `ExecutorCompletionService`. It returns the futures in the order of completion.

```
var service = new ExecutorCompletionService(executor);
for (Callable<T> task : tasks) service.submit(task);
for (int i = 0; i < tasks.size(); i++) {
    Process service.take().get()
    Do something else
}
```

The `invokeAny` method is like `invokeAll`, but it returns as soon as any one of the submitted tasks has completed normally, without throwing an exception. It then returns the value of its `Future`. The other tasks are cancelled. This is useful for a search that can conclude as soon as a match has been found. This code snippet locates a file containing a given word:

```
String word = ...;
Set<Path> files = ...;
var tasks = new ArrayList<Callable<Path>>();
for (Path p : files)
    tasks.add(() -> { if (word occurs in p) return p; else throw
... });
Path found = executor.invokeAny(tasks);
```

As you can see, the `ExecutorService` does a lot of work for you. Not only does it map tasks to threads, but it also deals with task results, exceptions, and cancellation.



Note: The `invokeAll` and `invokeAny` methods are examples of *structured concurrency*, a style that makes it easy to reason about concurrent programs. With structured concurrency, you don't just start concurrent tasks, but you wait for their completion in the same scope.

Structured tasks scopes, a preview feature of Java 21, provide more robust constructs for structured concurrency.

10.1.3. Thread Interruption

Suppose that you run concurrent tasks, and you are always satisfied with the first result. Once a result has arrived, all other tasks should be canceled.

In Java, a thread is canceled by calling its `interrupt` method. (There is no precise definition of what interruption means, but in practice it is used to indicate a cancellation request.) For example, the `invokeAny` method from the preceding section will interrupt the remaining threads once a result is available.

In Java, task cancellation is *cooperative*. Cancelled tasks are not simply aborted. Instead, tasks are notified about the cancellation so that they have a chance to carry out any necessary cleanup. This section covers what a task implemetor needs to know about cancellation.

First suppose a thread is calling a blocking method while it is interrupted. Then the blocking call is terminated with an `InterruptedException`. This is a

checked exception, and you must catch it inside the `run` method of a `Runnable`. The usual reaction to the exception is to end the `run` method.

In this example, the task calls the static `sleep` method, which makes the current thread sleep for a given period, so that some other threads have a chance to do work.

```
Runnable task = () -> {
    try {
        while (more work to do) {
            Do more work
            Thread.sleep(millis);
        }
    }
    catch (InterruptedException ex) {
        // Do nothing
    }
    Clean up
};
```

If the thread executing this task is interrupted in the `sleep` method, the method terminates with an `InterruptedException`. If the thread was interrupted before calling the `sleep` method, then the `sleep` method immediately throws an `InterruptedException`. Either way, that exception is caught and the `run` method exits.

What if the task never calls a blocking operation? Then it needs to manually check for the *interrupted status* that is set when someone calls the `interrupt` method.

A `Runnable` can check for this status:

```
Runnable task = () -> {
    while (!Thread.currentThread().isInterrupted() && more work to do) {
        Do more work
    }
    Clean up
};
```



Note: There is also a static `Thread.interrupted` method which gets the interrupted status of the current thread, then clears it, and returns the old status.



Tip: The `InterruptedException` may seem pesky, but you should not just catch and hide it when you call a blocking method. If you can't do anything else, at least set the interrupted status:

```
try {
    Thread.sleep(millis);
} catch (InterruptedException ex) {
    Thread.currentThread().interrupt();
}
```

Or better, simply propagate the exception to a competent handler:

```
public void mySubTask() throws InterruptedException {
    ...
    Thread.sleep(millis);
    ...
}
```

10.2. Thread Safety

Many programmers initially think that concurrent programming is pretty easy. You just divide your work into tasks, and that's it. What could possibly go wrong?

In the following sections, I show you what can go wrong, and give a high-level overview of what you can do about it.

10.2.1. Visibility

Even operations as simple as writing and reading a variable can be incredibly complicated with modern processors. Consider this example:

```
private static boolean done = false;

public static void main(String[] args) {
    Runnable hellos = () -> {
        for (int i = 1; i <= 1000; i++)
            System.out.println("Hello " + i);
        done = true;
    };
    Runnable goodbye = () -> {
        int i = 1;
        while (!done) i++;
        System.out.println("Goodbye " + i);
    };
    Executor executor = Executors.newCachedThreadPool();
    executor.execute(hellos);
    executor.execute(goodbye);
}
```

The first task prints “Hello” a thousand times, and then sets `done` to `true`. The second task waits for `done` to become `true`, and then prints “Goodbye” once, incrementing a counter while it is waiting for that happy moment.

You’d expect the output to be something like

```
Hello 1
...
Hello 1000
Goodbye 501249
```

When I run this program on my laptop, the program prints up to “Hello 1000” and never terminates. The effect of

```
done = true;
```

may not be *visible* to the thread running the second task.

Why wouldn’t it be visible? Modern compilers, virtual machines, and processors perform many optimizations. These optimizations assume that the code is sequential unless explicitly told otherwise.

One optimization is caching of memory locations. We think of a memory location such as `done` as bits somewhere in the transistors of a RAM chip. But RAM chips are slow—many times slower than modern processors. Therefore, a processor tries to hold the data that it needs in registers or an onboard memory cache, and eventually writes changes back to memory. This caching is simply indispensable for processor performance. There are operations for synchronizing cached copies, but they have a significant performance cost and are only issued when requested.

Another optimization is instruction reordering. The compiler, the virtual machine, and the processor are allowed to change the order of instructions to speed up operations, provided it does not change the sequential semantics of the program.

For example, consider a computation

```
x = Something not involving y;  
y = Something not involving x;  
z = x + y;
```

The first two steps must occur before the third, but they can occur in either order. A processor can (and often will) run the first two steps concurrently, or swap the order if the inputs to the second step are more quickly available.

In our case, the loop

```
while (!done) i++;
```

can be reordered as

```
if (!done) while (true) i++;
```

since the loop body does not change the value of `done`.

By default, optimizations assume that there are no concurrent memory accesses. If there are, the virtual machine needs to know, so that it can then emit processor instructions that inhibit improper reorderings.

There are several ways of ensuring that an update to a variable is visible. Here is a summary:

1. The value of a `final` variable is visible after initialization.

2. The initial value of a `static` variable is visible after static initialization.
3. Changes to a `volatile` variable are visible.
4. Changes that happen before releasing a lock are visible to anyone acquiring the same lock (see [Section 10.7.1](#)).

In our case, the problem goes away if you declare the shared variable `done` with the `volatile` modifier:

```
private static volatile boolean done;
```

Then the compiler generates instructions that cause the virtual machine to issue processor commands for cache synchronization. As a result, any change to `done` in one task becomes visible to the other tasks.

The `volatile` modifier happens to suffice to solve this particular problem. But as you will see in the next section, declaring shared variables as `volatile` is not a general solution.



Tip: It is an excellent idea to declare any field that does not change after initialization as `final`. Then you never have to worry about its visibility.

10.2.2. Race Conditions

Suppose multiple concurrent tasks update a shared integer counter.

```
private static volatile int count = 0;  
...  
count++; // Task 1  
...  
count++; // Task 2  
...
```

The variable has been declared as `volatile`, so the updates are visible. But that is not enough.

The update `count++` actually means

```
register = count + 1;  
count = register;
```

When these computations are interleaved, the wrong value can be stored back into the `count` variable. In the parlance of concurrency, we say that the increment operation is not *atomic*. Consider this scenario:

```
int count = 0; // Initial value  
register1 = count + 1; // Thread 1 computes count + 1  
... // Thread 1 is preempted  
register2 = count + 1; // Thread 2 computes count + 1  
count = register2; // Thread 2 stores 1 in count  
... // Thread 1 is running again  
count = register1; // Thread 1 stores 1 in count
```

Now `count` is 1, not 2. This kind of error is called a *race condition* because it depends on which thread wins the “race” for updating the shared variable.

Does this problem really happen? It certainly does. Run the demo program of the companion code. It has 100 threads, each incrementing the counter 1,000 times and printing the result.

```
for (int i = 1; i <= 100; i++) {  
    int taskId = i;  
    Runnable task = () -> {  
        for (int k = 1; k <= 1000; k++)  
            count++;  
        System.out.println(taskId + ": " + count);  
    };  
    executor.execute(task);  
}
```

The output usually starts harmlessly enough as something like

```
1: 1000  
3: 2000  
2: 3000  
6: 4000
```

After a while, it looks a bit scary:

```
72: 58196  
68: 59196  
73: 61196  
71: 60196  
69: 62196
```

But that might just be because some threads were paused at inopportune moments. What matters is what happens with the task that finished last. Did it bring up the counter to 100,000?

I ran the program dozens of times on my multicore laptop, and it fails every time. Years ago, when personal computers had a single CPU, race conditions were more difficult to observe, and programmers did not notice such dramatic failures often. But it doesn't matter whether a wrong value is computed within seconds or hours.

This example looks at the simple case of a shared counter in a toy program. [Exercise 34](#) shows the same problem in a realistic example. But it's not just counters. Race conditions are a problem whenever shared variables are mutated. For example, when adding a value to the head of a queue, the insertion code might look like this:

```
var n = new Node();  
if (head == null) head = n;  
else tail.next = n;  
tail = n;  
tail.value = newValue;
```

Lots of things can go wrong if this sequence of instructions is paused at an unfortunate time and another task gets control, accessing the queue while it is in an inconsistent state.

Work through [Exercise 42](#) to get a feel for how a data structure can get corrupted by concurrent mutation.

We need to ensure that the entire sequence of operation is carried out together. Such an instruction sequence is called a *critical section*. You can use a *lock* to protect critical sections and make critical sequences of operation atomic. You will learn how to program with locks in [Section 10.7.1](#).

While it is straightforward to use locks for protecting critical sections, locks are not a general solution for solving all concurrency problems. They are difficult to use properly, and it is easy to make mistakes that severely degrade performance or even cause “deadlock.”

10.2.3. Strategies for Safe Concurrency

In languages such as C and C++, programmers need to manually allocate and deallocate memory. That sounds dangerous—and it is. Many programmers have spent countless miserable hours chasing memory allocation bugs. In Java, there is a garbage collector, and few Java programmers need to worry about memory management.

Unfortunately, there is no equivalent mechanism for shared data access in a concurrent program. The best you can do is to follow a set of guidelines to manage the inherent dangers.

A highly effective strategy is *confinement*. Just say no when it comes to sharing data among tasks. For example, when your tasks need to count something, give each of them a private counter instead of updating a shared counter. When the tasks are done, they can hand off their results to another task that combines them.

Another good strategy is *immutability*. It is safe to share immutable objects. For example, instead of adding results to a shared collection, a task can generate an immutable collection of results. Another task combines the results into another immutable data structure. The idea is simple, but there are a few things to watch out for—see [Section 10.2.4](#).

The third strategy is *locking*. By granting only one task at a time access to a data structure, one can keep it from being damaged. In [Section 10.3](#), you will see data structures provided by the Java concurrency library that are safe to use concurrently. [Section 10.7.1](#) shows you how locking works, and how experts build these data structures.

Locking is error-prone, and it can be expensive since it reduces opportunities for concurrent execution. For example, if you have lots of tasks contributing results to a shared hash table, and the table is locked for each update, then that is a real bottleneck. If most tasks have to wait their turn, they aren’t doing useful work. Sometimes it is possible to *partition* data so that different pieces can be accessed

concurrently. Several data structures in the Java concurrency library use partitioning, as do the parallel algorithms in the streams library. Don't try this at home! It is really hard to get it right. Instead, use the data structures and algorithms from the Java library.

10.2.4. Immutable Classes

A class is immutable when its instances, once constructed, cannot change. It sounds at first as if you can't do much with them, but that isn't true. The ubiquitous `String` class is immutable, as are the date classes in the `java.time` package. Each date instance is immutable, but you can obtain new dates, such as the one that comes a day after a given one.

Or consider a set for collecting results. You could use a mutable `HashSet` and update it like this:

```
results.addAll(newResults);
```

But that is clearly dangerous.

An immutable set always creates new sets. You would update the results somewhat like this:

```
results = results.union(newResults);
```

There is still mutation, but it is much easier to control what happens to one variable than to a hash set with many methods.

It is not difficult to implement immutable classes, but you should pay attention to these issues:

1. Don't change the object state after construction. Be sure to declare instance variables `final`. There is no reason not to, and you gain an important advantage: the virtual machine ensures that a `final` instance variable is visible after construction ([Section 10.2.1](#)).
2. Of course, none of the methods can be mutators. You should make them `final`, or better, declare the class `final`, so that mutators cannot be added in subclasses.
3. Don't leak state that can be mutated externally. None of your (non-private) methods can return a reference to any innards that could be used for

mutation, such as an internal array or collection. When one of your methods calls a method of another class, it must not pass any such references either, since the called method might otherwise use them for mutation. Instead, pass a copy.

4. Conversely, don't store any reference to a mutable object that the constructor receives. Instead, make a copy.
5. Don't let the `this` reference escape in a constructor. When you call another method, you know not to pass any internal references, but what about `this`? That's perfectly safe after construction, but if you reveal `this` in the constructor, someone could observe the object in an incomplete state. Also beware of constructors giving out inner class references that contain a hidden `this` reference. Naturally, these situations are quite rare.

10.3. Threadsafe Data Structures

If multiple threads concurrently modify a data structure, such as a queue or hash table, it is easy to damage the internals of the data structure. For example, one thread may begin to insert a new element. Suppose it is preempted in the middle of rerouting links, and another thread starts traversing the same location. The second thread may follow invalid links and create havoc, perhaps throwing exceptions or even getting trapped in an infinite loop.

As you will see in [Section 10.7.1](#), you can use locks to ensure that only one thread can access the data structure at a given point in time, blocking any others. But you can do better than that. You can use a threadsafe data structure that someone else has implemented. The collections in the `java.util.concurrent` package use locks correctly and have clever optimizations that minimize locking.

10.3.1. Concurrent Hash Maps

A `ConcurrentHashMap` is, first of all, a hash map whose operations are threadsafe. No matter how many threads operate on the map at the same time, the internals are not corrupted. Of course, some threads may be temporarily blocked, but the map can efficiently support a large number of concurrent readers and a certain number of concurrent writers.

But that is not enough. Suppose we want to use a map to count how often certain features are observed. As an example, suppose multiple threads encounter words,

and we want to count their frequencies. Obviously, the following code for updating a count is not threadsafe:

```
var map = new ConcurrentHashMap<String, Long>();
...
Long oldValue = map.get(word);
Long newValue = oldValue == null ? 1 : oldValue + 1;
map.put(word, newValue); // Error—might not replace oldValue
```

Another thread might be updating the exact same count at the same time.

To update a value safely, use the `compute` method. It is called with a key and a function to compute the new value. That function receives the key and the associated value, or `null` if there is none, and computes the new value. For example, here is how we can update a count:

```
map.compute(word, (k, v) -> v == null ? 1 : v + 1);
```

The `compute` method is *atomic*—no other thread can mutate the map entry while the computation is in progress.

There are also variants `computeIfPresent` and `computeIfAbsent` that only compute a new value when there is already an old one, or when there isn't yet one.

Another atomic operation is `putIfAbsent`. A counter might be initialized as

```
map.putIfAbsent(word, 0L);
```

You often need to do something special when a key is added for the first time. The `merge` method makes this particularly convenient. It has a parameter for the initial value that is used when the key is not yet present. Otherwise, the function that you supplied is called, combining the existing value and the initial value. (Unlike `compute`, the function does not process the key.)

```
map.merge(word, 1L, (existingValue, newValue) -> existingValue
+ newValue);
```

or simply,

```
map.merge(word, 1L, Long::sum);
```

Of course, the functions passed to `compute` and `merge` should complete quickly, and they should not attempt to mutate the map.



Note: There are methods that atomically remove or replace an entry if it is currently equal to an existing one. Before the `compute` method was available, people would write code like this for incrementing a count:

```
do {  
    oldValue = map.get(word);  
    newValue = oldValue + 1;  
} while (!map.replace(word, oldValue, newValue));
```



Note: There are several *bulk operations* for searching, transforming, or visiting a `ConcurrentHashMap`. They operate on a snapshot of the data and can safely execute even while other threads operate on the map. In the API documentation, look for the operations whose names start with `search`, `reduce`, and `forEach`. There are variants that operate on the keys, values, and entries. The `reduce` methods have specializations for `int`-, `long`-, and `double`-valued reduction functions.

10.3.2. Blocking Queues

One commonly used tool for coordinating work between tasks is a *blocking queue*. Producer tasks insert items into the queue, and consumer tasks retrieve them. The queue lets you safely hand over data from one task to another. Also, by limiting the queue size, you can throttle producers if the consumers cannot keep up with production.

When you try to add an element and the queue is currently full, or you try to remove an element when the queue is empty, the operation blocks. In this way, the queue balances the workload. If the producer tasks run slower than the consumer tasks, the consumers block while waiting for the results. If the producers run faster, the queue fills up until the consumers catch up.

[**Table 10.1**](#) shows the methods for blocking queues. The blocking queue methods fall into three categories that differ by the action they perform when the queue is full or empty. In addition to the blocking methods, there are methods that throw an exception when they don't succeed, and methods that return with a failure indicator instead of throwing an exception if they cannot carry out their tasks.

Table 10.1: Blocking Queue Operations

Method	Normal action	Error action
put	Adds an element to the tail.	Blocks if the queue is full.
take	Removes and returns the head element.	Blocks if the queue is empty.
add	Adds an element to the tail.	Throws an <code>IllegalStateException</code> if the queue is full.
remove	Removes and returns the head element.	Throws a <code>NoSuchElementException</code> if the queue is empty.
element	Returns the head element.	Throws a <code>NoSuchElementException</code> if the queue is empty.
offer	Adds an element and returns <code>true</code> .	Returns <code>false</code> if the queue is full.
poll	Removes and returns the head element.	Returns <code>null</code> if the queue is empty.
peek	Returns the head element.	Returns <code>null</code> if the queue is empty.



Note: The `poll` and `peek` methods return `null` to indicate failure. Therefore, it is illegal to insert null values into these queues.

There are also variants of the `offer` and `poll` methods with a timeout. For example, the call

```
boolean success = q.offer(x, 100, TimeUnit.MILLISECONDS);
```

tries for 100 milliseconds to insert an element to the tail of the queue. If it succeeds, it returns `true`; otherwise, it returns `false` when it times out. Similarly, the call

```
Object head = q.poll(100, TimeUnit.MILLISECONDS)
```

tries for 100 milliseconds to remove the head of the queue. If it succeeds, it returns the head; otherwise, it returns `null` when it times out.

The `java.util.concurrent` package supplies several variations of blocking queues. A `LinkedBlockingQueue` is based on a linked list, and an `ArrayBlockingQueue` uses a circular array.

[Exercise 22](#) shows how to use blocking queues for analyzing files in a directory. One thread walks the file tree and inserts files into a queue. Several threads remove the files and search them. In this application, it is likely that the producer quickly fills up the queue with files and blocks until the consumers can catch up.

A common challenge with such a design is stopping the consumers. A consumer cannot simply quit when the queue is empty. After all, the producer might not yet have started, or it may have fallen behind. If there is a single producer, it can add a “last item” indicator to the queue, similar to a dummy suitcase with a label “last bag” in a baggage claim belt.

10.3.3. Other Threadsafe Data Structures

Just like you can choose between hash maps and tree maps in the `java.util` package, there is a concurrent map that is based on comparing keys, called `ConcurrentSkipListMap`. Use it if you need to traverse the keys in sorted order, or if you need one of the added methods in the `NavigableMap` interface (see

[Chapter 7](#)). Similarly, there is a `ConcurrentSkipListSet`.

The `CopyOnWriteArrayList` and `CopyOnWriteArraySet` are threadsafe collections in which all mutators make a copy of the underlying array. This arrangement is useful if the threads that iterate over the collection greatly outnumber the threads that mutate it. When you construct an iterator, it contains a reference to the current array. If the array is later mutated, the iterator still has the old array, but the collection's array is replaced. As a consequence, the older iterator has a consistent (but potentially outdated) view that it can access without any synchronization expense.

Suppose you want a large, threadsafe set instead of a map. There is no `ConcurrentHashSet` class, and you know better than trying to create your own. Of course, you can use a `ConcurrentHashMap` with bogus values, but that gives you a map, not a set, and you can't apply operations of the `Set` interface.

The static `newKeySet` method yields a `Set<K>` that is actually a wrapper around a `ConcurrentHashMap<K, Boolean>`. (All map values are `Boolean.TRUE`, but you don't actually care since you just use it as a set.)

```
Set<String> words = ConcurrentHashMap.newKeySet();
```

If you have an existing map, the `keySet` method yields the set of keys. That set is mutable. If you remove the set's elements, the keys (and their values) are removed from the map. But it doesn't make sense to add elements to the key set, because there would be no corresponding values to add. You can use a second `keySet` method, with a default value used when adding elements to the set:

```
Set<String> words = map.keySet(1L);
words.add("Java");
```

If "Java" wasn't already present in `words`, it now has a value of one.



Note: Concurrent collections yield *weakly consistent* iterators. That means that the iterators present elements appearing at onset of iteration, but may or may not reflect some or all of the modifications that were made after they were constructed. However, such an iterator will not throw a `ConcurrentModificationException`.

In contrast, an iterator of a collection in the `java.util` package throws a `ConcurrentModificationException` when the collection has been modified after construction of the iterator.

10.4. Parallel Algorithms

Before starting to parallelize your computations, you should check if the Java library has done this for you. The stream library or the `Arrays` class may already do what you need.

10.4.1. Parallel Streams

The stream library can automatically parallelize operations on large data sets. For example, if `coll` is a large collection of strings, and you want to find how many of them start with the letter A, call

```
long result = coll.parallelStream().filter(s ->
    s.startsWith("A")).count();
```

The `parallelStream` method yields a parallel stream. The stream is broken up into segments. The filtering and counting is done on each segment, and the results are combined. You don't need to worry about the details.



Caution: When you use parallel streams with lambdas (for example, as the argument to `filter` and `map` in the preceding examples), be sure to stay away from unsafe mutation of shared objects.

For parallel streams to work well, a number of conditions need to be fulfilled:

- There needs to be enough data. There is a substantial overhead for parallel streams that is only repaid for large data sets.
- The data should be in memory. It would be inefficient to have to wait for the data to arrive.
- The stream should be efficiently splittable into subregions. A stream backed by an array or a balanced binary tree works well, but a linked list or the result of `Stream.iterate` does not.

- The stream operations should do a substantial amount of work. If the total work load is not large, it does not make sense to pay for the cost of setting up the concurrent computation.
- The stream operations should not block.

In other words, don't turn all your streams into parallel streams. Use parallel streams only when you do a substantial amount of sustained computational work on data that is already in memory.

10.4.2. Parallel Array Operations

The `Arrays` class has a number of parallelized operations. Just as with the parallel stream operations of the preceding sections, the operations break the array into sections, work on them concurrently, and combine the results.

The static `Arrays.parallelSetAll` method fills an array with values computed by a function. The function receives the element index and computes the value at that location.

```
Arrays.parallelSetAll(values, i -> i % 10);
// Fills values with 0 1 2 3 4 5 6 7 8 9 0 1 2...
```

Clearly, this operation benefits from being parallelized. There are versions for all primitive type arrays and for object arrays.

The `parallelSort` method can sort an array of primitive values or objects. For example,

```
Arrays.parallelSort(words,
Comparator.comparing(String::length));
```

With all methods, you can supply the bounds of a range, such as

```
Arrays.parallelSort(values, values.length / 2, values.length);
// Sort the upper half
```



Note: At first glance, it seems a bit odd that these methods have `parallel` in their names—the user shouldn't care how the setting or sorting happens. However, the API designers wanted to make it clear that

the operations are parallelized. That way, users are on notice to avoid generator or comparison functions with side effects.

Finally, there is a `parallelPrefix` that is rather specialized—[Exercise 8](#) gives a simple example.

For other parallel operations on arrays, turn the arrays into parallel streams. For example, to compute the sum of a long array of integers, call

```
long sum = IntStream.of(values).parallel().sum();
```

10.5. Asynchronous Computations

A common approach to concurrent computation is to break up a task into subtasks, launch concurrent work on the subtasks, and block until all pieces have completed. Blocking works well with virtual threads. But with platform threads, blocking can be problematic since platform threads are a limited resource. A non-blocking or *asynchronous*, programming style can provide higher throughput. In the following sections, you will see the Java platform support for asynchronous computations.

10.5.1. Completable Futures

When you have a `Future` object, you need to call `get` to obtain the value, blocking until the value is available. The `CompletableFuture` class implements the `Future` interface, and it provides a second mechanism for obtaining the result. You register a *callback* that will be invoked (in some thread) with the result once it is available.

```
CompletableFuture<String> f = ...;
f.thenAccept((String s) -> Process the result s);
```

In this way, you can process the result, without blocking, as soon as it is available.

There are a few API methods that return `CompletableFuture` objects. For example, the `HttpClient` class can fetch a web page asynchronously:

```
HttpClient client = HttpClient.newHttpClient();
HttpRequest request =
HttpRequest.newBuilder(URI.create(urlString)).GET().build();
CompletableFuture<HttpResponse<String>> f = client.sendAsync(
    request, HttpResponse.BodyHandlers.ofString());
```

To run a task asynchronously and obtain a `CompletableFuture`, you don't submit it directly to an executor service. Instead, you call the static method `CompletableFuture.supplyAsync`:

```
Supplier<V> task = ...;
CompletableFuture<V> f = CompletableFuture.supplyAsync(task,
executor);
```

If you omit the executor, the task is run on a default executor (namely the executor returned by `ForkJoinPool.commonPool()`).

Note that the first parameter of this method is a `Supplier<V>`, not a `Callable<V>`. Both interfaces describe functions with no parameters and a return value of type `V`, but a `Supplier` function cannot throw a checked exception.

A `CompletableFuture` can complete in two ways: either with a result, or with an uncaught exception. In order to handle both cases, use the `whenComplete` method. The supplied function is called with the result (or `null` if none) and the exception (or `null` if none).

```
f.whenComplete((s, t) -> {
    if (t == null) { Process the result s; }
    else { Process the Throwable t; }
});
```

The `CompletableFuture` is called completable because you can manually set a completion value. (In other concurrency libraries, such an object is called a *promise*). Of course, when you create a `CompletableFuture` with `supplyAsync`, the completion value is implicitly set when the task has finished. But setting the result explicitly gives you additional flexibility. For example, two tasks can work simultaneously on computing an answer:

```
var f = new CompletableFuture<Integer>();
executor.execute(() -> {
    int n = workHard(arg);
    f.complete(n);
});
executor.execute(() -> {
    int n = workSmart(arg);
    f.complete(n);
});
```

To instead complete a future with an exception, call

```
Throwable t = ...;
f.completeExceptionally(t);
```



Note: It is safe to call `complete` or `completeExceptionally` on the same future in multiple threads. If the future is already completed, these calls have no effect.

The `isDone` method tells you whether a `Future` object has been completed (normally or with an exception). In the preceding example, the `workHard` and `workSmart` methods can use that information to stop working when the result has been determined by the other method.



Caution: Unlike a plain `Future`, the computation of a `CompletableFuture` is not interrupted when you invoke its `cancel` method. Canceling simply sets the `Future` object to be completed exceptionally, with a `CancellationException`. In general, this makes sense since a `CompletableFuture` may not have a single thread that is responsible for its completion. However, this restriction also applies to `CompletableFuture` instances returned by methods such as `supplyAsync`, which could in principle be interrupted. See [Exercise 58](#) for a workaround.

10.5.2. Composing Completable Futures

Nonblocking calls are implemented through callbacks. The programmer registers a callback for the action that should occur after a task completes. Of course, if the next action is also asynchronous, the next action after that is in a different callback. Even though the programmer thinks in terms of “first do step 1, then step 2, then step 3,” the program logic can become dispersed in “callback hell.” It gets even worse when you have to add error handling. Suppose step 2 is “the user logs in.” You may need to repeat that step since the user can mistype the credentials. Trying to implement such a control flow in a set of callbacks—or to understand it once it has been implemented—can be quite challenging.

The `CompletableFuture` class solves this problem by providing a mechanism for *composing* asynchronous tasks into a processing pipeline.

For example, suppose we want to extract all links from a web page in order to build a web crawler. Let’s say we have a method

```
public void CompletableFuture<String> readPage(URI url)
```

that yields the text of a web page when it becomes available. If the method

```
public static List<URI> getLinks(String page)
```

yields the URIs in an HTML page, you can schedule it to be called when the page is available:

```
CompletableFuture<String> contents = readPage(url);
CompletableFuture<List<URI>> links =
contents.thenApply(Parser::getLinks);
```

The `thenApply` method doesn’t block either. It returns another future. When the first future has completed, its result is fed to the `getLinks` method, and the return value of that method becomes the final result.

With completable futures, you just specify what you want to have done and in which order. It won’t all happen right away, of course, but what is important is that all the code is in one place.

Conceptually, `CompletableFuture` is a simple API, but there are many variants of methods for composing completable futures. Let us first look at those that deal with a single future (see [Table 10.2](#)). (For each method shown, there are also two

`Async` variants that I don't show. One of them uses a shared `ForkJoinPool`, and the other has an `Executor` parameter.) In the table, I use a shorthand notation for the ponderous functional interfaces, writing `T -> U` instead of `Function<? super T, U>`. These aren't actual Java types, of course.

You have already seen the `thenApply` method. Suppose `f` is a function that receives values of type `T` and returns values of type `U`. The calls

```
CompletableFuture<U> future.thenApply(f);  
CompletableFuture<U> future.thenApplyAsync(f);
```

return a future that applies the function `f` to the result of `future` when it is available. The second call runs `f` in yet another thread.

The `thenCompose` method, instead of taking a function mapping the type `T` to the type `U`, receives a function mapping `T` to `CompletableFuture<U>`. That sounds rather abstract, but it can be quite natural. Consider the action of reading a web page from a given URL. Instead of supplying a method

```
public String blockingReadPage(URI url)
```

it is more elegant to have that method return a future:

```
public CompletableFuture<String> readPage(URI url)
```

Now, suppose we have another method that gets the URL from user input, perhaps from a dialog that won't reveal the answer until the user has clicked the OK button. That, too, is an event in the future:

```
public CompletableFuture<URI> getURLInput(String prompt)
```

Here we have two functions `T -> CompletableFuture<U>` and `U -> CompletableFuture<V>`. Clearly, they compose to a function `T -> CompletableFuture<V>` if the second function is called when the first one has completed. That is exactly what `thenCompose` does.

In the preceding section, you saw the `whenComplete` method for handling exceptions. There is also a `handle` method that requires a function processing the result or exception and computing a new result. In many cases, it is simpler to call the `exceptionally` method instead:

```
CompletableFuture<String> contents = readPage(url)
    .exceptionally(t -> { Log t; return emptyPage; });
```

The supplied handler is only called if an exception occurred, and it produces a result to be used in the processing pipeline. If no exception occurred, the original result is used.

The methods in [Table 10.2](#) with `void` result are normally used at the end of a processing pipeline.

Table 10.2: Adding an Action to a `CompletableFuture<T>` Object

Method	Parameter	Description
<code>thenApply</code>	<code>T -> U</code>	Apply a function to the result.
<code>thenAccept</code>	<code>T -> void</code>	Like <code>thenApply</code> , but with <code>void</code> result.
<code>thenCompose</code>	<code>T -> CompletableFuture<U></code>	Invoke the function on the result and execute the returned future.
<code>thenRun</code>	<code>Runnable</code>	Execute the <code>Runnable</code> with <code>void</code> result.
<code>handle</code>	<code>(T, Throwable) -> U</code>	Process the result or error and yield a new result.
<code>whenComplete</code>	<code>(T, Throwable) -> void</code>	Like <code>handle</code> , but with <code>void</code> result.
<code>exceptionally</code>	<code>Throwable -> T</code>	Turn the error into a default result.

Method	Parameter	Description
exceptionallyCompose	Throwable -> CompletableFuture<U>	Invoke the function on the exception and execute the returned future.
completeOnTimeout	T, long, TimeUnit	Yield the given value as the result in case of timeout.
orTimeout	long, TimeUnit	Yield a TimeoutException in case of timeout.

Now let us turn to methods that combine multiple futures (see [Table 10.3](#)).

The first three methods run a `CompletableFuture<T>` and a `CompletableFuture<U>` action concurrently and combine the results.

The next three methods run two `CompletableFuture<T>` actions concurrently. As soon as one of them finishes, its result is passed on, and the other result is ignored.

Finally, the static `allOf` and `anyOf` methods take a variable number of completable futures and yield a `CompletableFuture<Void>` that completes when all of them, or any one of them, completes. The `allOf` method does not yield a result. The `anyOf` method does *not* terminate the remaining tasks. [Exercise 60](#) and [Exercise 62](#) show useful improvements of these two methods.

Table 10.3: Combining Multiple Composition Objects

Method	Parameters	Description
thenCombine	CompletableFuture<U>, (T, U) -> V	Execute both and combine the results with the given function.

Method	Parameters	Description
thenAcceptBoth	CompletableFuture<U>, (T, U) -> void	Like thenCombine, but with void result.
runAfterBoth	CompletableFuture<?>, Runnable	Execute the runnable after both complete.
applyToEither	CompletableFuture<T>, T -> V	When a result is available from one or the other, pass it to the given function.
acceptEither	CompletableFuture<T>, T -> void	Like applyToEither, but with void result.
runAfterEither	CompletableFuture<?>, Runnable	Execute the runnable after one or the other completes.
static allOf	CompletableFuture<?> ...	Complete with void result after all given futures complete.
static anyOf	CompletableFuture<?> ...	Complete after any of the given futures completes and yield its result.



Note: Technically speaking, the methods in this section have parameters of type `CompletionStage`, not `CompletableFuture`. The `CompletionStage` interface describes how to compose asynchronous computations, whereas the `Future` interface focuses on the result of a

computation. A `CompletableFuture` is both a `CompletionStage` and a `Future`.

10.5.3. Long-Running Tasks in User-Interface Callbacks

One of the reasons to use threads is to make your programs more responsive. This is particularly important in an application with a user interface. When your program needs to do something time-consuming, you cannot do the work in the user-interface thread, or the user interface will freeze. Instead, fire up another worker thread.

For example, if you want to read a web page when the user clicks a button, don't do this:

```
var read = new JButton("Read");
read.addActionListener(event -> {
    // Bad—long-running action is executed on UI thread
    var in = new Scanner(url.openStream());
    while (in.hasNextLine()) {
        String line = in.nextLine();
        ...
    }
});
```

Instead, do the work in a separate thread.

```
read.addActionListener(event -> {
    // Good—long-running action in separate thread
    Runnable task = () -> {
        var in = new Scanner(url.openStream());
        while (in.hasNextLine()) {
            String line = in.nextLine();
            ...
        }
    }
    executor.execute(task);
});
```

However, you cannot directly update the user interface from the thread that executes the long-running task. User interfaces such as Swing, JavaFX, or Android are not threadsafe. You cannot manipulate user-interface elements from multiple threads, or they risk becoming corrupted. In fact, JavaFX and Android check for this, and throw an exception if you try to access the user interface from a thread other than the UI thread.

