Elements of Programming

OUR GOAL IN THIS CHAPTER IS to convince you that writing a program is easier than writing a piece of text, such as a paragraph or essay. Writing prose is difficult: we spend many years in school to learn how to do it. By contrast, just a few building blocks suffice to enable us to write programs that can help solve all sorts of fascinating, but otherwise unapproachable, problems. In this chapter, we take you through these building blocks, get you started on programming in Java, and study a variety of interesting programs. You will be able to express yourself (by writing programs) within just a few weeks. Like the ability to write prose, the ability to program is a lifetime skill that you can continually refine well into the future.

In this book, you will learn the *Java programming language*. This task will be much easier for you than, for example, learning a foreign language. Indeed, programming languages are characterized by only a few dozen vocabulary words and rules of grammar. Much of the material that we cover in this book could be expressed in the Python or C++ languages, or any of several other modern programming languages. We describe everything specifically in Java so that you can get started creating and running programs right away. On the one hand, we will focus on learning to program, as opposed to learning details about Java. On the other hand, part of the challenge of programming is knowing which details are relevant in a given situation. Java is widely used, so learning to program in this language will enable you to write programs on many computers (your own, for example). Also, learning to program in Java will make it easy for you to learn other languages, including lower-level languages such as C and specialized languages such as Matlab.

1.1 Your First Program

In this section, our plan is to lead you into the world of Java programming by taking you through the basic steps required to get a simple program running. The *Java platform* (hereafter abbreviated *Java*) is a collection of applications, not unlike many of the other applications that you are accustomed to using (such as your word processor, email program, and web browser). As with any application, you need to be sure that Java is properly installed on your computer. It comes preloaded on many computers, or you can download it easily. You also need a text editor and a terminal application. Your first task is to find the instructions for installing such a Java programming environment on *your* computer by visiting

```
http://introcs.cs.princeton.edu/java
```

Programs in this section

We refer to this site as the *booksite*. It contains an extensive amount of supplementary information about the material in this book for your reference and use while programming.

Programming in Java To introduce you to developing Java programs, we break the process down into three steps. To program in Java, you need to:

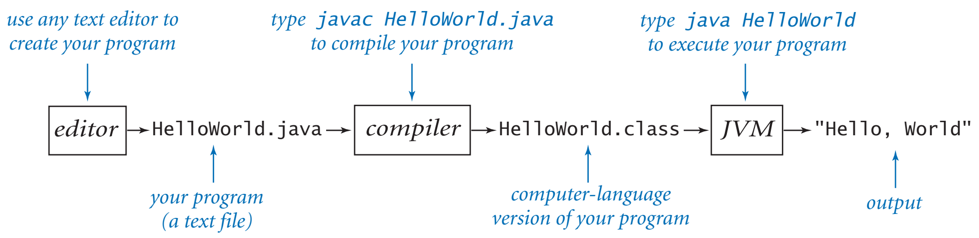
- *Create* a program by typing it into a file named, say, `MyProgram.java`.
- *Compile* it by typing `javac MyProgram.java` in a terminal window.
- *Execute* (or *run*) it by typing `java MyProgram` in the terminal window.

In the first step, you start with a blank screen and end with a sequence of typed characters on the screen, just as when you compose an email message or an essay. Programmers use the term *code* to refer to program text and the term *coding* to refer to the act of creating and editing the code. In the second step, you use a system application that *compiles* your program (translates it into a form more suitable for the computer) and puts the result in a file named `MyProgram.class`. In the third step, you transfer control of the computer from the system to your program (which returns control back to the system when finished). Many systems have several different ways to create, compile, and execute programs. We choose the sequence given here because it is the simplest to describe and use for small programs.

Creating a program. A Java program is nothing more than a sequence of characters, like a paragraph or a poem, stored in a file with a `.java` extension. To create one, therefore, you need simply define that sequence of characters, in the same way as you do for email or any other computer application. You can use any *text editor* for this task, or you can use one of the more sophisticated *integrated development environments* described on the booksite. Such environments are overkill for the sorts of programs we consider in this book, but they are not difficult to use, have many useful features, and are widely used by professionals.

Compiling a program. At first, it might seem that Java is designed to be best understood by the computer. To the contrary, the language is designed to be best understood by the programmer—that's you. The computer's language is far more primitive than Java. A *compiler* is an application that translates a program from the Java language to a language more suitable for execution on the computer. The compiler takes a file with a `.java` extension as input (your program) and produces a file with the same name but with a `.class` extension (the computer-language version). To use your Java compiler, type in a terminal window the `javac` command followed by the file name of the program you want to compile.

Executing (running) a program. Once you compile the program, you can execute (or run) it. This is the exciting part, where your program takes control of your computer (within the constraints of what Java allows). It is perhaps more accurate to say that your computer follows your instructions. It is even more accurate to say that a part of Java known as the *Java Virtual Machine* (*JVM*, for short) directs your computer to follow your instructions. To use the JVM to execute your program, type the `java` command followed by the program name in a terminal window.


use any text editor to create your program
type javac HelloWorld.java to compile your program
type java HelloWorld to execute your program
editor
HelloWorld.java
compiler
HelloWorld.class
JVM
"Hello, World"
your program (a text file)
computer-language version of your program
output

Developing a Java program

Program 1.1.1 Hello, World

```
public class HelloWorld
{
   public static void main(String[] args)
   {
      // Prints "Hello, World" in the terminal window.
      System.out.println("Hello, World");
   }
}
```

This code is a Java program that accomplishes a simple task. It is traditionally a beginner's first program. The box below shows what happens when you compile and execute the program. The terminal application gives a command prompt (`%` in this book) and executes the commands that you type (`javac` and then `java` in the example below). Our convention is to highlight in boldface the text that you type and display the results in regular face. In this case, the result is that the program prints the message `Hello, World` in the terminal window.

```
% javac HelloWorld.java
% java HelloWorld
Hello, World
```

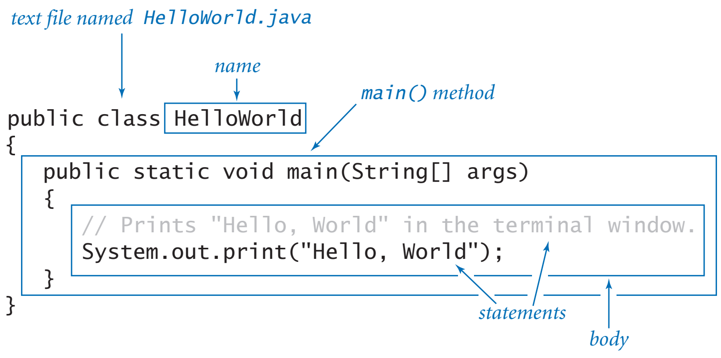
PROGRAM 1.1.1 is an example of a complete Java program. Its name is `HelloWorld`, which means that its code resides in a file named `HelloWorld.java` (by convention in Java). The program's sole action is to print a message to the terminal window. For continuity, we will use some standard Java terms to describe the program, but we will not define them until later in the book: PROGRAM 1.1.1 consists of a single *class* named `HelloWorld` that has a single *method* named `main()`. (When referring to a method in the text, we use `()` after the name to distinguish it from other kinds of names.) Until SECTION 2.1, all of our classes will have this same structure. For the time being, you can think of "class" as meaning "program."

The first line of a method specifies its name and other information; the rest is a sequence of *statements* enclosed in curly braces, with each statement typically followed by a semicolon. For the time being, you can think of "programming" as meaning "specifying a class name and a sequence of statements for its `main()` method," with the heart of the program consisting of the sequence of statements in the `main()` method (its *body*). Program 1.1.1 contains two such statements:

- The first statement is a *comment*, which serves to document the program. In Java a single-line comment begins with two '/' characters and extends to the end of the line. In this book, we display comments in gray. Java ignores comments—they are present only for human readers of the program.
- The second statement is a *print statement*. It calls the method named `System.out.println()` to print a text message—the one specified between the matching double quotes—to the terminal window.

In the next two sections, you will learn about many different kinds of statements that you can use to make programs. For the moment, we will use only comments and print statements, like the ones in `HelloWorld`.

When you type `java` followed by a class name in your terminal window, the system calls the `main()` method that you defined in that class, and executes its statements in order, one by one. Thus, typing `java HelloWorld` causes the system to call the `main()` method in Program 1.1.1 and execute its two statements. The first statement is a comment, which Java ignores. The second statement prints the specified message to the terminal window.



```
public class HelloWorld
{
   public static void main(String[] args)
   {
      // Prints "Hello, World" in the terminal window.
      System.out.print("Hello, World");
   }
}
```

Anatomy of a program

Since the 1970s, it has been a tradition that a beginning programmer's first program should print `Hello, World`. So, you should type the code in Program 1.1.1 into a file, compile it, and execute it. By doing so, you will be following in the footsteps of countless others who have learned how to program. Also, you will be checking that you have a usable editor and terminal application. At first, accomplishing the task of printing something out in a terminal window might not seem very interesting; upon reflection, however, you will see that one of the most basic functions that we need from a program is its ability to tell us what it is doing.

For the time being, all our program code will be just like Program 1.1.1, except with a different sequence of statements in `main()`. Thus, you do not need to start with a blank page to write a program. Instead, you can

- Copy `HelloWorld.java` into a new file having a new program name of your choice, followed by `.java`.
- Replace `HelloWorld` on the first line with the new program name.
- Replace the comment and print statements with a different sequence of statements.

Your program is characterized by its sequence of statements and its name. Each Java program must reside in a file whose name matches the one after the word `class` on the first line, and it also must have a `.java` extension.

Errors. It is easy to blur the distinctions among editing, compiling, and executing programs. You should keep these processes separate in your mind when you are learning to program, to better understand the effects of the errors that inevitably arise.

You can fix or avoid most errors by carefully examining the program as you create it, the same way you fix spelling and grammatical errors when you compose an email message. Some errors, known as *compile-time* errors, are identified when you compile the program, because they prevent the compiler from doing the translation. Other errors, known as *run-time* errors, do not show up until you execute the program.

