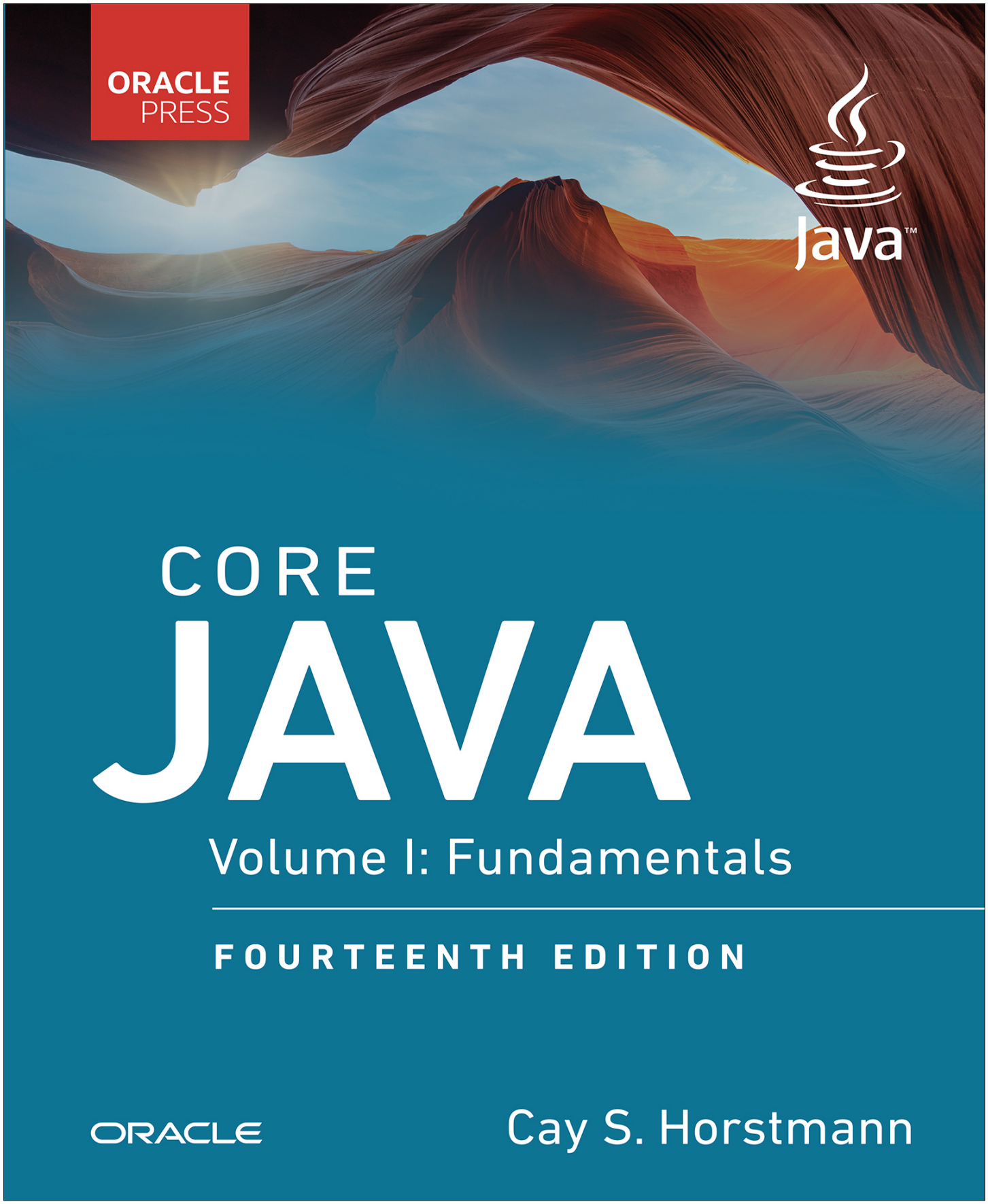

FREE SAMPLE CHAPTER |

This page intentionally left blank

Core Java

Volume I: Fundamentals

Fourteenth Edition

Cay S. Horstmann

Pearson

Cover image: emotionPicture/stock.adobe.com
Figure 1.1: Sourceforge
Figures 2.2, 3.2-3.5, 4.9, 5.4, 7.2, 10.5, 10.6, 11.1: Oracle Corporation
Figures 2.3-2.5, 12.2: Eclipse Foundation AISBL
Figure 4.2: Violet UML Editor

Visit us on the Web: informit.com

Library of Congress Control Number: 2025945021

ISBN-13: 978-0-13-555857-7
ISBN-10: 0-13-555857-3

Contents

This page intentionally left blank

Preface

To the Reader

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

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

The book that you are reading right now is the first volume of the fourteenth edition of *Core Java*. Each edition closely followed a release of the Java Development Kit, and each time, I rewrote the book to take advantage of the newest Java features. This edition has been updated to reflect the features of Java 25.

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

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

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

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

- The Java platform module system

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

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

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

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

A Tour of This Book

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

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

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

Object-oriented programming (OOP) is now in the mainstream of programming practice, and Java is an object-oriented programming language. **Chapter 4** introduces encapsulation, the first of two fundamental building blocks of object orientation, and the Java language mechanism to implement it—that is, classes and methods. In addition to the rules of the Java language, you will

also find advice on sound OOP design. Finally, I cover the marvelous `javadoc` tool that formats your code comments as a set of hyperlinked web pages. If you are familiar with C++, you can browse through this chapter quickly. Programmers coming from a non-object-oriented background should expect to spend some time mastering the OOP concepts before going further with Java.

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

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

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

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

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

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

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

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

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

Conventions

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

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

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

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

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

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

`java.lang.IO` 25

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

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

Listing 1.1: `NotHelloWorld.java`

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

Sample Code

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

This page intentionally left blank

Acknowledgments

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

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

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

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

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

Cay Horstmann
Düsseldorf, Germany
September 2025

Chapter 3

Fundamental Programming Structures in Java

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

3.1. A Simple Java Program

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

For example:

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

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

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

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

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

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

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

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

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

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

```
IO.println();
```

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

You compile the file with the command

You run the sample program with this command:

```
java FirstSample.java
```

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

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

```
javac FirstSample.java
```

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

```
java FirstSample
```

Remember to leave off the `.class` extension.

When you use

```
java ClassName
```

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

3.2. Comments

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

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

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

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

Listing 3.1: `FirstSample.java`

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

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

3.3. Data Types

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

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

3.3.1. Integer Types

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

Table 3.1: Java Integer Types

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

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

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

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

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

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

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

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

3.3.2. Floating-Point Types

The floating-point types denote numbers with fractional parts. The two floating-point types are shown in Table 3.2.

Table 3.2: Floating-Point Types

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.3.3. The `char` Type

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

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

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

```
void main()\u007BIO.println("Hello, World!");\u007D
```

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

Table 3.3: Escape Sequences for Special Characters

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

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

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

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

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

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

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

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

3.3.4. Unicode and the `char` Type

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

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

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

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

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

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

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

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

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

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

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

3.3.5. The `boolean` Type

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

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

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

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

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

3.4. Variables and Constants

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

3.4.1. Declaring Variables

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

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

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

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

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

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

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

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

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

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

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

You can declare multiple variables on a single line:

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

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

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

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

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

```
Box aBox;
```

3.4.2. Initializing Variables

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

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

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

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

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

```
int vacationDays = 12;
```

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

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

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

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

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

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

3.4.3. Constants

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

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

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

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

```
final double CM_PER_INCH = 2.54;

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

// CM_PER_INCH also accessible in other methods
```

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

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

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

3.4.4. Enumerated Types

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

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

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

Now you can declare variables of this type:

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

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

Enumerated types are discussed in greater detail in Chapter 5.