Therefore, you need to schedule any UI updates to happen on the UI thread. Each user-interface library provides some mechanism to schedule a `Runnable` for execution on the UI thread. Here is how to do it in Swing:

```
EventQueue.invokeLater(() -> message.appendText(line + "\n"));
```



Note: It is tedious to implement lengthy operations while giving users feedback on the progress, so user-interface libraries usually provide some kind of helper class for managing the details, such as `SwingWorker` in Swing and `AsyncTask` in Android. You specify actions for the long-running task (which is run on a separate thread), as well as progress updates and the final disposition (which are run on the UI thread).

10.6. Atomic Counters and Accumulators

If multiple threads update a shared counter, you need to make sure that this is done in a threadsafe way. There are a number of classes in the `java.util.concurrent.atomic` package that use safe and efficient machine-level instructions to guarantee atomicity of operations on integers, `long` and `boolean` values, object references, and arrays thereof. Using these classes correctly requires considerable expertise. However, atomic counters and accumulators are convenient for application-level programming.

For example, you can safely generate a sequence of numbers like this:

```
public static AtomicLong nextNumber = new AtomicLong();
// In some thread...
long id = nextNumber.incrementAndGet();
```

The `incrementAndGet` method atomically increments the `AtomicLong` and returns the post-increment value. That is, the operations of getting the value, adding 1, setting it, and producing the new value cannot be interrupted. It is guaranteed that the correct value is computed and returned, even if multiple threads access the same instance concurrently.

There are methods for atomically setting, adding, and subtracting values, but suppose you want to make a more complex update. One way is to use the `updateAndGet` method. For example, suppose you want to keep track of the largest value that is observed by different threads. The following won't work:

```
public static AtomicLong largest = new AtomicLong();
// In some thread...
largest.set(Math.max(largest.get(), observed)); // Error—race
condition!
```

This update is not atomic. Instead, call `updateAndGet` with a lambda expression for updating the variable. In our example, we can call

```
largest.updateAndGet(x -> Math.max(x, observed));
```

or

```
largest.accumulateAndGet(observed, Math::max);
```

The `accumulateAndGet` method takes a binary operator that is used to combine the atomic value and the supplied argument.

There are also methods `getAndUpdate` and `getAndAccumulate` that return the old value.



Note: These methods are also provided for the classes:

<code>AtomicInteger</code>	<code>AtomicLongFieldUpdater</code>
<code>AtomicIntegerArray</code>	<code>AtomicReference</code>
<code>AtomicIntegerFieldUpdater</code>	<code>AtomicReferenceArray</code>
<code>AtomicLongArray</code>	<code>AtomicReferenceFieldUpdater</code>

When you have a very large number of threads accessing the same atomic values, performance suffers because updates are carried out *optimistically*. That is, the operation computes a new value from a given old value, then does the replacement provided the old value is still the current one, or retries if it is not. Under heavy contention, updates require too many retries.

The classes `LongAdder` and `LongAccumulator` solve this problem for certain common updates. A `LongAdder` is composed of multiple variables whose collective sum is the current value. Multiple threads can update different summands, and new summands are automatically provided when the number of threads increases. This is efficient in the common situation where the value of the sum is not needed until after all work has been done. The performance improvement can be substantial—see [Exercise 18](#).

If you anticipate high contention, you should simply use a `LongAdder` instead of an `AtomicLong`. The method names are slightly different. Call `increment` to increment a counter or `add` to add a quantity, and `sum` to retrieve the total.

```
final LongAdder count = new LongAdder();
for (...) {
    executor.execute(() -> {
        while (...) {
            ...
            if (...) count.increment();
        }
    });
...
long total = count.sum();
```



Note: Of course, the `increment` method does *not* return the old value. Doing that would undo the efficiency gain of splitting the sum into multiple summands.

The `LongAccumulator` generalizes this idea to an arbitrary accumulation operation. In the constructor, you provide the operation as well as its neutral element. To incorporate new values, call `accumulate`. Call `get` to obtain the current value.

```

var accumulator = new LongAccumulator(Long::sum, 0);
// In some tasks...
accumulator.accumulate(value);
// When all work is done
long sum = accumulator.get();

```

Internally, the accumulator has variables a_1, a_2, \dots, a_n . Each variable is initialized with the neutral element (θ in our example).

When `accumulate` is called with value v , then one of them is atomically updated as $a_i = a_i \text{ op } v$, where op is the accumulation operation written in infix form. In our example, a call to `accumulate` computes $a_i = a_i + v$ for some i .

The result of `get` is $a_1 \text{ op } a_2 \text{ op } \dots \text{ op } a_n$. In our example, that is the sum of the accumulators, $a_1 + a_2 + \dots + a_n$.

If you choose a different operation, you can compute maximum or minimum (see [Exercise 20](#)). In general, the operation must be associative and commutative. That means that the final result must be independent of the order in which the intermediate values were combined.

There are also `DoubleAdder` and `DoubleAccumulator` that work in the same way, except with double values.



Tip: If you use a hash map of `LongAdder`, you can use the following idiom to increment the adder for a key:

```

ConcurrentHashMap<String, LongAdder> counts = ...;
counts.computeIfAbsent(key, k -> new
    LongAdder()).increment();

```

When the count for `key` is incremented the first time, a new adder is set.

10.7. Locks and Conditions

Now you have seen several tools that application programmers can safely use for structuring concurrent applications. You may be curious how one would build a

threadsafe counter or blocking queue. The following sections show you how it is done, so that you gain some understanding of the costs and complexities.

10.7.1. Locks

To avoid the corruption of shared variables, one needs to ensure that only one thread at a time can compute and set the new values. Code that must be executed in its entirety, without interruption, is called a *critical section*. One can use a *lock* to implement a critical section:

```
public class Counter {  
    private final Lock countLock = new ReentrantLock();  
        // Shared among multiple threads  
    private int count = 0; // Shared among multiple threads  
  
    public int increment() {  
        countLock.lock();  
        try {  
            count++; // Critical section  
            return count;  
        } finally {  
            countLock.unlock(); // Make sure the lock is unlocked  
        }  
    }  
}
```



Note: In this section, I use the `ReentrantLock` class to explain how locking works. As you will see in the next section, there is no requirement to use explicit locks since there are “implicit” locks that are used by the `synchronized` keyword. But it is easier to understand what goes on under the hood by looking at explicit locks.

The first thread to execute the `lock` method locks the `countLock` object and then proceeds into the critical section. If another thread tries to call `lock` on the same object, it is blocked until the first thread executes the call to `unlock`. In this way, it is guaranteed that only one thread at a time can execute the critical section.

Note that, by placing the `unlock` method into a `finally` clause, the lock is released if any exception happens in the critical section. Otherwise, the lock would be permanently locked, and no other thread would be able to proceed past it. This would clearly be very bad. Of course, in this case, the critical section can't throw an exception since it only executes an integer increment. But it is a common idiom to use the `try/finally` statement anyway, in case more code gets added later.

At first glance, it seems simple enough to use locks for protecting critical sections. However, the devil is in the details. Experience has shown that many programmers have difficulty writing correct code with locks. They might use the wrong locks, or create situations that *deadlock* when no thread can make progress because all of them wait for a lock.

For that reason, application programmers should use locks as a matter of last resort. First try to avoid sharing, by using immutable data or handing off mutable data from one thread to another. If you must share, use prebuilt threadsafe structures such as a `ConcurrentHashMap` or a `LongAdder`. Still, it is useful to know about locks so you can understand how such data structures can be implemented.

10.7.2. The synchronized Keyword

In the preceding section, I showed you how to use a `ReentrantLock` to implement a critical section. You don't have to use an explicit lock because in Java, *every object* has an *intrinsic lock*. To understand intrinsic locks, however, it helps to have seen explicit locks first.

The `synchronized` keyword is used to lock the intrinsic lock. It can occur in two forms. You can lock a block:

```
synchronized (obj) {  
    Critical section  
}
```

This essentially means

```
obj.intrinsicLock.lock();
try {
    Critical section
} finally {
    obj.intrinsicLock.unlock();
}
```

An object does not actually have a field that is an intrinsic lock. The code is just meant to illustrate what goes on when you use the `synchronized` keyword.

You can also declare a method as `synchronized`. Then its body is locked on the receiver parameter `this`. That is,

```
public synchronized void method() {
    Body
}
```

is the equivalent of

```
public void method() {
    this.intrinsicLock.lock();
    try {
        Body
    } finally {
        this.intrinsicLock.unlock();
    }
}
```

For example, a counter can simply be declared as

```
public class Counter {
    private int count = 0;
    public synchronized int increment() {
        count++;
        return count;
    }
}
```

By using the intrinsic lock of the `Counter` instance, there is no need to come up with an explicit lock.

As you can see, using the `synchronized` keyword yields code that is quite concise. Of course, to understand this code, you have to know that each object has an intrinsic lock.



Note: There is more to locks than atomicity. Locks also guarantee visibility. For example, consider the `done` variable that gave us so much grief in [Section 10.2.1](#). If you use a lock for both writing and reading the variable, then you are assured that the caller of `get` sees any update to the variable through a call by `set`.

```
public class Flag {  
    private boolean done;  
    public synchronized void set() { done = true; }  
    public synchronized boolean get() { return done; }  
}
```

Note that, unlike the `ReentrantLock` in the preceding section, the intrinsic lock is *public*. Anyone can acquire it. This can sometimes be useful. For example, Java 1.0 has a `Hashtable` class with synchronized methods for mutating the table. To safely iterate over such a table, you can acquire the lock like this:

```
synchronized (table) {  
    for (K key : table.keySet()) ...  
}
```

Here, `table` denotes both the hash table and the lock that its methods use. This “client side locking” is a common source of misunderstandings (see [Exercise 44](#)), and it can lead to deadlocks.

Synchronized methods were inspired by the *monitor* concept that was pioneered by Per Brinch Hansen and Tony Hoare in the 1970s. A monitor is essentially a class in which all instance variables are private and all methods are protected by a private lock. However, Java classes are not proper monitors since it is possible to

have public instance variables and to mix synchronized and unsynchronized methods.



Caution: In Java 21, virtual threads are *pinned* when acquiring an intrinsic lock. They cannot be unmounted from their carrier thread until the lock is relinquished. Pinning is not a problem if the critical section is immediately exited. But if the virtual thread blocks while pinned, this can seriously degrade throughput and even cause starvation of the carrier thread pool.

One remedy is to use explicit locks, such as `ReentrantLock`, instead of synchronized methods or blocks. If the pinning happens in framework or driver code, you should check if there are newer versions that have been adapted for virtual threads.

This limitation of virtual threads is expected to be removed in a future version of Java.

10.7.3. Waiting on Conditions

Consider a simple `Queue` class with methods for adding and removing objects. Synchronizing the methods ensures that these operations are atomic.

```
public class Queue {  
    class Node { Object value; Node next; };  
    private Node head;  
    private Node tail;  
  
    public synchronized void add(Object newValue) {  
        var n = new Node();  
        if (head == null) head = n;  
        else tail.next = n;  
        tail = n;  
        tail.value = newValue;  
    }  
  
    public synchronized Object remove() {
```

```

        if (head == null) return null;
        Node n = head;
        head = n.next;
        return n.value;
    }
}

```

Now suppose we want to turn the `remove` method into a method `take` that blocks if the queue is empty.

The check for emptiness must come inside the synchronized method because otherwise the inquiry would be meaningless—another thread might have emptied the queue in the meantime.

```

public synchronized Object take() {
    if (head == null) ... // Now what?
    Node n = head;
    head = n.next;
    return n.value;
}

```

But what should happen if the queue is empty? No other thread can add elements while the current thread holds the lock. This is where the `wait` method comes in.

If the `take` method finds that it cannot proceed, it calls the `wait` method:

```

public synchronized Object take() throws InterruptedException {
    while (head == null) wait();
    ...
}

```

The current thread is now deactivated and gives up the lock. This lets in another thread that can, we hope, add elements to the queue. This is called *waiting on a condition*.

Note that the `wait` method is a method of the `Object` class. It relates to the lock that is associated with the object.

There is an essential difference between a thread that is blocking to acquire a lock and a thread that has called `wait`. Once a thread calls the `wait` method, it enters a

`wait set` for the object. The thread is not made runnable when the lock is available. Instead, it stays deactivated until another thread has called the `notifyAll` method on the same object.

When another thread has added an element, it should call that method:

```
public synchronized void add(Object newValue) {  
    ...  
    notifyAll();  
}
```

The call to `notifyAll` reactivates all threads in the wait set. When the threads are removed from the wait set, they are again runnable and the scheduler will eventually activate them again. At that time, they will attempt to reacquire the lock. As one of them succeeds, it continues where it left off, returning from the call to `wait`.

At this time, the thread should test the condition again. There is no guarantee that the condition is now fulfilled—the `notifyAll` method merely signals to the waiting threads that it *may be* fulfilled at this time and that it is worth checking for the condition again. For that reason, the test is in a loop

```
while (head == null) wait();
```

A thread can only call `wait`, `notifyAll`, or `notify` on an object if it holds the lock on that object.



Caution: Another method, `notify`, unblocks only a single thread from the wait set. That is more efficient than unblocking all threads, but there is a danger. If the chosen thread finds that it still cannot proceed, it becomes blocked again. If no other thread calls `notify` again, the program deadlocks.



Note: When implementing data structures with blocking methods, the `wait`, `notify`, and `notifyAll` methods are appropriate. But they are not easy to use properly. Application programmers should never have a need

to use these methods. Instead, use prebuilt data structures such as `LinkedBlockingQueue` or `ConcurrentHashMap`.

10.8. Threads

As we are nearing the end of this chapter, the time has finally come to talk more about threads, the primitives that actually execute tasks. Normally, you are better off using executors that manage threads for you, but the following sections give you some background information about working directly with threads.

10.8.1. Running a Thread

Here is the simplest way for running a platform thread in Java:

```
Runnable task = () -> { ... };
var thread = new Thread(task);
thread.start();
```

Starting a virtual thread is even simpler:

```
Thread.startVirtualThread(task);
```

If you want to wait for a thread to finish, call the `join` method:

```
thread.join(); // Waits indefinitely
thread.join(timeoutMillis);
```

The call blocks until the thread ends or the timeout has been reached. You can call the `isAlive` method to determine whether the `join` method returned due to timeout.

A thread ends when its `run` method returns, either normally or because an exception was thrown. In the latter case, the *uncaught exception handler* of the thread is invoked. When the thread is created, that handler is set to the uncaught exception handler of the thread group, which is ultimately the global handler (see

[Chapter 5](#)). You can change the handler of a thread by calling the `setUncaughtExceptionHandler` method.



Note: The initial release of Java defined a `stop` method that immediately terminates a thread, and a `suspend` method that blocks a thread until another thread calls `resume`. Both methods have since been deprecated for removal.

The `stop` method is inherently unsafe. Suppose a thread is stopped in the middle of a critical section—for example, inserting an element into a queue. Then the queue is left in a partially updated state. However, the lock protecting the critical section is unlocked, and other threads can use the corrupted data structure. You should interrupt a thread when you want it to stop. The interrupted thread can then terminate when it is safe to do so.

The `suspend` method has a different problem. If a thread is suspended while it holds a lock, any other thread trying to acquire that lock blocks. If the thread that is going to call `resume` is among them, the program deadlocks.

10.8.2. Thread-Local Variables

Sometimes you want to make a task-specific object available to all methods that collaborate on the task, without having to pass the object as a parameter of each method call. You can use a *thread-local variable* for this purpose. This is not an actual variable, but an object whose `get` and `set` methods access a value that depends on the current thread.

Suppose, for example, that you want to share a database connection. Declare a variable

```
public static final ThreadLocal<Connection> CONNECTION = new  
ThreadLocal<>();
```

When the task starts, initialize the connection for this thread:

```
CONNECTION.set(connect(uri, username, password));
```

The task calls some methods, all within the same thread, and eventually one of them needs the connection:

```
Connection connection = CONNECTION.get();
var result = connection.executeQuery(query);
```

Note that the same call may happen on multiple threads. Each of them gets its own connection object.



Caution: When executing tasks with a thread pool, you don't want to make thread-local data available to the other tasks that are scheduled on the same thread. You also want to make sure they get garbage collected when your task is done. It is essential that you call

```
CONNECTION.remove();
```

upon task completion (successfully or with an exception).

An `InheritableThreadLocal` is a subclass of `ThreadLocal`, used to propagate thread-local data to subtasks. When a new thread is started, it receives a copy of the parent's inheritable thread locals. (The copy is necessary because the child thread might mutate the thread-local values.)



Caution: Sharing resource-intensive objects as thread locals can be a problem when migrating to virtual threads. There will likely be far more of them than threads in a thread pool, and now you have many more instances. In such a situation, you should rethink your sharing strategy.

One option may be scoped values, a preview feature of Java 21. They are similar to immutable thread-local variables, but have a lighter-weight implementation.



Note: The `java.util.Random` class is threadsafe. But it is still inefficient if multiple threads need to wait for a single shared generator. The `ThreadLocalRandom` class provides a remedy. The call

```
RandomGenerator generator = ThreadLocalRandom.current();
```

returns a random number generator with a per-thread state. This used to be implemented as a thread-local variable. But currently, the generator state is stored in special instance variables of the `Thread` object.

10.8.3. Miscellaneous Thread Properties

The `Thread` class exposes a number of properties for threads, but most of them are more useful for students of certification exams than application programmers. This section briefly reviews them.

Threads can be collected in groups, and there are API methods to manage thread groups, such as interrupting all threads in a group. Nowadays, executors are the preferred mechanism for managing groups of tasks.

You can set *priorities* for threads, where high-priority threads are scheduled to run before lower-priority ones. Hopefully, priorities are honored by the virtual machine and the host platform, but the details are highly platform-dependent. Therefore, using priorities is fragile and not generally recommended.

Threads have *states*, and you can tell whether a thread is new, running, blocked on input/output, waiting, or terminated. When you use threads as an application programmer, you should not need to inquire about their states.

Threads have names, and you can change the name for debugging purposes. For example:

```
Thread.currentThread().setName("Bitcoin-miner-1");
```

When a thread terminates due to an uncaught exception, the exception is passed to the thread's *uncaught exception handler*. By default, its stack trace is dumped to `System.err`, but you can install your own handler (see

[Chapter 5](#)).

A *daemon* is a thread that has no other role in life than to serve others. This is useful for threads that send timer ticks or clean up stale cache entries. When only daemon threads remain, the virtual machine exits.

To make a daemon thread, call `thread.setDaemon(true)` before starting the thread.



Note: All virtual threads are members of a dummy thread group that you cannot change. They all have normal priority and are daemon threads.

10.9. Processes

Up to now, you have seen how to execute Java code in separate threads within the same program. Sometimes, you need to execute another program. For this, use the `ProcessBuilder` and `Process` classes. The `Process` class executes a command in a separate operating system process and lets you interact with its standard input, output, and error streams. The `ProcessBuilder` class lets you configure a `Process` object.



Note: The `ProcessBuilder` class is a more flexible replacement for the `Runtime.exec` calls.

10.9.1. Building a Process

Start the building process by specifying the command that you want to execute. You can supply a `List<String>` or simply the strings that make up the command.

```
var builder = new ProcessBuilder("gcc", "myapp.c");
```



Caution: The first string must be an executable command, not a shell builtin. For example, to run the `dir` command in Windows, you need to

build a process with strings "cmd.exe", "/C", and "dir".

Each process has a *working directory*, which is used to resolve relative directory names. By default, a process has the same working directory as the virtual machine, which is typically the directory from which you launched the java program. You can change it with the `directory` method:

```
builder = builder.directory(path.toFile());
```



Note: Each of the methods for configuring a `ProcessBuilder` returns itself, so that you can chain commands. Ultimately, you will call

```
Process p = new  
ProcessBuilder(command).directory(file).start();
```

Next, you will want to specify what should happen to the standard input, output, and error streams of the process. By default, each of them is a pipe that you can access with

```
OutputStream processIn = p.getOutputStream();  
InputStream processOut = p.getInputStream();  
InputStream processErr = p.getErrorStream();
```

or, for text input and output

```
BufferedWriter processIn = p.outputWriter(charset);  
BufferedReader processOut = p.inputReader(charset);  
BufferedReader processErr = p.errorReader(charset);
```

Note that the input stream of the process is an output stream in the JVM! You write to that stream, and whatever you write becomes the input of the process. Conversely, you read what the process writes to the output and error streams. For you, they are input streams.

You can specify that the input, output, and error streams of the new process should be the same as the JVM. If the user runs the JVM in a console, any user

input is forwarded to the process, and the process output shows up in the console. Call

```
builder.inheritIO()
```

to make this setting for all three streams. If you only want to inherit some of the streams, pass the value

```
ProcessBuilder.Redirect.INHERIT
```

to the `redirectInput`, `redirectOutput`, or `redirectError` methods. For example,

```
builder.redirectOutput(ProcessBuilder.Redirect.INHERIT);
```

You can redirect the process streams to files by supplying `File` objects:

```
builder.redirectInput(inputFile)
    .redirectOutput(outputFile)
    .redirectError(errorFile)
```

The files for output and error are created or truncated when the process starts. To append to existing files, use

```
builder.redirectOutput(ProcessBuilder.Redirect.appendTo(outputFile));
```

It is often useful to merge the output and error streams, so you see the outputs and error messages in the sequence in which the process generates them. Call

```
builder.redirectErrorStream(true)
```

to activate the merging. If you do that, you can no longer call `redirectError` on the `ProcessBuilder` or `getErrorStream` on the `Process`.

Finally, you may want to modify the environment variables of the process. Here, the builder chain syntax breaks down. You need to get the builder's environment (which is initialized by the environment variables of the process running the JVM), then put or remove entries.

```
Map<String, String> env = builder.environment();
env.put("LANG", "fr_FR");
env.remove("JAVA_HOME");
Process p = builder.start();
```

If you want to pipe the output of one process into the input of another (as with the | operator in a shell), use the `startPipeline` method. Pass a list of process builders and read the result from the last process. Here is an example, enumerating the unique extensions in a directory tree:

```
List<Process> processes = ProcessBuilder.startPipeline(List.of(
    new ProcessBuilder("find", "/opt/jdk-17"),
    new ProcessBuilder("grep", "-o", "\\.\\.[^.]\\*$"),
    new ProcessBuilder("sort"),
    new ProcessBuilder("uniq")
));
Process last = processes.get(processes.size() - 1);
var result = new String(last.getInputStream().readAllBytes());
```

This is just an example to show the mechanics. Of course, this particular task would be more efficiently solved by making the directory walk in Java instead of running four processes.

10.9.2. Running a Process

After you have configured the builder, invoke its `start` method to start the process. If you configured the input, output, and error streams as pipes, you can now write to the input stream and read the output and error streams. For example,

```
Process process = new ProcessBuilder("/bin/ls", "-l")
    .directory(Path.of("/tmp").toFile())
    .start();
try (var in = new Scanner(process.getInputStream())) {
    while (in.hasNextLine())
        System.out.println(in.nextLine());
}
```



Caution: There is limited buffer space for the process streams. You should not flood the input, and you should read the output promptly. If there is a lot of input and output, you may need to produce and consume it in separate threads.

To wait for the process to finish, call

```
int result = process.waitFor();
```

or, if you don't want to wait indefinitely,

```
long delay = ...;
if (process.waitFor(delay, TimeUnit.SECONDS)) {
    int result = process.exitValue();
    ...
} else {
    process.destroyForcibly();
}
```

The first call to `waitFor` returns the exit value of the process (by convention, `0` for success or a nonzero error code). The second call returns `true` if the process didn't time out. Then you need to retrieve the exit value by calling the `exitValue` method.

Instead of waiting for the process to finish, you can just leave it running and occasionally call `isAlive` to see whether it is still alive. To kill the process, call `destroy` or `destroyForcibly`. The difference between these calls is platform-dependent. On Unix, the former terminates the process with `SIGTERM`, the latter with `SIGKILL`. (The `supportsNormalTermination` method returns `true` if the `destroy` method can terminate the process normally.)

Finally, you can receive an asynchronous notification when the process has completed. The call `process.onExit()` yields a `CompletableFuture<Process>` that you can use to schedule any action.

```
process.onExit().thenAccept(
    p -> System.out.println("Exit value: " + p.exitValue()));
```

10.9.3. Process Handles

To get more information about a process that your program started, or any other process that is currently running on your machine, use the `ProcessHandle` interface. You can obtain a `ProcessHandle` in four ways:

1. Given a `Process` object `p`, `p.toHandle()` yields its `ProcessHandle`.
2. Given a long operating system process ID, `ProcessHandle.of(id)` yields the handle of that process.
3. `ProcessHandle.current()` is the handle of the process that runs this Java virtual machine.
4. `ProcessHandle.allProcesses()` yields a `Stream<ProcessHandle>` of all operating system processes that are visible to the current process.

Given a process handle, you can get its process ID, its parent process, its children, and its descendants:

```
long pid = handle.pid();
Optional<ProcessHandle> parent = handle.parent();
Stream<ProcessHandle> children = handle.children();
Stream<ProcessHandle> descendants = handle.descendants();
```



Note: The `Stream<ProcessHandle>` instances that are returned by the `allProcesses`, `children`, and `descendants` methods are just snapshots in time. Any of the processes in the stream may be terminated by the time you get around to seeing them, and other processes may have started that are not in the stream.

The `info` method yields a `ProcessHandle.Info` object with methods for obtaining information about the process.

```
Optional<String[]> arguments()
Optional<String> command()
Optional<String> commandLine()
Optional<String> startInstant()
Optional<String> totalCpuDuration()
Optional<String> user()
```

All of these methods return `Optional` values since it is possible that a particular operating system may not be able to report the information.

For monitoring or forcing process termination, the `ProcessHandle` interface has the same `isAlive`, `supportsNormalTermination`, `destroy`, `destroyForcibly`, and `onExit` methods as the `Process` class. However, there is no equivalent to the `waitFor` method.

Exercises

1. Using parallel streams, find all files in a directory that contain a given word.
How do you find just the first one? Are the files actually searched concurrently?
2. How large does an array have to be for `Arrays.parallelSort` to be faster than `Arrays.sort` on your computer?
3. Implement a method yielding a task that reads through all words in a file, trying to find a given word. The task should finish immediately (with a debug message) when it is interrupted. For all files in a directory, schedule one task for each file. Interrupt all others when one of them has succeeded.
4. One parallel operation not discussed in [Section 10.4.2](#) is the `parallelPrefix` method that replaces each array element with the accumulation of the prefix for a given associative operation. Huh? Here is an example. Consider the array `[1, 2, 3, 4, ...]` and the `*` operation. After executing `Arrays.parallelPrefix(values, (x, y) -> x * y)`, the array contains

`[1, 1 × 2, 1 × 2 × 3, 1 × 2 × 3 × 4, ...]`

Perhaps surprisingly, this computation can be parallelized. First, join neighboring elements, as indicated here:

`[1, 1 × 2, 3, 3 × 4, 5, 5 × 6, 7, 7 × 8]`

The gray values are left alone. Clearly, one can make this computation concurrently in separate regions of the array. In the next step, update the indicated elements by multiplying them with elements that are one or two positions below:

```
[1, 1 × 2, 1 × 2 × 3, 1 × 2 × 3 × 4, 5, 5 × 6, 5 × 6 × 7, 5  
× 6 × 7 × 8]
```

This can again be done concurrently. After $\log(n)$ steps, the process is complete. This is a win over the straightforward linear computation if sufficient processors are available.

In this exercise, you will use the `parallelPrefix` method to parallelize the computation of Fibonacci numbers. We use the fact that the n th Fibonacci number is the top left coefficient of F^n , where $F = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$. Make an array filled with 2×2 matrices. Define a `Matrix` class with a multiplication method, use `parallelSetAll` to make an array of matrices, and use `parallelPrefix` to multiply them.

5. Produce an example that demonstrates escaping of `this` in a constructor of an immutable class (see [Section 10.2.3](#)). Try to come up with something convincing and scary. If you use an event listener (as many examples on the Web do), it should listen for something interesting, which isn't easy for an immutable class.
6. Write an application in which multiple threads read all words from a collection of files. Use a `ConcurrentHashMap<String, Set<File>>` to track in which files each word occurs. Use the `merge` method to update the map.
7. Repeat the preceding exercise, but use `computeIfAbsent` instead. What is the advantage of this approach?
8. In a `ConcurrentHashMap<String, Long>`, find the key with maximum value (breaking ties arbitrarily). Hint: `reduceEntries`.
9. Generate 1,000 threads, each of which increments a counter 100,000 times. Compare the performance of using `AtomicLong` versus `LongAdder`.
10. Use a `LongAccumulator` to compute the maximum or minimum of the accumulated elements.
11. Use a blocking queue for processing files in a directory. One thread walks the file tree and inserts files into a queue. Several threads remove the files and search each one for a given keyword, printing out any matches. When the producer is done, it should put a dummy file into the queue.
12. Repeat the preceding exercise, but instead have each consumer compile a map of words and their frequencies that are inserted into a second queue. A final thread merges the dictionaries and prints the ten most common words. Why don't you need to use a `ConcurrentHashMap`?

13. Repeat the preceding exercise, making a `Callable<Map<String, Integer>>` for each file and using an appropriate executor service. Merge the results when all are available. Why don't you need to use a `ConcurrentHashMap`?
14. Use an `ExecutorCompletionService` instead and merge the results as soon as they become available.
15. Repeat the preceding exercise, using a global `ConcurrentHashMap` for collecting the word frequencies.
16. Repeat the preceding exercise, using parallel streams. None of the stream operations should have any side effects.
17. Write a program that walks a directory tree and generates a thread for each file. In the threads, count the number of words in the files and, without using locks, update a shared counter that is declared as

```
public static long count = 0;
```

Run the program multiple times. What happens? Why?

18. Fix the program of the preceding exercise with using a lock.
19. Fix the program of the preceding exercise with using a `LongAdder`.
20. Consider this stack implementation:

```
public class Stack {
    class Node { Object value; Node next; };
    private Node top;

    public void push(Object newValue) {
        var n = new Node();
        n.value = newValue;
        n.next = top;
        top = n;
    }

    public Object pop() {
        if (top == null) return null;
        Node n = top;
        top = n.next;
        return n.value;
    }
}
```

Describe two different ways in which the data structure can fail to contain the correct elements.

21. Consider this queue implementation:

```
public class Queue {  
    class Node { Object value; Node next; };  
    private Node head;  
    private Node tail;  
  
    public void add(Object newValue) {  
        var n = new Node();  
        if (head == null) head = n;  
        else tail.next = n;  
        tail = n;  
        tail.value = newValue;  
    }  
  
    public Object remove() {  
        if (head == null) return null;  
        Node n = head;  
        head = n.next;  
        return n.value;  
    }  
}
```

Describe two different ways in which the data structure can fail to contain the correct elements.

22. What is wrong with this code snippet?

```
public class Stack {  
    private Object myLock = "LOCK";  
  
    public void push(Object newValue) {  
        synchronized (myLock) {  
            ...  
        }  
        ...  
    }  
}
```

23. What is wrong with this code snippet?

```
public class Stack {  
    public void push(Object newValue) {  
        synchronized (new ReentrantLock()) {  
            ...  
        }  
    }  
    ...  
}
```

24. What is wrong with this code snippet?

```
public class Stack {  
    private Object[] values = new Object[10];  
    private int size;  
  
    public void push(Object newValue) {  
        synchronized (values) {  
            if (size == values.length)  
                values = Arrays.copyOf(values, 2 * size);  
            values[size] = newValue;  
            size++;  
        }  
    }  
    ...  
}
```

25. Write a program that asks the user for a URL, reads the web page at that URL, and displays all the links. Use a `CompletableFuture` for each step. Don't call `get`.

26. Write a program that demonstrates a deadlock, using two threads.

27. Write a program that demonstrates a deadlock, using a single thread. Hint: `join`.

28. Write a method

```
public static <T> CompletableFuture<T> repeat(  
    Supplier<T> action, Predicate<T> until)
```

that asynchronously repeats the action until it produces a value that is accepted by the `until` function, which should also run asynchronously. Test with a function that reads a `java.net.PasswordAuthentication` from the console, and a function that simulates a validity check by sleeping for a second and then checking that the password is "secret". Hint: Use recursion.

29. Implement a static method `CompletableFuture<T> <T> supplyAsync(Supplier<T> action, Executor exec)` that returns an instance of a subclass of `CompletableFuture<T>` whose `cancel` method can interrupt the thread that executes the `action` method, provided the task is running. In a `Runnable`, capture the current thread, then call `action.get()`, and complete the `CompletableFuture` with the result or exception.
30. The method

```
static CompletableFuture<Void>
    CompletableFuture.allOf(CompletableFuture<?>... cfs)
```

does not yield the results of the arguments, which makes it a bit cumbersome to use. Implement a method that combines completable futures of the same type:

```
static <T> CompletableFuture<List<T>>
    allOf(List<CompletableFuture<T>> cfs)
```

Note that this method has a `List<T>` parameter since you cannot have variable arguments of a generic type.

31. The method

```
static CompletableFuture<Object>
    CompletableFuture.anyOf(CompletableFuture<?>... cfs)
```

returns as soon as any of the arguments completes, normally *or exceptionally*. This is markedly different from `ExecutorService.invokeAny` which keeps going until one of the tasks completes successfully and prevents the method from being used for a concurrent search. Implement a method

```
static CompletableFuture<T>
anyOf(List<Supplier<T>> actions, Executor exec)
```

that yields the first actual result, or a `NoSuchElementException` if all actions completed with exceptions.

32. Implement *sleep sort*, one of the worst sorting algorithms known to man. Given an array of positive integers, start a thread for each value v that sleeps for v seconds, then adds the value into a blocking queue. Print the queue when all elements have been added.

Chapter 11 ■ Annotations

Annotations are tags that you insert into your source code so that some tool can process them. The tools can operate on the source level, or they can process class files into which the compiler has placed annotations.

Annotations do not change the way your programs are compiled. The Java compiler generates the same virtual machine instructions with or without the annotations.

To benefit from annotations, you need to select a processing tool and use annotations that your processing tool understands, before you can apply that tool to your code.

There is a wide range of uses for annotations. For example, JUnit (available at <https://junit.org>) uses annotations to mark methods that execute tests and to specify how the tests should be run. Jakarta Persistence (<https://jakarta.ee/specifications/persistence/>) uses annotations to define mappings between classes and database tables, so that objects can be persisted automatically without the developer having to write SQL queries.

In this chapter, you will learn the details of the annotation syntax, how to define your own annotations, and how to write annotation processors that work at the source level or at runtime.



Note: For a compelling use of annotations, check out JCommander (<https://jcommander.org>) and picocli (<https://picocli.info>). These libraries use annotations for the processing of command-line arguments.