In general, errors in programs, also commonly known as *bugs*, are the bane of a programmer's existence: the error messages can be confusing or misleading, and the source of the error can be very hard to find. One of the first skills that you will learn is to identify errors; you will also learn to be sufficiently careful when coding, to avoid making many of them in the first place. You can find several examples of errors in the Q&A at the end of this section.

Program 1.1.2 Using a command-line argument

```
public class UseArgument
{
   public static void main(String[] args)
   {
      System.out.print("Hi, ");
      System.out.print(args[0]);
      System.out.println(". How are you?");
   }
}
```

This program shows the way in which we can control the actions of our programs: by providing an argument on the command line. Doing so allows us to tailor the behavior of our programs.

```
% javac UseArgument.java
% java UseArgument Alice
Hi, Alice. How are you?
% java UseArgument Bob
Hi, Bob. How are you?
```

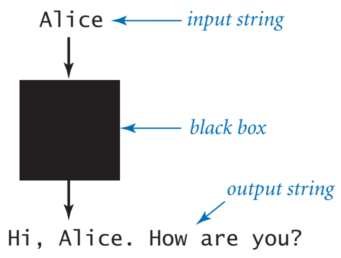
Input and output Typically, we want to provide *input* to our programs—that is, data that they can process to produce a result. The simplest way to provide input data is illustrated in `UseArgument` (Program 1.1.2). Whenever you execute the program `UseArgument`, it accepts the *command-line argument* that you type after the program name and prints it back out to the terminal window as part of the message. The result of executing this program depends on what you type after the program name. By executing the program with different command-line arguments, you produce different printed results. We will discuss in more detail the mechanism that we use to pass command-line arguments to our programs later, in Section 2.1. For now it is sufficient to understand that `args[0]` is the first command-line argument that you type after the program name, `args[1]` is the second, and so forth. Thus, you can use `args[0]` within your program's body to represent the first string that you type on the command line when it is executed, as in `UseArgument`.

In addition to the System.out.println() method, UseArgument calls the System.out.print() method. This method is just like System.out.println(), but prints just the specified string (and not a newline character).

Again, accomplishing the task of getting a program to print back out what we type in to it may not seem interesting at first, but upon reflection you will realize that another basic function of a program is its ability to respond to basic information from the user to control what the program does. The simple model that UseArgument represents will suffice to allow us to consider Java's basic programming mechanism and to address all sorts of interesting computational problems.

Stepping back, we can see that UseArgument does neither more nor less than implement a function that maps a string of characters (the command-line argument) into another string of characters (the message printed back to the terminal window). When using it, we might think of our Java program as a black box that converts our input string to some output string.

This model is attractive because it is not only simple but also sufficiently general to allow completion, in principle, of any computational task. For example, the Java compiler itself is nothing more than a program that takes one string of characters as input (a .java file) and produces another string of characters as output (the corresponding .class file). Later, you will be able to write programs that accomplish a variety of interesting tasks (though we stop short of programs as complicated as a compiler). For the moment, we will live with various limitations on the size and type of the input and output to our programs; in Section 1.5, you will see how to incorporate more sophisticated mechanisms for program input and output. In particular, you will see that we can work with arbitrarily long input and output strings and other types of data such as sound and pictures.



A bird's-eye view of a Java program

Q&A

Q. Why Java?

A. The programs that we are writing are very similar to their counterparts in several other languages, so our choice of language is not crucial. We use Java because it is widely available, embraces a full set of modern abstractions, and has a variety of automatic checks for mistakes in programs, so it is suitable for learning to program. There is no perfect language, and you certainly will be programming in other languages in the future.

Q. Do I really have to type in the programs in the book to try them out? I believe that you ran them and that they produce the indicated output.

A. Everyone should type in and run `HelloWorld`. Your understanding will be greatly magnified if you also run `UseArgument`, try it on various inputs, and modify it to test different ideas of your own. To save some typing, you can find all of the code in this book (and much more) on the booksite. This site also has information about installing and running Java on your computer, answers to selected exercises, web links, and other extra information that you may find useful while programming.

Q. What is the meaning of the words `public`, `static`, and `void`?

A. These keywords specify certain properties of `main()` that you will learn about later in the book. For the moment, we just include these keywords in the code (because they are required) but do not refer to them in the text.

Q. What is the meaning of the `//`, `/*`, and `*/` character sequences in the code?

A. They denote *comments*, which are ignored by the compiler. A comment is either text in between `/*` and `*/` or at the end of a line after `//`. Comments are indispensable because they help other programmers to understand your code and even can help you to understand your own code in retrospect. The constraints of the book format demand that we use comments sparingly in our programs; instead we describe each program thoroughly in the accompanying text and figures. The programs on the booksite are commented to a more realistic degree.

Q. What are Java's rules regarding tabs, spaces, and newline characters?

A. Such characters are known as *whitespace* characters. Java compilers consider all whitespace in program text to be equivalent. For example, we could write `HelloWorld` as follows:

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

But we do normally adhere to spacing and indenting conventions when we write Java programs, just as we indent paragraphs and lines consistently when we write prose or poetry.

Q. What are the rules regarding quotation marks?

A. Material inside double quotation marks is an exception to the rule defined in the previous question: typically, characters within quotes are taken literally so that you can precisely specify what gets printed. If you put any number of successive spaces within the quotes, you get that number of spaces in the output. If you accidentally omit a quotation mark, the compiler may get very confused, because it needs that mark to distinguish between characters in the string and other parts of the program.

Q. What happens when you omit a curly brace or misspell one of the words, such as `public` or `static` or `void` or `main`?

A. It depends upon precisely what you do. Such errors are called *syntax errors* and are usually caught by the compiler. For example, if you make a program `Bad` that is exactly the same as `HelloWorld` except that you omit the line containing the first left curly brace (and change the program name from `HelloWorld` to `Bad`), you get the following helpful message:

```
% javac Bad.java
Bad.java:1: error: '{' expected
public class Bad
                ^
1 error
```

From this message, you might correctly surmise that you need to insert a left curly brace. But the compiler may not be able to tell you exactly which mistake you made, so the error message may be hard to understand. For example, if you omit the second left curly brace instead of the first one, you get the following message:

```
% javac Bad.java
Bad.java:3: error: ';' expected
   public static void main(String[] args)
                                         ^
Bad.java:7: error: class, interface, or enum expected
}
^
2 errors
```

One way to get used to such messages is to intentionally introduce mistakes into a simple program and then see what happens. Whatever the error message says, you should treat the compiler as a friend, because it is just trying to tell you that something is wrong with your program.

Q. Which Java methods are available for me to use?

A. There are thousands of them. We introduce them to you in a deliberate fashion (starting in the next section) to avoid overwhelming you with choices.

Q. When I ran `UseArgument`, I got a strange error message. What's the problem?

A. Most likely, you forgot to include a command-line argument:

```
% java UseArgument
Hi, Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 0
        at UseArgument.main(UseArgument.java:6)
```

Java is complaining that you ran the program but did not type a command-line argument as promised. You will learn more details about array indices in Section 1.4. Remember this error message—you are likely to see it again. Even experienced programmers forget to type command-line arguments on occasion.

Exercises

1.1.1 Write a program that prints the `Hello, World` message 10 times.

1.1.2 Describe what happens if you omit the following in `HelloWorld.java`:

a. `public`
b. `static`
c. `void`
d. `args`

1.1.3 Describe what happens if you misspell (by, say, omitting the second letter) the following in `HelloWorld.java`:

a. `public`
b. `static`
c. `void`
d. `args`

1.1.4 Describe what happens if you put the double quotes in the print statement of `HelloWorld.java` on different lines, as in this code fragment:

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

1.1.5 Describe what happens if you try to execute `UseArgument` with each of the following command lines:

a. `java UseArgument java`
b. `java UseArgument @!&^%`
c. `java UseArgument 1234`
d. `java UseArgument.java Bob`
e. `java UseArgument Alice Bob`

1.1.6 Modify `UseArgument.java` to make a program `UseThree.java` that takes three names as command-line arguments and prints a proper sentence with the names in the reverse of the order given, so that, for example, `java UseThree Alice Bob Carol` prints `Hi Carol, Bob, and Alice.`

This page intentionally left blank

1.2 Built-in Types of Data

WHEN PROGRAMMING IN JAVA, YOU MUST always be aware of the type of data that your program is processing. The programs in SECTION 1.1 process strings of characters, many of the programs in this section process numbers, and we consider numerous other types later in the book. Understanding the distinctions among them is so important that we formally define the idea: a *data type* is a *set of values* and a *set of operations* defined on those values. You are familiar with various types of numbers, such as integers and real numbers, and with operations defined on them, such as addition and multiplication. In mathematics, we are accustomed to thinking of sets of numbers as being infinite; in computer programs we have to work with a finite number of possibilities. Each operation that we perform is well defined *only* for the finite set of values in an associated data type.

Programs in this section

There are eight *primitive* types of data in Java, mostly for different kinds of numbers. Of the eight primitive types, we most often use these: `int` for integers; `double` for real numbers; and `boolean` for true–false values. Other data types are available in Java libraries: for example, the programs in SECTION 1.1 use the type `String` for strings of characters. Java treats the `String` type differently from other types because its usage for input and output is essential. Accordingly, it shares some characteristics of the primitive types; for example, some of its operations are built into the Java language. For clarity, we refer to primitive types and `String` collectively as *built-in* types. For the time being, we concentrate on programs that are based on computing with built-in types. Later, you will learn about Java library data types and building your own data types. Indeed, programming in Java often centers on building data types, as you shall see in CHAPTER 3.