3.5. Operators

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

3.5.1. Arithmetic Operators

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

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

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

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

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

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

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

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

3.5.2. Mathematical Functions and Constants

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

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

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

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

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

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

The `Math` class supplies the usual trigonometric functions:

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

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

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

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

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

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

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

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

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

For example:

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

Static imports are covered in Chapter 4.

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

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

3.5.3. Conversions between Numeric Types

It is often necessary to convert from one numeric type to another. Figure 3.1 shows the legal conversions.

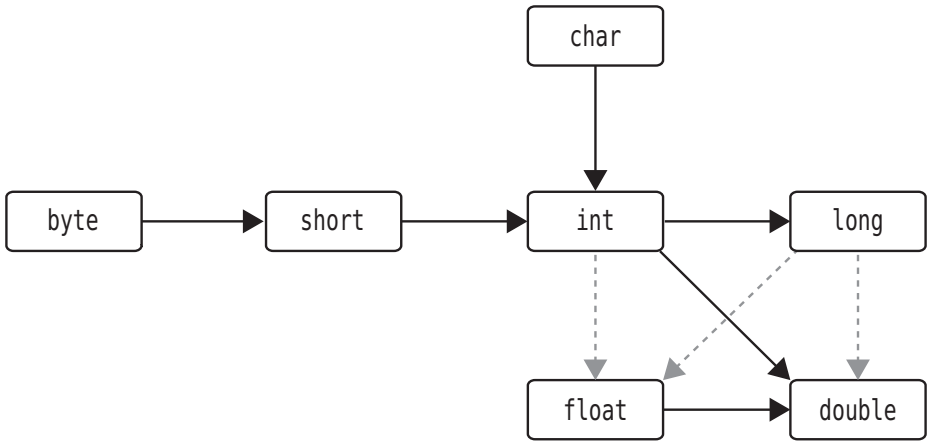

Figure 3.1: Legal conversions between numeric types

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

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

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

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

3.5.4. Casts

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

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

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

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

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

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

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

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

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

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

3.5.5. Assignment

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

```
x += 4;
```

is equivalent to

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

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

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

```
x += 3.5;
```

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

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

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

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

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

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

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

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

3.5.6. Increment and Decrement Operators

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

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

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

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

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

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

3.5.7. Relational and `boolean` Operators

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

```
3 == 7
```

is `false`.

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

```
3 != 7
```

is `true`.

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

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

expression$_1$ **&&** *expression*$_2$

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

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

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

Similarly, the value of *expression*$_1$ || *expression*$_2$ is automatically `true` if the first expression is `true`, without evaluating the second expression.

3.5.8. The Conditional Operator

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

condition ? $expression_1$: $expression_2$

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

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

gives the smaller of `x` and `y`.

3.5.9. Switch Expressions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.5.10. Bitwise Operators

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

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

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

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

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

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

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

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

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

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

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

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

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

3.5.11. Parentheses and Operator Hierarchy

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

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

means

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

Since += associates right to left, the expression

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

means

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

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

Table 3.4: Operator Precedence

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

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

3.6. Strings

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

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

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

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

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

3.6.1. Concatenation

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

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

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

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

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

sets `rating` to the string `"PG13"`.

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

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

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

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

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

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

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

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

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

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

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

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

3.6.2. Static and Instance Methods

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

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

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

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

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

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

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

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

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

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

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

3.6.3. Indexes and Substrings

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

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

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

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

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

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

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

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

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

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

You can compute where the next character starts:

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

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

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

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

creates a string consisting of the characters `"World"`.