The key points of this chapter are:

1. You can annotate declarations just as you use modifiers such as `public` or `static`.
2. You can also annotate types that appear in declarations, casts, `instanceof` checks, or method references.
3. An annotation starts with a @ symbol and may contain key/value pairs called elements.
4. Annotation values must be compile-time constants: primitive types, `enum` constants, `Class` literals, other annotations, or arrays thereof.
5. An item can have repeating annotations or annotations of different types.
6. To define an annotation, specify an annotation interface whose methods correspond to the annotation elements.
7. The Java library defines over a dozen annotations, and annotations are extensively used in the Java Enterprise Edition.
8. To process annotations in a running Java program, you can use reflection and query the reflected items for annotations.
9. Annotation processors process source files during compilation, using the Java language model API to locate annotated items.

11.1. Using Annotations

Here is an example of a simple annotation:

```
class CacheTest {  
    ...  
    @Test public void checkRandomInsertions()  
}
```

The annotation `@Test` annotates the `checkRandomInsertions` method. In Java, an annotation is used like a modifier (such as `public` or `static`). The name of each annotation is preceded by an @ symbol.

By itself, the `@Test` annotation does not do anything. It needs a tool to be useful. When running JUnit, it finds the methods that are annotated with `@Test` and calls them.

11.1.1. Annotation Elements

Annotations can have key/value pairs called *elements*, such as

```
@RepeatedTest(value=10, failureThreshold=3)
```

The names and types of the permissible elements are defined by each annotation (see [Section 11.2](#)). The elements can be processed by the tools that read the annotations.

An annotation element is one of the following:

- A primitive type value
- A String
- A Class object
- An instance of an enum
- An annotation
- An array of the preceding (but not an array of arrays)

To illustrate these possibilities, here is a @BugReport annotation, not from an actual library:

```
@BugReport(showStopper=true,  
           assignedTo="Harry",  
           testCase=CacheTest.class,  
           status=BugReport.Status.CONFIRMED,  
           ref=@Reference(id=11235811),  
           reportedBy={"Harry", "Fred"})
```



Note: Since annotations are processed by the compiler, all element values must be compile-time constants.



Caution: An annotation element can never have the value null.

If an element value is an array with a single element, you can omit the braces:

```
@BugReport(reportedBy="Harry") // Same as reportedBy=  
 {"Harry"}
```

Elements can have default values. For example, the `failureThreshold` element of the JUnit `@RepeatedTest` annotation has default `Integer.MAX_VALUE`. Therefore, the annotation `@RepeatedTest(value=10)` is equivalent to `@RepeatedTest(value=10, failureThreshold=Integer.MAX_VALUE)`. When running this test, it is repeated ten times, no matter how often it failed.

If the element name is `value`, and that is the only element you specify, you can omit `value=`. For example, `@RepeatedTest(10)` is the same as `@RepeatedTest(value=10)`.

11.1.2. Multiple and Repeated Annotations

An item can have multiple annotations:

```
@Test  
@Tag("localized")  
void testHello()
```

If the author of an annotation declared it to be repeatable, you can repeat the same annotation multiple times:

```
@Tag("localized")  
@Tag("showstopper")  
void testHello()
```

11.1.3. Annotating Declarations

So far, you have seen annotations applied to method declarations. There are many other places where annotations can occur. They fall into two categories: *declarations* and *type uses*. Declaration annotations can appear at the declarations of

- Classes (including `enum`) and interfaces (including annotation interfaces)
- Methods
- Constructors
- Instance variables (including `enum` constants)
- Local variables (including those declared in `for` and `try-with-resources` statements)
- Parameter variables and `catch` clause parameters
- Type parameters
- Packages and modules

For classes, interfaces, and modules, put the annotations before the `class`, `interface`, or `module` keyword and any modifiers:

```
@Entity public class User { ... }
```

For variables, put them before the type:

```
@SuppressWarnings("unchecked") List<User> users = ...;
public User getUser(@Param("id") String userId)
```

This also holds for record components:

```
public record Rectangle(@ToString(includeName=false) Point
    topLeft,
    int width, int height) {}
```

A type parameter in a generic class or method can be annotated like this:

```
public class Cache<@NonNullV> { ... }
```

A package is annotated in a file `package-info.java` that contains only the package statement preceded by annotations.

```
/**  
 * Package-level documentation  
 */  
@Generated("com.horstmann.generator")  
package com.horstmann.corejava.generated;  
import javax.annotation.processing.Generated;
```

Note that the `import` statement for the annotation comes *after* the package declaration.



Note: Annotations for local variables and packages are discarded when a class is compiled. Therefore, they can only be processed at the source level.

11.1.4. Annotating Type Uses

A declaration annotation provides some information about the item being declared. For example, in the declaration

```
public User getUser(@NonNull String userId)
```

it is asserted that the `userId` parameter is not `null`.



Note: The `@NonNull` annotation is a part of the Checker Framework (<http://types.cs.washington.edu/checker-framework>). With that framework, you can include assertions in your program, such that a parameter is non-null or that a `String` has been localized—that is, adapted to the user's local language and usage. . A static analysis tool then checks whether the assertions are valid in a given body of source code.

Now suppose we have a parameter of type `List<String>`, and we want to express that all of the strings are non-null. That is where type use annotations come in. Place the annotation before the type argument:

```
List<@NotNull String>
```

Type use annotations can appear in the following places:

- With generic type arguments: `List<@NotNull String>, Comparator.<@NotNull String> reverseOrder()`.
- In any position of an array: `@NotNull String[][] words (words[i] [j] is not null), String @NotNull [][] words (words is not null), String[] @NotNull [] words (words[i] is not null)`.
- With superclasses and implemented interfaces: `class Warning extends @Localized Message`.
- With constructor invocations: `new @Localized String(...)`.
- With nested types: `Map.<@Localized Entry`.
- With casts and `instanceof` checks: `(@Localized String) text, if (text instanceof @Localized String)`. (The annotations are only for use by external tools. They have no effect on the behavior of a cast or an `instanceof` check.)
- With exception specifications: `public String read() throws @Localized IOException`.
- With wildcards and type bounds: `List<@Localized ? extends Message>, List<? extends @Localized Message>`.
- With method and constructor references: `@Localized Message::getText`.

There are a few type positions that cannot be annotated:

```
@NotNull String.class // Error—cannot annotate class literal  
import java.lang.@NotNull String; // Error—cannot annotate import
```

You can place annotations before or after other modifiers such as `private` and `static`. It is customary (but not required) to put type use annotations

after other modifiers, and declaration annotations before other modifiers. For example,

```
private @NonNull String text; // Annotates the type use  
@Id private String userId; // Annotates the variable
```



Note: As you will see in [Section 11.2](#), an annotation author needs to specify where a particular annotation can appear. If an annotation is permissible both for a variable and a type use, and it is used in a variable declaration, then both the variable and the type use are annotated. For example, consider

```
public User getUser(@NonNull String userId)
```

if `@NonNull` can apply both to parameters and to type uses, the `userId` parameter is annotated, and the parameter type is `@NonNull String`.

11.1.5. Making Receivers Explicit

Suppose you want to annotate parameters that are not being mutated by a method.

```
public class Point {  
    public boolean equals(@Nullable Object other) { ... }  
}
```

Then a tool that processes this annotation would, upon seeing a call

```
p.equals(q)
```

reason that `q` has not been changed.

But what about `p`?

When the method is called, the receiver variable `this` is bound to `p`, but `this` is never declared, so you cannot annotate it.

Actually, you can declare it, with a rarely used syntax variant, just so that you can add an annotation:

```
public class Point {  
    public boolean equals(@NonNull Point this, @Nullable  
Object other) { ... }  
}
```

The first parameter is called the *receiver parameter*. It must be named `this`. Its type is the class that is being constructed.



Note: You can provide a receiver parameter only for methods, not for constructors. Conceptually, the `this` reference in a constructor is not an object of the given type until the constructor has completed. Instead, an annotation placed on the constructor describes a property of the constructed object.

A different hidden parameter is passed to the constructor of an inner class, namely the reference to the enclosing class object. You can make this parameter explicit as well:

```
static class Sequence {  
    private int from;  
    private int to;  
  
    class Iterator implements java.util.Iterator<Integer> {  
        private int current;  
  
        public Iterator(@NonNull Sequence Sequence.this) {  
            this.current = Sequence.this.from;  
        }  
        ...  
    }  
}
```

```
    }  
    ...  
}
```

The parameter must be named just like when you refer to it, *EnclosingClass.this*, and its type is the enclosing class.

11.2. Defining Annotations

Each annotation must be declared by an *annotation interface*, with the `@interface` syntax. The methods of the interface correspond to the elements of the annotation. For example, the JUnit `RepeatedTest` annotation is defined by the following interface:

```
@Target(ElementType.METHOD)  
@Retention(RetentionPolicy.RUNTIME)  
public @interface RepeatedTest {  
    int failureThreshold();  
    ...  
}
```

The `@interface` declaration creates an actual Java interface. Tools that process annotations receive objects that implement the annotation interface. When the JUnit test runner tool gets an object that implements `Test`, it simply invokes the `failureThreshold` method to retrieve the failure threshold of a particular `RepeatedTest` annotation.

The element declarations in the annotation interface are actually method declarations. The methods of an annotation interface can have no parameters and no `throws` clauses, and they cannot be generic.

The `@Target` and `@Retention` annotations are *meta-annotations*. They annotate the `RepeatedTest` annotation, indicating the places where the annotation can occur and where it can be accessed.

The value of the @Target meta-annotation is an array of ElementType objects, specifying the items to which the annotation can apply. You can specify any number of element types, enclosed in braces. For example,

```
@Target({ElementType.FIELD, ElementType.RECORD_COMPONENT,  
ElementType.TYPE})  
public @interface ToString
```

[Table 11.1](#) shows all possible targets. The compiler checks that you use an annotation only where permitted. For example, if you apply @ToString to a method, a compile-time error results.



Note: An annotation without an @Target restriction can be used with any declarations but not with type parameters and type uses. (These were the only possible targets in the first Java release that supported annotations.)

Table 11.1: Element Types for the @Target Annotation

Element type	Annotation applies to
ANNOTATION_TYPE	Annotation type declarations
PACKAGE	Packages
MODULE	Modules
TYPE	Classes (including enum) and interfaces (including annotation types)
METHOD	Methods
CONSTRUCTOR	Constructors

Element type	Annotation applies to
FIELD	Instance variables (including enum constants)
PARAMETER	Method or constructor parameters
RECORD_COMPONENT	Record components
LOCAL_VARIABLE	Local variables
TYPE_PARAMETER	Type parameters
TYPE_USE	Uses of a type

When you annotate a record component, the annotation is applied to the fields and methods to which the component gives rise, provided the annotation is permitted on fields or methods. For example, in the annotation

```
public record Rectangle(@ToString(includeName=false) Point
    topLeft,
    int width, int height) {}
```

the `topLeft` field receives the annotation if its target types include `ElementType.FIELD`.

The `@Retention` meta-annotation specifies where the annotation can be accessed. There are three choices.

1. `RetentionPolicy.SOURCE`: The annotation is available to source processors, but it is not included in class files.
2. `RetentionPolicy.CLASS`: The annotation is included in class files, but the virtual machine does not load them. This is the default.
3. `RetentionPolicy.RUNTIME`: The annotation is available at runtime and can be accessed through the reflection API.

You will see examples of all three scenarios later in this chapter.

There are several other meta-annotations—see [Section 11.3](#) for a complete list.

To specify a default value for an element, add a `default` clause after the method defining the element. For example,

```
public @interface RepeatedTest {  
    int failureThreshold() default Integer.MAX_VALUE;  
    ...  
}
```

This example shows how to denote a default of an empty array and a default for an annotation:

```
public @interface BugReport {  
    String[] reportedBy() default {};  
    // Defaults to empty array  
    Reference ref() default @Reference(id=0);  
    // Default for an annotation  
    ...  
}
```



Caution: Defaults are not stored with the annotation; instead, they are dynamically computed. If you change a default and recompile the annotation class, all annotated elements will use the new default, even in class files that have been compiled before the default changed.

All annotation interfaces implicitly extend the `java.lang.annotation.Annotation` interface. That interface is a regular interface, *not* an annotation interface.

You cannot extend annotation interfaces, and you never supply classes that implement annotation interfaces. Instead, source processing tools and the virtual machine generate proxy classes and objects when needed.

11.3. Annotations in the Java API

The Java API defines a number of annotation interfaces in the `java.lang`, `java.lang.annotation`, and `javax.annotation` packages. Four of them are meta-annotations that describe the behavior of annotation interfaces. The others are regular annotations that you use to annotate items in your source code. [Table 11.2](#) shows these annotations. I will discuss them in detail in the following two sections.

Table 11.2: Annotations in the Java API

Annotation interface	Applicable to	Purpose
Override	Methods	Checks that this method overrides a superclass method.
Serial	Methods	Checks that this method is a correct serialization method.
Deprecated	All declarations	Marks item as deprecated.
SuppressWarnings	All declarations except packages	Suppresses warnings of a given type.

Annotation interface	Applicable to	Purpose
SafeVarargs	Methods and constructors	Asserts that the varargs parameter is safe to use.
FunctionalInterface	Interfaces	Marks an interface as functional (with a single abstract method).
Generated	All declarations	Marks an item as source code that has been generated by a tool.
Target	Annotations	Specifies the locations to which this annotation can be applied.
Retention	Annotations	Specifies where this annotation can be used.
Documented	Annotations	Specifies that this annotation should be included in the documentation of annotated items.
Inherited	Annotations	Specifies that this annotation is

Annotation interface	Applicable to	Purpose
		inherited by subclasses.
Repeatable	Annotations	Specifies that this annotation can be applied multiple times to the same item.

11.3.1. Annotations for Compilation

The `@Deprecated` annotation can be attached to any items whose use is no longer encouraged. The compiler will warn when you use a deprecated item. This annotation has the same role as the `@deprecated` JavaDoc tag. However, the annotation persists until runtime.



Note: The `jdeprscan` utility which is part of the JDK can scan a set of JAR files for deprecated elements.

The `@Override` annotation makes the compiler check that the annotated method really overrides a method from the superclass. For example, if you declare

```
public class Point {
    @Override public boolean equals(Point other) { ... }
    ...
}
```

then the compiler will report an error—this `equals` method does not override the `equals` method of the `Object` class because that method has a parameter of type `Object`, not `Point`.

The `@Serial` annotation checks that methods used for serialization, which are not declared in interfaces, have the correct parameter types.

The `@SuppressWarnings` annotation tells the compiler to suppress warning messages of a particular category, for example

```
@SuppressWarnings("unchecked") T[] result  
= (T[]) Array.newInstance(cl, n);
```

The `@SafeVarargs` annotation asserts that a method does not corrupt its varargs parameter (see

[Chapter 6](#).

The @Generated annotation is intended for use by code generator tools. Any generated source code can be annotated to differentiate it from programmer-provided code. For example, a code editor can hide the generated code, or a code generator can remove older versions of generated code. Each annotation must contain a unique identifier for the code generator. A date string (in ISO 8601/RFC 3339 format) and a comment string are optional. For example,

```
@Generated(value="com.horstmann.generator",
    date="2015-01-04T12:08:56.235-0700");
```

You have seen the `FunctionalInterface` annotation in

[Chapter 3](#). It is used to annotate conversion targets for lambda expressions, such as

```
@FunctionalInterface  
public interface IntFunction<R> {  
    R apply(int value);  
}
```

If you later add another abstract method, the compiler will generate an error.

Of course, you should only add this annotation to interfaces that describe functions. There are other interfaces with a single abstract method (such as `AutoCloseable`) that are not conceptually functions.

11.3.2. Meta-Annotations

You have already seen the `@Target` and `@Retention` meta-annotations in [Section 11.2](#).

The `@Documented` meta-annotation gives a hint to documentation tools such as JavaDoc. Documented annotations should be treated just like other modifiers (such as `private` or `static`) for documentation purposes. In contrast, other annotations should not be included in the documentation.

For example, the `@SuppressWarnings` annotation is not documented. If a method or field has that annotation, it is an implementation detail that is of no interest to the JavaDoc reader. On the other hand, the `@FunctionalInterface` annotation is documented since it is useful for the programmer to know that the interface is intended to describe a function. [Figure 11.1](#) shows the documentation.

 **Predicate (Java SE 21 & JDK 21) – Firefox Developer Edition**

← → ⌂ https://docs.oracle.com/en/java/javase/21/docs/api/jav 120% ☆

OVERVIEW MODULE PACKAGE CLASS USE TREE PREVIEW NEW DEPRECATED INDEX HELP Java SE 21 & JDK 21

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD SEARCH X

Module java.base
Package java.util.function

Interface Predicate<T>

Type Parameters:

T - the type of the input to the predicate

Functional Interface:

This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

@FunctionalInterface
public interface Predicate<T>

Represents a predicate (boolean-valued function) of one argument.

This is a functional interface whose functional method is `test(Object)`.

Since:
1.8

Method Summary

All Methods	Static Methods	Instance Methods	Abstract Methods	Default Methods
Modifier and Type	Method	Description		
default	Predicate<T> and(Predicate<? super T> other)	Returns a composed predicate that represents a short-circuiting logical AND of this predicate and		

	another.
static <T> Predicate<T> isEqual(Object targetRef)	Returns a predicate that tests if two arguments are equal according to <code>Objects.equals(Object, Object)</code> .
default Predicate<T> negate()	Returns a predicate that represents the logical negation of this predicate.

Figure 11.1: A documented annotation

The `@Inherited` meta-annotation applies only to annotations for classes. When a class has an inherited annotation, then all of its subclasses automatically have the same annotation. This makes it easy to create annotations that work similar to marker interfaces (such as the `Serializable` interface).

Suppose you define an inherited annotation `@Persistent` to indicate that objects of a class can be saved in a database. Then the subclasses of persistent classes are automatically annotated as persistent.

```
@Inherited @interface Persistent { }

@Persistent class Employee { ... }
class Manager extends Employee { ... } // Also @Persistent
```

The `@Repeatable` meta-annotation makes it possible to apply the same annotation multiple times. For example, the JUnit `@Tag` annotation is repeatable. It can be used like this:

```
@Tag("localized")
@Tag("showstopper")
void testHello() { ... }
```

For historical reasons, the implementor of a repeatable annotation needs to provide a *container annotation* that holds the repeated annotations in an array.

Here is how to define the @Tag annotation and its container:

```
@Repeatable(Tag.class)
@interface Tag
{
    String value();
}

@interface Tags
{
    Tag[] value();
}
```

Whenever the user supplies two or more @Tag annotations, they are automatically wrapped into a @Tags annotation. This complicates processing of the annotation, as you will see in the next section.

11.4. Processing Annotations at Runtime

So far, you have seen how to add annotations to source files and how to define annotation types. Now the time has come to see what good can come out of that.

In this section, I show you a simple example of processing an annotation at runtime using the reflection API that you have already seen in

[Chapter 4](#). Suppose we want to reduce the tedium of implementing `toString` methods. Of course, one can write a generic `toString` method using reflection that simply includes all instance variable names and values. But suppose we want to customize that process. We may not want to include all instance variables, or we may want to skip class and variable names. For example, for the `Point` class we may prefer `[5, 10]` instead of `Point[x=5, y=10]`. Of course, any number of other enhancements would be plausible, but let's keep it simple. The point is to demonstrate what an annotation processor can do.

Annotate all classes that you want to benefit from this service with the `@ToString` annotation. In addition, all instance variables that should be included need to be annotated as well. The annotation is defined like this:

```
@Target({ElementType.FIELD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface ToString {
    boolean includeName() default true;
}
```

Here are annotated `Point` and `Rectangle` classes:

```
@ToString(includeName=false)
public class Point {
    @ToString(includeName=false) private int x;
    @ToString(includeName=false) private int y;
    ...
}

@ToString
public record Rectangle(
    @ToString(includeName=false) Point topLeft,
    @ToString int width,
    @ToString int height) {
}
```

The intent is for a rectangle to be represented as string as `Rectangle[[5, 10], width=20, height=30]`.

At runtime, we cannot modify the implementation of the `toString` method for a given class. Instead, let us provide a method that can format any object, discovering and using the `ToString` annotations if they are present.

The key are the methods

```
T getAnnotation(Class<T>)
T getDeclaredAnnotation(Class<T>)
T[] getAnnotationsByType(Class<T>)
T[] getDeclaredAnnotationsByType(Class<T>)
Annotation[] getAnnotations()
Annotation[] getDeclaredAnnotations()
boolean isAnnotationPresent(Class<? extends Annotation>
annotationClass)
```

of the `AnnotatedElement` interface. The reflection classes `Class`, `Field`, `Parameter`, `Method`, `Constructor`, and `Package` implement that interface.

As with other reflection methods, the methods with `Declared` in their name yield annotations in the class itself, whereas the others include inherited ones. In the context of annotations, this means that the annotation is `@Inherited` and applied to a superclass.

If an annotation is not repeatable, call `getAnnotation` to locate it. For example:

```
Class<?> cl = obj.getClass();
ToString ts = cl.getAnnotation(ToString.class);
if (ts != null && ts.includeName()) ...
```

Note that you pass the class object for the annotation (here, `ToString.class`) and you get back an object of some proxy class that implements the `ToString` interface. You can then invoke the interface

methods to get the values of the annotation elements. If the annotation is not present, the `getAnnotation` method returns `null`.

It gets a bit messy if an annotation is repeatable. If you call `getAnnotation` to look up a repeatable annotation, and the annotation was actually repeated, then you also get `null`. That is because the repeated annotations were wrapped inside the container annotation.

In this case, you should call `getAnnotationsByType`. That call “looks through” the container and gives you an array of the repeated annotations. If there was just one annotation, you get it in an array of length 1. With this method, you don’t have to worry about the container annotation.

The `getAnnotations` method gets all annotations (of any type) with which an item is annotated, with repeated annotations wrapped into containers.

Here is the implementation of the annotation-aware `toString` method:

```
public class ToStrings {  
    public static String toString(Object obj) {  
        if (obj == null) return "null";  
        Class<?> cl = obj.getClass();  
        ToString ts = cl.getAnnotation(ToString.class);  
        if (ts == null) return obj.toString();  
        var result = new StringBuilder();  
        if (ts.includeName()) result.append(cl.getName());  
        result.append("[");  
        boolean first = true;  
        for (Field f : cl.getDeclaredFields()) {  
            ts = f.getAnnotation(ToString.class);  
            if (ts != null) {  
                if (first) first = false; else  
                    result.append(", ");  
                f.setAccessible(true);  
  
                if (ts.includeName()) {  
                    result.append(f.getName());  
                    if (ts.value() != null)  
                        result.append(" = ");  
                    result.append(ts.value());  
                }  
            }  
        }  
        result.append("]");  
        return result.toString();  
    }  
}
```

```

        result.append(f.getName());
        result.append("=");
    }
    try {
        result.append(ToStrings.toString(f.get(obj)));
    } catch (ReflectiveOperationException ex) {
        ex.printStackTrace();
    }
}
result.append("]");
return result.toString();
}
}

```

When a class is annotated with `ToString`, the method iterates over its fields and prints the ones that are also annotated. If the `includeName` element is `true`, then the class or field name is included in the string.

Note that the method calls itself recursively. Whenever an object belongs to a class that isn't annotated, its regular `toString` method is used and the recursion stops.

This is a simple but typical use of the runtime annotation API. Look up classes, fields, and so on, using reflection; call `getAnnotation` or `getAnnotationsByType` on the potentially annotated elements to retrieve the annotations; then, invoke the methods of the annotation interfaces to obtain the element values.

11.5. Source-Level Annotation Processing

In the preceding section, you saw how to analyze annotations in a running program. Another use for annotation is the automatic processing of source files to produce more source code, configuration files, scripts, or whatever else one might want to generate.

To show you the mechanics, I will repeat the example of generating `toString` methods. However, this time, let's generate them in Java source. Then the methods will get compiled with the rest of the program, and they will run at full speed instead of using reflection.

11.5.1. Annotation Processors

Annotation processing is integrated into the Java compiler. During compilation, you can *invoke annotation processors* by running

```
javac -processor ProcessorClassName1,ProcessorClassName2,...  
sourceFiles
```

The compiler locates the annotations of the source files. Each annotation processor is executed in turn and given the annotations in which it expressed an interest. If an annotation processor creates a new source file, the process is repeated. Once a processing round yields no further source files, all source files are compiled.



Note: An annotation processor can only generate new source files. It cannot modify an existing source file.

An annotation processor implements the `Processor` interface, generally by extending the `AbstractProcessor` class. You need to specify which annotations your processor supports. In our case:

```
@SupportedAnnotationTypes("com.horstmann.annotations.ToString")  
@SupportedSourceVersion(SourceVersion.RELEASE_8)  
public class ToStringAnnotationProcessor extends  
AbstractProcessor {  
    public boolean process(Set<? extends TypeElement>  
annotations,  
        RoundEnvironment currentRound) {
```

```
    ...
}
```

A processor can claim specific annotation types, wildcards such as "`com.horstmann.*`" (all annotations in the `com.horstmann` package or any subpackage), or even "`*`" (all annotations).

The `process` method is called once for each round, with the set of all annotations that were found in any files during this round, and a `RoundEnvironment` reference that contains information about the current processing round.

11.5.2. The Language Model API

You use the *language model* API for analyzing source-level annotations. Unlike the reflection API, which presents the virtual machine representation of classes and methods, the language model API lets you analyze a Java program according to the rules of the Java language.

The compiler produces a tree whose nodes are instances of classes that implement the `javax.lang.model.element.Element` interface and its subinterfaces, `TypeElement`, `VariableElement`, `ExecutableElement`, and so on. These are the compile-time analogs to the `Class`, `Field/Parameter`, `Method/Constructor` reflection classes.

I do not want to cover the API in detail, but here are the highlights that you need to know for processing annotations.

- The `RoundEnvironment` gives you a set of all elements annotated with a particular annotation, by calling one of the methods

```
Set<? extends Element> getElementsAnnotatedWith(Class<? extends Annotation> a)
Set<? extends Element> getElementsAnnotatedWithAny(
    Set<Class<? extends Annotation>> annotations)
```

The second method is useful for repeated annotations.

- The source-level equivalent of the `AnnotatedElement` interface is `AnnotatedConstruct`. You use the methods

```
A getAnnotation(Class<A> annotationType)  
A[] getAnnotationsByType(Class<A> annotationType)
```

to get the annotation or repeated annotations for a given annotation class.

- A `TypeElement` represents a class or interface. The `getEnclosedElements` method yields a list of its fields and methods.
- Calling `getSimpleName` on an `Element` or `getQualifiedName` on a `TypeElement` yields a `Name` object that can be converted to a string with `toString`.

11.5.3. Using Annotations to Generate Source Code

Let us return to our task of automatically generating `toString` methods. We can't put these methods into the original classes—annotation processors can only produce new classes, not modify existing ones.

Therefore, we'll add all methods into a utility class `ToStrings`:

```
public class ToStrings {  
    public static String toString(Point obj) {  
        Generated code  
    }  
    public static String toString(Rectangle obj) {  
        Generated code  
    }  
    ...  
    public static String toString(Object obj) {  
        return Objects.toString(obj);  
    }  
}
```

Since we don't want to use reflection, we annotate accessor methods, not fields:

```
@ToString  
public class Rectangle {  
    ...  
    @ToString(includeName=false) public Point getTopLeft()  
    { return topLeft; }  
    @ToString public int getWidth() { return width; }  
    @ToString public int getHeight() { return height; }  
}
```

The annotation processor should then generate the following source code:

```
public static String toString(Rectangle obj) {  
    var result = new StringBuilder();  
    result.append("Rectangle");  
    result.append("[");  
    result.append(toString(obj.getTopLeft()));  
    result.append(",");  
    result.append("width=");  
    result.append(toString(obj.getWidth()));  
    result.append(",");  
    result.append("height=");  
    result.append(toString(obj.getHeight()));  
    result.append("]");  
    return result.toString();  
}
```

The “boilerplate” code is in gray. Here is an outline of the method that produces the `toString` method for a class with given `TypeElement`:

```
private void writeToStringMethod(PrintWriter out,  
TypeElement te) {  
    String className = te.getQualifiedName().toString();  
    Print method header and declaration of string builder
```

```

    ToString ann = te.getAnnotation(ToString.class);
    if (ann.includeName()) Print code to add class name
    for (Element c : te.getEnclosedElements()) {
        ann = c.getAnnotation(ToString.class);
        if (ann != null) {
            if (ann.includeName()) Print code to add field name
            Print code to append toString(obj.methodName())
        }
    }
    Print code to return string
}

```

And here is an outline of the `process` method of the annotation processor. It creates a source file for the helper class and writes the class header and one method for each annotated class.

```

public boolean process(Set<? extends TypeElement>
annotations,
                      RoundEnvironment currentRound) {
    if (annotations.size() == 0) return true;
    try {
        JavaFileObject sourceFile =
processingEnv.getFiler().createSourceFile(
            "com.horstmann.annotations.ToStrings");
        try (var out = new
PrintWriter(sourceFile.openWriter())) {
            Print code for package and class
            for (Element e :
currentRound.getElementsAnnotatedWith(ToString.class)) {
                if (e instanceof TypeElement te) {
                    writeToStringMethod(out, te);
                }
            }
            Print code for toString(Object)
        }
    } catch (IOException ex) {

```

```
        processingEnv.getMessager().printMessage(
            Kind.ERROR, ex.getMessage()));
    }
    return true;
}
```

For the tedious details, check the book's companion code.

Note that the `process` method is called in subsequent rounds with an empty list of annotations. It then returns immediately so it doesn't create the source file twice.



Tip: To see the rounds, run the `javac` command with the `-XprintRounds` flag:

Round 1:

```
    input files: {ch11.sec05.Point,
ch11.sec05.Rectangle,
        ch11.sec05.SourceLevelAnnotationDemo}
    annotations: [com.horstmann.annotations.ToString]
    last round: false
```

Round 2:

```
    input files: {com.horstmann.annotations.ToStrings}
    annotations: []
    last round: false
```

Round 3:

```
    input files: {}
    annotations: []
    last round: true
```

This example demonstrates how tools can harvest source file annotations to produce other files. The generated files don't have to be source files.

Annotation processors may choose to generate XML descriptors, property files, shell scripts, HTML documentation, and so on.



Note: You have now seen how to process annotations in source files and in a running program. A third possibility is to process annotations in class files, usually on the fly when loading them into the virtual machine. You need a tool such as ASM (<http://asm.ow2.org>) to locate and evaluate the annotations, and rewrite the byte codes.

Exercises

1. Describe how `Object.clone` could be modified to use a `@Cloneable` annotation instead of the `Cloneable` marker interface.
2. If annotations had existed in early versions of Java, then the `Serializable` interface would surely have been an annotation. Implement a `@Serializable` annotation. Choose a text or binary format for persistence. Provide classes for streams or readers/writers that persist the state of objects by saving and restoring all fields that are primitive values or themselves serializable. Don't worry about cyclic references for now.
3. Repeat the preceding assignment, but do worry about cyclic references.
4. Add a `@Transient` annotation to your serialization mechanism that acts like the `transient` modifier.
5. Define an annotation `@Todo` that contains a message describing whatever it is that needs to be done. Define an annotation processor that produces a reminder list from a source file. Include a description of the annotated item and the todo message.
6. Turn the annotation of the preceding exercise into a repeating annotation.
7. If annotations had existed in early versions of Java, they might have taken the role of JavaDoc. Define annotations `@Param`, `@Return`, and so on, and produce a basic HTML document from them with an annotation processor.
8. Implement the `@TestCase` annotation, generating a source file whose name is the name of the class in which the annotation occurs, followed

by `Test`. For example, if `MyMath.java` contains

```
@TestCase(args="4", expected="24")
@TestCase(args="0", expected="1")
public static long factorial(int n) { ... }
```

then generate a file `MyMathTest.java` with statements

```
assert(MyMath.factorial(4) == 24);
assert(MyMath.factorial(0) == 1);
```

You may assume that the test methods are static, and that `args` contains a comma-separated list of parameters of the correct type.

9. Implement the `@TestCase` annotation as a runtime annotation and provide a tool that checks it. Again, assume that the test methods are static and restrict yourself to a reasonable set of parameter and return types that can be described by strings in the annotation elements.
10. Implement a processor for the `@Resource` annotation that accepts an object of some class and looks for fields of type `String` annotated with `@Resource(name="URL")`. Then load the URL and “inject” the string variable with that content, using reflection.

Chapter 12 ■ The Java Platform Module System

An important characteristic of object-oriented programming is encapsulation. A class declaration consists of a public interface and a private implementation. A class can evolve by changing the implementation without affecting its users. A module system provides the same benefits for programming in the large. A module can make classes and packages selectively available so that its evolution can be controlled.

Several existing Java module systems rely on class loaders to isolate classes. However, Java 9 introduced a new system, called the Java Platform Module System, that is supported by the Java compiler and virtual machine. It was designed to modularize the large code base of the Java platform. You can, if you choose, use this system to modularize your own applications.

Whether or not you use Java platform modules in your own applications, you may be impacted by the modularized Java platform. This chapter shows you how to declare and use Java platform modules. You will also learn how to migrate your applications to work with the modularized Java platform and third-party modules.

The key points of this chapter are:

1. The Java Platform Module System was designed to modularize the Java platform.
2. You can use the Java Platform Module System to modularize applications and libraries.
3. A module is a collection of packages.
4. The properties of a module are defined in `module-info.java`.
5. A module declares on which other modules it depends.
6. A module provides encapsulation. Accessible packages must be explicitly exported.

7. A module may allow reflective access to private features by opening a package or the entire module.
8. The module system provides support for the `ServiceLoader` facility.
9. A modular JAR is a JAR with a `module-info.class` file that is placed on the module path.
10. By placing a regular JAR on the module path, it becomes an automatic module that exports and opens all of its packages.
11. All packages on the class path form the unnamed module.
12. To migrate existing applications, you may need to override access restrictions with command-line options.
13. The `jdeps` tool analyzes the dependencies of a given set of JAR files. The `jlink` tool produces an application with minimal dependencies.

12.1. The Module Concept

In object-oriented programming, the fundamental building block is the class. Classes provide encapsulation. Private features can only be accessed by code that has explicit permission, namely the methods of the class. This makes it possible to reason about access. If a private variable has changed, you can produce a set of all possible culprits. If you need to modify the private representation, you know which methods are affected.

In Java, packages provide the next larger organizational grouping. A package is a collection of classes. Packages also provide a level of encapsulation. Any feature with package access (neither public nor private) is accessible only from methods in the same package.

However, in large systems, this level of access control is not enough. Any public feature (that is, a feature that is accessible outside a package) is accessible everywhere. Suppose you want to modify or drop a rarely used feature. Once it is public, there is no way to reason about the impact of that change.

This is the situation that the Java platform designers faced. Over twenty years, the JDK grows by leaps and bounds, but clearly some features are essentially obsolete. Everyone's favorite example is CORBA. When is the

last time you used it? Yet, the `org.omg.corba` package is shipped with every JDK. As it happens, it would not be too difficult to put all of CORBA into a JAR file so that it can be used by those few who still need it.