After defining basic terms, we consider several sample programs and code fragments that illustrate the use of different types of data. These code fragments do not do much real computing, but you will soon see similar code in longer programs. Understanding data types (values and operations on them) is an essential step in beginning to program. It sets the stage for us to begin working with more intricate programs in the next section. Every program that you write will use code like the tiny fragments shown in this section.

type	*set of values*	*common operators*	*sample literal values*
`int`	integers	`+ - * / %`	`99 12 2147483647`
`double`	floating-point numbers	`+ - * /`	`3.14 2.5 6.022e23`
`boolean`	boolean values	`&& || !`	`true false`
`char`	characters		`'A' '1' '%' '\n'`
`String`	sequences of characters	`+`	`"AB" "Hello" "2.5"`

Basic built-in data types

Terminology To talk about data types, we need to introduce some terminology. To do so, we start with the following code fragment:

```
int a, b, c;
a = 1234;
b = 99;
c = a + b;
```

The first line is a *declaration statement* that declares the names of three *variables* using the *identifiers* `a`, `b`, and `c` and their type to be `int`. The next three lines are *assignment statements* that change the values of the variables, using the *literals* `1234` and `99`, and the *expression* `a + b`, with the end result that `c` has the value `1333`.

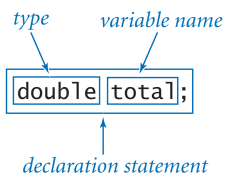
Literals. A *literal* is a Java-code representation of a data-type value. We use sequences of digits such as `1234` or `99` to represent values of type `int`; we add a decimal point, as in `3.14159` or `2.71828`, to represent values of type `double`; we use the keywords `true` or `false` to represent the two values of type `boolean`; and we use sequences of characters enclosed in matching quotes, such as `"Hello, World"`, to represent values of type `String`.

Operators. An *operator* is a Java-code representation of a data-type operation. Java uses + and * to represent addition and multiplication for integers and floating-point numbers; Java uses &&, `||`, and `!` to represent boolean operations; and so forth. We will describe the most commonly used operators on built-in types later in this section.

Identifiers. An *identifier* is a Java-code representation of a name (such as for a variable). Each identifier is a sequence of letters, digits, underscores, and currency symbols, the first of which is not a digit. For example, the sequences of characters abc, Ab$, abc123, and a_b are all legal Java identifiers, but Ab*, 1abc, and a+b are not. Identifiers are case sensitive, so Ab, ab, and AB are all different names. Certain *reserved words*—such as public, static, int, double, String, true, false, and null—are special, and you cannot use them as identifiers.

Variables. A *variable* is an entity that holds a data-type value, which we can refer to by name. In Java, each variable has a specific type and stores one of the possible values from that type. For example, an int variable can store either the value 99 or 1234 but not 3.14159 or "Hello, World". Different variables of the same type may store the same value. Also, as the name suggests, the value of a variable may *change* as a computation unfolds. For example, we use a variable named sum in several programs in this book to keep the running sum of a sequence of numbers. We create variables using *declaration statements* and compute with them in *expressions*, as described next.

Declaration statements. To create a variable in Java, you use a *declaration statement,* or just *declaration* for short A declaration includes a type followed by a variable name. Java reserves enough memory to store a data-type value of the specified type, and associates the variable name with that area of memory, so that it can access the value when you use the variable in later code. For economy, you can declare several variables of the same type in a single declaration statement.

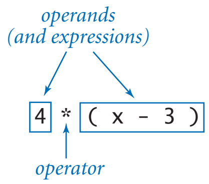


Anatomy of a declaration

Variable naming conventions. Programmers typically follow stylistic conventions when naming things. In this book, our convention is to give each variable a meaningful name that consists of a lowercase letter followed by lowercase letters, uppercase letters, and digits. We use uppercase letters to mark the words of a multi-word variable name. For example, we use the variable names i, x, y, sum, isLeapYear, and outDegrees, among many others. Programmers refer to this naming style as *camel case*.

Constant variables. We use the oxymoronic term *constant variable* to describe a variable whose value does not change during the execution of a program (or from one execution of the program to the next). In this book, our convention is to give each constant variable a name that consists of an uppercase letter followed by uppercase letters, digits, and underscores. For example, we might use the constant variable names SPEED_OF_LIGHT and DARK_RED.

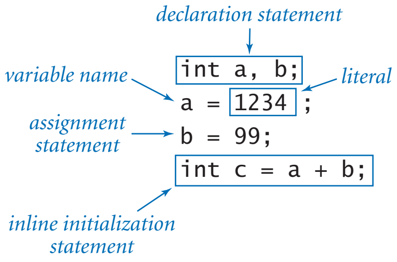
Expressions. An *expression* is a combination of literals, variables, and operations that Java *evaluates* to produce a value. For primitive types, expressions often look just like mathematical formulas, using *operators* to specify data-type operations to be performed on one more *operands*. Most of the operators that we use are *binary operators* that take exactly two operands, such as `x - 3` or `5 * x`. Each operand can be any expression, perhaps within parentheses. For example, we can write `4 * (x - 3)` or `5 * x - 6` and Java will understand what we mean. An expression is a directive to perform a sequence of operations; the expression is a representation of the resulting value.



Anatomy of an expression

Operator precedence. An expression is shorthand for a sequence of operations: in which order should the operators be applied? Java has natural and well defined *precedence* rules that fully specify this order. For arithmetic operations, multiplication and division are performed before addition and subtraction, so that `a - b * c` and `a - (b * c)` represent the same sequence of operations. When arithmetic operators have the same precedence, the order is determined by *left associativity*, so that `a - b - c` and `(a - b) - c` represent the same sequence of operations. You can use parentheses to override the rules, so you can write `a - (b - c)` if that is what you want. You might encounter in the future some Java code that depends subtly on precedence rules, but we use parentheses to avoid such code in this book. If you are interested, you can find full details on the rules on the booksite.

Assignment statements. An assignment statement associates a data-type value with a variable. When we write `c = a + b` in Java, we are not expressing mathematical equality, but are instead expressing an *action*: set the value of the variable `c` to be the value of `a` plus the value of `b`. It is true that the value of `c` is mathematically equal to the value of `a + b` immediately after the assignment statement has been executed, but the point of the statement is to change (or initialize) the value of `c`. The left-hand side of an assignment statement must be a single variable; the right-hand side can be any expression that produces a value of a compatible type. So, for example, both `1234 = a;` and `a + b = b + a;` are invalid statements in Java. In short, *the meaning of = is decidedly not the same as in mathematical equations.*



Using a primitive data type

Inline initialization. Before you can use a variable in an expression, you must first declare the variable and assign to it an initial value. Failure to do either results in a compile-time error. For economy, you can combine a declaration statement with an assignment statement in a construct known as an *inline initialization* statement. For example, the following code declares two variables `a` and `b`, and initializes them to the values `1234` and `99`, respectively:

```
int a = 1234;
int b = 99;
```

Most often, we declare and initialize a variable in this manner *at* the point of its first use in our program.

Tracing changes in variable values. As a final check on your understanding of the purpose of assignment statements, convince yourself that the following code *exchanges* the values of `a` and `b` (assume that `a` and `b` are `int` variables):

```
int t = a;
a = b;
b = t;
```

To do so, use a time-honored method of examining program behavior: study a table of the variable values after each statement (such a table is known as a *trace*).

	`a`	`b`	`t`
`int a, b;`	*undefined*	*undefined*	
`a = 1234;`	`1234`	*undefined*	
`b = 99;`	`1234`	`99`	
`int t = a;`	`1234`	`99`	`1234`
`a = b;`	`99`	`99`	`1234`
`b = t;`	`99`	`1234`	`1234`

Your first trace

Type safety. Java requires you to declare the type of every variable. This enables Java to check for type mismatch errors at compile time and alert you to potential bugs in your program. For example, you cannot assign a `double` value to an `int` variable, multiply a `String` with a `boolean`, or use an uninitialized variable within an expression. This situation is analogous to making sure that quantities have the proper units in a scientific application (for example, it does not make sense to add a quantity measured in inches to another measured in pounds).

Next, we consider these details for the basic built-in types that you will use most often (strings, integers, floating-point numbers, and true–false values), along with sample code illustrating their use. To understand how to use a data type, you need to know not just its defined set of values, but also which operations you can perform, the language mechanism for invoking the operations, and the conventions for specifying literals.

Characters and strings The `char` type represents individual alphanumeric characters or symbols, like the ones that you type. There are 2^{16} different possible `char` values, but we usually restrict attention to the ones that represent letters, numbers, symbols, and whitespace characters such as tab and newline. You can specify a `char` literal by enclosing a character within single quotes; for example, `'a'` represents the letter a. For tab, newline, backslash, single quote, and double quote, we use the special *escape sequences* `\t`, `\n`, `\\`, `\'`, and `\"`, respectively. The characters are encoded as 16-bit integers using an encoding scheme known as *Unicode*, and there are also escape sequences for specifying special characters not found on your keyboard (see the booksite). We usually do not perform any operations directly on characters other than assigning values to variables.

values	characters
typical literals	`'a'` `'\n'`

Java's built-in `char` data type

The `String` type represents sequences of characters. You can specify a `String` literal by enclosing a sequence of characters within double quotes, such as `"Hello, World"`. The `String` data type is *not* a primitive type, but Java sometimes treats it like one. For example, the *concatenation* operator (+) takes two `String` operands and produces a third `String` that is formed by appending the characters of the second operand to the characters of the first operand.

values	sequences of characters
typical literals	`"Hello, World"` `" * "`
operation	concatenate
operator	+

Java's built-in `String` data type

The concatenation operation (along with the ability to declare `String` variables and to use them in expressions and assignment statements) is sufficiently powerful to allow us to attack some nontrivial computing tasks. As an example, `Ruler` (Program 1.2.1) computes a table of values of the *ruler function* that describes the relative lengths of the marks on a ruler. One noteworthy feature of this computation is that it illustrates how easy it is to craft a short program that produces a huge amount of output. If you extend this program in the obvious way to print five lines, six lines, seven lines, and so forth, you will see that each time you add two statements to this program, you double the size of the output. Specifically, if the program prints n lines, the nth line contains $2^n - 1$ numbers. For example, if you were to add statements in this way so that the program prints 30 lines, it would print more than 1 *billion* numbers.

expression	*value*
`"Hi, " + "Bob"`	`"Hi, Bob"`
`"1" + " 2 " + "1"`	`"1 2 1"`
`"1234" + " + " + "99"`	`"1234 + 99"`
`"1234" + "99"`	`"123499"`