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

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

3.6.4. Strings Are Immutable

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

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

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

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

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

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

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

3.6.5. Testing Strings for Equality

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

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

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

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

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

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

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

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

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

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

3.6.6. Empty and Null Strings

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

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

or

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

or , for optimum efficiency

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

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

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

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

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

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

3.6.7. The `String` API

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

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

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

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

`java.lang.String` 1.0

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

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

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

3.6.8. Reading the Online API Documentation

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

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

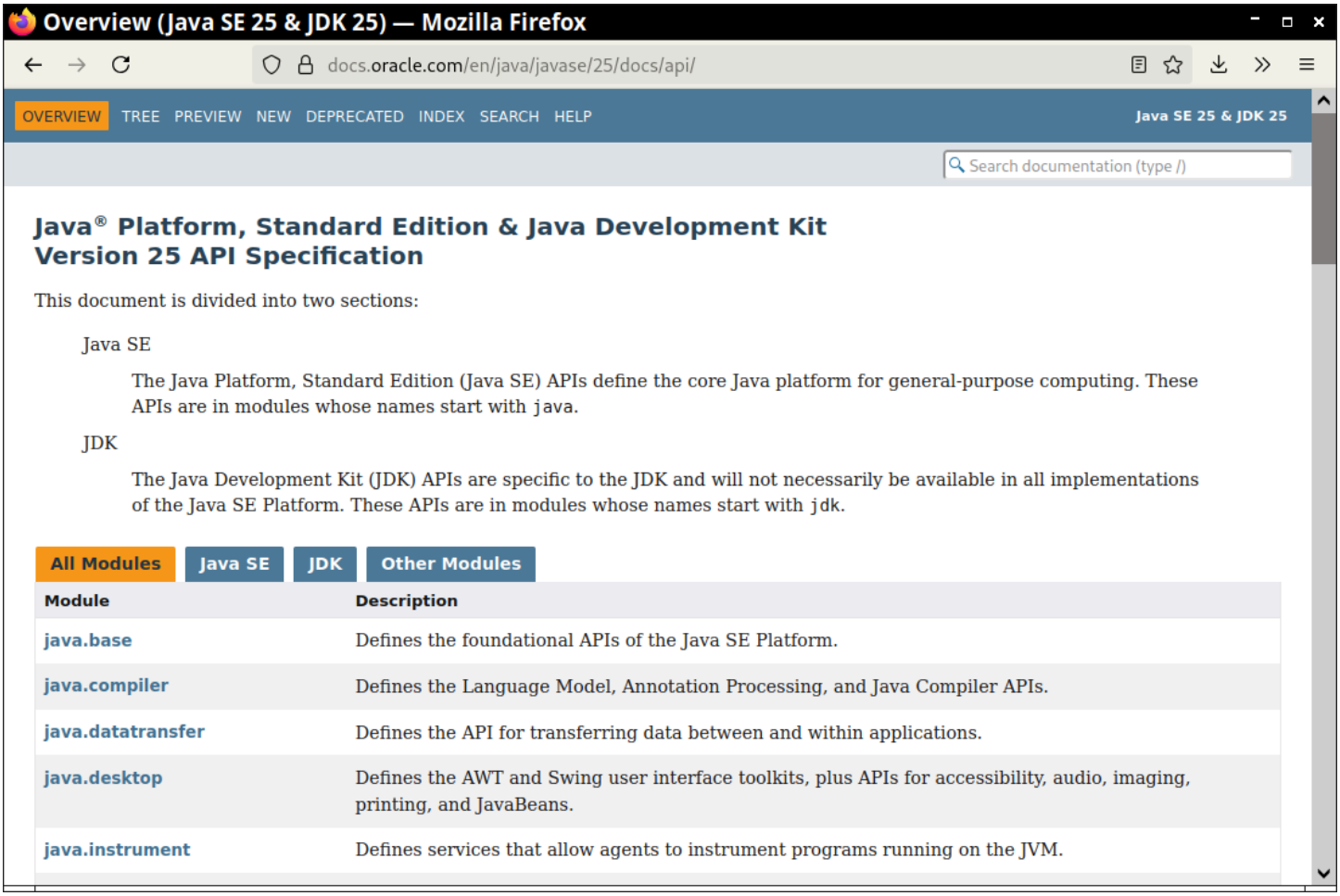

Figure 3.2: The Java API documentation

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

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

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

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

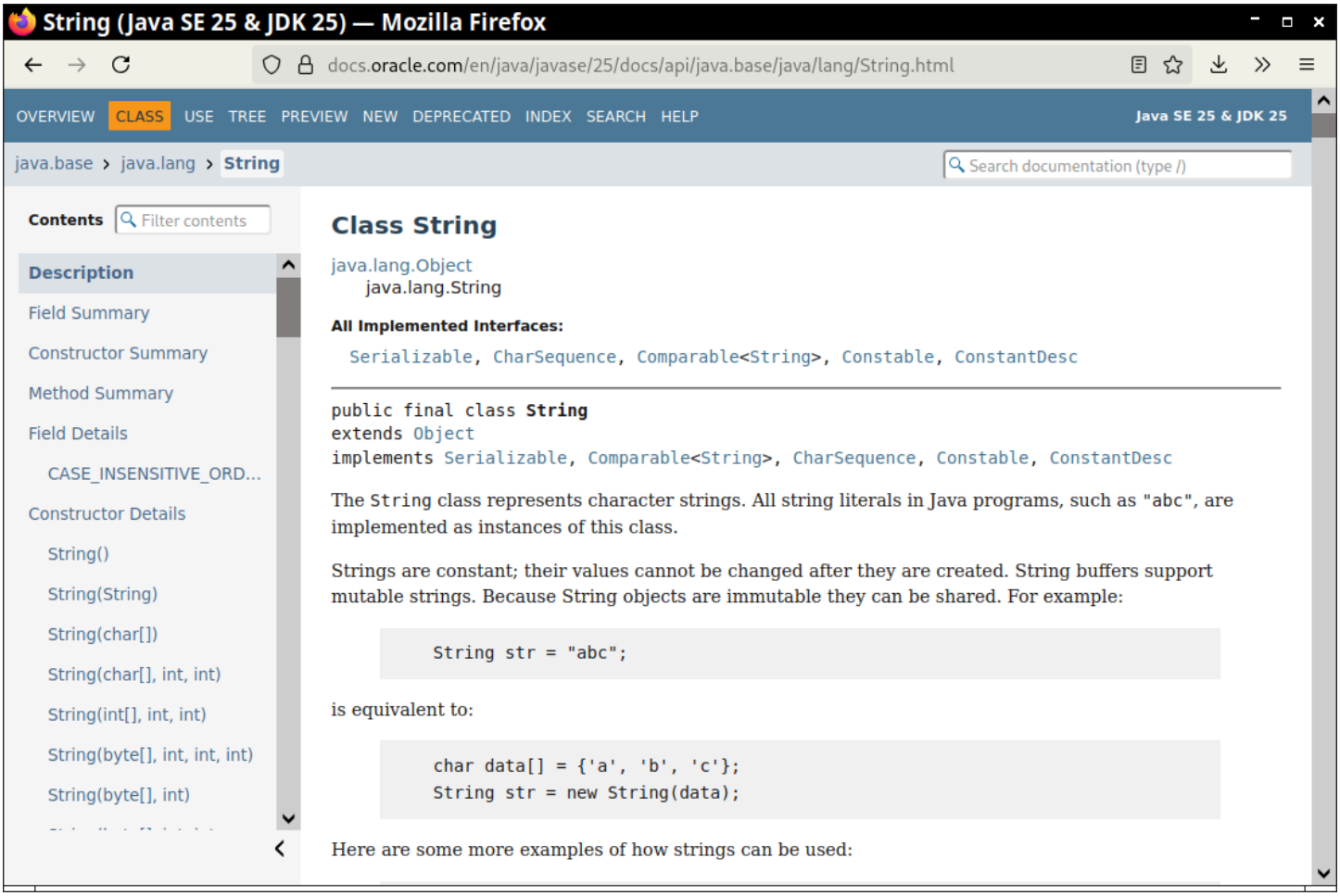

Figure 3.3: Class description for the `String` class

3.6.9. Building Strings

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

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

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

You can also provide initial content:

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

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

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

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

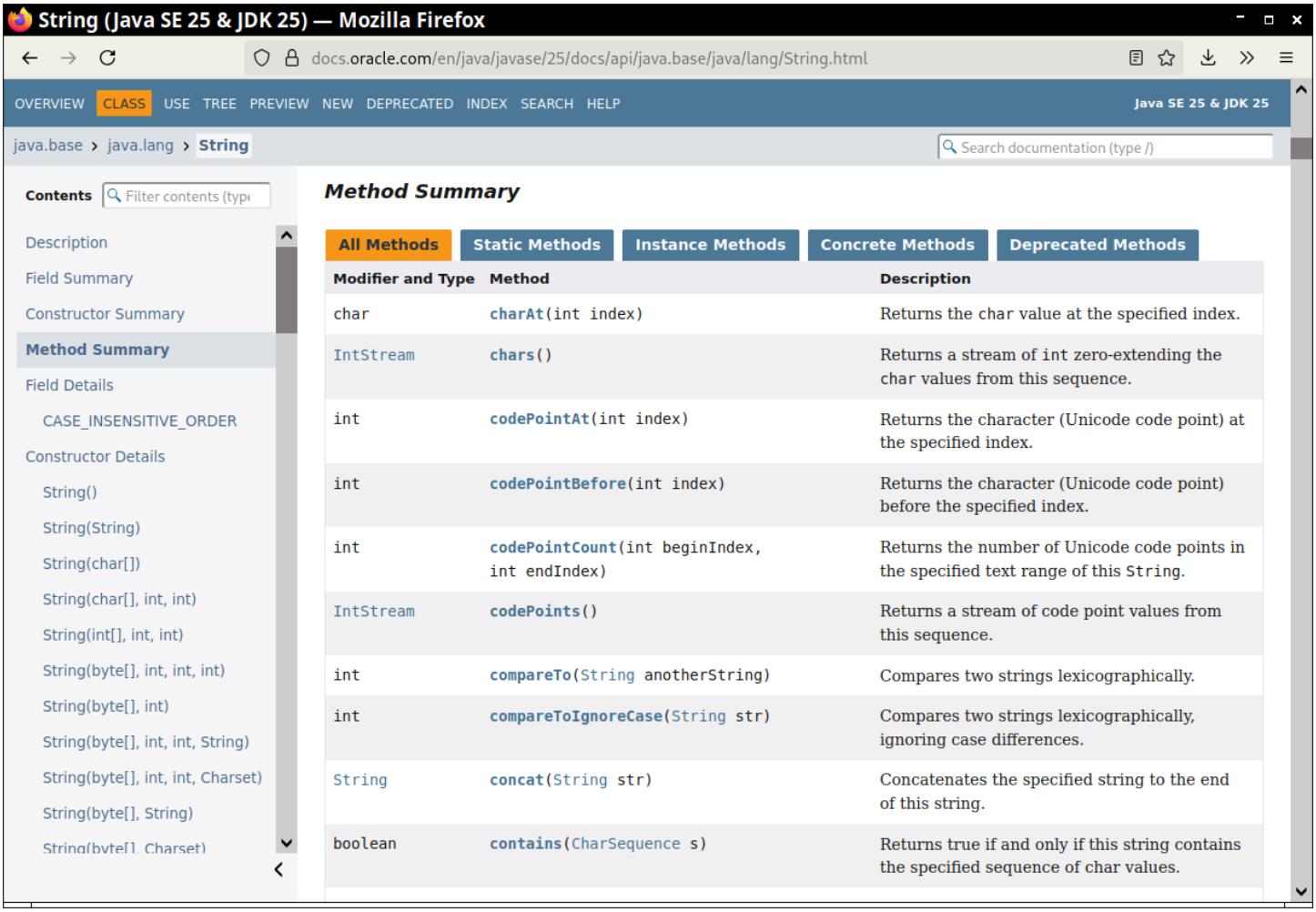

Figure 3.4: Method summary of the `String` class

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

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

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

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

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

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

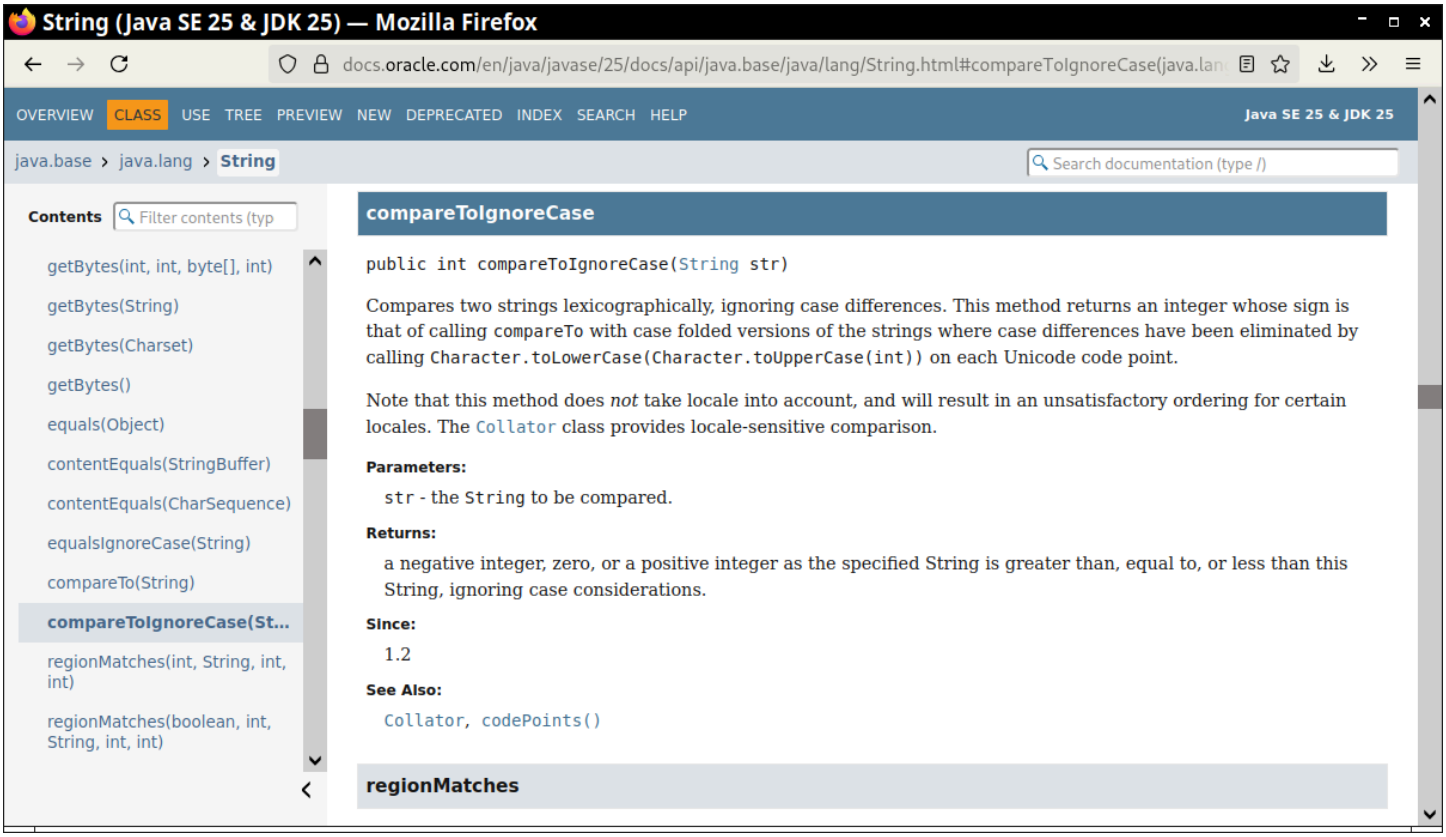

Figure 3.5: Detailed description of a `String` method

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

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

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

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

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

`java.lang.StringBuilder` 5.0

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

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

3.6.10. Text Blocks

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

is the same as

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

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

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

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

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

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

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

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

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

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

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

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

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

3.7. Input and Output

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

3.7.1. Reading Input

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

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

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

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

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

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

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

The program in Listing 3.2 asks for the user's name and age and then prints a message like

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

Listing 3.2: `InputDemo.java`

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

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

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

```
chcp 65001
```

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

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

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

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

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

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

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

The `nextLine` method reads a line of input.

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

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

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

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

Finally, include the line

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

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

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

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

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

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

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

`java.lang.IO` 25

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

`java.util.Scanner` 5.0

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

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

`java.lang.System` 1.0

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

`java.io.Console` 6

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

3.7.2. Formatting Output

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

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

prints

```
3333.3333333333335
```

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

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

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

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

```
3333.33
```

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

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

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

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

Table 3.5: Conversions for `formatted`

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

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

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

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

prints

```
3,333.33
```

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

Table 3.6: Flags for `printf`

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

Figure 3.6 shows a syntax diagram for format specifiers.