What about `java.awt`? It shouldn't be required in a server-side application, right? Except that the class `java.awt.DataFlavor` is used in the implementation of SOAP, an XML-based web services protocol.

The Java platform designers, faced with a giant hairball of code, decided that they needed a structuring mechanism that provides more control. They looked at existing module systems (such as OSGi) and found them unsuitable for their problem. Instead, they designed a new system, called the *Java Platform Module System*, that is now a part of the Java language and virtual machine. That system has been used successfully to modularize the Java API, and you can, if you so choose, use it with your own applications.

A Java platform module consists of:

- A collection of packages
- Optionally, resource files and other files such as native libraries
- A list of the accessible packages in the module
- A list of all modules on which this module depends

The Java platform enforces encapsulation and dependencies, both at compile time and in the virtual machine.

Why should you consider using the Java Platform Module System for your own programs instead of following the traditional approach of using JAR files on the class path? There are two advantages.

1. Strong encapsulation: You can control which of your packages are accessible, and you don't have to worry about maintaining code that you didn't intend for public consumption.
2. Reliable configuration: You avoid common class path problems such as duplicate or missing classes.

There are some issues that the Java Platform Module System does not address, such as versioning of modules. There is no support for specifying which version of a module is required, or for using multiple versions of a module in the same program. These can be desirable features, but you must use mechanisms other than the Java Platform Module System if you need them.

12.2. Naming Modules

A module is a collection of packages. The package names in the module need not be related. For example, the module `java.sql` contains packages `java.sql`, `javax.sql`, and `javax.transaction.xa`. Also, as you can see from this example, it is perfectly acceptable for the module name to be the same as a package name.

Just like a package name, a module name is made up of letters, digits, underscores, and periods. Also, just as with package names, there is no hierarchical relationship between modules. If you had a module `com.horstmann` and another module `com.horstmann.corejava`, they would be unrelated, as far as the module system is concerned.

When creating a module for use by others, it is important that its name is globally unique. It is expected that most module names will follow the “reverse domain name” convention, just like package names.

The easiest approach is to name a module after the top-level package that the module provides. For example, the SLF4J logging façade has a module `org.slf4j` with packages `org.slf4j`, `org.slf4j.spi`, `org.slf4j.event`, and `org.slf4j.helpers`.

This convention prevents package name conflicts in modules. Any given package can only be placed in one module. If your module names are unique and your package names start with the module name, then your package names will also be unique.

You can use shorter module names for modules that are not meant to be used by other programmers, such as a module containing an application program. Just to show that it can be done, I will do the same in this chapter. Modules with what could plausibly be library code will have names such as `com.horstmann.util`, and modules containing programs (with a class that has a `main` method) will have catchy names such as `ch12.sec03`.



Note: You only use module names in module declarations. In the source files for your Java classes, you never refer to module names. Instead, you use package names in the same way that they have always been used.

12.3. The Modular “Hello, World!” Program

Let us put the traditional “Hello, World!” program into a module. First we need to put the class into a package—the “unnamed package” cannot be contained in a module. Here it is:

```
package com.horstmann.hello;

public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, Modular World!");
    }
}
```

So far, nothing has changed. To make a module `ch12.sec03` containing this package, you need to add a module declaration. You place it in a file named `module-info.java`, located in the base directory (that is, the same directory that contains the `com` directory). By convention, the name of the base directory is the same as the module name.

```
ch12.sec03/
└ module-info.java
  com/
    └ horstmann/
      └ hello/
        └ HelloWorld.java
```

The `module-info.java` file contains the module declaration:

```
module ch12.sec03 {
}
```

This module declaration is empty because the module has nothing to offer to anyone, nor does it need anything.

Now you compile as usual:

```
javac ch12.sec03/module-info.java
ch12.sec03/com/horstmann/hello/HelloWorld.java
```

The `module-info.java` file doesn't look like a Java source file, and of course there can't be a class with the name `module-info`, since class names cannot contain hyphens. The `module` keyword, as well as keywords `requires`, `exports`, and so on, that you will see in the following sections, are “restricted keywords” that have a special meaning only in module declarations. The file is compiled into a class file `module-info.class` that contains the module definition in binary form.

To run this program as a modular application, you specify the *module path*, which is similar to the class path, but it contains modules. You also specify the main class in the format *modulename/classname*:

```
java --module-path ch12.sec03 --module
ch12.sec03/com.horstmann.hello.HelloWorld
```

Instead of `--module-path` and `--module`, you can use the single-letter options `-p` and `-m`:

```
java -p ch12.sec03 -m  
ch12.sec03/com.horstmann.hello.HelloWorld
```

Either way, the "Hello, Modular World!" greeting will appear, demonstrating that you have successfully modularized your first application.



Note: When you compile this module, you get two warnings:

```
warning: [module] module name component sec03 should  
avoid terminal digits  
warning: [module] module name component ch12 should  
avoid terminal digits
```

These warnings are intended to discourage programmers from adding version numbers to module names. You can ignore them, or suppress them with an annotation:

```
@SuppressWarnings("module")  
module ch12.sec03 {  
}
```

In this one respect, the `module` declaration is just like a class declaration: You can annotate it. (The annotation type must have target `ElementType.MODULE`.)

12.4. Requiring Modules

Let us make a new module `ch12.sec04` in which a program uses a `JOptionPane` to show the "Hello, Modular World!" message:

```
package com.horstmann.hello;  
  
import javax.swing.JOptionPane;  
  
public class HelloWorld {
```

```
public static void main(String[] args) {
    JOptionPane.showMessageDialog(null, "Hello, Modular
World!");
}
```

Now compilation fails with this message:

```
error: package javax.swing is not visible
(package javax.swing is declared in module java.desktop,
but module ch12.sec04 does not read it)
```

The JDK has been modularized, and the `javax.swing` package is now contained in the `java.desktop` module. Our module needs to declare that it relies on that module:

```
module ch12.sec04 {
    requires java.desktop;
}
```

It is a design goal of the module system that modules are explicit about their requirements, so that the virtual machine can ensure that all requirements are fulfilled before starting a program.

In the preceding section, the need for explicit requirements did not arise because we only used the `java.lang` package. This package is included in the `java.base` module which is required by default.

Note that our `ch12.sec04` module lists only its own module requirements. It requires the `java.desktop` module so that it can use the `javax.swing` package. The `java.desktop` module itself declares that it requires three other modules, namely `java.datatransfer`, `java.prefs`, and `java.xml`.

[Figure 12.1](#) shows the *module graph* whose nodes are modules. The edges of the graph (that is, the arrows joining nodes) are either declared requirements or, when no requirement is declared, the implied requirement of `java.base`.

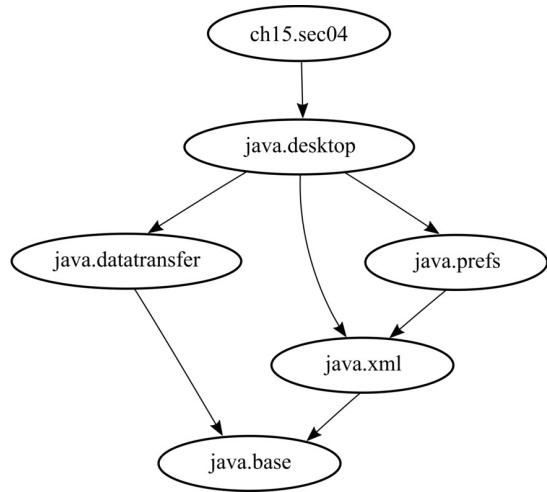


Figure 12.1: The module graph of the Swing “Hello, Modular World!” application

You cannot have cycles in the module graph. That is, a module cannot directly or indirectly require itself.

A module does not automatically pass on access rights to other modules. In our example, the `java.desktop` module declares that it requires `java.prefs`, and the `java.prefs` module declares that it requires `java.xml`. That does not give `java.desktop` the right to use packages from the `java.xml` module. It needs to explicitly declare that requirement. In mathematical terms, the `requires` relationship is not “transitive.” Generally, this behavior is desirable because it makes requirements explicit, but as you will see in [Section 12.11](#), you can relax it in some cases.



Note: The error message at the beginning of this section stated that our `ch12.sec04` module did not “read” the `java.desktop` module. In the parlance of the Java Platform Module System, module *M* *reads* module *N* in the following cases:

1. M requires N .
 2. M requires a module that transitively requires N (see [Section 12.11](#)).
 3. N is M or `java.base`.
-

12.5. Exporting Packages

In the preceding section, you saw that a module must require another module if it wants to use its packages. However, that does not automatically make all packages in the required module available. A module states which of its packages are accessible, using the `exports` keyword. For example, here is a part of the module declaration for the `java.xml` module:

```
module java.xml {  
    exports javax.xml;  
    exports javax.xml.catalog;  
    exports javax.xml.datatype;  
    exports javax.xml.namespace;  
    exports javax.xml.parsers;  
    ...  
}
```

This module makes many packages available, but hides others (such as `jdk.xml.internal`) by not exporting them.

When a package is exported, its `public` and `protected` classes and interfaces, and their `public` and `protected` members, are accessible outside the module. (As always, `protected` types and members are accessible only in subclasses and the same package.)

However, a package that is not exported is not accessible outside its own module. This is quite different from Java before modules. In the past, you were able to use public classes from any package, even if it was not part of the public API. For example, it was commonly recommended to use classes such as `sun.misc.BASE64Encoder` or

`com.sun.rowset.CachedRowSetImpl` when the public API did not provide the appropriate functionality.

Nowadays, you can no longer access unexported packages from the Java platform API since all of them are contained inside modules. As a result, some programs will no longer run with modern Java versions. Of course, nobody ever committed to keeping non-public APIs available, so this should not come as a shock.

Let us put exports to use in a simple situation. We will prepare a module `com.horstmann.greet` that exports a package, also called `com.horstmann.greet`, following the convention that a module that provides code for others should be named after the top-level package inside it. There is also a package `com.horstmann.greet.internal` that we don't export.

A public `Greeter` interface is in the first package.

```
package com.horstmann.greet;

public interface Greeter {
    static Greeter newInstance() {
        return new
com.horstmann.greet.internal.GreeterImpl();
    }

    String greet(String subject);
}
```

The second package has a class that implements the interface. The class is public since it is accessed in the first package.

```
package com.horstmann.greet.internal;

import com.horstmann.greet.Greeter;

public class GreeterImpl implements Greeter {
```

```
public String greet(String subject) {  
    return "Hello, " + subject + "!";  
}  
}
```

The `com.horstmann.greet` module contains both packages but only exports the first:

```
module com.horstmann.greet {  
    exports com.horstmann.greet;  
}
```

The second package is inaccessible outside the module.

We put our application into a second module, which will require the first module:

```
module ch12.sec05 {  
    requires com.horstmann.greet;  
}
```



Note: The `exports` statement is followed by a package name, whereas `requires` is followed by a module name.

Our application now uses a `Greeter` to obtain a greeting:

```
package com.horstmann.hello;  
  
import com.horstmann.greet.Greeter;  
  
public class HelloWorld {  
    public static void main(String[] args) {  
        Greeter greeter = Greeter.newInstance();
```

```
        System.out.println(greeter.greet("Modular World"));
    }
}
```

Here is the source file structure for these two modules:

```
com.horstmann.greet
└ module-info.java
com
└ horstmann
  └ greet
    └ Greeter.java
    └ internal
      └ GreeterImpl.java
ch12.sec05
└ module-info.java
com
└ horstmann
  └ hello
    └ HelloWorld.java
```

To build this application, first compile the `com.horstmann.greet` module:

```
javac com.horstmann.greet/module-info.java \
       com.horstmann.greet/com/horstmann/greet/Greeter.java \
       com.horstmann.greet/com/horstmann/greet/internal/GreeterImp
1.java
```

Then compile the application module with the first module on the module path:

```
javac -p com.horstmann.greet ch12.sec05/module-info.java \
       ch12.sec05/com/horstmann/hello/HelloWorld.java
```

Finally, run the program with both modules on the module path:

```
java -p ch12.sec05:com.horstmann.greet \
    -m ch12.sec05/com.horstmann.hello.HelloWorld
```

This example demonstrates how an implementation can be hidden in a module. The implementing class is inaccessible outside the module. Clients use an interface and a factory method, instead of instantiating the implementing class.



Note: In the Windows cmd shell, you have to enclose `ch12.sec05:com.horstmann.greet` in quotation marks because the colon character has a special meaning. You also need to use ^ instead of \ for multi-line commands.

You have now seen the `requires` and `exports` statements that form the backbone of the Java Platform Module System. As you can see, the module system is conceptually simple. Modules specify what modules they need, and which packages they offer to other modules. [Section 12.12](#) shows a minor variation of the `exports` statement.



Caution: A module does not provide a scope. You cannot have two packages with the same name in different modules. This is true even for hidden packages (that is, packages that are not exported.)

12.6. Modules and Reflective Access

In the preceding section, you saw that the module system enforces encapsulation. A module can only access explicitly exported packages from another module. In the past, it was always possible to overcome pesky access restrictions by using reflection. As you have seen in

[Chapter 4](#), reflection can access private members of any class.

However, in the modular world, that is no longer true. If a class is inside a module, reflective access to non-public members will fail. Specifically, recall how we accessed private fields:

```
Field f = obj.getClass().getDeclaredField("salary");
f.setAccessible(true);
double value = f.getDouble(obj);
f.setDouble(obj, value * 1.1);
```

The call `f.setAccessible(true)` succeeds unless a security manager disallows private field access. However, it is not common to run Java applications with security managers, and there are many libraries that use reflective access. Typical examples are object-relational mappers, such as JPA, that automatically persist objects in databases, and libraries that provide converters between Java objects and JSON or XML.

If you use such a library, and you also want to use modules, you have to be careful. To demonstrate this issue, I will use the Yasson implementation of the JSON-B standard. Here is a trivial class that illustrates the mechanism:

```
package com.horstmann.places;

public class Country {
    public Country() {}

    public Country(String name, double area) {
        this.name = name;
        this.area = area;
    }
    // ...
}
```

Here is a short program that demonstrates how to convert an object into JSON:

```

package com.horstmann.places;

import jakarta.json.bind.*;
import jakarta.json.bind.config.*;
import java.lang.reflect.*;

public class Demo {
    public static void main(String[] args) throws
JAXBException {
        var belgium = new Country("Belgium", 30510);

        var config = new JsonbConfig()
            .withPropertyVisibilityStrategy(
                new PropertyVisibilityStrategy() {
                    public boolean isVisible(Field field) {
return true; }
                    public boolean isVisible(Method method)
{ return false; }
                });
        Jsonb jsonb = JsonbBuilder.create(config);
        String json = jsonb.toJson(belgium);
        System.out.println(json);
    }
}

```

When you run the program, it prints

```
{"area":30510.0,"name":"Belgium"}
```

As you can see, there is nothing that the programmer needs to do to make this happen. Through reflection, the JSON-B library determines the names and values of the fields.

Now let us put the `Country` and `Demo` classes inside a module. When you do that, the `Demo` program will fail with an exception:

```
Exception in thread "main"
java.lang.reflect.InaccessibleObjectException:
    Unable to make field
private java.lang.String com.horstmann.places.Country.name
accessible:
module v2ch09.openpkg2 does not "opens
com.horstmann.places"
    to module org.eclipse.yasson
```

Of course, in pristine theory, it is wrong to violate encapsulation and poke around in the private members of an object. But mechanisms such as XML binding or object-relational mapping are so common that the module system must accommodate them.

Using the `opens` keyword, a module can *open* a package, which enables reflective access to all instances of classes in the given package. Here is what our module has to do:

```
module ch12.sec06 {
    requires jakarta.json.bind;
    opens com.horstmann.places;
}
```

With this change, JSON-B will work correctly.

A module can be declared as `open`, such as

```
open module ch12.sec06 {
    requires jakarta.json.bind;
}
```

An open module grants runtime access to all of its packages, as if all packages had been declared with `exports` and `opens`. However, only explicitly exported packages are accessible at compile time. Open modules combine the compile-time safety of the module system with the classic permissive runtime behavior.

JAR files can contain, in addition to class files and a manifest, *file resources* which can be loaded with the method `Class.getResourceAsStream`, and now also with `Module.getResourceAsStream`. If a resource is stored in a directory that matches a package in a module, then the package must be opened to the caller. Resources in other directories, as well as the class files and manifest, can be read by anyone.



Note: It is possible that future libraries will use *variable handles* instead of reflection for reading and writing fields. A `VarHandle` is similar to a `Field`. You can use it to read or write a specific field of any instance of a specific class. However, to obtain a `VarHandle`, the library code needs a `Lookup` object:

```
public Object getFieldValue(Object obj, String
fieldNames, Lookup lookup)
    throws NoSuchFieldException,
IllegalAccessException {
    Class<?> cl = obj.getClass();
    Field field = cl.getDeclaredField(fieldName);
    VarHandle handle = MethodHandles.privateLookupIn
(cl, lookup)
        .unreflectVarHandle(field);
    return handle.get(obj);
}
```

This works provided the `Lookup` object is generated in the module that has the permission to access the field. Some method in the module simply calls `MethodHandles.lookup()`, which yields an object encapsulating the access rights of the caller. In this way, one module can give permission for accessing private members to another module. The practical issue is how those permissions can be given with a minimum of hassle.

12.7. Modular JARs

So far, we have simply compiled modules into the directory tree of the source code. Clearly, that is not satisfactory for deployment. Instead, a module can be deployed by placing all its classes in a JAR file, with a `module-info.class` in the root. Such a JAR file is called a *modular JAR*.

To create a modular JAR file, you use the `jar` tool in the usual way. If you have multiple packages, it is best to compile with the `-d` option which places class files into a separate directory. The directory is created if it doesn't already exist. Then use the `-C` option of the `jar` command to change to that directory when collecting files.

```
javac -d modules/com.horstmann.greet \
       com.horstmann.greet/module-info.java \
       com.horstmann.greet/com/horstmann/greet/internal/GreeterImp
1.java \
       com.horstmann.greet/com/horstmann/greet/Greeter.java
jar -cvf com.horstmann.greet.jar -C
modules/com.horstmann.greet .
```

If you use a build tool such as Maven, Ant, or Gradle, just keep building your JAR file as you always do. As long as `module-info.class` is included, you get a modular JAR.

Then you can include the modular JAR in the module path, and the module will be loaded.



Caution: In the past, the classes of a package were sometimes distributed over multiple JAR files. (Such a package is called a “split package.”) This was probably never a good idea, and it is not possible with modules.

As with regular JAR files, you can specify a main class in a modular JAR:

```
javac -p com.horstmann.greet.jar -d modules/ch12.sec05 \
    ch12.sec05/module-info.java \
    ch12.sec05/com/horstmann/hello/HelloWorld.java
jar -c -v -f ch12.sec05.jar -e
com.horstmann.hello.HelloWorld -C modules/ch12.sec05 .
```

When you launch the program, you specify the module containing the main class:

```
java -p com.horstmann.greet.jar:ch12.sec05.jar -m
ch12.sec05
```

When creating a JAR file, you can optionally specify a version number. Use the `--module-version` argument and the version number to the JAR file name:

```
jar -c -v -f com.horstmann.greet-1.0.jar --module-version
1.0 -C com.horstmann.greet .
```

As already discussed, the version number is not used by the Java Platform Module System for resolving modules, but it can be queried by other tools and frameworks.



Note: You can find out the version number through the reflection API. In our example:

```
Optional<String> version
  =
Greeter.class.getModule().getDescriptor().rawVersion();
```

yields an `Optional` containing the version string "1.0".



Note: The module equivalent to a class loader is a *layer*. The Java Platform Module System loads the JDK modules and application

modules into the *boot layer*. A program can load other modules, using the layer API (which is not covered in this book). Such a program may choose to take module versions into account. It is expected that developers of programs such as Java EE application servers make use of the layer API to provide support for modules.



Tip: If you want to load a module into JShell, include the JAR on the module path and use the `--add-modules` option:

```
jshell --module-path com.horstmann.greet-1.0.jar \  
--add-modules com.horstmann.greet
```

12.8. Automatic Modules

You now know how to put the Java Platform Module System to use. If you start with a brand-new project in which you write all the code yourself, you can design modules, declare module dependencies, and package your application into modular JAR files.

However, that is an extremely uncommon scenario. Almost all projects rely on third-party libraries. Of course, you can wait until the providers of all libraries have turned them into modules, and then modularize your own code.

But what if you don't want to wait? The Java Platform Module System provides two mechanisms for crossing the chasm that separates today's premodular world and fully modular applications: automatic modules and the unnamed module.

For migration purposes, you can turn any JAR file into a module, simply by placing it onto a directory in the module path instead of the class path. A JAR without a `module-info.class` on the module path is called an *automatic module*. An automatic module has the following properties:

1. The module implicitly has a `requires` clause for all other modules.
2. All of its packages are exported and opened.
3. If there is an entry with key `Automatic-Module-Name` in the JAR file manifest `META-INF/MANIFEST.MF`, the value becomes the module name.
4. Otherwise the module name is obtained from the JAR file name, dropping any trailing version number and replacing sequences of non-alphanumeric characters with a dot.

The first two rules imply that the packages in the automatic module act as if they were on the class path. The reason for using the module path is for the benefit of other modules, allowing them to express dependencies on this module.

Suppose, for example, that you are implementing a module that processes CSV files and uses the Apache Commons CSV library. You would like to express in your `module-info.java` file that your module depends on Apache Commons CSV.

If you add `commons-csv-1.9.0.jar` onto the module path, then your modules can reference the module. Its name is `commons.csv` since the trailing version number `-1.9.0` is removed and the non-alphanumeric character `-` is replaced by a dot.

This name might be an acceptable module name because Commons CSV is well known and it is unlikely that someone else will try to use the same name for a different module. But it would be better if the maintainers of this JAR file could quickly agree to reserve a reverse DNS name, preferably the top-level package name `org.apache.commons.csv` as the module name. They just need to add a line

`Automatic-Module-Name: org.apache.commons.csv`

to the `META-INF/MANIFEST.MF` file inside the JAR. Eventually, hopefully, they will turn the JAR file into a true module by adding `module-info.java` with the reserved module name, and every other module that refers to the CSV module with that name will just continue to work.



Note: The migration plan to modules is a great social experiment, and nobody knows whether it will end well. Before you put third-party JARs on the module path, check whether they are modular, and if not, whether their manifest has a module name. If not, you can still turn the JAR into an automatic module, but be prepared to update the module name later.

12.9. The Unnamed Module

Any class that is not on the module path is part of an *unnamed module*. Technically, there may be more than one unnamed module, but all of them together act as if they were a single module, which is called *the unnamed module*. As with automatic modules, the unnamed module can access all other modules, and all of its packages are exported and opened.

However, *no explicit module* can access the unnamed module. (An explicit module is a module that is neither automatic nor unnamed—that is, a module with a `module-info.class` on the module path.) In other words, explicit modules are always free from the “class path hell.”

Suppose, for example, that you put `commons-csv-1.9.0.jar` onto the class path instead of the module path. Then the sample program for the preceding section won’t start:

```
Error occurred during initialization of boot layer
java.lang.module.FindException: Module commons.csv not
found, required by ch12.sec09
```

Therefore, migration to the Java Platform Module System is necessarily a bottom-up process:

1. The Java platform itself is modularized.
2. Next, libraries are modularized, either by using automatic modules or by turning them into explicit modules.

3. Once all libraries used by your application are modularized, you can turn the code of your application into a module.
-



Note: Automatic modules *can* read the unnamed module, so their dependencies can go onto the class path.

12.10. Command-Line Flags for Migration

Even if your programs do not use modules, you cannot escape the modular world when running with Java 9 and beyond. Even if the application code resides on the class path in an unnamed module and all packages are exported and opened, it interacts with the Java platform, which is modularized.

As of Java 11, compile-time encapsulation is strictly enforced. However, before Java 16, runtime access was permitted. The default behavior was to display a warning on the console for the first instance of each offense. As of Java 16, reflective access at runtime is also enforced. In order to give you time to prepare for that change, the `java` launcher in Java 9 through 16 had an `--illegal-access` flag with four possible settings:

1. `--illegal-access=permit` is the Java 9 default behavior, printing a message for the first instance of illegal access.
2. `--illegal-access=warn` prints a message for each illegal access.
3. `--illegal-access=debug` prints a message and stack trace for each illegal access.
4. `--illegal-access=deny` was the Java 16 default behavior, denying all illegal access.

The `--illegal-access` flag is no longer usable as of Java 17.

The `--add-exports` and `--add-opens` flags allows you to tweak legacy applications. Consider an application that uses an internal API that is no longer accessible, such as `com.sun.rowset.CachedRowSetImpl`. The best

remedy is to change the implementation. (As of Java 7, you can get a cached row set from a `RowSetProvider`.) But suppose you don't have access to the source code.

In that case, start the application with the `--add-exports` flag. Specify the module and the package that you want to export, and the module to which you want to export the package, which in our case is the unnamed module.

```
java --illegal-access=deny --add-exports  
java.sql.rowset/com.sun.rowset=ALL_UNNAMED \  
-jar MyApp.jar
```

Now suppose your application uses reflection to access private fields or methods. Reflection inside the unnamed module is OK, but it is no longer possible to reflectively access non-public members of the Java platform classes. For example, some libraries that dynamically generate Java classes call the protected `ClassLoader.defineClass` method through reflection. If an application uses such a library, add the flag

```
--add-opens java.base/java.lang=ALL_UNNAMED
```

When adding all those command-line options to get a legacy app to work, you may well end up with the command line from hell. To better manage a multitude of options, you can put options in one or more files which you specify with an @ prefix. For example,

```
java @options1 @options2 -jar MyProg.java
```

where the files `options1` and `options2` contain options for the `java` command.

There are a few syntax rules for the options files:

- Separate options with spaces, tabs, or newlines.
- Use double quotes around arguments that include spaces, such as "Program Files".
- A line ending in a \ is merged with the next line.

- Backslashes must be escaped, such as C:\\Users\\\\Fred.
- Comment lines start with #.

12.11. Transitive and Static Requirements

In [Section 12.4](#), you have seen the basic form of the `requires` statement. In this section, you will see two variants that are occasionally useful.

In some situations, it can be tedious for a user of a given module to declare all required modules. Consider, for example, the `java.desktop` module. It requires three modules: `java.prefs`, `java.datatransfer`, and `java.xml`. The `java.prefs` module is only used internally. However, classes from `java.datatransfer` and `java.xml` appear in the public API, in methods such as

```
java.awt.datatransfer.Clipboard
java.awt.Toolkit.getSystemClipboard()
java.beans.XMLDecoder(org.xml.sax.InputSource is)
```

That is not something that a user of the `java.desktop` module should have to think about. For that reason, the `java.desktop` module declares the requirement with the `transitive` modifier:

```
module java.desktop
{
    requires java.prefs;
    requires transitive java.datatransfer;
    requires transitive java.xml;
    ...
}
```

Any module that declares a requirement on `java.desktop` now automatically requires these two modules.



Note: Some programmers recommend that you should always use `requires transitive` when a package from another module is used in the public API. But that is not a rule of the Java language. Consider, for example, the `java.sql` module:

```
module java.sql {  
    requires transitive java.logging;  
    ...  
}
```

There is a single use of a package from the `java.logging` module in the entire `java.sql` API, namely the `java.sql.Driver.parentLogger` method that returns a `java.util.logging.Logger`. It would have been perfectly acceptable to not declare this module requirement as transitive. Then those modules—and only those—who actually use that method would need to declare that they require `java.logging`.

One compelling use of the `requires transitive` statement is an *aggregator* module—a module with no packages and only transitive requirements. One such module is the `java.se` module, declared like this:

```
module java.se {  
    requires transitive java.compiler;  
    requires transitive java.datatransfer;  
    requires transitive java.desktop;  
    ...  
    requires transitive java.sql;  
    requires transitive java.sql.rowset;  
    requires transitive java.xml;  
    requires transitive java.xml.crypto;  
}
```

A programmer who isn't interested in fine-grained module dependencies can simply require `java.se` and get all modules of the Java SE platform.

Finally, there is an uncommon `requires static` variant declaring that a module must be present at compile time but is optional at runtime. There are two use cases:

1. To access an annotation that is processed at compile time and declared in a different module.
2. To use a class in a different module if it is available, and otherwise do something else, such as:

```
try {
    new oracle.jdbc.driver.OracleDriver();
    ...
} catch (NoClassDefFoundError er) {
    // Do something else
}
```

12.12. Qualified Exporting and Opening

In this section, you will see a variant of the `exports` and `opens` statement that narrows their scope to a specified set of modules. For example, the `java.base` module contains a statement

```
exports sun.net to
    java.net.http,
    jdk.naming.dns;
```

Such a statement is called a *qualified export*. The listed modules can access the exported package, but other modules cannot.

Excessive use of qualified exports can indicate a poor modular structure. Nevertheless, they can arise when modularizing an existing code base. Here, the `sun.net` package is placed inside the `java.base` module because that is where it is mostly needed. However, a couple of other modules also

use that package. The Java platform designers didn't want to make `java.base` even bigger, and they didn't want to make the internal `sun.net` package generally available. In a greenfield project, one can instead design a more modular API.

Similarly, you can restrict the `opens` statement to specific modules. For example, in [Section 12.6](#) we could have used a qualified `opens` statement, like this:

```
module ch12.sec06 {  
    requires jakarta.json.bind;  
    opens com.horstmann.places to org.eclipse.yasson;  
}
```

Now the `com.horstmann.places` package is only opened to the `org.eclipse.yasson` module.

Admittedly, it seems rather brittle to put a dependency on a particular persistence mechanism into the module descriptor. Instead, you could place all classes that need to be persisted into a separate package and open up that package to all modules, so that any persistence mechanism can access it.

12.13. Service Loading

The `ServiceLoader` class (see [Section 12.13](#)) provides a lightweight mechanism for matching up service interfaces with implementations. The Java Platform Module System makes it easier to use this mechanism.

Here is a quick reminder of service loading. A service has an interface and one or more possible implementations. Here is an example of a simple interface:

```
public interface GreeterService {  
    String greet(String subject);  
    Locale getLocale();  
}
```

One or more modules provide implementations, such as

```
public class FrenchGreeter implements GreeterService {  
    public String greet(String subject) { return "Bonjour "  
+ subject; }  
    public Locale getLocale() { return Locale.FRENCH; }  
}
```

The service consumer must pick an implementation among all offered implementations, based on whatever criteria it deems appropriate.

```
ServiceLoader<GreeterService> greeterLoader =  
ServiceLoader.load(GreeterService.class);  
GreeterService chosenGreeter;  
for (GreeterService greeter : greeterLoader) {  
    if (...) {  
        chosenGreeter = greeter;  
    }  
}
```

In the past, implementations were offered by placing text files into the `META-INF/services` directory of the JAR file containing the implementation classes. The module system provides a better approach. Instead of text files, you add statements to the module descriptors.

A module providing an implementation of a service adds a `provides` statement that lists the service interface (which may be defined in any module) and the implementing classes (which must be a part of this module). Here is an example from the `jdk.security.auth` module:

```
module jdk.security.auth {  
    ...  
    provides javax.security.auth.spi.LoginModule with  
        com.sun.security.auth.module.Krb5LoginModule,  
        com.sun.security.auth.module.UnixLoginModule,  
        com.sun.security.auth.module.JndiLoginModule,  
        com.sun.security.auth.module.KeyStoreLoginModule,
```

```
        com.sun.security.auth.module.LdapLoginModule,  
        com.sun.security.auth.module.NTLoginModule;  
    }
```

The consuming modules contain a `uses` statement.

```
module java.base {  
    ...  
    uses javax.security.auth.spi.LoginModule;  
}
```

When code in a consuming module calls `ServiceLoader.load(serviceInterface.class)`, the matching provider classes will be loaded, even though they may not be in accessible packages.

12.14. Tools for Working with Modules

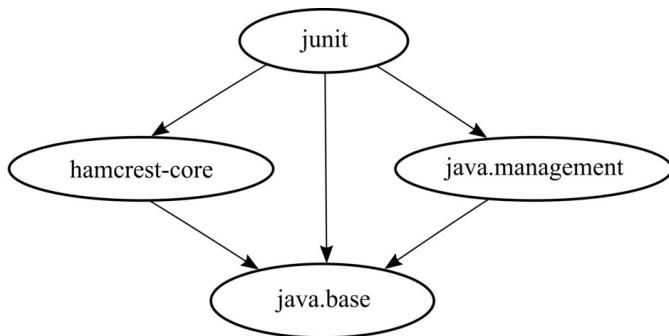
The `jdeps` tool analyzes the dependencies of a given set of JAR files. Suppose, for example, you want to modularize JUnit 4. Run

```
jdeps -s junit-4.12.jar hamcrest-core-1.3.jar
```

The `-s` flag generates a summary output:

```
hamcrest-core-1.3.jar -> java.base  
junit-4.12.jar -> hamcrest-core-1.3.jar  
junit-4.12.jar -> java.base  
junit-4.12.jar -> java.management
```

That tells you the module graph:



If you omit the `-s` flag, you get the module summary followed by a mapping from packages to required packages and modules. If you add the `-v` flag, the listing maps classes to required packages and modules.

The `--generate-module-info` option produces `module-info` files for each analyzed module:

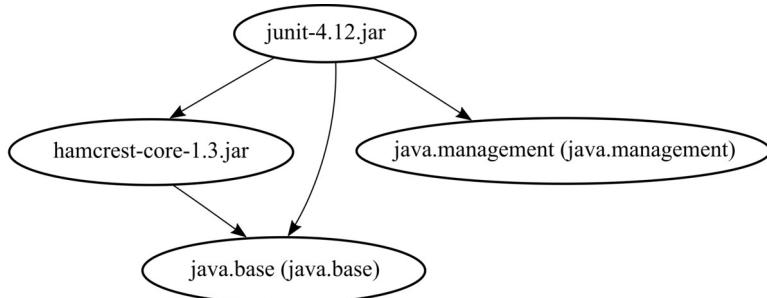
```
jdeps --generate-module-info /tmp/junit junit-4.12.jar
hamcrest-core-1.3.jar
```



Note: There is also an option to generate graphical output in the “dot” language for describing graphs. Assuming you have the dot tool installed, run these commands:

```
jdeps -s -dotoutput /tmp/junit junit-4.12.jar
hamcrest-core-1.3.jar
dot -Tpng /tmp/junit/summary.dot >
/tmp/junit/summary.png
```

Then you get this `summary.png` image:



You use the `jlink` tool to produce an application that executes without a separate Java runtime. The resulting image is much smaller than the entire JDK. You specify the modules that you want to have included and an output directory.

```
jlink --module-path
com.horstmann.greet.jar:ch12.sec05.jar:$JAVA_HOME/jmods \
--add-modules ch12.sec05 --output /tmp/hello
```

The output directory has a subdirectory `bin` with a `java` executable. If you run

```
bin/java -m ch12.sec05
```

the `main` method of the module's main class is invoked.