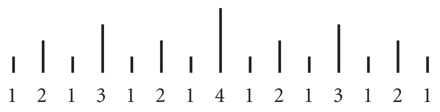
Typical `String` expressions

Program 1.2.1 String concatenation

```
public class Ruler
{
   public static void main(String[] args)
   {
      String ruler1 = "1";
      String ruler2 = ruler1 + " 2 " + ruler1;
      String ruler3 = ruler2 + " 3 " + ruler2;
      String ruler4 = ruler3 + " 4 " + ruler3;
      System.out.println(ruler1);
      System.out.println(ruler2);
      System.out.println(ruler3);
      System.out.println(ruler4);
   }
}
```

This program prints the relative lengths of the subdivisions on a ruler. The nth line of output is the relative lengths of the marks on a ruler subdivided in intervals of $1/2^n$ of an inch. For example, the fourth line of output gives the relative lengths of the marks that indicate intervals of one-sixteenth of an inch on a ruler.

```
% javac Ruler.java
% java Ruler
1
1 2 1
1 2 1 3 1 2 1
1 2 1 3 1 2 1 4 1 2 1 3 1 2 1
```



The ruler function for n = 4

Our most frequent use (by far) of the concatenation operation is to put together results of computation for output with `System.out.println()`. For example, we could simplify `UseArgument` (Program 1.1.2) by replacing its three statements in `main()` with this single statement:

```
System.out.println("Hi, " + args[0] + ". How are you?");
```

We have considered the `String` type first precisely because we need it for output (and command-line arguments) in programs that process not only strings but other types of data as well. Next we consider two convenient mechanisms in Java for converting numbers to strings and strings to numbers.

Converting numbers to strings for output. As mentioned at the beginning of this section, Java's built-in `String` type obeys special rules. One of these special rules is that you can easily convert a value of any type to a `String` value: whenever we use the + operator with a `String` as one of its operands, Java automatically converts the other operand to a `String`, producing as a result the `String` formed from the characters of the first operand followed by the characters of the second operand. For example, the result of these two code fragments

```
String a = "1234";             String a = "1234";
String b = "99";               int b = 99;
String c = a + b;              String c = a + b;
```

are both the same: they assign to `c` the value `"123499"`. We use this automatic conversion liberally to form `String` values for use with `System.out.print()` and `System.out.println()`. For example, we can write statements like this one:

```
System.out.println(a + " + " + b + " = " + c);
```

If `a`, `b`, and `c` are `int` variables with the values `1234`, `99`, and `1333`, respectively, then this statement prints the string `1234 + 99 = 1333`.

Converting strings to numbers for input. Java also provides library methods that convert the strings that we type as command-line arguments into numeric values for primitive types. We use the Java library methods `Integer.parseInt()` and `Double.parseDouble()` for this purpose. For example, typing `Integer.parseInt("123")` in program text is equivalent to typing the `int` literal `123`. If the user types `123` as the first command-line argument, then the code `Integer.parseInt(args[0])` converts the `String` value `"123"` into the `int` value `123`. You will see several examples of this usage in the programs in this section.

WITH THESE MECHANISMS, OUR VIEW OF each Java program as a black box that takes string arguments and produces string results is still valid, but we can now interpret those strings as numbers and use them as the basis for meaningful computations.

Integers The `int` type represents integers (natural numbers) between `-2147483648` (-2^{31}) and `2147483647` ($2^{31}-1$). These bounds derive from the fact that integers are represented in binary with 32 binary digits; there are 2^{32} possible values. (The term *binary digit* is omnipresent in computer science, and we nearly always use the abbreviation *bit*: a bit is either 0 or 1.) The range of possible `int` values is asymmetric because zero is included with the positive values. You can see the Q&A at the end of this section for more details about number representation, but in the present context it suffices to know that an `int` is one of the finite set of values in the range just given. You can specify an `int` literal with a sequence of the decimal digits `0` through `9` (that, when interpreted as decimal numbers, fall within the defined range). We use `int`s frequently because they naturally arise when we are implementing programs.

Standard arithmetic operators for addition/subtraction (`+` and `-`), multiplication (`*`), division (`/`), and remainder (`%`) for the `int` data type are built into Java. These operators take two `int` operands and produce an `int` result, with one significant exception—division or remainder by zero is not allowed. These operations are defined as in grade school (keeping in mind that all results must be integers): given two `int` values `a` and `b`, the value of `a / b` is the number of times `b` goes into `a` *with the fractional part discarded*, and the value of `a % b` is the remainder that you get when you divide `a` by `b`. For example, the value of `17 / 3` is `5`, and the value of `17 % 3` is `2`. The `int` results that we get from arithmetic operations are just what we expect, except that if the result is too large to fit into `int`'s 32-bit representation, then it will be truncated in a well-defined manner. This situation is known

expression	*value*	*comment*
`99`	`99`	*integer literal*
`+99`	`99`	*positive sign*
`-99`	`-99`	*negative sign*
`5 + 3`	`8`	*addition*
`5 - 3`	`2`	*subtraction*
`5 * 3`	`15`	*multiplication*
`5 / 3`	`1`	*no fractional part*
`5 % 3`	`2`	*remainder*
`1 / 0`		*run-time error*
`3 * 5 - 2`	`13`	** has precedence*
`3 + 5 / 2`	`5`	*/ has precedence*
`3 - 5 - 2`	`-4`	*left associative*
`( 3 - 5 ) - 2`	`-4`	*better style*
`3 - ( 5 - 2 )`	`0`	*unambiguous*

Typical `int` expressions

values	integers between -2^{31} and $+2^{31}-1$					
typical literals	`1234 99 0 1000000`					
operations	*sign*	*add*	*subtract*	*multiply*	*divide*	*remainder*
operators	`+ -`	`+`	`-`	`*`	`/`	`%`

Java's built-in `int` data type

Program 1.2.2 Integer multiplication and division

```
public class IntOps
{
   public static void main(String[] args)
   {
      int a = Integer.parseInt(args[0]);
      int b = Integer.parseInt(args[1]);
      int p = a * b;
      int q = a / b;
      int r = a % b;
      System.out.println(a + " * " + b + " = " + p);
      System.out.println(a + " / " + b + " = " + q);
      System.out.println(a + " % " + b + " = " + r);
      System.out.println(a + " = " + q + " * " + b + " + " + r);
   }
}
```

Arithmetic for integers is built into Java. Most of this code is devoted to the task of getting the values in and out; the actual arithmetic is in the simple statements in the middle of the program that assign values to p, q, and r.

```
% javac IntOps.java
% java IntOps 1234 99
1234 * 99 = 122166
1234 / 99 = 12
1234 % 99 = 46
1234 = 12 * 99 + 46
```

as *overflow*. In general, we have to take care that such a result is not misinterpreted by our code. For the moment, we will be computing with small numbers, so you do not have to worry about these boundary conditions.

Program 1.2.2 illustrates three basic operations (multiplication, division, and remainder) for manipulating integers,. It also demonstrates the use of `Integer.parseInt()` to convert `String` values on the command line to `int` values, as well as the use of automatic type conversion to convert `int` values to `String` values for output.

Three other built-in types are different representations of integers in Java. The `long`, `short`, and `byte` types are the same as `int` except that they use 64, 16, and 8 bits respectively, so the range of allowed values is accordingly different. Programmers use `long` when working with huge integers, and the other types to save space. You can find a table with the maximum and minimum values for each type on the booksite, or you can figure them out for yourself from the numbers of bits.

Floating-point numbers The `double` type represents *floating-point* numbers, for use in scientific and commercial applications. The internal representation is like scientific notation, so that we can compute with numbers in a huge range. We use floating-point numbers to represent real numbers, but they are decidedly not the same as real numbers! There are infinitely many real numbers, but we can represent only a finite number of floating-point numbers in any digital computer representation. Floating-point numbers do approximate real numbers sufficiently well that we can use them in applications, but we often need to cope with the fact that we cannot always do exact computations.

You can specify a `double` literal with a sequence of digits with a decimal point. For example, the literal `3.14159` represents a six-digit approximation to π. Alternatively, you specify a `double` literal with a notation like scientific notation: the literal `6.022e23` represents the number 6.022×10^{23}. As with integers, you can use these conventions to type floating-point literals in your programs or to provide floating-point numbers as string arguments on the command line.

expression	*value*
`3.141 + 2.0`	`5.141`
`3.141 - 2.0`	`1.111`
`3.141 / 2.0`	`1.5705`
`5.0 / 3.0`	`1.6666666666666667`
`10.0 % 3.141`	`0.577`
`1.0 / 0.0`	`Infinity`
`Math.sqrt(2.0)`	`1.4142135623730951`
`Math.sqrt(-1.0)`	`NaN`

Typical `double` expressions

The arithmetic operators +, -, *, and / are defined for `double`. Beyond these built-in operators, the Java `Math` library defines the square root function, trigonometric functions, logarithm/exponential functions, and other common functions for floating-point numbers. To use one of these functions in an expression, you type the name of the function followed by its argument in parentheses. For ex-

values	real numbers (specified by IEEE 754 standard)			
typical literals	`3.14159`	`6.022e23`	`2.0`	`1.4142135623730951`
operations	*add*	*subtract*	*multiply*	*divide*
operators	`+`	`-`	`*`	`/`

Java's built-in `double` data type

Program 1.2.3 Quadratic formula

```
public class Quadratic
{
   public static void main(String[] args)
   {
      double b = Double.parseDouble(args[0]);
      double c = Double.parseDouble(args[1]);
      double discriminant = b*b - 4.0*c;
      double d = Math.sqrt(discriminant);
      System.out.println((-b + d) / 2.0);
      System.out.println((-b - d) / 2.0);
   }
}
```

This program prints the roots of the polynomial $x^2 + bx + c$, using the quadratic formula. For example, the roots of $x^2 - 3x + 2$ are 1 and 2 since we can factor the equation as $(x - 1)(x - 2)$; the roots of $x^2 - x - 1$ are ϕ and $1 - \phi$, where ϕ is the golden ratio; and the roots of $x^2 + x + 1$ are not real numbers.

```
% javac Quadratic.java
% java Quadratic -3.0 2.0
2.0
1.0
```

```
% java Quadratic -1.0 -1.0
1.618033988749895
-0.6180339887498949
% java Quadratic 1.0 1.0
NaN
NaN
```

ample, the code `Math.sqrt(2.0)` evaluates to a `double` value that is approximately the square root of 2. We discuss the mechanism behind this arrangement in more detail in Section 2.1 and more details about the `Math` library at the end of this section.