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

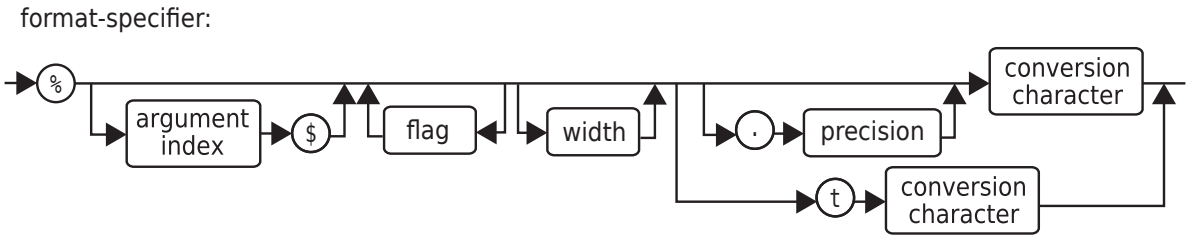

Figure 3.6: Format specifier syntax

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

yields the output

```
 3333,33
```

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

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

3.8. Control Flow

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

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

3.8.1. Block Scope

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

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

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

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

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

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

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

3.8.2. Conditional Statements

The conditional statement in Java has the form

`if (`*condition*`)` *statement*

The condition must be surrounded by parentheses.

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

```
{
    statement1
    statement2
    . . .
}
```

For example:

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

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

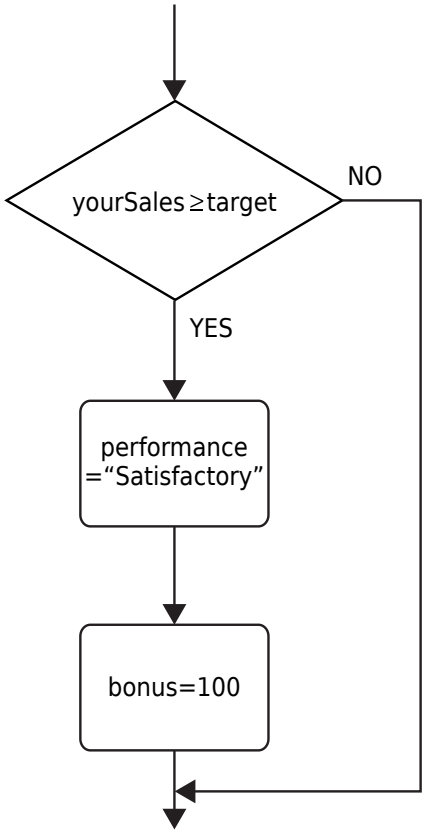

Figure 3.7: Flowchart for the `if` statement

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

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

`if (`*condition*`)` *statement*$_1$ `else` *statement*$_2$

For example:

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

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

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

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

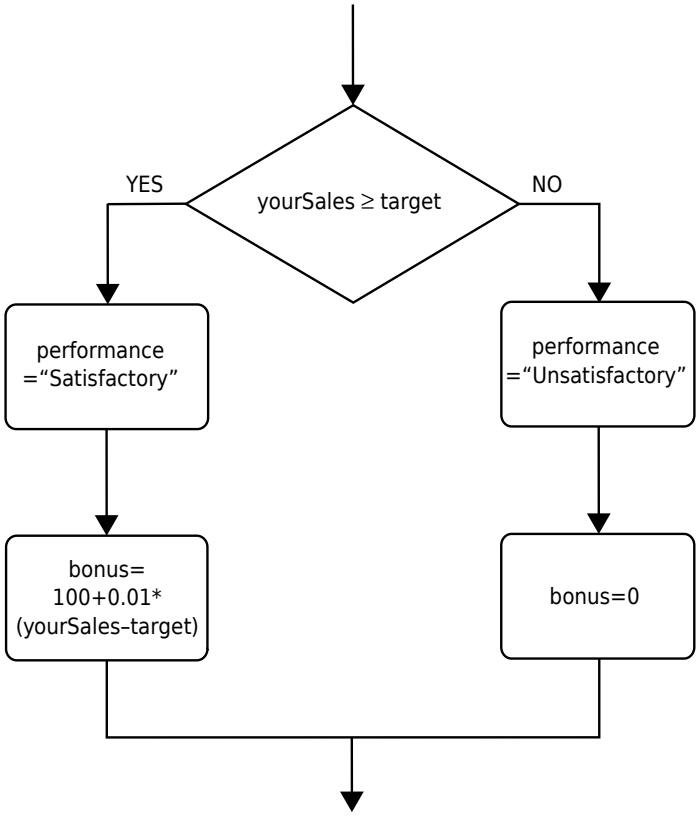

Figure 3.8: Flowchart for the `if/else` statement

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

Repeated `if . . . else if . . .` alternatives are common (see Figure 3.9). For example:

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

3.8.3. Loops

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

```
while (condition) statement
```

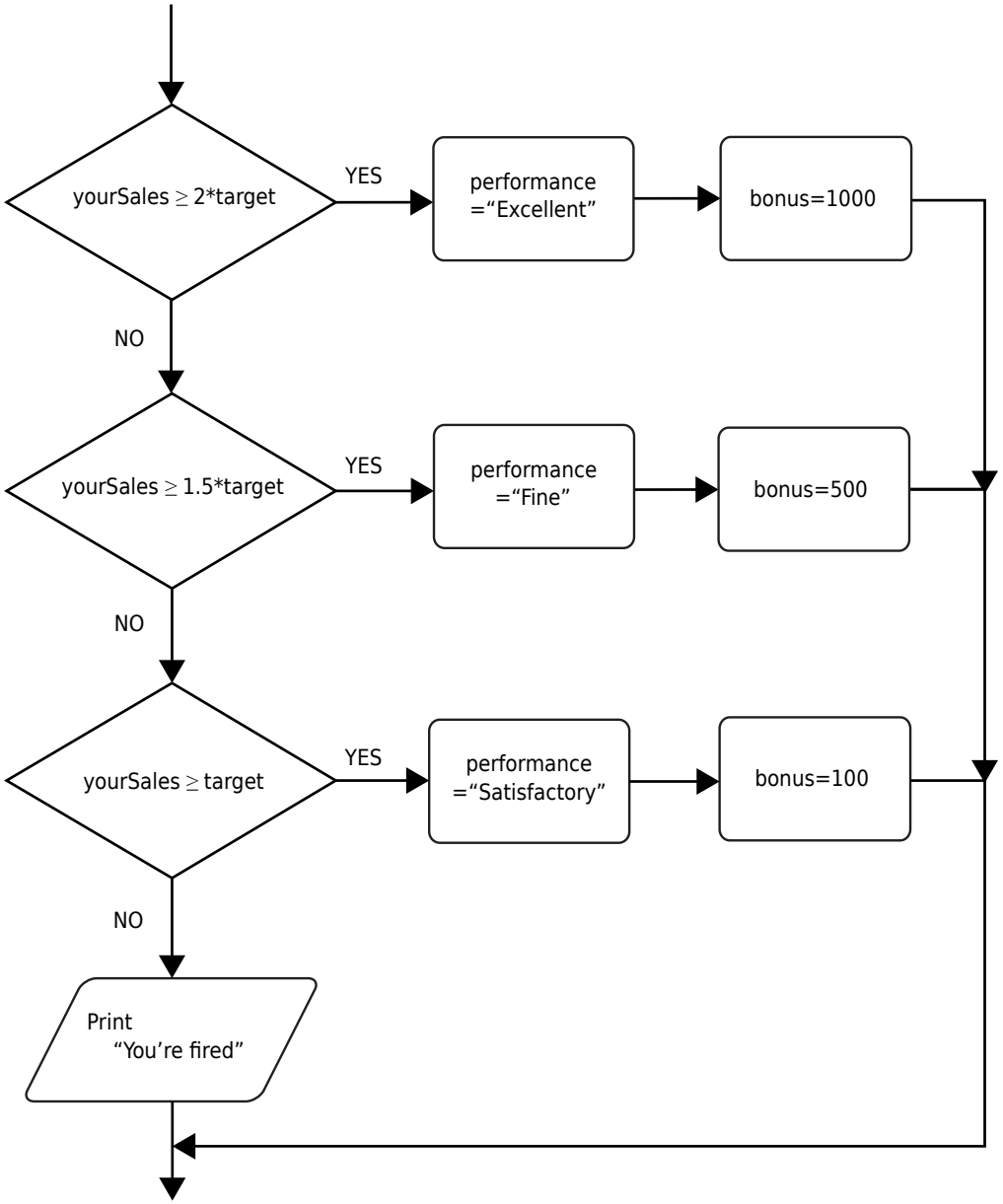

Figure 3.9: Flowchart for the `if/else if` (multiple branches)