The point of `jlink` is that it bundles up the minimal set of modules that is required to run the application. You can list them all:

```
bin/java --list-modules
```

In this example, the output is

```
ch12.sec05
com.horstmann.greet
java.base@21
```

All modules are included in a *runtime image* file `lib/modules`. On my computer, that file is 29MB, whereas the runtime image of all JDK modules

takes up 133MB. The entire application takes up 55MB, a fraction of the size of the JDK.

This can be the basis of a useful tool for packaging an application. You would still need to produce file sets for multiple platforms and launch scripts for the application.



Note: You can inspect the runtime image with the `jimage` command. However, the format is internal to the JVM, and runtime images are not meant to be generated or used by other tools.

Finally, the `jmod` tool builds and inspects the module files that are included with the JDK. When you look into the `jmods` directory inside the JDK, you will find a file with extension `jmod` for each module. There is no longer a `rt.jar` file.

Like JAR files, these files contain class files. In addition, they can hold native code libraries, commands, header files, configuration files, and legal notices. The JMOD files use the ZIP format. You can inspect their contents with any ZIP tool.

Unlike JAR files, JMOD files are only useful for linking; that is, for producing runtime images. There is no need for you to produce JMOD files unless you also want to bundle binary files such as native code libraries with your modules.

Exercises

1. The “restricted keywords” `module`, `exports`, `requires`, `uses`, `to`, and so on, have specific meanings in module declarations. Can you use them as names for classes? Packages? Modules? In particular, can you make a module called `module`? Try creating a context where you can produce declarations such as the following:

```
requires requires;  
exports exports;  
opens to to opens;
```

How about a module `transitive`? Can you require it?

2. Try accessing `GreeterImpl` in the program in [Section 12.5](#) from the `HelloWorld` class. What happens? Is it a compile-time or a runtime error?
3. In the program in [Section 12.5](#), use `java.util.logging.Level` to make a `Greeter` return an empty string when the level is less than `Level.INFO`. What is the effect on the module descriptors?
4. What happens if you put the Apache CSV JAR onto the class path as an unnamed module and try accessing its packages from a module? What should you do instead?
5. Develop an example that demonstrates a compelling use for a `requires transitive` dependency on a module such as `java.sql`, `java.xml`, or `java.desktop`.
6. Develop examples for the two uses cases of `requires static`. Would you ever want to have `requires transitive static`?
7. In the program in [Section 12.13](#), what happens if the `provides` or `uses` statements are omitted? Why aren't these compile-time errors?
8. In the program in [Section 12.13](#), use a service provider factory; that is, a class with a public static method `provider()` that returns the service object.
9. Reorganize the program in [Section 12.7](#) so that the service interface and implementation are defined in separate modules.
10. Download the open source JFreeChart program and use `jdeps` to analyze the dependencies of the demo program and the JAR files in the `lib` subdirectory.
11. Turn the demo program of JFreeChart into a module and the JAR files in the `lib` subdirectory into automatic modules.
12. Run `jlink` to get a runtime image of the JFreeChart demo program.
13. Try running a Java 8 version of the JavaFX SceneBuilder program under a newer version of Java. What command-line flags do you need to start it? How did you find out?

Index

Symbols

- ! operator
 - in property files [7.5.1](#)
 - operator [1.4, 1.4.5](#)
- != operator [1.4, 1.4.5](#)
 - for wrapper classes [1.8.4](#)
- # (number sign)
 - flag (for output) [1.6.2](#)
 - in javadoc hyperlinks [2.8.6](#)
 - in option files [12.10](#)
 - in property files [7.5.1](#)
- \$ (dollar sign)
 - flag (for output) [1.6.2](#)
 - in regular expressions [9.4.1, 9.4.6, 9.4.7](#)
 - in variable names [1.3.2](#)
- % (percent sign)
 - conversion character [1.6.2](#)
 - operator [1.4, 1.4.2](#)
- %o% [5.3.5](#)
- %o= [1.4](#)
- %og [5.3.5](#)
- %oh [5.3.5](#)
- %ot [5.3.5](#)
- %ou [5.3.5](#)
- & (ampersand) [1.4, 1.4.5](#)
- && operator
 - in regular expressions [9.4.1](#)
 - operator [1.4, 1.4.5](#)
- &= [1.4](#)
- > (right angle bracket)
 - in shell syntax [1.6.1](#)
 - operator [1.4.5](#)
- >>=, >>> operators [1.4](#)
- >=, >>, >>> operators [1.4, 1.4.5](#)
- < (left angle bracket)
 - flag (for output) [1.6.2](#)
 - in shell syntax [1.6.1](#)
 - operator [1.4.5](#)
- <>
 - for constructors of generic classes [6.1](#)
- << operator [1.4, 1.4.5](#)
- <=< [1.4](#)
- <...>
 - for element types, in array lists [1.8.3](#)
 - for type parameters [3.3.1, 6.1](#)
 - in javadoc hyperlinks [2.8.6](#)
 - in regular expressions [9.4.1](#)
- <= operator [1.4, 1.4.5](#)
- "" [1.5.3, 1.5.4, 4.3.1](#)
 - """ [1.5.7](#)
 - ..."
 - for strings [1.1.3](#)
 - in javadoc hyperlinks [2.8.6](#)
 - in text blocks [1.5.7](#)
 - '...'
 - for character literals [1.2.3](#)
- ((left parenthesis) [1.6.2](#)
- (...)
 - empty, for anonymous classes [3.9.2](#)
 - for casts [1.4.4, 3.1.5](#)
 - in regular expressions [9.4.1, 9.4.4](#)
 - operator [1.4](#)
- * (asterisk)
 - for annotation processors [11.5.1](#)
 - in documentation comments [2.8.2](#)
 - in regular expressions [9.4.1, 9.4.4](#)
 - operator [1.4, 1.4.2](#)
- wildcard
 - in class path [2.6.3](#)
 - in imported classes [2.6.5](#)
- *= [1.4](#)
- + (plus sign)
 - flag (for output) [1.6.2](#)
 - in regular expressions [9.4.1](#)
 - operator [1.4, 1.4.2](#)
 - for strings [1.5.1, 1.5.4, 4.3.1](#)
- ++ operator [1.4, 1.4.2](#)
- += [1.4](#)
- , (comma)
 - flag (for output) [1.6.2](#)
 - normalizing [9.4.6](#)
 - trailing, in arrays [1.8.2](#)

- (minus sign)
 - flag (for output) [1.6.2](#)
 - in regular expressions [9.4.1](#)
 - operator [1.4, 1.4.2](#)
 - > operator [3.4.1, 3.5.1](#)
- operator
 - in command-line options [2.6.2](#)
 - operator [1.4, 1.4.2](#)
- = [1.4, 1.4.1](#)
- . (period) [9.2.1](#)
 - in method calls [1.1.3](#)
 - in package names [1.1.2, 2.6.1](#)
 - in regular expressions [9.4.1, 9.4.7](#)
 - operator [1.4](#)
- ... (ellipsis) [1.9.3](#)
- / (slash)
 - file separator (Unix) [7.5.1, 9.2.1](#)
 - in javac path segments [1.1.2](#)
 - operator [1.4, 1.4.2](#)
 - root component [9.2.1](#)
- /**...*/ [2.8, 2.8.1](#)
- // /*...*/ comments [1.1.1](#)
- /= [1.4](#)
- 0
 - as default value [2.3.4, 2.3.7](#)
 - flag (for output) [1.6.2](#)
 - prefix (for octal literals) [1.2.1](#)
- 0b [1.2.1](#)
- 0x [1.2.1, 1.6.2](#)
- 0xFEFF [9.1.4](#)
- : (colon)
 - in assertions [5.2.1](#)
 - in switch statement [1.7.2](#)
 - path separator (Unix) [2.6.3, 7.5.1](#)
- switch [1.7.2](#)
 - switch statement [1.7.2, 4.5.4](#)
- :: (C++ operator) [3.5.1, 4.1.8](#)
- ; (semicolon)
 - path separator (Windows) [2.6.3, 7.5.1](#)
- = operator [1.4, 1.4.1](#)
- == operator [1.4, 1.4.5, 4.3.2](#)
 - for class objects [4.6.1](#)
 - for enumerations [4.4.1](#)
 - for strings [1.5.3](#)
 - for wrapper classes [1.8.4](#)
- ? (question mark)
 - in regular expressions [9.4.1](#)
 - replacement character [9.1.6](#)
- wildcard, for types [6.4.1, 6.4.5, 6.7.1](#)
- ? : [1.4, 1.4.5](#)
- @ (at sign)
 - in java command [12.10](#)
 - in javadoc comments [2.8.1](#)
- [...]
 - for arrays [1.8.1, 1.8.9](#)
 - in regular expressions [9.4.1](#)
 - operator [1.4](#)
- [I] [4.3.1, 4.6.1](#)
- [L] [4.6.1](#)
- \ (backslash)
 - character literal [1.2.3](#)
 - file separator (Windows) [7.5.1, 9.2.1](#)
 - in option files [12.10](#)
 - in regular expressions [9.4.1, 9.4.6](#)
 - in text blocks [1.5.7](#)
- \0 [9.4.1](#)
- \a, \A, in regular expressions [9.4.1](#)
- \b (backslash character literal) [1.2.3](#)
- \b, \B, in regular expressions [9.4.1](#)
- \c [9.4.1](#)
- \d, \D, in regular expressions [9.4.1](#)
- \e, \E, in regular expressions [9.4.1](#)
- \f (form feed character literal) [9.4.1](#)
- \G [9.4.1](#)
- \h, \H, in regular expressions [9.4.1](#)
- \k [9.4.1](#)
- \n (newline character literal)
 - for character literals [1.2.3](#)
 - in property files [7.5.1](#)
 - in regular expressions [9.4.1, 9.4.7](#)
- \p, \P, in regular expressions [9.4.1](#)
- \Q [9.4.1](#)
- \r (carriage return character literal)
 - for character literals [1.2.3](#)
 - in property files [7.5.1](#)
- \r, \R, in regular expressions [9.4.1](#)
- \s, \S, in regular expressions [9.4.1](#)
- \t (tab character literal)
 - in regular expressions [9.4.1](#)
 - tab, for character literals [1.2.3](#)
- \u (Unicode character literal)
 - for character literals [1.2.3](#)
 - in regular expressions [9.4.1](#)
- \v, \V, in regular expressions [9.4.1](#)
- \w, \W, in regular expressions [9.4.1](#)
- \x [9.4.1](#)

\z, \Z, in regular expressions [9.4.1](#)
^ (caret)
 for function parameters [3.4.1](#)
 in regular expressions [9.4.1, 9.4.7](#)
 operator [1.4, 1.4.5](#)
^= [1.4](#)
_ (underscore)
 in number literals [1.2.1](#)
 in variable names [1.3.2, 2.2.5](#)
{...}
 in annotation elements [11.1.1](#)
 in lambda expressions [3.4.1](#)
 in regular expressions [9.4.1, 9.4.6](#)
 with arrays [1.8.2](#)
{...} [4.1.7](#)
| operator
 in regular expressions [9.4.1](#)
 operator [1.4, 1.4.5](#)
|= [1.4](#)
|| operator [1.4, 1.4.5](#)
~ operator [1.4, 1.4.5](#)

A

a, A conversion characters [1.6.2](#)
abstract keyword [3.1.3, 4.2.2](#)
abstract classes [4.2.2](#)
abstract methods [3.4.2](#)
AbstractCollection class [3.2.2](#)
AbstractMethodError class [3.2.2](#)
AbstractProcessor class [11.5.1](#)
accept [3.6.2](#)
acceptEither method [10.5.2](#)
AccessibleObject class
 setAccessible method [4.7.2, 4.7.4](#)
 trySetAccessible method [4.7.2](#)
accessors [2.1.1](#)
accumulate [10.6](#)
accumulateAndGet method [10.6](#)
accumulator functions [8.12](#)
ActionListener interface [3.3.4](#)
add method
 of ArrayDeque [7.5.4](#)
 of ArrayList [1.8.3, 2.1.1](#)
 of BlockingQueue [10.3.2](#)
 of Collection [7.1](#)
 of List [7.1](#)

of ListIterator [7.2](#)
of LongAdder [10.6](#)
addAll method
 of Collection [6.4.3, 7.1](#)
 of Collections [7.1](#)
 of List [7.1](#)
addExact method [1.4.3](#)
addition [1.4.2](#)
 identity for [8.12](#)
addSuppressed method [5.1.5](#)
aggregators [12.11](#)
allMatch method [8.6](#)
allOf method
 of CompletableFuture [10.5.2](#)
 of EnumSet [7.5.3](#)
allProcesses method [10.9.3](#)
and, andNot methods (BitSet) [7.5.2](#)
and, andThen methods (functional
 interfaces) [3.6.2](#)
Android [3.3.4, 10.5.3](#)
AnnotatedConstruct interface [11.5.2](#)
AnnotatedElement interface [11.4](#)
Annotation interface
 extending [11.2](#)
annotation interfaces [11.2](#)
annotation processors [11.5.1](#)
annotations
 accessing [11.2, 12.11](#)
 applicability of [11.3](#)
 container [11.3.2, 11.4](#)
 declaration [11.1.3](#)
 documented [11.3, 11.3.2](#)
 generating source code with [11.5.3](#)
 inherited [11.3, 11.3.2, 11.4](#)
 key/value pairs in [11.1.1, 11.2](#)
 meta [11.2, 11.3.2](#)
 modifiers and [11.1.4](#)
 multiple [11.1.2](#)
 processing
 at runtime [11.4](#)
 source-level [11.5](#)
 repeatable [11.1.2, 11.3, 11.3.2, 11.4](#)
 standard [11.3](#)
 type use [11.1.4](#)
anonymous classes [3.9.2](#)
anyMatch method [8.6](#)
anyOf method [10.5.2](#)
Apache Commons CSV [12.8](#)

API documentation [1.5.5](#)
generating [2.8](#)

Applet [4.6.2](#)

apply, applyAsXXX methods (functional interfaces) [3.6.2](#)

applyToEither method [10.5.2](#)

arithmetic operations [1.4](#)

ArrayList class [4.7.6](#)

array lists [1.8.3](#)
anonymous [4.1.7](#)
checking for nulls [6.4.4](#)
constructing [1.8.3](#)
converting between [6.4.1](#)
copying [1.8.6](#)
elements of [1.8.3, 1.8.5](#)
filling [1.8.7](#)
instantiating with type variables [6.6.3](#)
size of [1.8.3](#)
sorting [1.8.7](#)
variables of [1.8.3](#)

array variables
assigning values to [1.8.2](#)
copying [1.8.6](#)
declaring [1.8.1](#)
initializing [1.8.1](#)

ArrayBlockingQueue class [10.3.2](#)

ArrayDeque class [7.5.4](#)

ArrayIndexOutOfBoundsException
class [1.8.1](#)

ArrayList class [1.8.3, 7.1](#)
add method [1.8.3, 2.1.1](#)
clone method [4.3.4](#)
forEach method [3.5.1](#)
get, remove methods [1.8.3](#)
removeIf method [3.4.2](#)
set, size methods [1.8.3](#)

arrays [1.8.1, 1.8.2](#)
accessing nonexisting elements in [1.8.1](#)
allocating [6.6.3](#)
annotating [11.1.4](#)
asList method [7.6.1](#)
casting [4.7.6](#)
checking [4.7.6](#)
comparing [4.3.2](#)
computing values of [10.4.2](#)
constructing [1.8.1, 1.8.2](#)
constructor references with [3.5.2](#)
converting

to a reference of type Object [4.3](#)
to/from streams [8.8, 8.14, 10.4.2](#)

copying [1.8.6](#)
copyOf method [1.8.6, 4.7.6](#)
covariant [6.4](#)
deepToString method [4.3.1](#)
equals method [4.3.2](#)
fill method [1.8.7](#)
filling [1.8.2, 1.8.7](#)
generating Class objects for [4.6.1](#)
growing [4.7.6](#)
hash codes of [4.3.3](#)
hashCode method [4.3.3](#)
length of [1.8.1, 1.8.2, 3.7.2](#)
multidimensional [1.8.9, 4.3.1](#)
of bytes [9.1.1](#)
of generic types [3.5.2, 6.6.4](#)
of objects [1.8.2, 10.4.2](#)
of primitive types [10.4.2](#)
of strings [9.4.5](#)
parallelXXX methods [1.8.7, 10.4.2](#)
passing into methods [1.9.2](#)
printing [1.8.7, 1.8.9, 4.3.1](#)
serializable [9.5.1](#)
setAll method [3.6.1](#)
sort method [1.8.7, 3.3.1, 3.3.2, 3.4.2, 3.5.1](#)
sorting [1.8.7, 3.3.1, 10.4.2](#)
stream method [8.2, 8.13](#)
superclass assignment in [4.1.5](#)
toString method [1.8.7, 4.3.1](#)
using class literals with [4.6.1](#)

ArrayStoreException class [4.1.5, 6.4, 6.6.4](#)

ASCII [1.5.6, 9.1.4](#)

ASM tool [11.5.3](#)

asMatchPredicate, asPredicate methods
(Pattern) [9.4.2](#)

assert keyword [5.2.1](#)

AssertionError class [5.2.1](#)

assertions [5.2](#)
checking [11.1.4](#)
enabling/disabling [5.2.2](#)

assignment operators [1.4.1](#)

associative operations [8.12](#)

asSubclass method [6.7.1](#)

asynchronous computations [10.5](#)

AsyncTask [10.5.3](#)

atomic operations [10.2.2, 10.3.1, 10.6, 10.7.3](#)
performance and [10.6](#)

Atomic`Xxx` classes [10.6](#)
@author annotation [2.8.2](#), [2.8.8](#)
autoboxing [1.8.4](#), [3.6.3](#)
AutoCloseable interface [5.1.5](#), [6.3](#)
 close method [5.1.5](#)
automatic modules [12.8](#), [12.9](#)
availableCharsets method [9.1.4](#)
availableProcessors method [10.1.1](#)
average method (`XxxStream`) [8.13](#)

B

b, B conversion characters [1.6.2](#)
Base classes. See Superclasses
BasicFileAttributes interface [9.2.4](#)
BeanInfo interface [4.7.5](#)
BiConsumer interface [3.6.2](#)
BiFunction interface [3.6.2](#), [3.6.3](#)
big-endian format [9.1.4](#), [9.1.8](#), [9.1.10](#)
BigDecimal class [1.2.2](#), [1.4.6](#), [9.5.3](#)
BigInteger class [1.2.1](#), [1.4.6](#)
binary data, reading/writing [9.1.8](#)
binary numbers [1.2.1](#), [1.2.2](#)
binary trees [7.3](#)
BinaryOperator interface [3.6.2](#)
binarySearch method [7.1](#)
BiPredicate interface [3.6.2](#)
BitSet class [7.5.2](#)
 collecting streams into [8.12](#)
 methods of [7.5.2](#)
bitwise operators [1.4.5](#)
block statement, labeled [1.7.4](#)
blocking queues [10.3.2](#)
BlockingQueue interface [10.3.2](#)
boolean type [1.2.4](#), [1.8.4](#)
 default value of [2.3.4](#), [2.3.7](#)
 formatting for output [1.6.2](#)
 reading/writing [9.1.8](#)
 streams of [8.13](#)
BooleanSupplier interface [3.6.2](#)
bootstrap class loader [4.6.3](#)
boxed [8.13](#)
branches [1.7.1](#)
break keyword [1.7.2](#), [1.7.4](#)
bridge methods [6.5.3](#)
 clashes of [6.6.6](#)
BufferedReader class [9.1.5](#)

build [9.3.2](#)
bulk operations [10.3.1](#)
byte type [1.2.1](#), [1.8.4](#), [9.1.2](#)
 MAX_VALUE, MIN_VALUE constants [1.2.1](#)
streams of [8.13](#)
 toUnsignedInt method [1.2.1](#)
 type conversions of [1.4.4](#)
byte codes [1.1.2](#)
byte order mark [9.1.4](#)
ByteArray`Xxx`Stream classes [9.1.1](#)
ByteBuffer class [9.1.10](#)
bytes
 arrays of [9.1.1](#)
 converting to strings [9.1.4](#)
 reading [9.1.2](#)
 skipping [9.1.2](#)
 writing [9.1.3](#)

C

C# programming language, type parameters
 in [6.4.3](#)
c, C conversion characters [1.6.2](#)
C/C++
 #include directive in [2.6.5](#)
 allocating memory in [10.2.3](#)
 integer types in [1.2.1](#)
 pointers in [2.1.2](#)
C:\ [9.2.1](#)
CachedRowSetImpl [12.10](#)
calculators [4.4.3](#)
calendars [2.1](#)
call by reference [2.2.6](#)
Callable interface [3.3.3](#)
 call method [10.1.2](#)
callbacks [3.3.4](#), [10.5.2](#)
 registering [10.5.1](#)
camel case [1.3.2](#)
cancel method
 of CompletableFuture [10.5.1](#)
 of Future [10.1.2](#)
cancellation requests [10.1.3](#)
CancellationException class [10.5.1](#)
cardinality [7.5.2](#)
carriage return [1.2.3](#)
case keyword [1.7.2](#)
cast [6.7.1](#)

cast operator [1.4.4](#)
casts [1.4.4](#), [3.1.5](#), [4.1.6](#)
 annotating [11.1.4](#)
 generic types and [6.6.2](#)
 inserting [6.5.2](#)
catch keyword [5.1.4](#)
 annotating parameters of [11.1.3](#)
 in try-with-resources [5.1.5](#)
 no type variables in [6.6.7](#)
ceiling [7.3](#)
Channel interface [3.1.7](#)
channels [9.1.10](#)
char type [1.2.3](#)
 streams of [8.13](#)
 type conversions of [1.4.4](#)
Character class [1.8.4](#)
character classes [9.4.1](#)
character encodings [9.1.4](#)
 native [9.1.4](#)
 partial [9.1.4](#), [9.1.6](#)
character literals [1.2.3](#)
characters [9.1](#)
 formatting for output [1.6.2](#)
 reading/writing [9.1.8](#)
charAt method [1.5.6](#)
CharSequence interface [1.5.5](#), [8.2](#)
 chars, codePoints methods [8.13](#)
Charset class
 availableCharsets method [9.1.4](#)
 defaultCharset method [9.1.4](#)
 forName method [9.1.4](#)
checked exceptions [5.1.2](#), [5.1.3](#)
 combining in a superclass [5.1.3](#)
 declaring [5.1.3](#)
 documenting [5.1.3](#)
 generic types and [6.6.7](#)
 in lambda expressions [5.1.3](#)
 no-argument constructors and [4.7.4](#)
 not allowed in a method [5.1.7](#)
 rethrowing [5.1.7](#)
checked views [6.6.2](#), [7.6.3](#)
checkedXXX methods (Collections) [7.1](#),
 [7.6.3](#)
Checker Framework [11.1.4](#)
checkIndex method [5.1.9](#)
Child classes. See Subclasses
Church, Alonzo [3.4.1](#)
Class class [4.6.1](#), [6.7.2](#)
asSubclass, cast methods [6.7.1](#)
comparing objects of [4.6.1](#)
forName method [4.6.1](#), [4.6.3](#), [4.6.4](#), [5.1.2](#),
[5.1.8](#)
generic [6.7.1](#)
getCanonicalName method [4.6.1](#)
getClassLoader method [4.6.1](#)
getComponentType method [4.6.1](#), [4.7.6](#)
getConstructor(s) methods [4.6.1](#), [4.7.1](#),
[4.7.4](#), [6.7.1](#)
getDeclaredConstructor(s)
methods [4.6.1](#), [4.7.1](#), [6.7.1](#)
getDeclaredField(s) methods [4.6.1](#)
getDeclaredMethod(s) methods [4.6.1](#),
[4.7.3](#)
getDeclaringClass method [4.6.1](#)
getEnclosingXxx methods [4.6.1](#)
getEnumConstants method [6.7.1](#)
getField(s) methods [4.6.1](#), [4.7.1](#)
getInterfaces method [4.6.1](#)
getMethod(s) methods [4.6.1](#), [4.7.1](#), [4.7.3](#)
getModifiers method [4.6.1](#)
getName method [4.6.1](#)
getPackage method [4.6.1](#)
getPackageName method [4.6.1](#)
getPermittedSubclasses method [4.6.1](#)
getRecordComponents method [4.6.1](#)
getResource method [4.6.2](#)
getResourceAsStream method [4.6.1](#), [4.6.2](#)
getSimpleName method [4.6.1](#)
getSuperclass method [4.6.1](#), [6.7.1](#)
getTypeName method [4.6.1](#)
getTypeParameters method [6.7.2](#)
isXXX methods [4.6.1](#), [4.7.6](#)
newInstance method [4.7.4](#), [6.7.1](#)
toGenericString, toString methods [4.6.1](#)
class declarations
 annotations in [11.1.3](#), [11.3.2](#)
 initialization blocks in [2.3.5](#)
class files [1.1.2](#), [4.6.3](#)
 paths of [2.6.1](#)
 processing annotations in [11.5.3](#)
class literals [4.6.1](#)
 no annotations for [11.1.4](#)
 no type variables in [6.6.2](#)
class loaders [4.6.3](#), [4.6.4](#)
class objects [4.6.1](#)
class path [2.6.3](#), [12.1](#)

ClassCastException class [3.1.5](#), [6.6.2](#)
classes [1.1.1](#), [2.1](#)
 abstract [3.1.3](#), [3.2.3](#), [4.2.2](#)
 accessing from a different module [12.11](#)
 adding to packages [2.6.4](#)
 anonymous [3.9.2](#)
 companion [3.2.1](#), [3.2.2](#)
 constructing objects of [1.3.1](#)
 deprecated [2.8.5](#), [11.3](#), [11.3.1](#)
 deserialization of [9.5.7](#)
 documentation comments for [2.8.1](#), [2.8.2](#)
 encapsulation of [12](#), [12.1](#)
 evolving [9.5.6](#)
 extending [4.1](#)
 fields of [4](#)
 final [4.2.1](#)
 generic [1.8.3](#)
 immutable [1.5.5](#), [10.2.4](#)
 implementing [2.2](#), [4.3.4](#)
 importing [2.6.5](#)
 inner [2.7.2](#)
 instances of [1.1.3](#), [2.2.1](#), [2.5.5](#)
 loading [4.7.1](#)
 local [3.9.1](#)
 members of [4](#)
 enumerating [4.4.4](#), [4.7.1](#)
 naming [1.3.2](#), [2.6](#), [4.6.1](#)
 nested [2.7](#), [11.1.4](#)
 not known at compile time [4.6.1](#), [4.7.7](#)
 protected [4.2.3](#)
 public [2.6.4](#), [12.5](#)
 sealed [4.2.4](#)
 serializable [9.5.3](#)
 static initialization of [4.6.3](#)
 static methods of [2.5.4](#)
 system [5.2.2](#)
 testing [2.6.4](#)
 utility [2.6.4](#), [4.6.4](#)
 wrapper [1.8.4](#)
classes win rule [4.3.3](#)
classifier functions [8.10](#)
ClassLoader class
 defineClass method [12.10](#)
 findClass, loadClass methods [4.6.3](#)
 setXxxAssertionStatus methods [5.2.2](#)
classloader inversion [4.6.4](#)
ClassNotFoundException class [5.1.2](#)
CLASSPATH [2.6.3](#)

clear method
 of BitSet [7.5.2](#)
 of Collection [7.1](#)
 of Map [7.4](#)
clone method
 of ArrayList [4.3.4](#)
 of Enum [4.4.1](#)
 of Message [4.3.4](#)
 of Object [4.2.3](#), [4.3](#), [4.3.4](#), [4.7.3](#)
 protected [4.3.4](#)
Cloneable interface [4.3.4](#)
CloneNotSupportedException class [4.3.4](#),
 [4.4.1](#)
cloning [4.3.4](#)
close method
 of AutoCloseable, Closeable [5.1.5](#)
 of PrintWriter [5.1.5](#)
 throwing exceptions [5.1.5](#)
Closeable interface [3.1.7](#)
 close method [5.1.5](#)
closures [3.7.2](#)
code [2.8.1](#)
code generator tools [11.3.1](#)
code points [1.5.6](#), [8.3](#), [9.1.4](#)
code units [1.2.3](#), [1.5.6](#), [8.13](#)
 in regular expressions [9.4.1](#)
codePoints method
 of CharSequence [8.13](#)
Collator class [1.5.3](#)
collect [8.8](#), [8.12](#)
collectingAndThen method [8.11](#)
Collection interface [3.2.1](#), [3.2.2](#), [7.1](#)
 add method [7.1](#)
 addAll method [6.4.3](#), [7.1](#)
 clear, contains, containsAll
 methods [7.1](#)
 isEmpty method [7.1](#)
 iterator method [7.1](#)
 parallelStream method [7.1](#), [8.1](#), [8.14](#),
 [10.4.1](#)
 remove, removeXxx, retainAll
 methods [7.1](#)
 size method [7.1](#)
 spliterator method [7.1](#)
 stream method [7.1](#), [8.1](#)
 toArray method [7.1](#)
Collections class [3.2.1](#), [7](#), [7.1](#), [7.6.3](#)
 addAll method [7.1](#)

binarySearch method [7.1](#)
branching [8.11](#)
copy method [7.1](#)
disjoint method [7.1](#)
fill method [1.8.7, 7.1](#)
frequency method [7.1](#)
generic [7.6.3](#)
given elements of [7.6.1](#)
indexOfSubList, lastIndexOfSubList
methods [7.1](#)
iterating over elements of [8.1](#)
mutable [7.6.1](#)
nCopies method [7.1](#)
processing [7.1](#)
replaceAll method [7.1](#)
reverse method [1.8.7, 7.1](#)
rotate method [7.1](#)
serializable [9.5.1](#)
shuffle method [1.8.7, 7.1](#)
sort method [1.8.7, 6.4.3, 6.7.2, 7.1](#)
swap method [7.1](#)
synchronizedXxx methods [7.1](#)
threadsafe [10.3.3](#)
unmodifiable views of [7.6.1, 7.6.3](#)
unmodifiableXxx methods [7.1](#)
vs. streams [8.1](#)

Collector interface [8.8](#)
Collectors class [2.6.6](#)
 counting method [8.11](#)
 filtering method [8.11](#)
 flatMapping method [8.11](#)
 groupingBy method [8.10, 8.11](#)
 groupingByConcurrent method [8.10, 8.14](#)
 joining method [8.8](#)
 mapping method [8.11](#)
 maxBy, minBy methods [8.11](#)
 partitioningBy method [8.10, 8.11](#)
 reducing method [8.11](#)
 summarizingXxx methods [8.8, 8.11](#)
 summingXxx methods [8.11](#)
 teeing method [8.11](#)
 toCollection method [8.8](#)
 toConcurrentMap method [8.9](#)
 toMap method [8.9](#)
 toSet method [8.8, 8.11](#)

Collectors class
 collectingAndThen method [8.11](#)
 mapping method [8.11](#)

command-line arguments [1.8.8](#)
comments [1.1.1](#)
 documentation [2.8](#)
commonPool method [8.14, 10.5.1](#)
companion classes [3.2.1, 3.2.2](#)
Comparable interface [3.3.1, 4.4.1, 6.4.3, 7.3](#)
 compareTo method [3.3.1](#)
 priority queues with [7.5.4](#)
 streams of [8.5](#)

Comparator interface [2.6.6, 3.3.2, 3.8, 7.3](#)
 comparing, comparingXxx methods [3.8.3](#)
 naturalOrder method [3.8.3](#)
 nullsFirst, nullsLast methods [3.8.3](#)
 priority queues with [7.5.4](#)
 reversed method [3.8.2](#)
 reverseOrder method [3.8.3](#)
 streams of [8.5](#)
 thenComparing method [3.8.3](#)

compare [3.3.1](#)
compareTo method
 of Enum [4.4.1](#)
 of String [1.5.3, 3.3.1](#)

compareToIgnoreCase method [3.5.1](#)
compareUnsigned method [1.4.3](#)
compatibility, drawbacks of [6.5](#)
compilation [1.1.2](#)
compile [9.4.2, 9.4.7](#)
compile-time errors [1.3.3, 3.1.6](#)
compiler
 instruction reordering in [10.2.1](#)

completable futures [10.5.1, 10.5.2](#)
 combining [10.5.2](#)
 composing [10.5.2](#)
 interrupting [10.5.1](#)

CompletableFuture class [10.5.1, 10.5.2](#)
 acceptEither method [10.5.2](#)
 allOf, anyOf methods [10.5.2](#)
 applyToEither method [10.5.2](#)
 cancel method [10.5.1](#)
 complete, completeExceptionally
 methods [10.5.1](#)
 completeOnTimeout method [10.5.2](#)
 exceptionally method [10.5.2](#)
 exceptionallyCompose method [10.5.2](#)
 handle method [10.5.2](#)
 isDone method [10.5.1](#)
 orTimeout method [10.5.2](#)
 runAfterXxx methods [10.5.2](#)

supplyAsync method [10.5.1](#)
thenAccept method [10.5.1](#), [10.5.2](#)
thenAcceptBoth method [10.5.2](#)
thenApply, thenApplyAsync
methods [10.5.2](#)
thenCombine method [10.5.2](#)
thenCompose method [10.5.2](#)
thenRun method [10.5.2](#)
whenComplete method [10.5.1](#), [10.5.2](#)
CompletionStage interface [10.5.2](#)
compose [3.6.2](#)
computations
asynchronous [10.5](#)
mutator [2.1.1](#)
precision of [1.2.2](#)
compute method
of ConcurrentHashMap [10.3.1](#)
of Map [7.4](#)
computeIfXxx methods
of ConcurrentHashMap [10.3.1](#)
of Map [7.4](#)
concat [8.4](#)
concatenation [1.5.1](#)
objects with strings [4.3.1](#)
concurrent programming [10](#), [10.9.2](#)
access errors in [3.7.2](#)
strategies for [10.2.3](#)
ConcurrentHashMap class [10.3.1](#), [10.7.3](#)
compute method [10.3.1](#)
computeIfXxx methods [10.3.1](#)
forEachXxx methods [10.3.1](#)
keySet method [10.3.3](#)
merge method [10.3.1](#)
newKeySet method [10.3.3](#)
no null values in [7.4](#)
putIfAbsent method [10.3.1](#)
reduceXxx, searchXxx methods [10.3.1](#)
threadsafe [10.7.1](#)
ConcurrentModificationException
class [7.2](#), [10.3.3](#)
ConcurrentSkipListXxx classes [10.3.3](#)
conditional operator [1.4.5](#)
configuration files
editing [5.3.4](#)
locating [4.6.2](#)
resolving paths for [9.2.1](#)
confinement [10.2.3](#)
connect [9.3.1](#)
Console class [1.6.1](#)
ConsoleHandler class [5.3.5](#), [5.3.6](#)
constants [1.3.4](#), [3.1.9](#)
naming [1.3.4](#)
static [2.5.2](#)
using in another class [1.3.4](#)
Constructor class [4.7.1](#)
getModifiers, getName methods [4.7.1](#)
newInstance method [4.7.4](#)
constructor references [3.5.2](#)
annotating [11.1.4](#)
constructors [2.3](#)
abstract classes and [4.2.2](#)
annotating [6.6.4](#), [11.1.3](#), [11.1.4](#)
canonical, compact, custom [2.4.2](#)
documentation comments for [2.8.1](#)
executing [2.3.1](#)
for subclasses [4.1.4](#)
implementing [2.3.1](#)
invoking another constructor from [2.3.3](#)
no-argument [2.3.7](#), [4.1.4](#), [4.7.4](#)
overloading [2.3.2](#)
public [2.3.1](#), [4.7.1](#)
references in [10.2.4](#)
Consumer interface [3.6.2](#)
contains method (String) [1.5.5](#)
contains, containsAll methods
(Collection) [7.1](#)
containsXxx methods (Map) [7.4](#)
context class loaders [4.6.4](#)
continue keyword [1.7.4](#)
control flow [1.7](#)
conversion characters [1.6.2](#)
cooperative cancellation [10.1.3](#)
Cooperative scheduling [10.1.1](#)
copy method
of Collections [7.1](#)
of Files [9.1.3](#), [9.2.3](#), [9.2.5](#)
copyOf method [1.8.6](#), [4.7.6](#)
CopyOnWriteArrayXxx classes [10.3.3](#)
CORBA [12.1](#)
count [8.1](#), [8.6](#)
counters
atomic [10.6](#)
de/incrementing [5.1.6](#)
counting [8.11](#)
country codes [8.10](#)
covariance [6.4](#)