When working with floating-point numbers, one of the first things that you will encounter is the issue of *precision*. For example, printing `5.0/2.0` results in `2.5` as expected, but printing `5.0/3.0` results in `1.6666666666666667`. In Section 1.5, you will learn Java's mechanism for controlling the number of significant digits that you see in output. Until then, we will work with the Java default output format.

The result of a calculation can be one of the special values `Infinity` (if the number is too large to be represented) or `NaN` (if the result of the calculation is undefined). Though there are myriad details to consider when calculations involve these values, you can use `double` in a natural way and begin to write Java programs instead of using a calculator for all kinds of calculations. For example, PROGRAM 1.2.3 shows the use of `double` values in computing the roots of a quadratic equation using the quadratic formula. Several of the exercises at the end of this section further illustrate this point.

As with `long`, `short`, and `byte` for integers, there is another representation for real numbers called `float`. Programmers sometimes use `float` to save space when precision is a secondary consideration. The `double` type is useful for about 15 significant digits; the `float` type is good for only about 7 digits. We do not use `float` in this book.

Booleans The `boolean` type represents truth values from logic. It has just two values: `true` and `false`. These are also the two possible `boolean` literals. Every `boolean` variable has one of these two values, and every `boolean` operation has operands and a result that takes on just one of these two values. This simplicity is deceiving—`boolean` values lie at the foundation of computer science.

values	*true or false*		
literals	`true`	`false`	
operations	and	or	not
operators	&&	||	!

Java's built-in `boolean` data type

The most important operations defined for `booleans` are *and* (&&), *or* (||), and *not* (!), which have familiar definitions:

- `a && b` is `true` if both operands are `true`, and `false` if either is `false`.
- `a || b` is `false` if both operands are `false`, and `true` if either is `true`.
- `!a` is `true` if `a` is `false`, and `false` if `a` is `true`.

Despite the intuitive nature of these definitions, it is worthwhile to fully specify each possibility for each operation in tables known as *truth tables*. The *not* function has only one operand: its value for each of the two possible values of the operand is

a	!a
true	false
false	true

a	b	a && b	a || b
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

Truth-table definitions of `boolean` operations specified in the second column. The *and* and *or* functions each have two operands: there are four different possibilities for operand values, and the values of the functions for each possibility are specified in the right two columns.

We can use these operators with parentheses to develop arbitrarily complex expressions, each of which specifies a well-defined boolean function. Often the same function appears in different guises. For example, the expressions `(a && b)` and `!(!a || !b)` are equivalent.

`a`	`b`	`a && b`	`!a`	`!b`	`!a || !b`	`!(!a || !b)`
`false`	`false`	`false`	`true`	`true`	`true`	`false`
`false`	`true`	`false`	`true`	`false`	`true`	`false`
`true`	`false`	`false`	`false`	`true`	`true`	`false`
`true`	`true`	`true`	`false`	`false`	`false`	`true`

Truth-table proof that `a && b` and `!(!a || !b)` are identical

The study of manipulating expressions of this kind is known as *Boolean logic*. This field of mathematics is fundamental to computing: it plays an essential role in the design and operation of computer hardware itself, and it is also a starting point for the theoretical foundations of computation. In the present context, we are interested in boolean expressions because we use them to control the behavior of our programs. Typically, a particular condition of interest is specified as a boolean expression, and a piece of program code is written to execute one set of statements if that expression is `true` and a different set of statements if the expression is `false`. The mechanics of doing so are the topic of Section 1.3.

Comparisons Some *mixed-type* operators take operands of one type and produce a result of another type. The most important operators of this kind are the comparison operators ==, !=, <, <=, >, and >=, which all are defined for each primitive numeric type and produce a `boolean` result. Since operations are defined only with respect to data types, each of these symbols stands for many operations, one for each data type. It is required that both operands be of the same type.

non-negative discriminant?	`(b*b - 4.0*a*c) >= 0.0`
beginning of a century?	`(year % 100) == 0`
legal month?	`(month >= 1) && (month <= 12)`

Typical comparison expressions

Program 1.2.4 Leap year

```
public class LeapYear
{
   public static void main(String[] args)
   {
      int year = Integer.parseInt(args[0]);
      boolean isLeapYear;
      isLeapYear = (year % 4 == 0);
      isLeapYear = isLeapYear && (year % 100 != 0);
      isLeapYear = isLeapYear || (year % 400 == 0);
      System.out.println(isLeapYear);
   }
}
```

This program tests whether an integer corresponds to a leap year in the Gregorian calendar. A year is a leap year if it is divisible by 4 (2004), unless it is divisible by 100 in which case it is not (1900), unless it is divisible by 400 in which case it is (2000).

```
% javac LeapYear.java
% java LeapYear 2004
true
% java LeapYear 1900
false
% java LeapYear 2000
true
```

Even without going into the details of number representation, it is clear that the operations for the various types are quite different. For example, it is one thing to compare two `int`s to check that `(2 <= 2)` is `true`, but quite another to compare two `double`s to check whether `(2.0 <= 0.002e3)` is `true`. Still, these operations are well defined and useful to write code that tests for conditions such as `(b*b - 4.0*a*c) >= 0.0`, which is frequently needed, as you will see.

The comparison operations have lower precedence than arithmetic operators and higher precedence than boolean operators, so you do not need the parentheses in an expression such as `(b*b - 4.0*a*c) >= 0.0`, and you could write an expression such as `month >= 1 && month <= 12` without parentheses to test whether the value of the `int` variable `month` is between 1 and 12. (It is better style to use the parentheses, however.)

Comparison operations, together with boolean logic, provide the basis for decision making in Java programs. Program 1.2.4 is an example of their use, and you can find other examples in the exercises at the end of this section. More importantly, in Section 1.3 we will see the role that boolean expressions play in more sophisticated programs.

operator	*meaning*	`true`	`false`
`==`	*equal*	`2 == 2`	`2 == 3`
`!=`	*not equal*	`3 != 2`	`2 != 2`
`<`	*less than*	`2 < 13`	`2 < 2`
`<=`	*less than or equal*	`2 <= 2`	`3 <= 2`
`>`	*greater than*	`13 > 2`	`2 > 13`
`>=`	*greater than or equal*	`3 >= 2`	`2 >= 3`

Comparisons with `int` operands and a `boolean` result

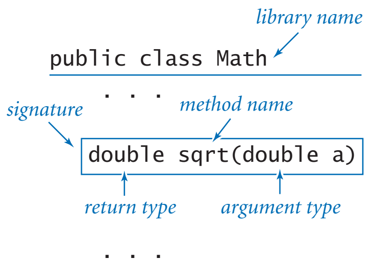
Library methods and APIs As we have seen, many programming tasks involve using Java library methods in addition to the built-in operators. The number of available library methods is vast. As you learn to program, you will learn to use more and more library methods, but it is best at the beginning to restrict your attention to a relatively small set of methods. In this chapter, you have already used some of Java's methods for printing, for converting data from one type to another, and for computing mathematical functions (the Java `Math` library). In later chapters, you will learn not just how to use other methods, but how to create and use your own methods.

For convenience, we will consistently summarize the library methods that you need to know how to use in tables like this one:

`void`	`System.out.print(String s)`	*print s*
`void`	`System.out.println(String s)`	*print s, followed by a newline*
`void`	`System.out.println()`	*print a newline*

Note: Any type of data can be used as argument (and will be automatically converted to `String`).

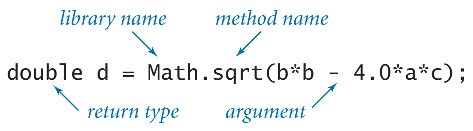
Java library methods for printing strings to the terminal



Anatomy of a method signature

Such a table is known as an *application programming interface* (*API*). Each method is described by a line in the API that specifies the information you need to know to use the method. The code in the tables is *not* the code that you type to use the method; it is known as the method's *signature*. The signature specifies the type of the arguments, the method name, and the type of the result that the method computes (the *return value*).

In your code, you can call a method by typing its name followed by arguments, enclosed in parentheses and separated by commas. When Java executes your program, we say that it *calls* (or *evaluates*) the method with the given arguments and that the method *returns* a value. A method call is an expression, so you can use a method call in the same way that you use variables and literals to build up more complicated expressions. For example, you can write expressions like `Math.sin(x) * Math.cos(y)` and so on. An argument is also an expression, so you can write code like `Math.sqrt(b*b - 4.0*a*c)` and Java knows what you mean—it evaluates the argument expression and passes the resulting value to the method.



Using a library method

The API tables on the facing page show some of the commonly used methods in Java's `Math` library, along with the Java methods we have seen for printing text to the terminal window and for converting strings to primitive types. The following table shows several examples of calls that use these library methods:

method call	*library*	*return type*	*value*
`Integer.parseInt("123")`	`Integer`	`int`	`123`
`Double.parseDouble("1.5")`	`Double`	`double`	`1.5`
`Math.sqrt(5.0*5.0 - 4.0*4.0)`	`Math`	`double`	`3.0`
`Math.log(Math.E)`	`Math`	`double`	`1.0`
`Math.random()`	`Math`	`double`	*random in* $[0, 1)$
`Math.round(3.14159)`	`Math`	`long`	`3`
`Math.max(1.0, 9.0)`	`Math`	`double`	`9.0`

Typical calls to Java library methods

`public class Math`

`double`	`abs(double a)`	*absolute value of* `a`
`double`	`max(double a, double b)`	*maximum of* `a` *and* `b`
`double`	`min(double a, double b)`	*minimum of* `a` *and* `b`

Note 1: `abs()`, `max()`, *and* `min()` *are defined also for* `int`, `long`, *and* `float`.