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

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

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

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

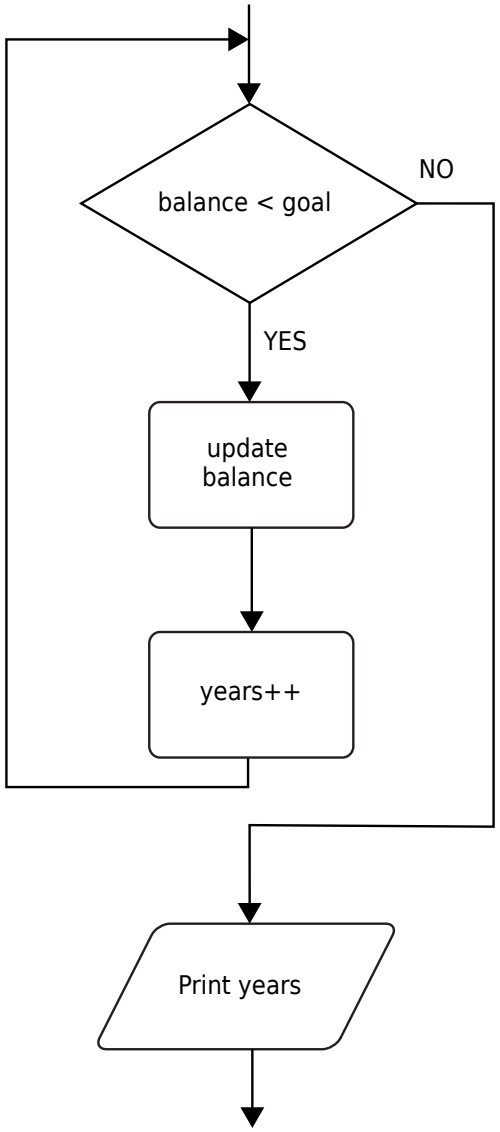

Figure 3.10: Flowchart for the `while` statement

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

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

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

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

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

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

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

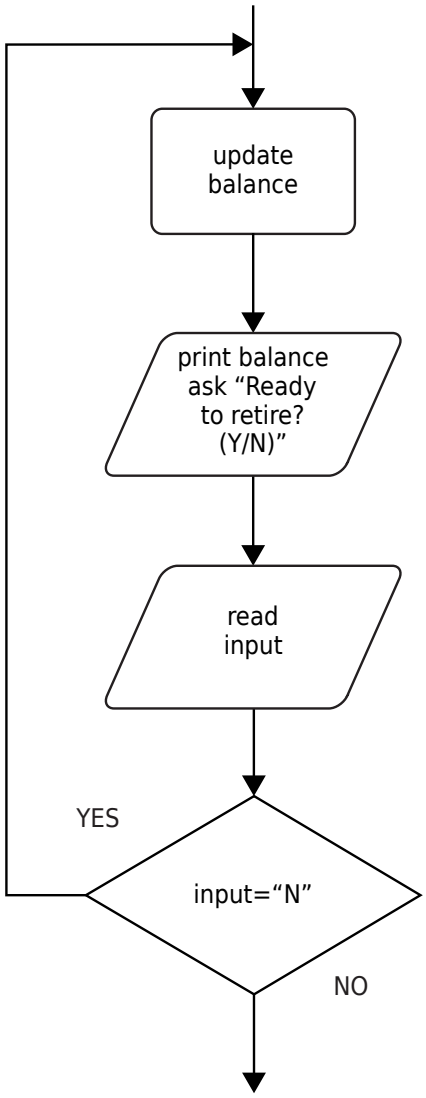

Figure 3.11: Flowchart for the do/while statement

Listing 3.3: Retirement.java

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

```
    double payment
        = Double.parseDouble(IO.readln("How much money will you contribute every year? "));
    double interestRate = Double.parseDouble(IO.readln("Interest rate in %: "));

    double balance = 0;
    int years = 0;

    // update account balance while goal isn't reached
    while (balance < goal) {
        // add this year's payment and interest

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

    IO.println("You can retire in " + years + " years.");
}
```

Listing 3.4: Retirement2.java

```
/**
 * This program demonstrates a <code>do/while</code> loop.
 */
void main() {
    double payment = Double.parseDouble(
        IO.readln("How much money will you contribute every year? "));
    double interestRate = Double.parseDouble(IO.readln("Interest rate in %: "));

    double balance = 0;
    int year = 0;

    String input;

    // update account balance while user isn't ready to retire
    do {
        // add this year's payment and interest
        balance += payment;
        double interest = balance * interestRate / 100;
        balance += interest;

        year++;

        // print current balance
        IO.println("After year %d, your balance is %,.2f".formatted(year,
                balance));

        // ask if ready to retire and get input
        input = IO.readln("Ready to retire? (Y/N) ");
    }
    while (input.equals("N"));
}
```

3.8.4. Determinate Loops

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

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

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

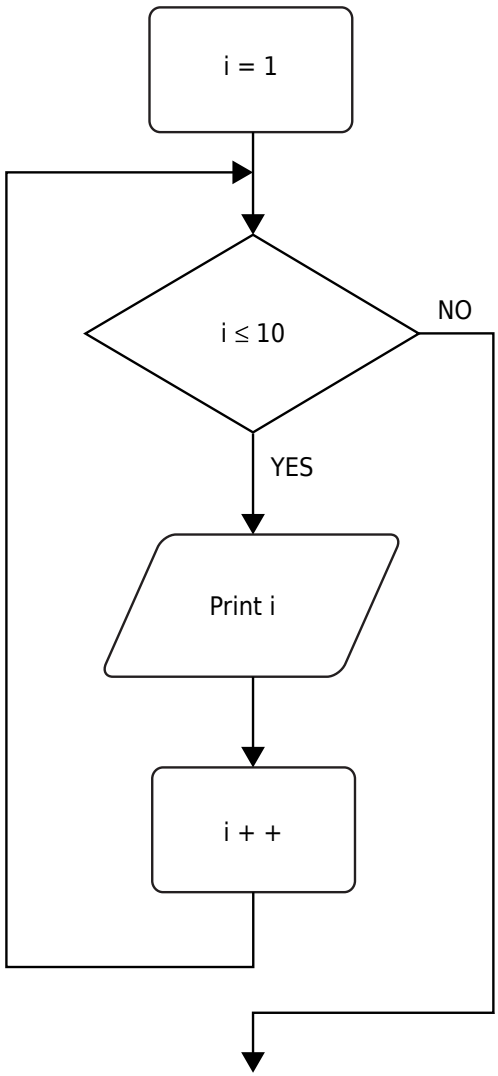

Figure 3.12: Flowchart for the `for` statement

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

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

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

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

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

might never end. Because of roundoff errors, the final value might not be reached exactly. In this example, `x` jumps from `9.99999999999998` to `10.09999999999998` because there is no exact binary representation for `0.1`.