`createDirectory`, `createDirectories`,
 `createFile` methods (`Files`) [9.2.2](#)
`createInstance` method [4.6.4](#)
`createTempXxx` methods (`Files`) [9.2.2](#)
critical sections [10.2.2](#), [10.7.1](#), [10.8.1](#)
current method
 of `ProcessHandle` [10.9.3](#)
 of `ThreadLocalRandom` [10.8.2](#)

D

D [1.2.2](#)
 conversion character [1.6.2](#)
 daemon threads [10.8.3](#)
Databases [11](#)
 persisting objects in [12.6](#)
DataInput/Output interfaces [9.1.8](#)
 `read/writeXxx` methods [9.1.8](#), [9.1.9](#)
DataXxxStream classes [9.1.8](#)
date classes
 immutability of [10.2.4](#)
DayOfWeek enumeration [2.1](#)
deadlocks [10.2.2](#), [10.7.1](#), [10.7.3](#), [10.8.1](#)
debugging
 messages for [5.1.1](#)
 overriding methods for [4.2.1](#)
 primary arrays for [1.8.7](#)
 streams [8.5](#)
 threads [10.8.3](#)
 using anonymous subclasses for [4.1.7](#)
 with assertions [5.2](#)
DecimalFormat class [2.5.5](#)
declaration-site variance [6.4.3](#)
decomposition
 of classes [1.9](#)
decrement operator [1.4.2](#)
decrementExact method [1.4.3](#)
deep copies [4.3.4](#)
deepToString method [4.3.1](#)
default keyword [3.2.2](#), [11.2](#)
default label (in switch) [1.7.2](#)
default methods [3.2.2](#), [3.2.3](#)
 conflicts of [3.2.3](#), [4.2.5](#)
 in interfaces [4.3.3](#)
defaultCharset method [9.1.4](#)
defaultReadObject method [9.5.3](#), [9.5.6](#)
defaultWriteObject method [9.5.3](#)

defensive programming [5.2](#)
deferred execution [3.6.1](#)
defineClass method [12.10](#)
delete, deleteIfExists methods
 (`Files`) [9.2.3](#)
delimiters, for scanners [9.1.5](#)
@deprecated annotation [2.8.5](#), [11.3](#), [11.3.1](#)
Deque interface [7.1](#), [7.5.4](#)
Derived classes. See Subclasses
destroy, destroyForcibly methods
 of `Process` [10.9.2](#)
 of `ProcessHandle` [10.9.3](#)
diamond syntax
 for array lists [1.8.3](#)
 for constructors of generic classes [6.1](#)
directories [9.2.1](#)
 checking for existence [9.2.2](#), [9.2.3](#)
 creating [9.2.2](#), [9.2.3](#)
 deleting [9.2.3](#), [9.2.4](#)
 moving [9.2.3](#)
 temporary [9.2.2](#)
 user [9.2.1](#)
 visiting [9.2.4](#)
 working [10.9.1](#)
directory [10.9.1](#)
disjoint [7.1](#)
distinct [8.5](#), [8.14](#)
divideUnsigned method [1.4.3](#)
division [1.4.2](#)
do [1.7.3](#)
doc-files [2.8.1](#)
documentation comments [2.8](#)
@Documented annotation [11.3](#), [11.3.2](#)
domain names
 for modules [12.2](#)
 for packages [2.6.1](#)
dot notation [1.1.3](#), [1.3.4](#)
double type [1.2.2](#), [1.8.4](#)
 atomic operations on [10.6](#)
 compare method [3.3.1](#)
 equals method [4.3.2](#)
 functional interfaces for [3.6.2](#)
 `isFinite`, `isInfinite` methods [1.2.2](#)
 `Nan`, `NEGATIVE_INFINITY`,
 `POSITIVE_INFINITY` values [1.2.2](#)
parseDouble method [1.5.4](#)
streams of [8.13](#)
toString method [1.5.4](#)

type conversions of [1.4.4](#)
double brace initialization [4.1.7](#)
`DoubleAccumulator`, `DoubleAdder`
 classes [10.6](#)
`DoubleConsumer`, `DoubleXxxOperator`,
 `DoublePredicate`, `DoubleSupplier`,
 `DoubleToXxxFunction` interfaces [3.6.2](#)
`DoubleFunction` interface [3.6.2](#), [6.6.1](#)
doubles [8.13](#)
`DoubleStream` interface [8.13](#)
`DoubleSummaryStatistics` class [8.8](#), [8.13](#)
downstream collectors [8.11](#), [8.14](#)
`Driver.parentLogger` method [12.11](#)
`dropWhile` method [8.4](#)
dynamic method lookup [4.1.5](#), [6.5.3](#)

E

E [1.4.3](#)
e, **E**
 conversion characters [1.6.2](#)
Eclipse [1.1.2](#)
effectively final variables [3.7.2](#)
efficiency, and `final` modifier [4.2.1](#)
`element` [10.3.2](#), [11.5.2](#)
`Elements` [11.1.1](#), [11.2](#)
`else` keyword [1.7.1](#)
`em` [2.8.1](#)
empty method
 of `Optional` [8.7.5](#)
 of `Stream` [8.2](#)
empty string [1.5.3](#), [4.3.1](#)
 concatenating [1.5.4](#)
encapsulation [2.1](#), [12](#), [12.1](#), [12.6](#)
Encodings. See Character encodings
end [9.4.3](#), [9.4.4](#)
End-of-line character. See Line feed
`endsWith` method [1.5.5](#)
enhanced for loop [1.8.5](#), [1.8.9](#), [3.7.2](#)
 for collections [7.2](#)
 for enumerations [4.4.1](#)
 for iterators [4.6.5](#)
 for paths [9.2.1](#)
enhanced form [1.7.2](#), [4.5.4](#)
`Entry` [6.5.1](#)
`entrySet` method [7.4](#)
`enum` keyword [1.3.4](#), [4.4](#), [4.4.1](#)
enum instances
 adding methods to [4.4.3](#)
 construction [4.4.2](#)
 referred by name [4.4.5](#)
enumeration sets [7.5.3](#)
enumerations [4.4](#)
 annotating [11.1.3](#)
 comparing [4.4.1](#)
 constructing [4.4.2](#)
 customizing serialization of [9.5.4](#)
 defining [1.3.4](#)
 nested inside classes [4.4.4](#)
 serialization of [9.5.5](#)
 static members of [4.4.4](#)
 traversing instances of [4.4.1](#)
 using in `switch` [4.4.5](#)
`EnumMap`, `EnumSet` classes [7.5.3](#)
environment variables [10.9.1](#)
equality, testing for [1.4.5](#)
`equals` method
 final [4.3.2](#)
 null-safe [4.3.2](#)
 of `Arrays` [4.3.2](#)
 of `Double` [4.3.2](#)
 of `Object` [4.3](#), [4.3.2](#)
 of `Objects` [4.3.2](#)
 of records [2.4.1](#)
 of `String` [1.5.3](#)
 of subclasses vs. superclass [4.3.2](#)
 of wrapper classes [1.8.4](#)
 overriding [4.3.2](#)
 symmetric [4.3.2](#)
 values from different classes and [4.3.2](#)
`equalsIgnoreCase` method [1.5.3](#)
`Error` class [5.1.2](#)
error messages, for generic methods [6.2](#)
`errorReader` method [10.9.1](#)
errors
 `AbstractMethodError` [3.2.2](#)
 `AssertionError` [5.2.1](#)
even numbers [1.4.2](#)
`Exception` class [5.1.2](#)
exceptionally [10.5.2](#)
`exceptionallyCompose` method [10.5.2](#)
exceptions [5.1](#)
 annotating [11.1.4](#)
 `ArrayIndexOutOfBoundsException` [1.8.1](#)
 `ArrayStoreException` [4.1.5](#), [6.4](#), [6.6.4](#)

CancellationException [10.5.1](#)
catching [5.1.4](#), [5.1.6](#)
chaining [5.1.7](#)
checked [4.7.4](#), [5.1.2](#), [5.1.3](#)
ClassCastException [3.1.5](#), [6.6.2](#)
ClassNotFoundException [5.1.2](#)
CloneNotSupportedException [4.3.4](#), [4.4.1](#)
combining in a superclass [5.1.3](#)
ConcurrentModificationException [7.2](#),
[10.3.3](#)
creating [5.1.2](#)
documenting [5.1.3](#)
ExecutionException [10.1.2](#)
FileNotFoundException [5.1.2](#)
generic types and [6.6.7](#)
hierarchy of [5.1.2](#)
IllegalArgumentException [5.2](#)
IllegalStateException [8.9](#), [10.3.2](#)
InaccessibleObjectException [4.7.2](#),
[12.6](#)
IndexOutOfBoundsException [5.1.9](#)
InterruptedException [10.1.3](#)
InvalidAccessException [9.5.6](#)
InvalidPathException [9.2.1](#)
IOException [5.1.2](#), [5.1.5](#), [9.1.5](#)
NoSuchElementException [8.7.4](#), [10.3.2](#)
NullPointerException [1.5.3](#), [1.8.2](#), [2.1.2](#),
[2.3.4](#), [5.1.2](#), [5.1.9](#), [7.4](#), [8.6](#)
NumberFormatException [5.1.2](#)
ReflectiveOperationException [4.6.1](#)
rethrowing [5.1.5](#), [5.1.7](#)
RuntimeException [5.1.2](#)
SecurityException [4.7.2](#)
ServletException [5.1.7](#)
suppressed [5.1.5](#)
throwing [5.1.1](#)
TimeoutException [10.1.2](#)
uncaught [5.1.8](#)
unchecked [5.1.2](#)
UncheckedIOException [9.1.5](#)
exec [10.9](#)
Executable class
 getModifiers method [4.7.4](#)
 getName method [4.7.4](#)
 getParameters method [4.7.1](#), [4.7.4](#)
ExecutableElement interface [11.5.2](#)
ExecutionException class [10.1.2](#)
Executor interface [10.5.2](#)
executor services [10.1.1](#), [10.5.1](#)
ExecutorCompletionService class [10.1.2](#)
Executors class
 newFixedThreadPool method [10.1.1](#)
 newVirtualThreadPerTaskExecutor
 method [10.1.1](#)
ExecutorService interface
 execute method [10.1.1](#)
 invokeAll, invokeAny methods [10.1.2](#)
exhaustiveness [1.7.2](#)
exists [9.2.2](#), [9.2.3](#)
exitValue method [10.9.2](#)
exports keyword [12.3](#), [12.5](#)
 qualified [12.12](#)
extends keyword [3.1.7](#), [4.1.1](#), [6.3](#), [6.4.3](#)
Externalizable interface,
 read/writeExternal methods [9.5.4](#)

F

F [1.2.2](#), [1.6.2](#)
factory methods [2.3.1](#), [2.5.5](#)
failures, logging [5.1.7](#)
false literal [1.2.4](#)
 as default value [2.3.4](#), [2.3.7](#)
Field class [4.7.1](#)
 get method [4.7.2](#), [4.7.4](#)
 getBoolean, getByte, getChar, getDouble,
 getFloat, getInt, getLong methods [4.7.2](#),
[4.7.4](#)
 getModifiers, getName methods [4.7.1](#),
[4.7.4](#)
 getShort method [4.7.2](#), [4.7.4](#)
 getType method [4.7.1](#)
 set, setXxx methods [4.7.4](#)
fields [4](#)
 enumerating [4.7.1](#)
 final [10.2.1](#)
 provided [4.2.3](#)
 public [4.7.1](#)
 retrieving values of [4.7.2](#)
 setting [4.7.2](#)
 transient [9.5.2](#)
File class [9.2.1](#)
file attributes
 copying [9.2.3](#)
 filtering paths by [9.2.4](#)

file handlers [5.3.5](#)
file pointers [9.1.9](#)
`file.separator` package [7.5.1](#)
`FileChannel` class
 `get, getXxx` methods [9.1.10](#)
 `lock` method [9.1.11](#)
 `open` method [9.1.10](#)
 `put, putXxx` methods [9.1.10](#)
 `tryLock` method [9.1.11](#)
`FileFilter` interface [3.6.2](#)
`FileHandler` class [5.3.5, 5.3.6](#)
`FileNotFoundException` class [5.1.2](#)
files
 archiving [9.2.5](#)
 channels to [9.1.10](#)
 checking for existence [5.1.2, 9.2.2, 9.2.3](#)
 closing [5.1.5](#)
 `copy` method [9.1.3, 9.2.3, 9.2.5](#)
 copying [9.2.3](#)
 `createTempXxx` methods [9.2.2](#)
 `createXxx` methods [9.2.2](#)
 creating [9.2.1, 9.2.3](#)
 `delete, deleteIfExists` methods [9.2.3](#)
 deleting [9.2.3](#)
 empty [9.2.2](#)
 encoding of [9.1.4](#)
 `exists` method [9.2.2, 9.2.3](#)
 `find` method [9.2.3, 9.2.4](#)
 `isDirectory, isRegularFile`
 methods [9.2.2, 9.2.3](#)
 `lines` method [8.2, 8.14, 9.1.5](#)
 `list` method [9.2.4](#)
 locking [9.1.11](#)
 memory-mapped [8.14, 9.1.10](#)
 `move` method [9.2.3](#)
 moving [9.2.3](#)
 `newBufferedReader` method [9.1.5](#)
 `newBufferedWriter` method [9.1.6, 9.2.3](#)
 `newXxxStream` methods [9.1.1, 9.2.3, 9.5.1](#)
 random-access [9.1.9](#)
 `read` method [9.1.2](#)
 `readAllBytes` method [9.1.2](#)
 `readAllLines` method [9.1.5](#)
 reading from/writing to [1.6.1, 5.1.2, 9.1.2](#)
 `readNBytes` method [9.1.2](#)
 `skipNBytes` method [9.1.2](#)
 temporary [9.2.2](#)
 `walk` method [9.2.3, 9.2.5](#)
`walkFileTree` method [9.2.3, 9.2.4](#)
`write` method [9.1.6, 9.2.3](#)
`FileSystem, FileSystems` classes [9.2.5](#)
`FileVisitor` interface [9.2.4](#)
fill method
 of `Arrays` [1.8.7](#)
 of `Collections` [1.8.7, 7.1](#)
`Filter` class [5.3.6](#)
 of `Optional` [8.7.3](#)
 of `Stream` [8.1, 8.3, 8.6](#)
filtering [8.11](#)
final keyword [1.3.4, 2.3.6, 4.2.1](#)
final fields [10.2.1](#)
final methods [10.2.4](#)
final variables [10.2.1, 10.2.4](#)
finalize [4.3](#)
finally keyword [5.1.6](#)
 for locks [10.7.1](#)
 return statements in [5.1.6](#)
financial calculations [1.2.2](#)
find [9.2.3, 9.2.4](#)
 `findAll` method [9.4.3](#)
 `findAny` method [8.6](#)
 `findClass` method [4.6.3](#)
 `findFirst` method [4.6.5, 8.6](#)
first [7.3](#)
flag bits, sequences of [7.5.2](#)
`flatMap` method
 of `Optional` [8.7.6, 8.7.7](#)
 of `Stream` [8.3](#)
`flatMapting` method [8.11](#)
flip [7.5.2](#)
float type [1.2.2, 1.8.4](#)
 streams of [8.13](#)
 type conversions of [1.4.4](#)
floating-point types [1.2.2](#)
 binary number system and [1.2.2](#)
 comparing [3.3.1](#)
 division of [1.4.2](#)
 formatting for output [1.6.2](#)
 in hexadecimal notation [1.2.2](#)
 type conversions of [1.4.4](#)
floor [7.3](#)
`floorMod` method [1.4.2](#)
for keyword [1.7.3](#)
 declaring variables for [1.7.5](#)
 enhanced [1.8.5, 1.8.9, 3.7.2, 4.4.1, 7.2, 9.2.1](#)
 multiple variables in [1.7.3](#)

forEach method
 of ArrayList [3.5.1](#)
 of Map [7.4](#)
 of Stream [8.8](#)
forEachOrdered method [8.8](#)
forEachXxx methods
 (ConcurrentHashMap) [10.3.1](#)
ForkJoinPool class [10.5.2](#)
 commonPool method [8.14, 10.5.1](#)
format specifiers [1.6.2](#)
formatted [1.6.2](#)
formatted output [1.6.2](#)
Formatter class [5.3.6](#)
forms, posting data from [9.3.1, 9.3.2](#)
forName method
 of Charset [9.1.4](#)
 of Class [4.6.1, 4.6.3, 4.6.4, 5.1.2, 5.1.8](#)
frequency [7.1](#)
Function interface [3.6.2, 8.9](#)
function types [3.4, 3.6.2](#)
functional interfaces [3.4.2, 11.3, 11.3.1](#)
 as method parameters [6.4.2](#)
 common [3.6.2](#)
 contravariant in parameter types [6.4.2](#)
 for primitive types [3.6.2](#)
 implementing [3.6.3](#)
@FunctionalInterface annotation [3.6.3, 11.3, 11.3.1, 11.3.2](#)
functions [2.1](#)
 higher-order [3.8](#)
Functions. See Methods
Future interface [10.1.2](#)
 cancel, isCancelled, isDone
 methods [10.1.2](#)
 get method [10.1.2, 10.5.1](#)
futures [10.1.2](#)
 completable [10.5.1, 10.5.2](#)

G

g, G
 conversion characters [1.6.2](#)
gadget chains [9.5.7](#)
garbage collector [7.5.5](#)
generate [8.2, 8.13](#)
@Generated annotation [11.3, 11.3.1](#)
generators, converting to streams [8.14](#)

generic classes [1.8.3, 6.1](#)
 constructing objects of [6.1](#)
 information available at runtime [6.7.2](#)
 instantiating [6.1](#)
generic collections [7.6.3](#)
generic constructors [6.7.2](#)
generic methods [6.2](#)
 calling [6.2](#)
 declaring [6.2](#)
 information available at runtime [6.7.2](#)
generic type declarations [6.7.2](#)
generic types [3.3.1](#)
 annotating [11.1.4](#)
 arrays of [3.5.2](#)
 casting [6.6.2](#)
 exceptions and [6.6.7](#)
 in JVM [6.5](#)
 invariant [6.4, 6.4.2](#)
 lambda expressions and [6.4.2](#)
 reflection and [6.7](#)
 restrictions on [6.6](#)
GenericArrayType interface [6.7.2](#)
get method
 of Array [4.7.6](#)
 of ArrayList [1.8.3](#)
 of BitSet [7.5.2](#)
 of Field [4.7.2, 4.7.4, 9.1.10](#)
 of Future [10.1.2, 10.5.1](#)
 of List [7.1](#)
 of LongAccumulator [10.6](#)
 of Map [7.4](#)
 of Optional [8.7.4, 8.7.7](#)
 of Path [9.2.1](#)
 of ServiceLoader.Provider [4.6.5](#)
 of Supplier [3.6.2](#)
GET requests [9.3.2](#)
getAndXxx methods (AtomicXxx) [10.6](#)
getAnnotation, getAnnotationsByType
 methods
 of AnnotatedConstruct [11.5.2](#)
 of AnnotatedElement [11.4](#)
getAsXxx methods
 of OptionalXxx [8.13](#)
 of XxxSupplier [3.6.2](#)
getAudioClip method [4.6.2](#)
getAverage method [8.8](#)
getBoolean method
 of Array [4.7.6](#)

of Field [4.7.2](#), [4.7.4](#)
of FileChannel [9.1.10](#)

getByte method
 of Array [4.7.6](#)
 of Field [4.7.2](#), [4.7.4](#)
 of FileChannel [9.1.10](#)

getCanonicalName method [4.6.1](#)

getChar method
 of Array [4.7.6](#)
 of Field [4.7.2](#), [4.7.4](#)
 of FileChannel [9.1.10](#)

getClass method [4.2.1](#), [4.3](#), [4.3.2](#), [4.6.1](#),
[6.6.2](#), [6.7.1](#)

getClassLoader method [4.6.1](#)

getComponentType method [4.6.1](#), [4.7.6](#)

getConstructor(s) methods (Class) [4.6.1](#),
[4.7.1](#), [4.7.4](#), [6.7.1](#)

getContextClassLoader method [4.6.4](#)

getCountry method [8.10](#)

getCurrencyInstance method [2.5.5](#)

getDayOfMonth methods
 of LocalDate [2.1](#)

getDeclaredAnnotationXxx methods
 (AnnotatedElement) [11.4](#)

getDeclaredConstructor(s) methods
 (Class) [4.6.1](#), [4.7.1](#), [6.7.1](#)

getDeclaredField(s) methods (Class) [4.6.1](#)

getDeclaredMethod(s) methods
 (Class) [4.6.1](#), [4.7.3](#)

getDeclaringClass method
 of Class [4.6.1](#)
 of Enum [4.4.1](#)

getDefault method
 of RandomGenerator [1.1.3](#)

getDouble method
 of Array [4.7.6](#)
 of Field [4.7.2](#), [4.7.4](#)
 of FileChannel [9.1.10](#)

getElementsAnnotatedWith method [11.5.2](#)

getEnclosedElements method [11.5.2](#)

getEnclosingXxx methods (Class) [4.6.1](#)

getEnumConstants method [6.7.1](#)

getErrorStream method [10.9.1](#)

getField(s) methods (Class) [4.6.1](#), [4.7.1](#)

getFileName method [9.2.1](#)

getFilePointer method [9.1.9](#)

getFloat method
 of Array [4.7.6](#)

of Field [4.7.2](#), [4.7.4](#)
of FileChannel [9.1.10](#)

getHead method [5.3.6](#)

getHeaderFields method [9.3.1](#)

getInputStream method
 of Process [10.9.1](#)
 of URL [9.3](#)
 ofURLConnection [9.3.1](#)

getInstant method [5.3.6](#)

getInt method
 of Array [4.7.6](#)
 of Field [4.7.2](#), [4.7.4](#)
 of FileChannel [9.1.10](#)

getInterfaces method [4.6.1](#)

getLength method [4.7.6](#)

getLevel method [5.3.6](#)

getLogger method [5.3.2](#), [5.3.3](#)

getLoggerName method [5.3.6](#)

getLong method
 of Array [4.7.6](#)
 of Field [4.7.2](#), [4.7.4](#)
 of FileChannel [9.1.10](#)

getLongThreadID method [5.3.6](#)

getMax method [8.8](#)

getMessage method [5.3.6](#)

getMethod(s) methods (Class) [4.6.1](#), [4.7.1](#),
[4.7.3](#)

getModifiers method
 of Class [4.6.1](#)
 of Constructor [4.7.1](#)
 of Executable [4.7.4](#)
 of Field [4.7.1](#), [4.7.4](#)
 of Method [4.7.1](#)

getMonthValue method
 of LocalDate [2.1](#)

getName method
 of Class [4.6.1](#)
 of Constructor [4.7.1](#)
 of Executable [4.7.4](#)
 of Field [4.7.1](#), [4.7.4](#)
 of Method [4.7.1](#)
 of Parameter [4.7.4](#)
 of Path [9.2.1](#)
 ofPropertyDescriptor [4.7.5](#)
 of System.Logger [5.3.3](#)

getOrDefault method [7.4](#)

getOutputStream method
 of Process [10.9.1](#)

of URLConnection [9.3.1](#)
getPackage method [4.6.1](#)
getPackageName method [4.6.1](#)
getParameters method
 of Executable [4.7.1, 4.7.4](#)
 of LogRecord [5.3.6](#)
getParent method [9.2.1](#)
getPath method [9.2.5](#)
getPercentInstance method [2.5.5](#)
getPermittedSubclasses method [4.6.1](#)
getProperties method [7.5.1](#)
getProperty method [4.6.3, 5.1.9, 7.5.1](#)
getPropertyDescriptors method [4.7.5](#)
getPropertyType, getReadMethod methods
 (PropertyDescriptor) [4.7.5](#)
getQualifiedName method [11.5.2](#)
getRecordComponents method [4.6.1](#)
getResource method [4.6.2](#)
getResourceAsStream method
 of Class [4.6.1, 4.6.2](#)
 of Module [12.6](#)
getResourceBundle, getResourceBundleName
 methods (LogRecord) [5.3.6](#)
getRoot method [9.2.1](#)
getSequenceNumber method [5.3.6](#)
getShort method
 of Array [4.7.6](#)
 of Field [4.7.2, 4.7.4](#)
 of FileChannel [9.1.10](#)
getSimpleName method
 of Class [4.6.1](#)
 of Element [11.5.2](#)
getSourceXxxName methods
 (LogRecord) [5.3.6](#)
getSuperclass method [4.6.1, 6.7.1](#)
getSuppressed method [5.1.5](#)
getTail method [5.3.6](#)
Getter/setter pairs. See Properties
getThrown method [5.3.6](#)
getType method
 of Field [4.7.1](#)
 of Parameter [4.7.4](#)
getTypeName method [4.6.1](#)
getTypeParameters method [6.7.2](#)
getURLs method [4.6.3](#)
getValue method [2.1](#)
getWriteMethod method [4.7.5](#)
Goetz, Brian [10](#)

graphemeClusters method
 of String [8.4](#)
group [9.4.3, 9.4.4](#)
grouping [8.10](#)
 classifier functions of [8.10](#)
 reducing to numbers [8.11](#)
groupingBy method [8.10, 8.11](#)
groupingByConcurrent method [8.10, 8.14](#)
GUI
 callbacks in [3.3.4](#)
 long-running tasks in [10.5.3](#)

H

h, H conversion characters [1.6.2](#)
handle [10.5.2](#)
Hansen, Per Brinch [10.7.2](#)
hash [4.3.3](#)
hash codes [4.3.3](#)
 computing in String class [4.3.3](#)
 formatting for output [1.6.2](#)
hash functions [4.3.3, 7.3](#)
hash maps
 concurrent [10.3.1](#)
 weak [7.5.5](#)
hash tables [7.3](#)
hashCode method
 of Arrays [4.3.3](#)
 of Enum [4.4.1](#)
 of Object [4.3, 4.3.2, 4.3.3](#)
 of records [2.4.1](#)
HashMap class [7.4](#)
 null values in [7.4](#)
HashSet class [7.3](#)
 readObject, writeObject methods [9.5.3](#)
Hashtable class [10.7.2](#)
hasNext method
 declaring [3.1.2](#)
 of Iterator [7.2](#)
 of Scanner [1.6.1, 9.1.5](#)
hasNextXxx methods (Scanner) [1.6.1, 9.1.5](#)
headMap method [7.6.2](#)
headSet method
 of NavigableSet [7.3](#)
 of SortedSet [7.3, 7.6.2](#)
heap pollution [6.6.2, 7.6.3](#)
Hello, World! program [1.1.1](#)

modular [12.3](#)
helper methods [6.4.5](#)
hexadecimal numbers [1.2.1, 1.2.2](#)
 formatting for output [1.6.2](#)
higher [7.3](#)
higher-order functions [3.8](#)
hn, hr elements (HTML) [2.8.1](#)
Hoare, Tony [10.7.2](#)
HTML
 generating documentation in [11.5.3](#)
 including code in [1.5.7](#)
HTTP connections [9.3](#)
HTTP/2 support [9.3](#)
HttpClient class [9.3, 9.3.2](#)
 enabling logging for [9.3.2](#)
 newBuilder, newHttpClient
 methods [9.3.2, 10.5.1](#)
HttpHeaders class [9.3.2](#)
HttpResponse interface [9.3.2](#)
HttpURLConnection class [9.3.1](#)
hyperlinks
 in documentation comments [2.8.6](#)
 regular expressions for [9.4](#)

I

IDE [1.1.2](#)
identity method
 of Function [3.6.2, 8.9](#)
 of UnaryOperator [3.6.2](#)
identity values [8.12](#)
if keyword [1.7.1](#)
ifPresent, ifPresentOrElse methods
 (Optional) [8.7.2](#)
IllegalArgumentException class [5.2](#)
IllegalStateException class [8.9, 10.3.2](#)
ImageIcon class [4.6.2](#)
images, locating [4.6.2](#)
img [2.8.1](#)
immutability [10.2.3](#)
immutable classes [10.2.4](#)
implements keyword [3.1.3](#)
import keyword [1.1.3, 2.6.5](#)
 no annotations for [11.1.4](#)
 static [2.6.6](#)
import static [4.4.5](#)
InaccessibleObjectException class [4.7.2, 12.6](#)
increment [10.6](#)
increment operator [1.4.2](#)
incrementAndGet method [10.6](#)
incrementExact method [1.4.3](#)
indexOf method
 of List [7.1](#)
 of String [1.5.5](#)
indexOfSubList method [7.1](#)
IndexOutOfBoundsException class [5.1.9](#)
info [10.9.3](#)
Information hiding. See Encapsulation
inheritance [4.1, 4.3.4](#)
 classes win rule [4.2.5, 4.3.3](#)
 default methods and [4.2.5](#)
@Inherited annotation [11.3, 11.3.2](#)
initCause method [5.1.7](#)
initialization blocks [2.3.5](#)
 static [2.5.3](#)
Inlining [4.2.1](#)
inner classes [2.7.2](#)
 anonymous [3.9.2](#)
 invoking methods of outer classes [2.7.2](#)
 local [3.7.2, 3.9](#)
 syntax for [2.7.3](#)
input
 reading [1.6.1, 9.1.5](#)
 splitting along delimiters [9.4.5](#)
input prompts [1.6.2](#)
input streams [9.1](#)
 copying [9.1.3](#)
 obtaining [9.1.1](#)
 reading from [9.1.2](#)
inputReader method [10.9.1](#)
InputStream class [9.1.2](#)
 transferTo method [9.1.3](#)
InputStreamReader class [9.1.5](#)
INSTANCE [9.5.5](#)
instance methods [1.1.3, 2.2.4](#)
instance variables [2.2.1, 2.2.5](#)
 abstract classes and [4.2.2](#)
 annotating [11.1.3](#)
 comparing [4.3.2](#)
 default values of [2.3.4](#)
 final [2.3.6](#)
 in records [2.4.1, 2.4.2](#)
 initializing [2.3.5, 4.1.4](#)

not accessible from static methods [2.5.4](#)
of deserialized objects [9.5.5](#), [9.5.6](#)
protected [4.2.3](#)
setting [2.3.1](#)
transient [9.5.2](#)
vs. local [2.3.4](#)
instanceof keyword [3.1.5](#), [4.1.6](#), [4.3.2](#)
annotating [11.1.4](#)
with pattern matching [3.1.6](#)
instances [1.1.1](#), [1.1.3](#)
instruction reordering [10.2.1](#)
int type [1.2.1](#)
functional interfaces for [3.6.2](#)
processing values of [3.6.1](#)
random number generator for [1.1.3](#)
streams of [8.13](#)
type conversions of [1.4.4](#)
using class literals with [4.6.1](#)
IntBinaryOperator interface [3.6.2](#)
IntConsumer interface [3.6.1](#), [3.6.2](#)
Integer class [1.8.4](#)
compare method [3.3.1](#)
MAX_VALUE, MIN_VALUE constants [1.2.1](#)
parseInt method [1.5.4](#), [5.1.2](#)
toString method [1.5.4](#)
unsigned division in [1.2.1](#)
xxxUnsigned methods [1.4.3](#)
integer types [1.2.1](#)
comparing [3.3.1](#)
computing [1.4.2](#), [1.4.3](#)
formatting for output [1.6.2](#)
in hexadecimal notation [1.2.1](#)
reading/writing [9.1.8](#), [9.1.9](#)
type conversions of [1.4.4](#)
values of
 even/odd [1.4.2](#)
 signed [1.2.1](#)
interface keyword [3.1.2](#), [11.2](#)
 sealed [4.2.4](#)
interface methods [3.2.2](#), [3.2.3](#)
interfaces [3.1](#)
 annotating [11.1.3](#), [11.1.4](#)
 compatibility of [3.2.2](#)
 declarations of [3.1.2](#)
 defining variables in [3.1.9](#)
 documentation comments for [2.8.1](#)
 evolution of [3.2.2](#)
 extending [3.1.7](#)
functional [3.4.2](#), [11.3](#), [11.3.1](#)
implementing [3.1.3](#)
 multiple [3.1.8](#)
methods of [3.1.2](#), [3.1.3](#)
nested, enumerating [4.7.1](#)
no instance variables in [3.1.9](#)
no redefining methods of the Object class
in [4.3.3](#)
views of [7.6](#)
interrupted [10.1.3](#)
interrupted status [10.1.3](#)
InterruptedException class [10.1.3](#)
intersects [7.5.2](#)
IntFunction interface [3.6.2](#), [6.6.1](#)
IntPredicate interface [3.6.2](#)
intrinsic locks [10.7.2](#)
ints [8.13](#)
IntSequence [3.1.3](#), [3.9.1](#)
IntStream interface [8.13](#)
 parallel method [8.14](#)
IntSummaryStatistics class [8.8](#), [8.13](#)
IntSupplier, IntToXxxFunction,
 IntUnaryOperator interfaces [3.6.2](#)
InvalidClassException class [9.5.6](#)
InvalidPathException class [9.2.1](#)
InvocationHandler interface [4.7.7](#)
invoke [4.7.3](#), [4.7.4](#)
invokeAll, invokeAny methods
 (ExecutorService) [10.1.2](#)
IOException class [5.1.2](#), [9.1.5](#)
 addSuppressed, getSuppressed
 methods [5.1.5](#)
isAbstract method [4.6.1](#), [4.7.1](#)
isAlive method
 of Process [10.9.2](#)
 of ProcessHandle [10.9.3](#)
 of Thread [10.8.1](#)
isAnnotation method [4.6.1](#)
isAnonymousClass method [4.6.1](#)
isArray method [4.6.1](#), [4.7.6](#)
isAssignableFrom method [4.6.1](#)
isCancelled method [10.1.2](#)
isDirectory method [9.2.2](#), [9.2.3](#)
isDone method
 of CompletableFuture [10.5.1](#)
 of Future [10.1.2](#)
isEmpty method
 of BitSet [7.5.2](#)

of Collection [7.1](#)
of Map [7.4](#)
isEnum method [4.6.1](#)
isEqual method [3.6.2](#)
isFinite, isInfinite methods
(Double) [1.2.2](#)
isInstance method [4.6.1](#)
isInterface method [4.6.1, 4.7.1](#)
isInterrupted method [10.1.3](#)
isLocalClass method [4.6.1](#)
isLoggable method
of Filter [5.3.6](#)
of System.Logger [5.3.3](#)
isMemberClass method [4.6.1](#)
isNamePresent method [4.7.4](#)
isNative method [4.6.1, 4.7.1](#)
isNull method [3.5.1](#)
ISO 8601 format [11.3.1](#)
ISO 8859-1 encoding [9.1.4, 9.1.6](#)
isPresent method [8.7.4, 8.7.7](#)
isPrimitive method [4.6.1](#)
isPrivate, isProtected, isPublic methods
(Modifier) [4.6.1, 4.7.1](#)
isRecord method [4.6.1](#)
isRegularFile method [9.2.2, 9.2.3](#)
isSealed method [4.6.1](#)
isStatic, isStrict, isSynchronized methods
(Modifier) [4.6.1, 4.7.1](#)
isSynthetic method [4.6.1](#)
isVolatile method [4.6.1, 4.7.1](#)
Iterable interface [7.2, 9.2.1](#)
iterator method [7.2](#)
iterate [8.2, 8.5, 8.13, 10.4.1](#)
iterator method
next, hasNext methods [7.2](#)
of Collection [7.1](#)
of ServiceLoader [4.6.5](#)
of Stream [8.8](#)
remove, removeIf methods [7.2](#)
iterators [7.2, 8.8](#)
converting to streams [8.2, 8.14](#)
invalid [7.2](#)
traversing [4.6.5](#)
weakly consistent [10.3.3](#)