`double`	`sin(double theta)`	*sine of* `theta`
`double`	`cos(double theta)`	*cosine of* `theta`
`double`	`tan(double theta)`	*tangent of* `theta`

Note 2: Angles are expressed in radians. Use `toDegrees()` *and* `toRadians()` *to convert.*
Note 3: Use `asin()`, `acos()`, *and* `atan()` *for inverse functions.*

`double`	`exp(double a)`	*exponential* (e^a)
`double`	`log(double a)`	*natural log* ($\log_e$ `a`, *or* ln `a`)
`double`	`pow(double a, double b)`	*raise* `a` *to the* `b`*th power* (a^b)
`long`	`round(double a)`	*round* `a` *to the nearest integer*
`double`	`random()`	*random number in* [0, 1)
`double`	`sqrt(double a)`	*square root of* `a`
`double`	`E`	*value of e (constant)*
`double`	`PI`	*value of* π *(constant)*

See booksite for other available functions.

Excerpts from Java's `Math` *library*

`void`	`System.out.print(String s)`	*print* `s`
`void`	`System.out.println(String s)`	*print* `s`, *followed by a newline*
`void`	`System.out.println()`	*print a newline*

Java library methods for printing strings to the terminal

`int`	`Integer.parseInt(String s)`	*convert* `s` *to an* `int` *value*
`double`	`Double.parseDouble(String s)`	*convert* `s` *to a* `double` *value*
`long`	`Long.parseLong(String s)`	*convert* `s` *to a* `long` *value*

Java library methods for converting strings to primitive types

With three exceptions, the methods on the previous page are *pure*—given the same arguments, they always return the same value, without producing any observable *side effect*. The method `Math.random()` is impure because it returns potentially a different value each time it is called; the methods `System.out.print()` and `System.out.println()` are impure because they produce side effects—printing strings to the terminal. In APIs, we use a verb phrase to describe the behavior of a method that produces side effects; otherwise, we use a noun phrase to describe the return value. The keyword `void` designates a method that does not return a value (and whose main purpose is to produce side effects).

The `Math` library also defines the constant values `Math.PI` (for π) and `Math.E` (for e), which you can use in your programs. For example, the value of `Math.sin(Math.PI/2)` is `1.0` and the value of `Math.log(Math.E)` is `1.0` (because `Math.sin()` takes its argument in radians and `Math.log()` implements the natural logarithm function).

These APIs are typical of the online documentation that is the standard in modern programming. The extensive online documentation of the Java APIs is routinely used by professional programmers, and it is available to you (if you are interested) directly from the Java website or through our booksite. You do not need to go to the online documentation to understand the code in this book or to write similar code, because we present and explain in the text all of the library methods that we use in APIs like these and summarize them in the endpapers. More important, in Chapters 2 and 3 you will learn in this book how to develop your own APIs and to implement methods for your own use.

Type conversion One of the primary rules of modern programming is that you should always be aware of the type of data that your program is processing. Only by knowing the type can you know precisely which set of values each variable can have, which literals you can use, and which operations you can perform. For example, suppose that you wish to compute the average of the four integers `1`, `2`, `3`, and `4`. Naturally, the expression `(1 + 2 + 3 + 4) / 4` comes to mind, but it produces the `int` value `2` instead of the `double` value `2.5` because of type conversion conventions. The problem stems from the fact that the operands are `int` values but it is natural to expect a `double` value for the result, so conversion from `int` to `double` is necessary at some point. There are several ways to do so in Java.

Implicit type conversion. You can use an `int` value wherever a `double` value is expected, because Java automatically converts integers to doubles when appropriate. For example, `11*0.25` evaluates to `2.75` because `0.25` is a `double` and both operands need to be of the same type; thus, `11` is converted to a `double` and then the result of dividing two `double`s is a `double`. As another example, `Math.sqrt(4)` evaluates to `2.0` because `4` is converted to a `double`, as expected by `Math.sqrt()`, which then returns a `double` value. This kind of conversion is called *automatic promotion* or *coercion*. Automatic promotion is appropriate because your intent is clear and it can be done with no loss of information. In contrast, a conversion that might involve loss of information (for example, assigning a `double` value to an `int` variable) leads to a compile-time error.

Explicit cast. Java has some built-in type conversion conventions for primitive types that you can take advantage of when you are aware that you might lose information. You have to make your intention to do so explicit by using a device called a *cast*. You cast an expression from one primitive type to another by prepending the desired type name within parentheses. For example, the expression `(int) 2.71828` is a cast from `double` to `int` that produces an `int` with value `2`. The conversion methods defined for casts throw away information in a reasonable way (for a full list, see the booksite). For example, casting a floating-point number to an integer discards the fractional part by rounding toward zero. `RandomInt` (Program 1.2.5) is an example that uses a cast for a practical computation.

expression	*expression type*	*expression value*
`(1 + 2 + 3 + 4) / 4.0`	`double`	`2.5`
`Math.sqrt(4)`	`double`	`2.0`
`"1234" + 99`	`String`	`"123499"`
`11 * 0.25`	`double`	`2.75`
`(int) 11 * 0.25`	`double`	`2.75`
`11 * (int) 0.25`	`int`	`0`
`(int) (11 * 0.25)`	`int`	`2`
`(int) 2.71828`	`int`	`2`
`Math.round(2.71828)`	`long`	`3`
`(int) Math.round(2.71828)`	`int`	`3`
`Integer.parseInt("1234")`	`int`	`1234`

Typical type conversions

Casting has higher precedence than arithmetic operations—any cast is applied to the value that immediately follows it. For example, if we write `int value = (int) 11 * 0.25`, the cast is no help: the literal `11` is already an integer, so the cast `(int)` has no effect. In this example, the compiler produces a `possible loss of precision` error message because there would be a loss

Program 1.2.5 Casting to get a random integer

```
public class RandomInt
{
   public static void main(String[] args)
   {
      int n = Integer.parseInt(args[0]);
      double r = Math.random();    // uniform between 0.0 and 1.0
      int value = (int) (r * n);   // uniform between 0 and n-1
      System.out.println(value);
   }
}
```

This program uses the Java method `Math.random()` to generate a random number `r` between `0.0` (inclusive) and `1.0` (exclusive); then multiplies `r` by the command-line argument `n` to get a random number greater than or equal to `0` and less than `n`; then uses a cast to truncate the result to be an integer `value` between `0` and `n-1`.

```
% javac RandomInt.java
% java RandomInt 1000
548
% java RandomInt 1000
141
% java RandomInt 1000000
135032
```

of precision in converting the resulting value (`2.75`) to an `int` for assignment to `value`. The error is helpful because the intended computation for this code is likely `(int) (11 * 0.25)`, which has the value `2`, not `2.75`.

Explicit type conversion. You can use a method that takes an argument of one type (the value to be converted) and produces a result of another type. We have already used the `Integer.parseInt()` and `Double.parseDouble()` library methods to convert `String` values to `int` and `double` values, respectively. Many other methods are available for conversion among other types. For example, the library method `Math.round()` takes a `double` argument and returns a `long` result: the nearest integer to the argument. Thus, for example, `Math.round(3.14159)` and `Math.round(2.71828)` are both of type `long` and have the same value (`3`). If you want to convert the result of `Math.round()` to an `int`, you must use an explicit cast.

Beginning programmers tend to find type conversion to be an annoyance, but experienced programmers know that paying careful attention to data types is a key to success in programming. It may also be a key to avoiding failure: in a famous incident in 1985, a French rocket exploded in midair because of a type-conversion problem. While a bug in your program may not cause an explosion, it is well worth your while to take the time to understand what type conversion is all about. After you have written just a few programs, you will see that an understanding of data types will help you not only compose compact code but also make your intentions explicit and avoid subtle bugs in your programs.

Photo: ESA

Explosion of Ariane 5 rocket

Summary *A data type is a set of values and a set of operations on those values.* Java has eight primitive data types: `boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, and `double`. In Java code, we use operators and expressions like those in familiar mathematical expressions to invoke the operations associated with each type. The `boolean` type is used for computing with the logical values `true` and `false`; the `char` type is the set of character values that we type; and the other six numeric types are used for computing with numbers. In this book, we most often use `boolean`, `int`, and `double`; we do not use `short` or `float`. Another data type that we use frequently, `String`, is not primitive, but Java has some built-in facilities for `String`s that are like those for primitive types.

When programming in Java, we have to be aware that every operation is defined only in the context of its data type (so we may need type conversions) and that all types can have only a finite number of values (so we may need to live with imprecise results).

The `boolean` type and its operations—`&&`, `||`, and `!`—are the basis for logical decision making in Java programs, when used in conjunction with the mixed-type comparison operators `==`, `!=`, `<`, `>`, `<=`, and `>=`. Specifically, we use boolean expressions to control Java's conditional (`if`) and loop (`for` and `while`) constructs, which we will study in detail in the next section.

The numeric types and Java's libraries give us the ability to use Java as an extensive mathematical calculator. We write arithmetic expressions using the built-in operators `+`, `-`, `*`, `/`, and `%` along with Java methods from the `Math` library.

Although the programs in this section are quite rudimentary by the standards of what we will be able to do after the next section, this class of programs is quite useful in its own right. You will use primitive types and basic mathematical functions extensively in Java programming, so the effort that you spend now in understanding them will certainly be worthwhile.

Q&A (Strings)

Q. How does Java store strings internally?

A. Strings are sequences of characters that are encoded with Unicode, a modern standard for encoding text. Unicode supports more than 100,000 different characters, including more than 100 different languages plus mathematical and musical symbols.

Q. Can you use < and > to compare `String` values?

A. No. Those operators are defined only for primitive-type values.

Q. How about == and `!=`?

A. Yes, but the result may not be what you expect, because of the meanings these operators have for nonprimitive types. For example, there is a distinction between a `String` and its value. The expression `"abc" == "ab" + x` is `false` when `x` is a `String` with value `"c"` because the two operands are stored in different places in memory (even though they have the same value). This distinction is essential, as you will learn when we discuss it in more detail in Section 3.1.