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

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

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

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

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

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

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

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

can be rewritten as follows:

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

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

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

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

Listing 3.5 shows a typical example of a `for` loop.

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

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

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

possible outcomes. The following `for` loop computes this value:

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

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

Listing 3.5: `LotteryOdds.java`

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

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

3.8.5. Multiple Selections with `switch`

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

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

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

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

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

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

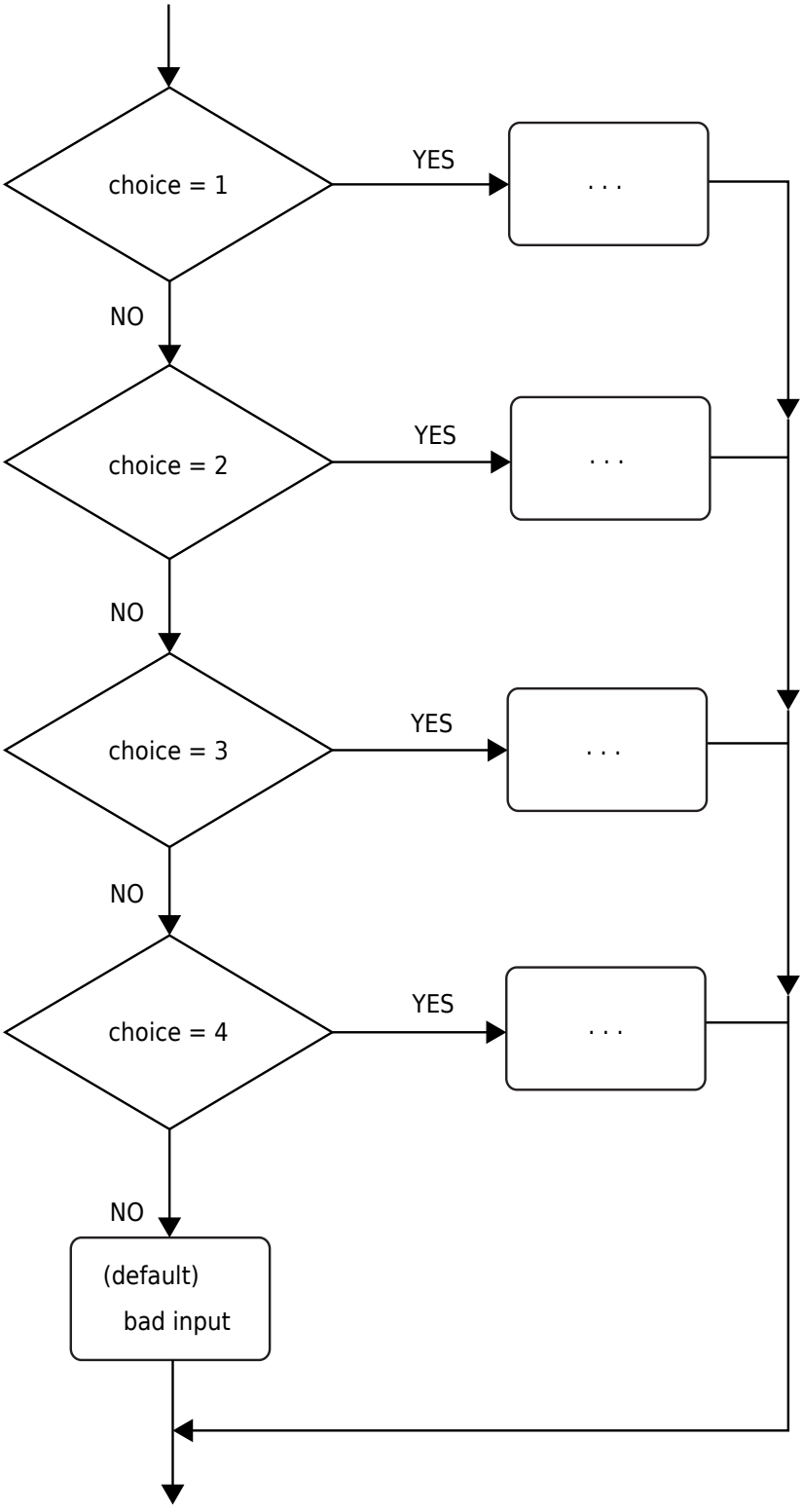

Figure 3.13: Flowchart for the `switch` statement

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

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

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

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

For symmetry, Java 14 also introduced a `switch` expression with fallthrough, for a total of four forms of `switch`. Table 3.7 shows them all.

Table 3.7: The four forms of `switch`

	Expression	Statement

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

Fallthrough	`int numLetters = switch (seasonName) {`
	`  case "Spring":`
	`  IO.println("spring time!");`
	`  case "Summer", "Winter":`
	`  yield 6;`
	`  case "Fall":`
	`  yield 4;`
	`  default:`
	`  yield -1;`
	`};`
		`switch (seasonName) {`
		`  case "Spring":`
		`  IO.println("spring time!");`
		`  case "Summer", "Winter":`
		`  numLetters = 6;`
		`  break;`
		`  case "Fall":`
		`  numLetters = 4;`
		`  break;`
		`  default:`
		`  numLetters = -1;`
		`}`

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

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

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

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

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

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

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

Exceptions are covered in detail in Chapter 7.

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

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

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

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

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

For example, consider:

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

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

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

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

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

3.8.6. Statements That Break Control Flow

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.9. Big Numbers

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

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

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

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

```
BigInteger reallyBig
    = new BigInteger("22223224462942044552973989346190996720666693909649976499097960 0");
```

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

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

```
0.1000000000000000055511151231257827021181583404541015625
```

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

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

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

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

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

The program in Listing 3.5 computed the statement

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

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

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

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

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

to the top of the program, in addition to the general modifications described in the notes in Section 3.1 and Section 3.7.1.

Listing 3.6: `BigIntegerDemo.java`

```
/**
 * This program uses big numbers to compute the odds of winning the grand prize
 * in a lottery.
 */
void main() {
    IO.print("How many numbers do you need to draw? ");
    int k = Integer.parseInt(IO.readln());

    IO.print("What is the highest number you can draw? ");
    BigInteger n = new BigInteger(IO.readln());

    // Binomial coefficient n*(n-1)*(n-2)*...*(n-k+1)/(1*2*3*...*k)

    BigInteger lotteryOdds = BigInteger.ONE;

    for (int i = 1; i <= k; i++)
        lotteryOdds = lotteryOdds
                .multiply(n.subtract(BigInteger.valueOf(i - 1)))
                .divide(BigInteger.valueOf(i));

    IO.println("Your odds are 1 in " + lotteryOdds + " Good luck!");
}
```

`java.math.BigInteger` 1.1

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

java.math.BigDecimal 1.1

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

3.10. Arrays

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

3.10.1. Declaring Arrays

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

```
int[] a;
```

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

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

This statement declares and initializes an array of 100 integers.