J

jar [2.6.2](#)
--module-version option [12.7](#)
-C option [12.7](#)
-d option [12.7](#)
JAR files [2.6.2](#)
dependencies in [12.14](#)
for split packages [12.7](#)
manifest for [12.8](#)
modular [12.7](#)
processing order of [2.6.3](#)
resources in [4.6.2](#)
scanning for deprecated elements [11.3.1](#)
Java [1.1.2](#)
--add-exports, --add-opens
options [12.10](#)
--add-module option [12.7](#)
--illegal-access option [12.10](#)
-cp (--class-path, -classpath)
option [2.6.3](#)
-da (-disableassertions) option [5.2.2](#)
-ea (-enableassertions) option [5.2.2](#)
-esa (-enablesystemassertions)
option [5.2.2](#)
-m, -p (--module, -module-path)
options [12.3, 12.7](#)
compatibility with older versions of [4.2.4, 4.2.5, 6.5](#)
online API documentation on [1.5.5](#)
option files for [12.10](#)
option names in [2.6.2](#)
strongly typed [1.3.1](#)
Unicode support in [1.5.6](#)
uniformity of [1.1.1, 3.2.3](#)
Java Persistence Architecture [11](#)
Java Platform Module System [12](#)
layers in [12.7](#)
migration to [12.8, 12.9](#)
no support for versioning in [12.1, 12.3, 12.7](#)
service loading in [12.13](#)
java.awt package [2.6.4, 12.1](#)
java.awt.geom package [9.5.3](#)
java.base package [12.4](#)
java.class.path package [7.5.1](#)
java.desktop package [12.4](#)
java.home package [7.5.1](#)

`java.io.tmpdir` package [7.5.1](#)
`java.lang`, `java.lang.annotation` packages [11.3](#)
`java.lang.reflect` package [4.7.1](#)
`java.logging` package [12.11](#)
`java.time` package
 immutability of classes [10.2.4](#)
`java.util` package [1.1.3, 10.3.3](#)
`java.util.concurrent` package [10.3, 10.3.2](#)
`java.util.concurrent.atomic` package [10.6](#)
`java.util.logging` package [5.3.1, 5.3.4](#)
`java.util.random` package [3.1.1](#)
`java.version` package [7.5.1](#)
JavaBeans [4.7.5](#)
javac [1.1.2](#)
 -`author` option [2.8.8](#)
 -`cp` (`--class-path`, `-classpath`) option [2.6.3](#)
 -`d` option [2.6.1, 2.8.8](#)
 -`link`, `-linksources` options [2.8.8](#)
 -`parameters` option [4.7.1](#)
 -`processor` option [11.5.1](#)
 -`version` option [2.8.8](#)
 -`XprintRounds` option [11.5.3](#)
javadoc [2.8](#)
 including annotations in [11.3.2](#)
JavaFX [3.3.4, 10.5.3](#)
`javan.log` files [5.3.5](#)
JavaServer Faces framework [7.4](#)
`javax.annotation` package [11.3](#)
`javax.swing` package [12.4](#)
JAXB [12.6](#)
JCommander [11](#)
jconsole [5.3.4](#)
jdeprscan [11.3.1](#)
jdeps [12.14](#)
JDK [1.1.2](#)
 obsolete features in [12.1](#)
JEP 246 (platform logging API) [5.3.1](#)
jlink [12.14](#)
jmod [12.14](#)
job scheduling [7.5.4](#)
join method
 of `String` [1.5.1](#)
 of `Thread` [10.8.1](#)
joining [8.8](#)
JPA [12.6](#)

JShell [1.1.4](#)
 imported packages in [1.1.4](#)
 loading modules into [12.7](#)
JSON [4.2.4](#)
JUnit [11, 11.1](#)

K

key/value pairs
 adding new keys to [7.4](#)
 in annotations [11.1.1, 11.2](#)
 removed by garbage collector [7.5.5](#)
 values of [7.4](#)
Key/value pairs. See Properties
keySet method
 of `ConcurrentHashMap` [10.3.3](#)
 of `Map` [7.4, 7.6](#)
keywords [1.3.2](#)
 contextual [4.2.4](#)

L

L [1.2.1](#)
`L64X128MixRandom` [3.1.1](#)
labeled statements [1.7.4](#)
lambda expressions [3.4](#)
 annotating targets for [11.3.1](#)
 capturing variables in [3.7.2](#)
 executing [3.6.1](#)
 for loggers [5.3.3](#)
 generic types and [6.4.2](#)
 parameters of [3.4.1](#)
 processing [3.6](#)
 return type of [3.4.1](#)
 scope of [3.7.1](#)
 this reference in [3.7.1](#)
 throwing exceptions in [5.1.3](#)
 using with streams [8.3, 10.4.1](#)
language codes [8.10](#)
language model API [11.5.2](#)
last [7.3](#)
lastIndexOf method
 of `List` [7.1](#)
 of `String` [1.5.5](#)
lastIndexOfSubList method [7.1](#)
lazy operations [8.1, 8.2, 8.5, 9.4.5](#)
length method

of arrays [1.8.1](#)
of RandomAccessFile [9.1.9](#)
of String [1.1.3, 1.5.6](#)
.level [5.3.4](#)
lib/modules [12.14](#)
limit [8.4, 8.14](#)
line feed [1.5.7](#)
 character literal for [1.2.3](#)
 formatting for output [1.6.2](#)
 in regular expressions [9.4.1](#)
line.separator package [7.5.1](#)
lines method
 of BufferedReader [9.1.5](#)
 of Files [8.2, 8.14, 9.1.5](#)
@link annotation [2.8.6](#)
linked lists [7.1, 7.2](#)
LinkedBlockingQueue class [10.3.2, 10.7.3](#)
LinkedHashMap class [7.4](#)
LinkedList class [7.1](#)
List class [6.4.3, 7.1, 9.2.4](#)
 add, addAll, get, indexOf, lastIndexOf,
 listIterator methods [7.1](#)
 of method [1.8.3, 1.8.6, 7.1, 7.6.1](#)
 remove, replaceAll, set, sort
 methods [7.1](#)
 subList method [7.1, 7.6.2](#)
ListIterator interface [7.2](#)
lists
 converting to streams [8.14](#)
 mutable [7.6.1](#)
 printing elements of [3.5.1](#)
 removing null values from [3.5.1](#)
 sublists of [7.6.2](#)
 unmodifiable views of [7.6.3](#)
literals
 character [1.2.3](#)
 floating-point [1.2.2](#)
 integer [1.2.1](#)
 string [1.5.3, 1.5.7](#)
little-endian format [9.1.4](#)
load [4.6.5, 12.13](#)
load balancing [9.5](#)
loadClass method [4.6.3](#)
local classes [3.9.1](#)
local variables [1.7.5](#)
 annotating [11.1.3](#)
 vs. instance [2.3.4](#)
LocalDate class [2.1](#)
getXXX methods [2.1](#)
now method [2.3.1, 2.5.4](#)
of method [2.1, 2.3.1](#)
plus, plusXXX methods [2.1, 2.1.1, 2.1.2](#)
Locale class [8.9](#)
 getCountry method [8.10](#)
locales [8.9, 8.11](#)
LocalTime class
 final [4.2.1](#)
lock method
 of FileChannel [9.1.11](#)
 of ReentrantLock [10.7.1](#)
locks [10.2.2](#)
 error-prone [10.2.3](#)
 intrinsic [10.7.2](#)
 reentrant [10.7.1](#)
 releasing [5.1.6, 10.2.1](#)
log handlers [5.3.5](#)
 filtering/formatting [5.3.6](#)
Log4j [5.3.1](#)
Logback [5.3.1](#)
Logger class (java.util.logging) [12.11](#)
Logger interface (System) [5.3.2, 5.3.3](#)
 getName method [5.3.3](#)
 isLoggable method [5.3.3](#)
 log method [5.3.2, 5.3.3](#)
loggers
 filtering/formatting [5.3.6](#)
 hierarchy of [5.3.4](#)
 naming [5.3.2](#)
logging [5.3](#)
 configuring [5.3.4](#)
 failures [5.1.7](#)
 levels of [5.3.3, 5.3.4](#)
 overriding methods for [4.2.1](#)
LogRecord class, methods of [5.3.6](#)
long type [1.2.1, 1.8.4](#)
 atomic operations on [10.6](#)
 functional interfaces for [3.6.2](#)
 MAX_VALUE, MIN_VALUE constants [1.2.1](#)
 streams of [8.13](#)
 type conversions of [1.4.4](#)
 unsigned division in [1.2.1](#)
 xxxUnsigned methods [1.4.3](#)
long-term persistence [9.5.6](#)
LongAccumulator class [10.6](#)
 accumulate, get methods [10.6](#)
LongAdder class [10.6](#)

add, increment, sum methods [10.6](#)
threadsafe [10.7.1](#)
LongConsumer, LongXxxOperator,
LongPredicate, LongSupplier,
LongToXxxFunction interfaces [3.6.2](#)
LongFunction interface [3.6.2, 6.6.1](#)
longs [8.13](#)
LongStream interface [8.13](#)
LongSummaryStatistics class [8.8, 8.13](#)
Lookup [12.6](#)
lookup method (MethodHandles) [12.6](#)
loops [1.7.3](#)
 exiting [1.7.4](#)
 infinite [1.7.3](#)
lower [7.3](#)

M

main [1.1.1, 1.1.3](#)
decomposing [1.9](#)
string array parameter of [1.8.8](#)
Map interface [7.1](#)
 clear method [7.4](#)
 compute method [7.4](#)
 computeIfXxx methods [7.4](#)
 containsXxx methods [7.4](#)
 entrySet method [7.4](#)
 forEach method [7.4](#)
 get, getOrDefault methods [7.4](#)
 isEmpty method [7.4](#)
 keySet method [7.4, 7.6](#)
 merge method [7.4](#)
 of method [7.4, 7.6.1](#)
 of Optional [8.7.3](#)
 of Stream [8.3](#)
 ofEntries method [7.6.1](#)
 put method [7.4](#)
 putAll method [7.4](#)
 putIfAbsent method [7.4](#)
 remove method [7.4](#)
 replace, replaceAll methods [7.4](#)
 size method [7.4](#)
 values method [7.4, 7.6](#)
mapping method [8.11](#)
 of Collectors [8.11](#)
maps [7.4](#)
 concurrent [7.4, 8.9](#)

empty [7.4](#)
iterating over [7.4](#)
of stream elements [8.9, 8.14](#)
order of elements in [7.4](#)
views of [7.4](#)
 unmodifiable [7.6.3](#)
mapToInt method [8.12](#)
mapToXxx methods (*XxxStream*) [8.13](#)
marker interfaces [4.3.4](#)
Matcher class [9.4.2, 9.4.4](#)
 methods of [9.4.6](#)
matcher, matches methods (Pattern) [9.4.2](#)
MatchResult interface [9.4.3, 9.4.6](#)
Math class
 E constant [1.4.3](#)
 floorMod method [1.4.2](#)
 max, min methods [1.4.3](#)
 PI constant [1.4.3, 2.5.2, 2.6.6](#)
 pow method [1.4.3, 2.5.4, 2.6.6](#)
 round method [1.4.4](#)
 sqrt method [1.4.3](#)
 TAU constant [1.4.3](#)
 xxxExact methods [1.4.3, 1.4.4](#)
max method
 of Stream [8.6](#)
 of XxxStream [8.13](#)
MAX_VALUE [1.2.1](#)
maxBy method
 of BinaryOperator [3.6.2](#)
 of Collectors [8.11](#)
memory
 allocating [10.2.3](#)
 caching [10.2.1](#)
 concurrent access to [10.2.1](#)
memory-mapped files [9.1.10](#)
merge method
 of ConcurrentHashMap [10.3.1](#)
 of Map [7.4](#)
Message [4.3.4](#)
meta-annotations [11.2, 11.3.2](#)
META-INF/MANIFEST.MF [12.8](#)
META-INF/services [12.13](#)
Method class [4.7.1](#)
 getModifiers, getName methods [4.7.1](#)
 invoke method [4.7.3, 4.7.4](#)
method calls [1.1.3](#)
 receiver of [2.2.4](#)
method expressions [3.5.1, 4.1.8](#)

method references [3.5.1](#), [6.6.3](#)
annotating [11.1.4](#)

`MethodHandles.lookup` method [12.6](#)

methods [1.1.1](#)
abstract [3.4.2](#), [4.2.2](#)
accessor [2.1.1](#), [2.4.1](#)
annotating [6.6.4](#), [11.1.3](#)
atomic [10.3.1](#)
body of [2.2.3](#)
chaining calls of [2.1](#)
clashes of [6.6.6](#)
compatible [4.3.3](#)
declarations of [2.2.2](#)
default [3.2.2](#), [3.2.3](#)
deprecated [2.8.5](#), [11.3](#), [11.3.1](#)
documentation comments for [2.8.1](#), [2.8.3](#)
enumerating [4.7.1](#)
factory [2.3.1](#), [2.5.5](#)
final [4.2.1](#), [10.2.4](#)
for throwing exceptions [5.1.9](#)
header of [2.2.2](#)
inlining [4.2.1](#)
instance [2.2.4](#)
invoking [4.7.3](#)
modifying functions [3.8.2](#)
mutator [2.1.1](#), [7.6.3](#), [10.2.4](#)
naming [1.3.2](#), [2.4.1](#)
native [2.5.2](#)
overloading [2.3.2](#), [3.5.1](#)
overriding [3.2.2](#), [4.1.3](#), [4.2.1](#), [5.1.3](#), [11.3](#),
[11.3.1](#)
parameters of [4.7.1](#)
 null checks for [5.1.9](#)
passing arrays into [1.9.2](#)
private [3.2.4](#)
proxied [4.7.7](#)
public [3.1.2](#), [3.1.3](#), [4.7.1](#)
restricted to subclasses [4.2.3](#)
return value of [1.1.1](#), [2.2.3](#)
returning functions [3.8.1](#)
static [1.9.1](#), [2.5.4](#), [2.5.5](#), [2.6.6](#), [3.2.1](#)
storing in variables [1.1.3](#)
symmetric [4.3.2](#)
synchronized [10.7.2](#), [10.7.3](#)
used for serialization [11.3](#), [11.3.1](#)
utility [2.6.4](#)
variable number of arguments of [1.9.3](#)

Microsoft Notepad [9.1.4](#)

Microsoft Windows
line ending in [1.5.7](#)
path separator in [2.6.3](#), [7.5.1](#)

min method
 of Math [1.4.3](#)
 of Stream [8.6](#)
 of XxxStream [8.13](#)

MIN_VALUE [1.2.1](#)

minBy method
 of BinaryOperator [3.6.2](#)
 of Collectors [8.11](#)

Modifier class
 isXxx methods [4.6.1](#), [4.7.1](#)
 toString method [4.6.1](#)

modifiers, checking [4.7.1](#)

module keyword [12.3](#)

module path [12.3](#), [12.7](#), [12.8](#), [12.9](#)

module-info.class [12.3](#), [12.7](#)

module-info.java [12.3](#)

`Module.getResourceAsStream` method [12.6](#)

modules [12](#)
 aggregator [12.11](#)
 annotating [12.3](#)
 automatic [12.8](#), [12.9](#)
 bundling up the minimal set of [12.14](#)
 declaration of [12.2](#), [12.3](#)
 documentation comments for [2.8.1](#), [2.8.7](#)
 explicit [12.9](#)
 illegal access to [12.10](#)
 inspecting files in [12.14](#)
 loading into JShell [12.7](#)
 naming [12.2](#), [12.8](#)
 open [12.6](#)
 reflective access for [4.7.1](#), [4.7.2](#)
 required [12.4](#), [12.11](#)
 tools for [12.14](#)
 transitive [12.11](#)
 unnamed [12.9](#)
 versioning and [12.1](#), [12.3](#), [12.7](#)

monitors (classes) [10.7.2](#)

move [9.2.3](#)

multiplication [1.4.2](#)

mutators [2.1.1](#)
 unmodifiable views and [7.6.3](#)

N

n

- conversion character [1.6.2](#)
- name [4.4.1](#)
- NaN [1.2.2](#)
- native encoding [9.1.4](#)
- native methods [2.5.2](#)
- naturalOrder method [3.8.3](#)
- navigable maps/sets [7.6.3](#)
- NavigableMap interface [10.3.3](#)
- NavigableSet interface [7.1, 7.3, 7.6.2](#)
 - methods of [7.3](#)
- nCopies method [7.1](#)
- negate [3.6.2](#)
- negateExact method [1.4.3](#)
- negative values [1.2.1](#)
- NEGATIVE_INFINITY [1.2.2](#)
- nested classes [2.7](#)
 - annotating [11.1.4](#)
 - enumerating [4.7.1](#)
 - inner [2.7.2](#)
 - public [2.7.1](#)
 - static [2.7.1](#)
- new keyword [1.1.3, 1.3.1, 1.4, 2.3.1](#)
 - as constructor reference [3.5.2](#)
 - for anonymous classes [3.9.2](#)
 - for arrays [1.8.1, 1.8.2, 1.8.9](#)
- newBufferedReader method [9.1.5](#)
- newBufferedWriter method [9.1.6, 9.2.3](#)
- newBuilder method [9.3.2, 10.5.1](#)
- newFileSystem method [9.2.5](#)
- newFixedThreadPool method
 - of Executors [10.1.1](#)
- newHttpClient method [9.3.2, 10.5.1](#)
- newInputStream method [9.1.1, 9.2.3, 9.5.1](#)
- newInstance method
 - of Array [4.7.6](#)
 - of Class [4.7.4, 6.7.1](#)
 - of Constructor [4.7.4](#)
- newKeySet method [10.3.3](#)
- newOutputStream method [9.1.1, 9.2.3, 9.5.1](#)
- newProxyInstance method [4.7.7](#)
- newVirtualThreadPerTaskExecutor method
 - of Executors [10.1.1](#)
- next method
 - declaring [3.1.2](#)

- of Iterator [7.2](#)
- of Scanner [1.6.1](#)
- nextClearBit method [7.5.2](#)
- nextDouble method
 - common for all generators [3.1.1](#)
 - of Scanner [1.6.1, 9.1.5](#)
- nextInt method
 - common for all generators [3.1.1](#)
 - of Random [1.1.3](#)
 - of Scanner [1.6.1](#)
- nextLine method [1.6.1](#)
- nextSetBit method [7.5.2](#)
- nominal typing [3.6.2](#)
- non-sealed keyword [4.2.4](#)
- noneMatch method [8.6](#)
- noneOf method [7.5.3](#)
- noninterference, of stream operations [8.2](#)
- @NonNull annotation [11.1.4](#)
- normalize [9.2.1](#)
- NoSuchElementException class [8.7.4, 10.3.2](#)
- notify, notifyAll methods (Object) [10.7.3](#)
- now method
 - of LocalDate [2.3.1, 2.5.4](#)
- null literal [1.5.3, 2.1.2](#)
 - as default value [2.3.4, 2.3.7](#)
 - checking parameters for [5.1.9](#)
 - comparing against [4.3.2](#)
 - converting to strings [4.3.1](#)
- NullPointerException class [1.5.3, 1.8.2, 2.1.2, 2.3.4, 5.1.2, 5.1.9, 7.4](#)
 - vs. Optional [8.6](#)
- nullsFirst, nullsLast methods
 - (Comparator) [3.8.3](#)
- NumberFormat class
 - getXXXInstance methods [2.5.5](#)
- NumberFormatException class [5.1.2](#)
- numbers
 - average of [3.1.2, 3.1.3](#)
 - big [1.4.6](#)
 - comparing [3.3.1](#)
 - converting to strings [1.5.4](#)
 - default value of [2.3.4, 2.3.7](#)
 - even or odd [1.4.2](#)
 - formatting [1.6.2](#)
 - from grouped elements [8.11](#)
 - in regular expressions [9.4.1](#)
 - non-negative [5.2.1, 7.5.2](#)
 - printing [1.6.2](#)

random [1.1.3](#), [3.1.1](#), [8.2](#), [8.4](#), [8.13](#)
reading/writing [9.1.5](#), [9.1.8](#), [9.1.9](#)
rounding [1.2.2](#), [1.4.4](#)
type conversions of [1.4.4](#)
unsigned [1.2.1](#), [1.4.3](#)
with fractional parts [1.2.2](#)

O

o [1.6.2](#)
Object class [4.3](#)
clone method [4.2.3](#), [4.3](#), [4.3.4](#), [4.7.3](#)
equals method [4.3](#), [4.3.2](#)
finalize method [4.3](#)
getClass method [4.2.1](#), [4.3](#), [4.3.2](#), [4.6.1](#),
[6.6.2](#), [6.7.1](#)
hashCode method [4.3](#), [4.3.2](#), [4.3.3](#)
notify, notifyAll methods [10.7.3](#)
toString method [4.3](#), [4.3.1](#)
wait method [10.7.3](#)
object references [2.1.2](#)
attempting to change [2.2.6](#)
comparing [4.3.2](#)
default value of [2.3.4](#), [2.3.7](#)
null [2.1.2](#)
passed by value [2.2.6](#)
serialization and [9.5.1](#)
object-oriented programming [2](#)
encapsulation in [12](#), [12.1](#)
object-relational mappers [12.6](#)
ObjectInputStream class [9.5.1](#)
defaultReadObject method [9.5.3](#), [9.5.6](#)
readDouble method [9.5.3](#)
readFields method [9.5.6](#)
readObject method [9.5.1](#), [9.5.7](#)
ObjectInputValidation interface [9.5.7](#)
ObjectOutputStream class [9.5.1](#)
defaultWriteObject method [9.5.3](#)
writeDouble method [9.5.3](#)
writeObject method [9.5.1](#), [9.5.3](#)
objects [1.1.1](#), [2.1](#)
calling methods on [1.1.3](#)
casting [3.1.5](#)
checkIndex method [5.1.9](#)
cloning [4.3.4](#)
comparing [1.8.4](#), [4.3.2](#)
constructing [1.1.3](#), [2.3](#), [4.7.4](#)

converting
to JSON [12.6](#)
to strings [4.3.1](#)
converting to streams [8.2](#)
deep/shallow copies of [4.3.4](#)
deserialized [9.5.5](#), [9.5.6](#)
equals method [4.3.2](#)
hash method [4.3.3](#)
immutable [2.1.1](#)
initializing variables with [1.3.1](#)
inspecting [4.7.2](#)
invoking static methods on [2.5.4](#)
isNull method [3.5.1](#)
mutable [2.3.6](#)
requireNonNull, requireNonNullXxx
methods [5.1.9](#)
serializable [9.5.1](#)
sorting [3.3.1](#)
state of [2.1](#)
ObjXxxConsumer interfaces [3.6.2](#)
octal numbers [1.2.1](#)
formatting for output [1.6.2](#)
octonions [1.5.6](#)
odd numbers [1.4.2](#)
of method
of EnumSet [7.5.3](#)
of IntStream [8.13](#)
of List [1.8.3](#), [1.8.6](#), [7.1](#), [7.6.1](#)
of LocalDate [2.1](#), [2.3.1](#)
of Map [7.4](#), [7.6.1](#)
of Optional [8.7.5](#)
of Path [9.2.1](#), [9.2.5](#)
of ProcessHandle [10.9.3](#)
of Set [7.6.1](#)
of Stream [8.2](#)
ofEntries method [7.6.1](#)
offer [10.3.2](#)
ofNullable method
of Optional [8.7.5](#)
of Stream [8.2](#), [8.7.7](#)
ofString method [9.3.2](#)
onExit method
of Process [10.9.2](#)
of ProcessHandle [10.9.3](#)
open keyword [9.1.10](#)
open keyword [12.6](#)
openConnection method [9.3.1](#)
opens keyword [12.6](#)

qualified [12.12](#)
openStream method [9.1.1](#)
Operation [4.4.3](#)
operations
 associative [8.12](#)
 atomic [10.2.2](#), [10.3.1](#), [10.6](#), [10.7.3](#)
 bulk [10.3.1](#)
 lazy [8.1](#), [8.2](#), [8.5](#), [9.4.5](#)
 parallel [10.4](#)
 performed optimistically [10.6](#)
 stateless [8.14](#)
 threadsafe [10.3](#)
operators [1.4](#)
 cast [1.4.4](#)
 precedence of [1.4](#)
option files [12.10](#)
Optional class [8.6](#), [8.7.6](#)
 creating values of [8.7.5](#)
 empty method [8.7.5](#)
 filter method [8.7.3](#)
 flatMap method [8.7.6](#), [8.7.7](#)
 for empty streams [8.12](#)
 for processes [10.9.3](#)
 get method [8.7.4](#), [8.7.7](#)
 ifPresent method [8.7.2](#)
 ifPresentOrElse method [8.7.2](#)
 isPresent method [8.7.4](#), [8.7.7](#)
 map method [8.7.3](#)
 of, ofNullable methods [8.7.5](#)
 or method [8.7.3](#)
 orElse method [8.6](#)
 orElseThrow method [8.7.1](#), [8.7.4](#)
 proper usage of [8.7.4](#)
 stream method [8.7.7](#)
OptionalXxx classes [8.13](#)
or method
 of BitSet [7.5.2](#)
 of Predicate, BiPredicate [3.6.2](#)
order [9.1.10](#)
ordinal [4.4.1](#)
orElseThrow method [8.7.1](#), [8.7.4](#)
org.omg.corba package [12.1](#)
orTimeout method [10.5.2](#)
os.arch, os.name, os.version system properties [7.5.1](#)
OSGi [12.1](#)
output
 formatted [1.6.2](#)
writing [9.1.6](#)
output streams [9.1](#)
 closing [9.1.3](#)
 obtaining [9.1.1](#)
 writing to [9.1.3](#)
OutputStream class [9.5.1](#)
 write method [9.1.3](#)
OutputStreamWriter class [9.1.6](#)
outputWriter method [10.9.1](#)
@Override annotation [4.1.3](#), [9.5.3](#), [9.5.4](#), [11.3](#), [11.3.1](#)
overriding [4.1.3](#)
 for logging/debugging [4.2.1](#)
overview.xhtml package [2.8.7](#)

P

package keyword [2.6.1](#)
package declarations [2.6.1](#)
package-info.java [2.8.7](#), [11.1.3](#)
packages [1.1.1](#), [2.6](#)
 accessing [2.6.4](#), [4.2.3](#), [12.1](#), [12.5](#), [12.6](#), [12.8](#)
 adding classes to [2.6.4](#)
 annotating [11.1.3](#)
 default [2.6.1](#)
 documentation comments for [2.8.1](#), [2.8.7](#)
 exporting [12.5](#), [12.6](#)
 naming [2.6.1](#)
 not nesting [2.6.1](#)
 split [12.7](#)
parallel [8.14](#)
parallel streams [10.4.1](#)
parallelStream method [7.1](#), [8.1](#), [8.14](#), [10.4.1](#)
parallelXxx methods (Arrays) [1.8.7](#), [10.4.2](#)
@param annotation [2.8.3](#)
Parameter class [4.7.4](#)
parameter variables [2.2.6](#)
 annotating [11.1.3](#)
 scope of [1.7.5](#)
Parameterized types. See Type parameters
ParameterizedType interface [6.7.2](#)
Parent classes. See Superclasses
parentLogger method (Driver) [12.11](#)
parseDouble method [1.5.4](#)
parseInt method [1.5.4](#), [5.1.2](#)
partitioning [10.2.3](#)
partitioningBy method [8.10](#), [8.11](#)

Pascal triangle [1.8.9](#)
passwords [1.6.1](#)
Path interface [3.2.1, 9.2.1](#)
 get method [9.2.1](#)
 getXXX methods [9.2.1](#)
 normalize method [9.2.1](#)
 of method [9.2.1, 9.2.5](#)
 relativize method [9.2.1](#)
 resolve, resolveSibling methods [9.2.1](#)
 subpath method [9.2.1](#)
 toAbsolutePath, toFile methods [9.2.1](#)
path separators [9.2.1](#)
path.separator package [7.5.1](#)
Paths class [3.2.1, 9.2.1](#)
 absolute vs. relative [9.2.1](#)
 combining [9.2.1](#)
 filtering [9.2.4](#)
 resolving [9.2.1](#)
 taking apart [9.2.1](#)
Pattern class
 asMatchPredicate, asPredicate
 methods [9.4.2](#)
 compile method [9.4.2, 9.4.7](#)
 flags [9.4.7](#)
 matcher, matches methods [9.4.2](#)
 split method [9.4.5](#)
 splitAsStream method [8.2, 9.4.5](#)
 splitWithDelimiters method [9.4.5](#)
pattern variables [5.3.5](#)
Pattern.quote method [9.4.1](#)
PECS (producer extends, consumer
super) [6.4.2](#)
peek method
 of BlockingQueue [10.3.2](#)
 of Stream [8.5](#)
performance
 atomic operations and [10.6](#)
 big numbers and [1.4.6](#)
 combined operators and [1.4.2](#)
 memory caching and [10.2.1](#)
permits keyword [4.2.4](#)
@Persistent annotation [11.3.2](#)
PI [1.4.3, 2.5.2, 2.6.6](#)
Picocli [11](#)
platform class loader [4.6.3](#)
platform logging API [5.3.1, 5.3.4](#)
Platform threads [10.1.1](#)
plugins, loading [4.6.3](#)
plus, plusXXX methods
 of LocalDate [2.1, 2.1.1, 2.1.2](#)
Point class [4.3.1](#)
poll [10.3.2](#)
pollXXX methods (NavigableSet) [7.3](#)
pools, for parallel streams [8.14](#)
pop [7.5.4](#)
POSITIVE_INFINITY [1.2.2](#)
POST requests [9.3.2](#)
postVisitDirectory method [9.2.4](#)
pow [1.4.3, 2.5.4, 2.6.6](#)
predefined character classes [9.4.1](#)
Predicate interface [3.4.2, 3.6.2](#)
 and method [3.6.2](#)
 isEqual method [3.6.2](#)
 or, negate methods [3.6.2](#)
 test method [3.6.2, 6.4.2](#)
predicate functions [8.10](#)
previous method
 of ListIterator [7.2](#)
previousClearBit method [7.5.2](#)
previousSetBit method [7.5.2](#)
preVisitDirectory method [9.2.4](#)
primitive types [1.2](#)
 comparing [4.3.2](#)
 converting to strings [4.3.1](#)
 functions interfaces for [3.6.2](#)
 passed by value [2.2.6](#)
 streams of [8.12, 8.13](#)
 type parameters and [6.6.1](#)
 variables of, no updating for [2.2.6](#)
 wrapper classes for [1.8.4](#)
printStackTrace method [5.1.8](#)
PrintStream class [1.1.3, 4.3.1, 9.1.6](#)
 print method [1.1.3, 1.6.2, 5.3, 9.1.6](#)
 printf method [1.6.2, 1.9.3, 9.1.6](#)
 println method [1.1.3, 1.6.1, 1.6.2, 1.8.7, 3.5.1, 9.1.6](#)
PrintWriter class [9.1.6](#)
 close method [5.1.5](#)
 print method [9.1.6](#)
 printf method [9.1.6](#)
 println method [9.1.6](#)
priority queues [7.5.4](#)
private keyword [1.1.1, 2.6.4](#)
 for enum constructors [4.4.2](#)
Process class [10.9](#)
 destroy, destroyForcibly methods [10.9.2](#)

errorReader method [10.9.1](#)
exitValue method [10.9.2](#)
getErrorStream method [10.9.1](#)
getInputStream, getOutputStream
methods [10.9.1](#)
inputReader method [10.9.1](#)
isAlive method [10.9.2](#)
onExit method [10.9.2](#)
outputWriter method [10.9.1](#)
supportsNormalTermination
method [10.9.2](#)
toHandle method [10.9.3](#)
waitFor method [10.9.2](#)
ProcessBuilder class [10.9](#)
directory method [10.9.1](#)
redirectXxx methods [10.9.1](#)
start, startPipeline methods [10.9.1](#)
processes [10.9](#)
building [10.9.1](#)
getting info about [10.9.3](#)
killing [10.9.2](#)
running [10.9.2](#)
ProcessHandle interface [10.9.3](#)
allProcesses method [10.9.3](#)
current method [10.9.3](#)
destroy, destroyForcibly methods [10.9.3](#)
info method [10.9.3](#)
isAlive method [10.9.3](#)
of method [10.9.3](#)
onExit method [10.9.3](#)
supportsNormalTermination
method [10.9.3](#)
processing pipeline [10.5.2, 10.9.1](#)
Processor interface [11.5.1](#)
programming languages
functional [3](#)
object-oriented [1.1.1](#)
programs
compiling [1.1.2](#)
configuration options for [7.5.1](#)
packaging [12.14](#)
responsive [10.5.3](#)
running [1.1.2](#)
testing [5.2](#)
promises (in concurrent libraries) [10.5.1](#)
properties [4.7.5, 7.5.1](#)
loading from file [7.5.1](#)
naming [4.7.5](#)
read-only/write-only [4.7.5](#)
testing for [6.4.2](#)
property files
encoding [7.5.1](#)
generating [11.5.3](#)
protected keyword [4.2.3](#)
Provider.get, Provider.type methods [4.6.5](#)
provides keyword [12.13](#)
Proxy class [4.7.7](#)
newProxyInstance method [4.7.7](#)
public keyword [1.1.1, 2.6.4](#)
for interface methods [3.1.2, 3.1.3](#)
method overriding and [4.1.3](#)
push [7.5.4](#)
put method
of BlockingQueue [10.3.2](#)
of FileChannel [9.1.10](#)
of Map [7.4](#)
putAll method [7.4](#)
putBoolean method
of FileChannel [9.1.10](#)
putByte method [9.1.10](#)
putChar method [9.1.10](#)
putDouble, putFloat methods
of FileChannel [9.1.10](#)
putIfAbsent method
of ConcurrentHashMap [10.3.1](#)
of Map [7.4](#)
putInt, putLong methods
of FileChannel [9.1.10](#)
putShort method [9.1.10](#)