Q. How can I compare two strings like words in a book index or dictionary?

A. We defer discussion of the `String` data type and associated methods until Section 3.1, where we introduce object-oriented programming. Until then, the string concatenation operation suffices.

Q. How can I specify a string literal that is too long to fit on a single line?

A. You can't. Instead, divide the string literal into independent string literals and concatenate them together, as in the following example:

```
String dna = "ATGCGCCCACAGCTGCGTCTAAACCGGACTCTG" +
             "AAGTCCGGAAATTACACCTGTTAG";
```

Q&A (Integers)

Q. How does Java store integers internally?

A. The simplest representation is for small positive integers, where the *binary number system* is used to represent each integer with a fixed amount of computer memory.

Q. What's the binary number system?

A. In the *binary number system*, we represent an integer as a sequence of *bits*. A bit is a single binary (base 2) digit—either 0 or 1—and is the basis for representing information in computers. In this case the bits are coefficients of powers of 2. Specifically, the sequence of bits $b_n b_{n-1} \ldots b_2 b_1 b_0$ represents the integer

$$b_n 2^n + b_{n-1} 2^{n-1} + \ldots + b_2 2^2 + b_1 2^1 + b_0 2^0$$

For example, 1100011 represents the integer

$$99 = 1\cdot 64 + 1\cdot 32 + 0\cdot 16 + 0\cdot 8 + 0\cdot 4 + 1\cdot 2 + 1\cdot 1$$

The more familiar *decimal number system* is the same except that the digits are between 0 and 9 and we use powers of 10. Converting a number to binary is an interesting computational problem that we will consider in the next section. Java uses 32 bits to represent `int` values. For example, the decimal integer 99 might be represented with the 32 bits `00000000000000000000000001100011`.

Q. How about negative numbers?

A. Negative numbers are handled with a convention known as *two's complement*, which we need not consider in detail. This is why the range of `int` values in Java is –2147483648 (-2^{31}) to 2147483647 ($2^{31} - 1$). One surprising consequence of this representation is that `int` values can become negative when they get large and *overflow* (exceed 2147483647). If you have not experienced this phenomenon, see Exercise 1.2.10. A safe strategy is to use the `int` type when you know the integer values will be fewer than ten digits and the `long` type when you think the integer values might get to be ten digits or more.

Q. It seems wrong that Java should just let `int`s overflow and give bad values. Shouldn't Java automatically check for overflow?

A. Yes, this issue is a contentious one among programmers. The short answer for now is that the lack of such checking is one reason such types are called *primitive* data types. A little knowledge can go a long way in avoiding such problems. Again, it is fine to use the `int` type for small numbers, but when values run into the billions, you cannot.

Q. What is the value of `Math.abs(-2147483648)`?

A. `-2147483648`. This strange (but true) result is a typical example of the effects of integer overflow and two's complement representation.

Q. What do the expressions `1 / 0` and `1 % 0` evaluate to in Java?

A. Each generates a run-time *exception*, for division by zero.

Q. What is the result of division and remainder for negative integers?

A. The quotient `a / b` rounds toward `0`; the remainder `a % b` is defined such that `(a / b) * b + a % b` is always equal to `a`. For example, `-14 / 3` and `14 / -3` are both `-4`, but `-14 % 3` is `-2` and `14 % -3` is `2`. Some other languages (including Python) have different conventions when dividing by negative integers.

Q. Why is the value of `10 ^ 6` not `1000000` but `12`?

A. The `^` operator is not an exponentiation operator, which you must have been thinking. Instead, it is the *bitwise exclusive or* operator, which is seldom what you want. Instead, you can use the literal `1e6`. You could also use `Math.pow(10, 6)` but doing so is wasteful if you are raising 10 to a known power.

Q&A (Floating-Point Numbers)

Q. Why is the type for real numbers named `double`?

A. The decimal point can "float" across the digits that make up the real number. In contrast, with integers the (implicit) decimal point is fixed after the least significant digit.

Q. How does Java store floating-point numbers internally?

A. Java follows the IEEE 754 standard, which supported in hardware by most modern computer systems. The standard specifies that a floating-point number is stored using three fields: sign, mantissa, and exponent. If you are interested, see the booksite for more details. The IEEE 754 standard also specifies how special floating-point values—positive zero, negative zero, positive infinity, negative infinity, and NaN (not a number)—should be handled. In particular, floating-point arithmetic never leads to a run-time exception. For example, the expression `-0.0/3.0` evaluates to `-0.0`, the expression `1.0/0.0` evaluates to positive infinity, and `Math.sqrt(-2.0)` evaluates to NaN.

Q. Fifteen digits for floating-point numbers certainly seems enough to me. Do I really need to worry much about precision?

A. Yes, because you are used to mathematics based on real numbers with infinite precision, whereas the computer always deals with finite approximations. For example, the expression `(0.1 + 0.1 == 0.2)` evaluates to `true` but the expression `(0.1 + 0.1 + 0.1 == 0.3)` evaluates to `false`! Pitfalls like this are not at all unusual in scientific computing. Novice programmers should avoid comparing two floating-point numbers for equality.

Q. How can I initialize a `double` variable to NaN or infinity?

A. Java has built-in constants available for this purpose: `Double.NaN`, `Double.POSITIVE_INFINITY`, and `Double.NEGATIVE_INFINITY`.

Q. Are there functions in Java's `Math` library for other trigonometric functions, such as cosecant, secant, and cotangent?

A. No, but you could use `Math.sin()`, `Math.cos()`, and `Math.tan()` to compute them. Choosing which functions to include in an API is a tradeoff between the convenience of having every function that you need and the annoyance of having to find one of the few that you need in a long list. No choice will satisfy all users, and the Java designers have many users to satisfy. Note that there are plenty of redundancies even in the APIs that we have listed. For example, you could use `Math.sin(x)/Math.cos(x)` instead of `Math.tan(x)`.

Q. It is annoying to see all those digits when printing a `double`. Can we arrange `System.out.println()` to print just two or three digits after the decimal point?

A. That sort of task involves a closer look at the method used to convert from `double` to `String`. The Java library function `System.out.printf()` is one way to do the job, and it is similar to the basic printing method in the C programming language and many modern languages, as discussed in Section 1.5. Until then, we will live with the extra digits (which is not all bad, since doing so helps us to get used to the different primitive types of numbers).

Q&A (Variables and Expressions)

Q. What happens if I forget to declare a variable?

A. The compiler complains when you refer to that variable in an expression. For example, `IntOpsBad` is the same as Program 1.2.2 except that the variable `p` is not declared (to be of type `int`).

```
% javac IntOpsBad.java
IntOpsBad.java:7: error: cannot find symbol
      p = a * b;
      ^
  symbol:   variable p
  location: class IntOpsBad
IntOpsBad.java:10: error: cannot find symbol
      System.out.println(a + " * " + b + " = " + p);
                                                 ^
  symbol:   variable p
  location: class IntOpsBad
2 errors
```

The compiler says that there are two errors, but there is really just one: the declaration of `p` is missing. If you forget to declare a variable that you use often, you will get quite a few error messages. A good strategy is to correct the *first* error and check that correction before addressing later ones.

Q. What happens if I forget to initialize a variable?

A. The compiler checks for this condition and will give you a `variable might not have been initialized` error message if you try to use the variable in an expression before you have initialized it.

Q. Is there a difference between the `=` and `==` operators?

A. Yes, they are quite different! The first is an assignment operator that changes the value of a variable, and the second is a comparison operator that produces a `boolean` result. Your ability to understand this answer is a sure test of whether you understood the material in this section. Think about how you might explain the difference to a friend.

Q. Can you compare a `double` to an `int`?

A. Not without doing a type conversion, but remember that Java usually does the requisite type conversion automatically. For example, if `x` is an `int` with the value `3`, then the expression `(x < 3.1)` is `true`—Java converts `x` to `double` (because `3.1` is a `double` literal) before performing the comparison.

Q. Will the statement `a = b = c = 17;` assign the value `17` to the three integer variables `a`, `b`, and `c`?

A. Yes. It works because an assignment statement in Java is also an expression (that evaluates to its right-hand side) and the assignment operator is right associative. As a matter of style, we do not use such *chained assignments* in this book.

Q. Will the expression `(a < b < c)` test whether the values of three integer variables `a`, `b`, and `c` are in strictly ascending order?

A. No, it will not compile because the expression `a < b` produces a `boolean` value, which would then be compared to an `int` value. Java does not support *chained comparisons*. Instead, you need to write `(a < b && b < c)`.

Q. Why do we write `(a && b)` and not `(a & b)`?

A. Java also has an `&` operator that you may encounter if you pursue advanced programming courses.

Q. What is the value of `Math.round(6.022e23)`?

A. You should get in the habit of typing in a tiny Java program to answer such questions yourself (and trying to understand why your program produces the result that it does).

Q. I've heard Java referred to as a *statically typed* language. What does this mean?

A. Static typing means that the type of every variable and expression is known at compile time. Java also verifies and enforces type constraints at compile time; for example, your program will not compile if you attempt to store a value of type `double` in a variable of type `int` or call `Math.sqrt()` with a `String` argument.

Exercises

1.2.1 Suppose that `a` and `b` are `int` variables. What does the following sequence of statements do?

```
int t = a; b = t; a = b;
```

1.2.2 Write a program that uses `Math.sin()` and `Math.cos()` to check that the value of $\cos^2\theta + \sin^2\theta$ is approximately 1 for any θ entered as a command-line argument. Just print the value. Why are the values not always exactly 1?

1.2.3 Suppose that `a` and `b` are `boolean` variables. Show that the expression

```
(!(a && b) && (a || b)) || ((a && b) || !(a || b))
```

evaluates to `true`.

1.2.4 Suppose that `a` and `b` are `int` variables. Simplify the following expression: `(!(a < b) && !(a > b))`.

1.2.5 The *exclusive or* operator `^` for `boolean` operands is defined to be `true` if they are different, `false` if they are the same. Give a truth table for this function.