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

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

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

Note: You can define an array variable either as

```
int[] a;
```

or as

```
int a[];
```

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

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

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

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

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

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

You can declare an *anonymous array*:

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

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

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

is shorthand for

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

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

```
new elementType[0]
```

or

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

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

3.10.2. Accessing Array Elements

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

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

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

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

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

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

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

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

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

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

3.10.3. The "for each" Loop

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

The *enhanced* `for` loop

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

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

implements the `Iterable` interface, such as `ArrayList`. Array lists are covered in Chapter 5 and the `Iterable` interface in Chapter 9.

For example,

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

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

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

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

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

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

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

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

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

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

3.10.4. Array Copying

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

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

Figure 3.14 shows the result.

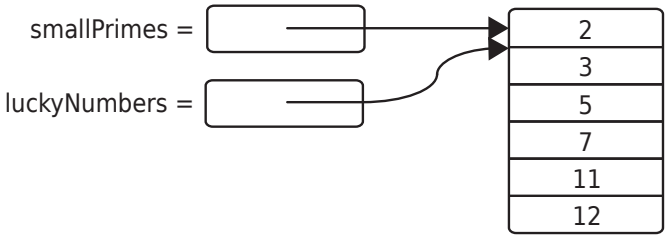

Figure 3.14: Copying an array variable

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

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

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

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

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

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

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

is not the same as

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

but rather

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

3.10.5. Command-Line Arguments

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

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

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

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

If the program is called as

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

or

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

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

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

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

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

3.10.6. Array Sorting

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

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

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

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

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

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

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

A second array holds the numbers to be drawn:

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

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

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

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

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

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

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

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

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

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

Listing 3.7: `LotteryDrawing.java`

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

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

java.util.Arrays 1.2

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

3.10.7. Multidimensional Arrays

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Then compute the other rows, as follows:

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

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

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

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

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

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

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

The output is formatted like this:

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

Listing 3.8: `CompoundInterest.java`

```
/**
 * This program shows how to store tabular data in a 2D array.
 */
void main() {
    final double STARTRATE = 5;
    final int NRATES = 6;
    final int NYEARS = 10;

    // set interest rates to 5 . . . 10%
    double[] interestRates = new double[NRATES];
    for (int j = 0; j < interestRates.length; j++)
        interestRates[j] = (STARTRATE + j) / 100.0;

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

    // set initial balances to 10000
    for (int j = 0; j < balances[0].length; j++)
        balances[0][j] = 10000;

    // compute interest for future years
    for (int i = 1; i < balances.length; i++) {
        for (int j = 0; j < balances[i].length; j++) {
            // get last year's balances from previous row
            double oldBalance = balances[i - 1][j];

            // compute interest
            double interest = oldBalance * interestRates[j];

            // compute this year's balances
            balances[i][j] = oldBalance + interest;
        }
    }

    printTable(interestRates, balances);
}
```

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

3.10.8. Ragged Arrays

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

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

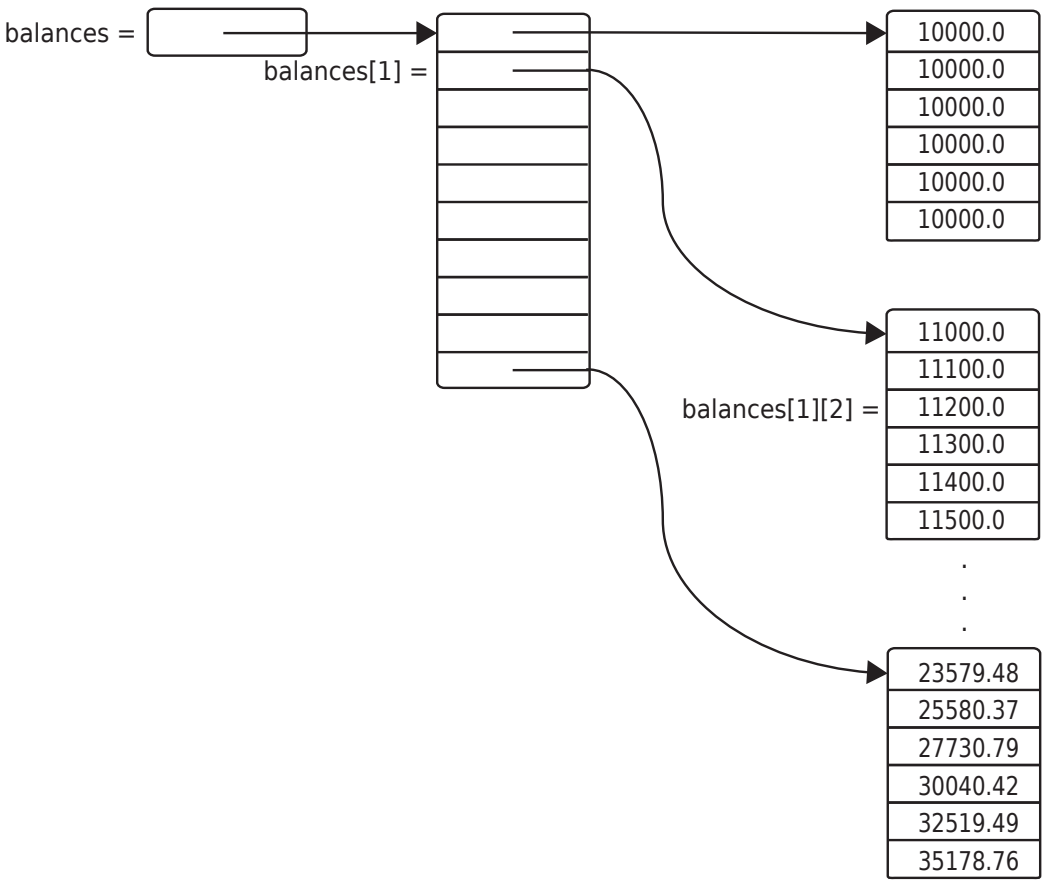

Figure 3.15: A two-dimensional array

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

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

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

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

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

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

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

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

Next, allocate the rows:

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

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

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

Listing 3.9 gives the complete program.

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

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

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

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

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

Note: The Java declaration

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

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

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

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

Listing 3.9: `LotteryArray.java`

```
/**
 * This program demonstrates a triangular array.
 */
void main() {
    final int NMAX = 10;

    // allocate triangular array
    int[][] odds = new int[NMAX + 1][];
    for (int n = 0; n <= NMAX; n++)
        odds[n] = new int[n + 1];

    // fill triangular array
    for (int n = 0; n < odds.length; n++)
        for (int k = 0; k < odds[n].length; k++) {
            /*
             * compute binomial coefficient
             * n*(n-1)*(n-2)*...*(n-k+1)/(1*2*3*...*k)
             */
            int lotteryOdds = 1;
            for (int i = 1; i <= k; i++)
                lotteryOdds = lotteryOdds * (n - i + 1) / i;

            odds[n][k] = lotteryOdds;
        }

```

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

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

This page intentionally left blank

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y