Q

qualified exports [12.12](#)
Queue class [7.1, 7.5.4](#)
synchronizing methods in [10.7.3](#)
using ArrayDeque with [7.5.4](#)
quote method (Pattern) [9.4.1](#)
quoteReplacement method [9.4.6](#)

R

race conditions [8.14, 10.2.2](#)
Random class [1.1.3, 3.1.1](#)
nextInt method [1.1.3](#)
random numbers [1.1.3, 3.1.1](#)

streams of [8.2](#), [8.4](#), [8.13](#)
RandomAccess interface [7.1](#)
RandomAccessFile class [9.1.9](#)
 getFilePointer method [9.1.9](#)
 length method [9.1.9](#)
 seek method [9.1.9](#)
RandomGenerator interface [3.1.1](#)
 getDefaultValue method [1.1.3](#)
 methods of [8.13](#)
RandomNumbers [2.5.4](#)
range [7.5.3](#)
range, rangeClosed methods
 (XxxStream) [8.13](#)
ranges [7.6.2](#)
 converting to streams [8.14](#)
raw types [6.5.1](#), [6.6.2](#)
read method
 of Files [9.1.2](#)
 of InputStream [9.1.2](#)
 of InputStreamReader [9.1.5](#)
readAllXxx methods (Files) [9.1.2](#), [9.1.5](#)
readByte, readChar methods
 (DataInput) [9.1.8](#)
readDouble method
 of DataInput [9.1.8](#)
 of ObjectInputStream [9.5.3](#)
Reader class [9.1.5](#)
readers [9.1](#)
readExternal method [9.5.4](#)
readFields method [9.5.6](#)
readFloat, readFully methods
 (DataInput) [9.1.8](#)
readInt method [9.1.8](#), [9.1.9](#)
readLine method
 of BufferedReader [9.1.5](#)
 of Console [1.6.1](#)
readLong method [9.1.8](#)
readNBytes method [9.1.2](#)
readObject method
 of HashSet [9.5.3](#)
 of ObjectInputStream [9.5.1](#), [9.5.7](#)
readPassword method [1.6.1](#)
readResolve method [9.5.5](#)
readShort method [9.1.8](#)
readUnsignedXxx, readUTF methods
 (DataOutput) [9.1.8](#)
receiver parameters [2.2.4](#), [11.1.5](#)
records [2.4](#)
customizing serialization of [9.5.4](#)
 Serializable [9.5.7](#)
redirection syntax [1.6.1](#)
redirectXxx methods
 (ProcessBuilder) [10.9.1](#)
reduce [8.12](#)
reduceXxx methods
 (ConcurrentHashMap) [10.3.1](#)
reducing [8.11](#)
reductions [8.6](#), [8.12](#)
ReentrantLock class [10.7.1](#)
 lock, unlock methods [10.7.1](#)
reflection [4.7](#)
 generic types and [6.6.3](#), [6.7](#)
 module system and [4.7.1](#), [4.7.2](#), [12.6](#), [12.10](#)
 processing annotations with [11.4](#)
 security and [9.5.7](#)
ReflectiveOperationException class [4.6.1](#)
regular expressions [9.4](#)
 flags for [9.4.7](#)
 groups in [9.4.4](#)
 replacing matches with [9.4.6](#)
 splitting input with [9.4.5](#)
 testing matches of [9.4.2](#), [9.4.3](#)
 turning into predicates [9.4.2](#)
relational operators [1.4.5](#)
relativize [9.2.1](#)
remainderUnsigned method [1.4.3](#)
remove method
 of ArrayDeque [7.5.4](#)
 of ArrayList [1.8.3](#)
 of BlockingQueue [10.3.2](#)
 of Collection [7.1](#)
 of Iterator [7.2](#)
 of List [7.1](#)
 of Map [7.4](#)
removeAll method [7.1](#)
removeIf method
 of ArrayList [3.4.2](#)
 of Collection [7.1](#)
 of Iterator [7.2](#)
@Repeatable annotation [11.3](#), [11.3.2](#)
@RepeatedTest annotation [11.1.1](#)
replace method
 of Map [7.4](#)
 of String [1.5.5](#)
replaceAll method
 of Collections [7.1](#)

of List [7.1](#)
of Map [7.4](#)
of Matcher [9.4.6](#)
of String [9.4.6](#)
replaceFirst method [9.4.6](#)
requireNonNull, requireNonNullXxx methods
(Objects) [5.1.9](#)
requires keyword [12.3, 12.4, 12.5, 12.8,](#)
[12.11](#)
Reserved words. See Keywords
resolve, resolveSibling methods
(Path) [9.2.1](#)
resources [4.6](#)
loading [4.6.2, 12.6](#)
managing [5.1.5](#)
resume [10.8.1](#)
retainAll method [7.1](#)
@Retention annotation [11.2, 11.3](#)
return keyword [1.9.1, 2.2.3, 2.8.3](#)
in finally blocks [5.1.6](#)
in lambda expressions [3.4.1](#)
return types, covariant [4.1.3, 6.5.3](#)
return values
as arrays [1.9.2](#)
missing [8.6](#)
providing type of [1.9.1](#)
reverse [1.8.7, 7.1](#)
reverse domain name convention [2.6.1, 12.2](#)
reversed [3.8.2](#)
reverseOrder method [3.8.3](#)
rotate [7.1](#)
round [1.4.4](#)
RoundEnvironment interface [11.5.2](#)
roundoff errors [1.2.2](#)
RowSetProvider class [12.10](#)
rt.jar package [12.14](#)
runAfterXxx methods
(CompletableFuture) [10.5.2](#)
Runnable interface [3.3.3, 3.6.2, 10.1.1, 10.1.2](#)
executing on the UI thread [10.5.3](#)
run method [3.6.2, 10.1.1, 10.1.3, 10.8.1](#)
using class literals with [4.6.1](#)
runtime
availableProcessors method [10.1.1](#)
exec method [10.9](#)
raw types at [6.6.2](#)
safety checks at [6.5.2](#)
runtime image file [12.14](#)

RuntimeException class [5.1.2](#)

S

s, S conversion characters [1.6.2](#)
safety checks, as runtime [6.5.2](#)
@SafeVarargs annotation [6.6.4, 11.3, 11.3.1](#)
sample code [1.1.2](#)
Scala [6.4.3](#)
Scanner class [1.6.1](#)
findAll method [9.4.3](#)
hasNext, hasNextXxx methods [1.6.1, 9.1.5](#)
next, nextXxx methods [1.6.1, 9.1.5](#)
tokens method [8.2, 9.1.5](#)
sealed keyword [4.2.4](#)
sealed types [4.2.4](#)
searchXxx methods
(ConcurrentHashMap) [10.3.1](#)
security [2.6.4, 9.5.7](#)
SecurityException class [4.7.2](#)
@see annotation [2.8.6](#)
seek [9.1.9](#)
sequences, producing [8.2](#)
@Serial annotation [9.5.3, 9.5.4, 11.3, 11.3.1](#)
serial numbers [9.5.1](#)
Serializable interface [9.5.1](#)
readResolve, writeReplace methods [9.5.5](#)
serialization [9.5](#)
filters for [9.5.7](#)
serialVersionUID method [9.5.6](#)
server-side software [9.5](#)
ServiceLoader class [4.6.5, 12.13](#)
iterator method [4.6.5](#)
load method [4.6.5, 12.13](#)
ServiceLoader.Provider interface [4.6.5](#)
services
configurable [4.6.5](#)
loading [4.6.5, 12.13](#)
ServletException [5.1.7](#)
set method [7.1, 10.3.3](#)
of Array [4.7.6](#)
of ArrayList [1.8.3](#)
of BitSet [7.5.2](#)
of Field [4.7.4](#)
of List [7.1](#)
of ListIterator [7.2](#)
of method [7.6.1](#)

working with EnumSet [7.5.3](#)
setAccessible method [4.7.2, 4.7.4](#)
setAll method [3.6.1](#)
setBoolean, setByte, setChar methods
 of Array [4.7.6](#)
 of Field [4.7.4](#)
setClassAssertionStatus method [5.2.2](#)
setContextClassLoader method [4.6.4](#)
setDaemon method [10.8.3](#)
setDefaultAssertionStatus method [5.2.2](#)
setDefaultUncaughtExceptionHandler
 method [5.1.8](#)
setDoOutput method [9.3.1](#)
setDouble, setFloat, setInt, setLong
 methods
 of Array [4.7.6](#)
 of Field [4.7.4](#)
setOut method [2.5.2](#)
setPackageAssertionStatus method [5.2.2](#)
setProperty method [5.3.4](#)
setRequestProperty method [9.3.1](#)
sets [7.3](#)
 immutable [10.2.4](#)
 threadsafe [10.3.3](#)
 unmodifiable views of [7.6.3](#)
setShort method
 of Array [4.7.6](#)
 of Field [4.7.4](#)
setUncaughtExceptionHandler
 method [10.8.1](#)
shallow copies [4.3.4](#)
shared variables [10.2.1, 10.2.3](#)
 atomic mutations of [10.6](#)
 locking [10.7.1](#)
shell
 redirection syntax of [1.6.1](#)
 scripts for, generating [11.5.3](#)
shift operators [1.4.5](#)
Shift_JIS encoding [9.1.4](#)
short type [1.2.1, 1.8.4](#)
 MAX_VALUE, MIN_VALUE constants [1.2.1](#)
 streams of [8.13](#)
 type conversions of [1.4.4](#)
short circuit evaluation [1.4.5](#)
short-term persistence [9.5.6](#)
shuffle [1.8.7, 7.1](#)
SimpleFileVisitor class [9.2.4](#)
@since annotation [2.8.5](#)
singletons [9.5.5](#)
size method
 of ArrayList [1.8.3](#)
 of Collection [7.1](#)
 of Map [7.4](#)
skip [8.4](#)
skipNBytes method [9.1.2](#)
sleep [10.1.3](#)
SLF4J [5.3.1, 12.2](#)
SOAP protocol [12.1](#)
SocketHandler class [5.3.5](#)
sort method
 of Arrays [1.8.7, 3.3.1, 3.3.2, 3.4.2, 3.5.1](#)
 of Collections [1.8.7, 6.4.3, 6.7.2, 7.1](#)
 of List [7.1](#)
sorted [8.5](#)
sorted maps [7.6.2, 7.6.3](#)
sorted sets [7.1, 7.6.2](#)
 traversing [7.3](#)
 unmodifiable views of [7.6.3](#)
sorted streams [8.14](#)
SortedMap interface [7.6.2](#)
SortedSet interface [7.1, 7.3](#)
 first method [7.3](#)
 headSet method [7.3, 7.6.2](#)
 last method [7.3](#)
 subSet, tailSet methods [7.3, 7.6.2](#)
sorting
 array lists [1.8.7](#)
 arrays [1.8.7, 3.3.1](#)
 chaining comparators for [3.8.3](#)
 changing order of [3.8.1](#)
 streams [8.5](#)
 strings [1.5.3, 3.5.1](#)
source code, generating [11.3, 11.3.1, 11.5.3](#)
source files
 documentation comments for [2.8.7](#)
 placing, in a file system [2.6.1](#)
space flag (for output) [1.6.2](#)
spaces
 in regular expressions [9.4.1](#)
 removing [1.5.5](#)
split method
 of Pattern [9.4.5](#)
 of String [1.5.2, 9.4.5](#)
splitAsStream method [8.2, 9.4.5](#)
spliterator [7.1](#)
Spliterators class

spliteratorUnknownSize method [8.2](#)
splitWithDelimiters method [9.4.5](#)
SQL [1.5.7](#)
sqrt [1.4.3](#)
square root, computing [8.7.6](#)
Stack class [7.5.4](#)
stack trace [5.1.8, 5.1.9](#)
StackWalker class [5.1.8](#)
standard output [1.1.1](#)
StandardCharsets class [9.1.4](#)
start method
 of Matcher, MatchResult [9.4.3, 9.4.4](#)
 of ProcessBuilder [10.9.2](#)
 of Thread [10.8.1](#)
startPipeline method [10.9.1](#)
startsWith method [1.5.5](#)
startVirtualThread method
 of Thread [10.8.1](#)
stateless operations [8.14](#)
statements, combining [1.8.1](#)
static keyword [1.1.1, 1.3.4, 1.9.1, 2.5, 4.4.4](#)
 for modules [12.11](#)
static constants [2.5.2](#)
static imports [2.6.6](#)
static initialization [4.6.3](#)
static methods [1.9.1, 2.5.4, 2.5.5](#)
 accessing static variables from [2.5.4](#)
 importing [2.6.6](#)
 in interfaces [3.2.1, 3.2.2](#)
static nested classes [2.7.1](#)
static variables [2.5.1](#)
 accessing from static methods [2.5.4](#)
 importing [2.6.6](#)
 visibility of [10.2.1](#)
stop [10.8.1](#)
Stream interface
 anyMatch method [8.6](#)
 collect method [8.8, 8.12](#)
 concat method [8.4](#)
 count method [8.1, 8.6](#)
 distinct method [8.5, 8.14](#)
 dropWhile method [8.4](#)
 empty method [8.2](#)
 filter method [8.1, 8.3, 8.6](#)
 findAny method [8.6](#)
 findFirst method [4.6.5, 8.6](#)
 flatMap method [8.3](#)
 forEach, forEachOrdered methods [8.8](#)
 generate method [8.2, 8.13](#)
 iterate method [8.2, 8.5, 8.13, 10.4.1](#)
 iterator method [8.8](#)
 limit method [8.4, 8.14](#)
 map method [8.3](#)
 mapToInt method [8.12](#)
 max, min methods [8.6](#)
 noneMatch method [8.6](#)
 of Arrays [8.2, 8.13](#)
 of BitSet [7.5.2](#)
 of Collection [7.1, 8.1](#)
 of method [8.2](#)
 of Optional [8.7.7](#)
 of StreamSupport [8.2](#)
 ofNullable method [8.2, 8.7.7](#)
 peek method [8.5](#)
 reduce method [8.12](#)
 skip method [8.4](#)
 sorted method [8.5](#)
 takeWhile method [8.4](#)
 toArray method [3.5.2, 8.8](#)
 toList method [8.2](#)
 unordered method [8.14](#)
streams [8, 8.2](#)
 collecting elements of [8.8, 8.9](#)
 combining [8.4](#)
 computing values from [8.12](#)
 converting to/from arrays [8.2, 8.8, 8.14, 10.4.2](#)
 creating [8.2](#)
 debugging [8.5](#)
 empty [8.2, 8.6, 8.12](#)
 filtering [8.7.7](#)
 finite [8.2](#)
 flattening [8.3, 8.7.6](#)
 infinite [8.1, 8.2, 8.4, 8.5](#)
 intermediate operations for [8.1](#)
 locating services with [4.6.5](#)
 noninterference of [8.2](#)
 of primitive type values [8.12, 8.13](#)
 of random numbers [8.13](#)
 ordered [8.14](#)
 parallel [8.1, 8.6, 8.8, 8.9, 8.10, 8.12, 8.14, 10.4.1](#)
 processed lazily [8.1, 8.2, 8.5](#)
 reductions of [8.6](#)
 removing duplicates from [8.5](#)
 returned by Files.lines [8.14](#)

sorting [8.5](#)
splitting [8.4](#)
summarizing [8.8, 8.13](#)
terminal operation for [8.1, 8.6](#)
transformations of [8.3, 8.13](#)
vs. collections [8.1](#)

StreamSupport class
 stream method [8.2](#)

String class [1.1.3, 1.5.5](#)
 charAt method [1.5.6](#)
 compareTo method [1.5.3, 3.3.1](#)
 compareToIgnoreCase method [3.5.1](#)
 contains method [1.5.5](#)
 endsWith method [1.5.5](#)
 equals method [1.5.3](#)
 equalsIgnoreCase method [1.5.3](#)
 final [4.2.1](#)
 formatted method [1.6.2](#)
 graphemeClusters method [8.4](#)
 hash codes [4.3.3](#)
 immutability of [1.5.5, 10.2.4](#)
 indexOf, lastIndexOf methods [1.5.5](#)
 join method [1.5.1](#)
 length method [1.1.3, 1.5.6](#)
 replace method [1.5.5](#)
 replaceAll method [9.4.6](#)
 split method [1.5.2, 9.4.5](#)
 startsWith method [1.5.5](#)
 substring method [1.5.2](#)
 toLowerCase method [1.5.5, 8.3](#)
 toUpperCase method [1.5.5](#)

StringBuilder class [1.5.1](#)
strings [1.1.3, 1.5](#)
 comparing [1.5.3](#)
 concatenating [1.5.1, 4.3.1](#)
 converting
 from byte arrays [9.1.4](#)
 from objects [4.3.1](#)
 to numbers [1.5.4](#)
 converting to code points [8.3](#)
 empty [1.5.3, 1.5.4, 4.3.1](#)
 formatting for output [1.6.2](#)
 internal representation of [1.5.6](#)
 sorting [1.5.3, 3.5.1](#)
 splitting [1.5.2, 8.2](#)
 transforming to lower/uppercase [8.3](#)

StringWriter class [9.1.6](#)
strip [1.5.5](#)

strong [2.8.1](#)
subclasses [4.1.1](#)
 anonymous [4.1.7, 4.4.3](#)
 calling `toString` method in [4.3.1](#)
 constructors for [4.1.4](#)
 inheriting annotations [11.3](#)
 initializing instance variables in [4.1.4](#)
 methods in [4.1.2](#)
 preventing [4.2.1](#)
 public [4.1.3](#)
 superclass assignments in [4.1.5](#)

subList method [7.1, 7.6.2](#)
subMap method [7.6.2](#)
subpath [9.2.1](#)
subSet method
 of `NavigableSet` [7.3](#)
 of `SortedSet` [7.3, 7.6.2](#)

substring [1.5.2](#)
subtractExact method [1.4.3](#)
subtraction [1.4.2](#)
 accurate [1.4.6](#)
 not associative [8.12](#)

subtypes [3.1.4](#)
 wildcards for [6.4.1](#)

sum method
 of `LongAdder` [10.6](#)
 of `XxxStream` [8.13](#)

summarizingXxx methods (`Collectors`) [8.8, 8.11](#)

summaryStatistics method [8.13](#)
summingXxx methods (`Collectors`) [8.11](#)

super keyword [3.2.3, 4.1.3, 4.1.4, 4.1.8, 6.4.2, 6.4.3](#)

superclasses [4.1.1](#)
 annotating [11.1.4](#)
 calling `equals` method on [4.3.2](#)
 default methods of [4.2.5](#)
 methods of [4.1.3](#)
 public [4.1.3](#)
 serializability of [9.5.1](#)

supertypes [3.1.4, 3.1.5, 3.1.8](#)
 wildcards for [6.4.2](#)

Supplier interface [3.6.2, 10.5.1](#)
supplyAsync method [10.5.1](#)

supportsNormalTermination method
 of `Process` [10.9.2](#)
 of `ProcessHandle` [10.9.3](#)

@SuppressWarnings annotation [6.6.2](#), [11.3](#),
[11.3.1](#), [11.3.2](#), [12.3](#)
suspend [10.8.1](#)
swap [7.1](#)
Swing GUI toolkit [3.3.4](#), [10.5.3](#)
SwingConstants interface [3.1.9](#)
SwingWorker class [10.5.3](#)
switch keyword [1.7.2](#)
enhanced [1.7.2](#), [4.5.4](#)
exhaustive [1.7.2](#)
fall-through variant of [1.7.2](#)
using enumerations in [4.4.5](#)
with pattern matching [4.2.4](#)
symbolic links [9.2.3](#), [9.2.4](#)
synchronized keyword [10.7.1](#), [10.7.3](#)
synchronized views [7.6.3](#)
synchronizedXxx methods
(Collections) [7.1](#)
System class
getLogger method [5.3.2](#), [5.3.3](#)
getProperties method [7.5.1](#)
getProperty method [4.6.3](#), [5.1.9](#), [7.5.1](#)
setOut method [2.5.2](#)
setProperty method [5.3.4](#)
system class loader [4.6.3](#), [4.6.4](#)
system classes, enabling/disabling assertions
for [5.2.2](#)
system properties [7.5.1](#)
System.err [5.1.8](#), [5.3.5](#), [10.8.3](#)
System.in [1.6.1](#)
System.Logger interface [5.3.2](#), [5.3.3](#)
getName method [5.3.3](#)
isLoggable method [5.3.3](#)
log method [5.3.2](#), [5.3.3](#)
System.Logger.Level enumeration [5.3.3](#)
System.out [1.1.3](#), [1.3.4](#), [1.6.1](#), [1.6.2](#), [1.8.7](#),
[1.9.3](#), [2.5.2](#), [3.5.1](#), [5.3](#), [9.1.6](#)

T

t, T conversion characters [1.6.2](#)
tab completion [1.1.4](#)
tagging interfaces [4.3.4](#)
tailMap method [7.6.2](#)
tailSet method
of NavigableSet [7.3](#)
of SortedSet [7.3](#), [7.6.2](#)

take [10.3.2](#)
takeWhile method [8.4](#)
tar [2.6.2](#)
@Target annotation [11.2](#), [11.3](#)
tasks [10.1](#)
canceling [10.1.2](#)
combining results from [10.1.2](#)
computationally intensive [10.1.1](#)
coordinating work between [10.3.2](#)
defining [3.3.3](#)
executing [3.3.3](#), [10.1.1](#)
groups of [10.8.3](#)
long-running [10.5.3](#)
running [10.1.1](#)
submitting [10.1.2](#)
vs. threads [10.1.1](#)
working simultaneously [10.5.1](#)
TAU [1.4.3](#)
teeing [8.11](#)
terminal window [1.1.2](#)
test method [11.1](#), [11.2](#)
of BiPredicate [3.6.2](#)
of Predicate [3.6.2](#), [6.4.2](#)
of XxxPredicate [3.6.2](#)
text blocks [1.5.7](#)
thenAccept method [10.5.1](#), [10.5.2](#)
thenAcceptBoth method [10.5.2](#)
thenApply, thenApplyAsync methods
(CompletableFuture) [10.5.2](#)
thenCombine method [10.5.2](#)
thenComparing method [3.8.3](#)
thenCompose method [10.5.2](#)
thenRun method [10.5.2](#)
third-party libraries [12.8](#)
this keyword [2.2.5](#)
annotating [11.1.5](#)
in constructors [2.3.3](#), [10.2.4](#)
in lambda expressions [3.7.1](#)
in method references [3.5.1](#)
Thread class
getContextClassLoader method [4.6.4](#)
interrupted, isInterrupted
methods [10.1.3](#)
isAlive method [10.8.1](#)
join method [10.8.1](#)
properties [10.8.3](#)
resume method (deprecated) [10.8.1](#)
setContextClassLoader method [4.6.4](#)

setDaemon method [10.8.3](#)
setDefaultUncaughtExceptionHandler method [5.1.8](#)
setUncaughtExceptionHandler method [10.8.1](#)
sleep method [10.1.3](#)
start method [10.1.1, 10.8.1](#)
startVirtualThread method [10.8.1](#)
stop, suspend methods (deprecated) [10.8.1](#)
ThreadLocal class [10.8.2](#)
ThreadLocalRandom.current method [10.8.2](#)
threads [10.1.1, 10.8](#)
 atomic mutations in [10.6](#)
 creating [3.3.3](#)
 daemon [10.8.3](#)
 groups of [10.8.3](#)
 interrupting [10.1.2, 10.1.3](#)
 locking [10.7.1](#)
 names of [10.8.3](#)
 platform [10.1.1](#)
 priorities of [10.8.3](#)
 race conditions in [8.14, 10.2.2](#)
 running tasks in [3.3.3](#)
 starting [10.8.1](#)
 states of [10.8.3](#)
 terminating [10.1.1](#)
 uncaught exception handlers of [10.8.3](#)
 virtual [10.1.1](#)
 visibility and [10.2.1, 10.7.2](#)
 vs. tasks [10.1.1](#)
 waiting on conditions [10.7.3](#)
 worker [10.5.3](#)
throw keyword [5.1.1](#)
Throwable class [5.1.2](#)
 in assertions [5.2.1](#)
 initCause method [5.1.7](#)
 no generic subtypes for [6.6.7](#)
 printStackTrace method [5.1.8](#)
@throws annotation [2.8.3, 5.1.3](#)
 type variables in [6.6.7](#)
TimeoutException class [10.1.2](#)
Timestamp class [4.3.2](#)
toAbsolutePath method [9.2.1](#)
toArray method
 of Collection [7.1](#)
 of Stream [3.5.2, 8.8](#)
 of XxxStream [8.13](#)
toByteArray method
of BitSet [7.5.2](#)
of ByteArrayOutputStream [9.1.1](#)
toCollection method [8.8](#)
toConcurrentMap method [8.9](#)
ToDoubleFunction interface [3.6.2, 6.6.1](#)
toFile method [9.2.1](#)
toGenericString method [4.6.1](#)
toHandle method [10.9.3](#)
toIntExact method [1.4.4](#)
ToIntFunction interface [3.6.2, 6.6.1](#)
tokens [8.2, 9.1.5](#)
toList method [8.2](#)
toLongArray method [7.5.2](#)
ToLongFunction interface [3.6.2, 6.6.1](#)
toLowerCase method [1.5.5, 8.3](#)
toMap method [8.9](#)
toPath method [9.2.1](#)
toSet method [8.8, 8.11](#)
toString method
 calling from subclasses [4.3.1](#)
 of Arrays [1.8.7, 4.3.1](#)
 of BitSet [7.5.2](#)
 of Class [4.6.1](#)
 of Double [1.5.4](#)
 of Enum [4.4.1](#)
 of Integer [1.5.4](#)
 of Modifier [4.6.1](#)
 of Object [4.3, 4.3.1](#)
 of Point [4.3.1](#)
 of records [2.4.1](#)
toUnsignedInt method [1.2.1](#)
toUpperCase method [1.5.5](#)
transferTo method [9.1.3](#)
transient keyword [9.5.2, 9.5.3](#)
transitive keyword [12.11](#)
TreeMap class [7.4, 8.9](#)
TreeSet class [7.3](#)
Troubleshooting. See Debugging
true literal [1.2.4](#)
try keyword [5.1.4, 5.1.6](#)
 for visiting directories [9.2.4](#)
try-with-resources [5.1.5](#)
 closing output streams with [9.1.3](#)
 for file locking [9.1.11](#)
tryLock method [9.1.11](#)
trySetAccessible method [4.7.2](#)
Type interface [6.7.2](#)
type bounds [6.3, 6.7.2](#)

annotating [11.1.4](#)
type erasure [6.5, 6.6.5](#)
 clashes after [6.6.6](#)
type method
 (*ServiceLoader.Provider*) [4.6.5](#)
type parameters [3.3.1, 6.1](#)
 annotating [11.1.3](#)
 primitive types and [6.1, 6.6.1](#)
type variables
 exceptions and [6.6.7](#)
 in static context [6.6.5](#)
 no instantiating of [6.6.3](#)
 wildcards with [6.4.3](#)
TypeElement interface [11.5.2](#)
TypeVariable interface [6.7.2](#)

U

U+ [1.5.6](#)
UnaryOperator interface [3.6.2](#)
uncaught exception handlers [10.8.1, 10.8.3](#)
unchecked exceptions [5.1.2](#)
 documenting [5.1.3](#)
 generic types and [6.6.7](#)
UncheckedIOException class [9.1.5](#)
Unicode [1.5.6, 8.13, 9.1.4](#)
 replacement character in [9.1.6](#)
Unit tests [11](#)
Unix
 executable files in [1.1.2](#)
 path separator in [2.6.3, 7.5.1](#)
 wildcard in classpath in [2.6.3](#)
unlock [10.7.1](#)
unmodifiableXxx methods
 (*Collections*) [7.1](#)
unordered [8.14](#)
updateAndGet method [10.6](#)
URI class [9.3.2](#)
URL class [9.3.2](#)
 final [4.2.1](#)
 getInputStream method [9.3](#)
 openConnection method [9.3.1](#)
 openStream method [9.1.1](#)
URLClassLoader class [4.6.3](#)
URLConnection class [9.3.1](#)
 connect method [9.3.1](#)
 getHeaderFields method [9.3.1](#)

getInputStream method [9.3.1](#)
getOutputStream method [9.3.1](#)
setDoOutput method [9.3.1](#)
setRequestProperty method [9.3.1](#)
URLs, reading from [9.1.1, 9.3](#)
user.dir, *user.home*, *user.name* system
 properties [7.5.1](#)
uses keyword [12.13](#)
UTF-16 [1.2.3, 1.5.6, 8.13, 9.1.4](#)
 in regular expressions [9.4.1](#)
UTF-8 [9.1.4](#)
 modified [9.1.8](#)
Util.createInstance [4.6.4](#)
utility classes [2.6.4, 4.6.4](#)

V

validateObject method [9.5.7](#)
valueOf method
 of *BitSet* [7.5.2](#)
 of *Enum* [4.4.1](#)
values method
 of *Enum* [4.4.1](#)
 of *Map* [7.4, 7.6](#)
var keyword [1.3.1, 1.3.3](#)
varargs parameters
 declaring [1.9.3](#)
 safety of [11.3, 11.3.1](#)
VarHandle class [12.6](#)
variable handles [12.6](#)
VariableElement interface [11.5.2](#)
variables [1.1.3, 1.3](#)
 atomic mutations of [10.6](#)
 capturing, in lambda expressions [3.7.2](#)
 declaring [1.3.1, 1.3.3](#)
 defined in interfaces [3.1.9](#)
 deprecated [2.8.5, 11.3, 11.3.1](#)
 documentation comments for [2.8.1, 2.8.4](#)
 effectively final [3.7.2](#)
 final [10.2.1, 10.2.4](#)
 holding object references [2.1.2](#)
 initializing [1.3.1, 1.3.4](#)
 instance [2.2.1, 2.2.5, 2.3.1, 2.3.6, 2.4.1, 2.4.2, 2.5.4, 4.1.4, 4.2.2, 4.2.3, 4.3.2, 9.5.2, 9.5.5, 9.5.6](#)
 local [1.7.5](#)
 naming [1.3.2](#)

parameter [2.2.6](#)
private [2.2.1](#), [2.6.4](#)
public static final [3.1.9](#)
redefining [1.7.5](#)
scope of [1.7.5](#), [2.6.4](#)
shared [10.2.1](#), [10.2.3](#), [10.7.1](#)
static [2.5.1](#), [2.5.4](#), [2.6.6](#), [10.2.1](#)
thread-local [10.8.2](#)
using an abstract class as type of [4.2.2](#)
visibility of [10.2.1](#), [10.7.2](#)
volatile [10.2.1](#), [10.2.2](#)
@version annotation [2.8.2](#), [2.8.8](#)
versioning [9.5.6](#)
views [7.6](#)
virtual machine [1.1.2](#)
instruction reordering in [10.2.1](#)
Virtual threads [10.1.1](#)
visibility [10.2.1](#)
guaranteed with locks [10.7.2](#)
visitFile, visitFileFailed methods
(FileVisitor) [9.2.4](#)
void keyword [1.1.1](#), [1.9.1](#)
using class literals with [4.6.1](#)
volatile keyword [10.2.1](#), [10.2.2](#)

W

wait [10.7.3](#)
waitFor method [10.9.2](#)
waiting on a condition [10.7.3](#)
walk [9.2.3](#), [9.2.5](#)
walkFileTree method [9.2.3](#), [9.2.4](#)
warning [11.3](#)
warnings, suppressing [6.6.2](#), [6.6.4](#), [11.3.1](#)
weak references [7.5.5](#)
weaker access privilege [4.1.3](#)
WeakHashMap class [7.5.5](#)
weakly consistent iterators [10.3.3](#)
WeakReference class [7.5.5](#)
web pages
extracting links from [10.5.2](#)
reading [10.5.2](#), [10.5.3](#)
whenComplete method [10.5.1](#), [10.5.2](#)
while keyword [1.7.3](#)
breaking/continuing [1.7.4](#)
declaring variables for [1.7.5](#)
white space
in regular expressions [9.4.1](#)
in text blocks [1.5.7](#)
removing [1.5.5](#)
wildcards
annotating [11.1.4](#)
capturing [6.4.5](#)
for annotation processors [11.5.1](#)
for types [6.4.1](#), [6.4.2](#)
in class path [2.6.3](#)
unbounded [6.4.4](#)
with imported classes [2.6.5](#)
with type variables [6.4.3](#)
WildcardType interface [6.7.2](#)
Window class [2.6.4](#)
WindowAdapter class [3.2.2](#)
WindowListener interface [3.2.2](#)
words
in regular expressions [9.4.1](#)
reading from a file [9.1.5](#)
working directory [9.2.1](#), [10.9.1](#)
wrapper classes [1.8.4](#)
write method
of Files [9.1.6](#), [9.2.3](#)
of OutputStream [9.1.3](#)
of Writer [9.1.6](#)
writeByte, writeChar methods
(DataOutput) [9.1.8](#)
writeDouble method
of DataOutput [9.1.8](#)
of ObjectOutputStream [9.5.3](#)
writeExternal method [9.5.4](#)
writeFloat, writeFully methods
(DataOutput) [9.1.8](#)
writeInt method [9.1.8](#), [9.1.9](#)
writeLong method [9.1.8](#)
writeObject method
of HashSet [9.5.3](#)
of ObjectOutputStream [9.5.1](#), [9.5.3](#)
Writer class [9.1.6](#)
write method [9.1.6](#)
writeReplace method [9.5.5](#)
writers [9.1](#)
writeShort, writeUnsignedXxx, writeUTF
methods (DataOutput) [9.1.8](#)

X

x, X
conversion characters [1.6.2](#)
XML descriptors, generating [11.5.3](#)
xor [7.5.2](#)
Xoroshiro128PlusPlus [3.1.1](#)

Y

yield keyword [1.7.2](#)
yield method (Thread) [10.1.1](#)

Z

ZIP file systems [9.2.5](#)
ZipInputStream, ZipOutputStream
classes [9.2.5](#)