1.2.6 Why does `10/3` give `3` and not `3.333333333`?

Solution. Since both `10` and `3` are integer literals, Java sees no need for type conversion and uses integer division. You should write `10.0/3.0` if you mean the numbers to be `double` literals. If you write `10/3.0` or `10.0/3`, Java does implicit conversion to get the same result.

1.2.7 What does each of the following print?

a. `System.out.println(2 + "bc");`

b. `System.out.println(2 + 3 + "bc");`

c. `System.out.println((2+3) + "bc");`

d. `System.out.println("bc" + (2+3));`

e. `System.out.println("bc" + 2 + 3);`

Explain each outcome.

1.2.8 Explain how to use Program 1.2.3 to find the square root of a number.

1.2.9 What does each of the following print?

a. `System.out.println('b');`

b. `System.out.println('b' + 'c');`

c. `System.out.println((char) ('a' + 4));`

Explain each outcome.

1.2.10 Suppose that a variable `a` is declared as `int a = 2147483647` (or equivalently, `Integer.MAX_VALUE`). What does each of the following print?

a. `System.out.println(a);`

b. `System.out.println(a+1);`

c. `System.out.println(2-a);`

d. `System.out.println(-2-a);`

e. `System.out.println(2*a);`

f. `System.out.println(4*a);`

Explain each outcome.

1.2.11 Suppose that a variable `a` is declared as `double a = 3.14159`. What does each of the following print?

a. `System.out.println(a);`

b. `System.out.println(a+1);`

c. `System.out.println(8/(int) a);`

d. `System.out.println(8/a);`

e. `System.out.println((int) (8/a));`

Explain each outcome.

1.2.12 Describe what happens if you write `sqrt` instead of `Math.sqrt` in PROGRAM 1.2.3.

1.2.13 Evaluate the expression `(Math.sqrt(2) * Math.sqrt(2) == 2)`.

1.2.14 Write a program that takes two positive integers as command-line arguments and prints `true` if either evenly divides the other.

1.2.15 Write a program that takes three positive integers as command-line arguments and prints `false` if any one of them is greater than or equal to the sum of the other two and `true` otherwise. (*Note*: This computation tests whether the three numbers could be the lengths of the sides of some triangle.)

1.2.16 A physics student gets unexpected results when using the code

```
double force = G * mass1 * mass2 / r * r;
```

to compute values according to the formula $F = Gm_1m_2 / r^2$. Explain the problem and correct the code.

1.2.17 Give the value of the variable `a` after the execution of each of the following sequences of statements:

```
int a = 1;    boolean a = true;    int a = 2;
a = a + a;    a = !a;              a = a * a;
a = a + a;    a = !a;              a = a * a;
a = a + a;    a = !a;              a = a * a;
```

1.2.18 Write a program that takes two integer command-line arguments `x` and `y` and prints the Euclidean distance from the point (`x`, `y`) to the origin (0, 0).

1.2.19 Write a program that takes two integer command-line arguments `a` and `b` and prints a random integer between `a` and `b`, inclusive.

1.2.20 Write a program that prints the sum of two random integers between `1` and `6` (such as you might get when rolling dice).

1.2.21 Write a program that takes a `double` command-line argument `t` and prints the value of $\sin(2t) + \sin(3t)$.

1.2.22 Write a program that takes three `double` command-line arguments x_0, v_0, and t and prints the value of $x_0 + v_0 t - g t^2/2$, where g is the constant 9.80665. (*Note*: This value is the displacement in meters after t seconds when an object is thrown straight up from initial position x_0 at velocity v_0 meters per second.)

1.2.23 Write a program that takes two integer command-line arguments `m` and `d` and prints `true` if day `d` of month `m` is between 3/20 and 6/20, `false` otherwise.

Creative Exercises

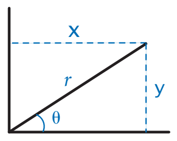
1.2.24 *Continuously compounded interest.* Write a program that calculates and prints the amount of money you would have after t years if you invested P dollars at an annual interest rate r (compounded continuously). The desired value is given by the formula Pe^{rt}.

1.2.25 *Wind chill.* Given the temperature T (in degrees Fahrenheit) and the wind speed v (in miles per hour), the National Weather Service defines the effective temperature (the wind chill) as follows:

$$w = 35.74 + 0.6215\,T + (0.4275\,T - 35.75)\,v^{0.16}$$

Write a program that takes two `double` command-line arguments `temperature` and `velocity` and prints the wind chill. Use `Math.pow(a, b)` to compute a^b. *Note*: The formula is not valid if T is larger than 50 in absolute value or if v is larger than 120 or less than 3 (you may assume that the values you get are in that range).

1.2.26 *Polar coordinates.* Write a program that converts from Cartesian to polar coordinates. Your program should accept two `double` command-line arguments `x` and `y` and print the polar coordinates r and θ. Use the method `Math.atan2(y, x)` to compute the arctangent value of `y/x` that is in the range from $-\pi$ to π.



Polar coordinates

1.2.27 *Gaussian random numbers.* Write a program `RandomGaussian` that prints a random number r drawn from the *Gaussian distribution*. One way to do so is to use the *Box–Muller* formula

$$r = \sin(2\,\pi\,v)\,(-2\ln u)^{1/2}$$

where u and v are real numbers between 0 and 1 generated by the `Math.random()` method.

1.2.28 *Order check.* Write a program that takes three `double` command-line arguments `x`, `y`, and `z` and prints `true` if the values are strictly ascending or descending ($x < y < z$ or $x > y > z$), and `false` otherwise.

1.2.29 *Day of the week.* Write a program that takes a date as input and prints the day of the week that date falls on. Your program should accept three `int` command-line arguments: `m` (month), `d` (day), and `y` (year). For `m`, use `1` for January, `2` for February, and so forth. For output, print `0` for Sunday, `1` for Monday, `2` for Tuesday, and so forth. Use the following formulas, for the Gregorian calendar:

$$\begin{aligned} y_0 &= y - (14 - m) / 12 \\ x &= y_0 + y_0 / 4 - y_0 / 100 + y_0 / 400 \\ m_0 &= m + 12 \times ((14 - m) / 12) - 2 \\ d_0 &= (d + x + (31 \times m_0) / 12) \% 7 \end{aligned}$$

Example: On which day of the week did February 14, 2000 fall?

$$\begin{aligned} y_0 &= 2000 - 1 = 1999 \\ x &= 1999 + 1999 / 4 - 1999 / 100 + 1999 / 400 = 2483 \\ m_0 &= 2 + 12 \times 1 - 2 = 12 \\ d_0 &= (14 + 2483 + (31 \times 12) / 12) \% 7 = 2500 \% 7 = 1 \end{aligned}$$

Answer: Monday.

1.2.30 *Uniform random numbers.* Write a program that prints five uniform random numbers between 0 and 1, their average value, and their minimum and maximum values. Use `Math.random()`, `Math.min()`, and `Math.max()`.

1.2.31 *Mercator projection.* The *Mercator projection* is a conformal (angle-preserving) projection that maps latitude φ and longitude λ to rectangular coordinates (x, y). It is widely used—for example, in nautical charts and in the maps that you print from the web. The projection is defined by the equations $x = \lambda - \lambda_0$ and $y = 1/2 \ln((1 + \sin\varphi) / (1 - \sin\varphi))$, where λ_0 is the longitude of the point in the center of the map. Write a program that takes λ_0 and the latitude and longitude of a point from the command line and prints its projection.

1.2.32 *Color conversion.* Several different formats are used to represent color. For example, the primary format for LCD displays, digital cameras, and web pages, known as the *RGB format,* specifies the level of red (R), green (G), and blue (B) on an integer scale from 0 to 255. The primary format for publishing books and magazines, known as the *CMYK format*, specifies the level of cyan (C), magenta

(M), yellow (Y), and black (K) on a real scale from 0.0 to 1.0. Write a program `RGBtoCMYK` that converts RGB to CMYK. Take three integers—`r`, `g`, and `b`—from the command line and print the equivalent CMYK values. If the RGB values are all 0, then the CMY values are all 0 and the K value is 1; otherwise, use these formulas:

$$\begin{aligned} w &= \max\,(\,r/255,\, g/255,\, b/255\,) \\ c &= (w - (r/255))\,/\,w \\ m &= (w - (g/255))\,/\,w \\ y &= (w - (b/255))\,/\,w \\ k &= 1 - w \end{aligned}$$

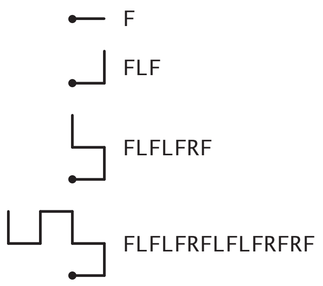
1.2.33 *Great circle.* Write a program `GreatCircle` that takes four `double` command-line arguments—`x1`, `y1`, `x2`, and `y2`—(the latitude and longitude, in degrees, of two points on the earth) and prints the great-circle distance between them. The great-circle distance (in nautical miles) is given by the following equation:

$$d = 60 \arccos(\sin(x_1)\sin(x_2) + \cos(x_1)\cos(x_2)\cos(y_1 - y_2))$$

Note that this equation uses degrees, whereas Java's trigonometric functions use radians. Use `Math.toRadians()` and `Math.toDegrees()` to convert between the two. Use your program to compute the great-circle distance between Paris (48.87° N and −2.33° W) and San Francisco (37.8° N and 122.4° W).

1.2.34 *Three-sort.* Write a program that takes three integer command-line arguments and prints them in ascending order. Use `Math.min()` and `Math.max()`.

1.2.35 *Dragon curves.* Write a program to print the instructions for drawing the dragon curves of order 0 through 5. The instructions are strings of `F`, `L`, and `R` characters, where `F` means "draw line while moving 1 unit forward," `L` means "turn left," and `R` means "turn right." A dragon curve of order n is formed when you fold a strip of paper in half n times, then unfold to right angles. The key to solving this problem is to note that a curve of order n is a curve of order $n-1$ followed by an `L` followed by a curve of order $n-1$ traversed in reverse order, and then to figure out a similar description for the reverse curve.



Dragon curves of order 0, 1, 2, and 3