CHAPTER 1

Getting Started

1 Getting Started

She starts—she moves—she seems to feel
The thrill of life along her keel.

Henry Wadsworth Longfellow, *The Building of the Ship*

Introduction

This chapter introduces you to the Java language and gives you enough details to allow you to write simple programs involving expressions, assignments, and console output. The details about assignments and expressions are similar to that of most other high-level languages. Every language has its own way of handling strings and console output, so even the experienced programmer should look at that material. Even if you are already an experienced programmer in some language other than Java, you should read at least the subsection entitled "A Sample Java Application Program" in Section 1.1 and preferably all of Section 1.1, and you should read all of Section 1.3 on strings and at least skim Section 1.4 to find out about Java defined constants and comments.

Prerequisites

This book is self contained and requires no preparation other than some simple high school algebra.

1.1 Introduction to Java

Eliminating the middle man is not necessarily a good idea.

Found in my old economics class notes

In this section we give you an overview of the Java programming language.

ORIGINS OF THE JAVA LANGUAGE

Java is well-known as a programming language for Internet applications. However, this book, and many other books and programmers, view Java as a general-purpose programming language that is suitable for most any application whether it involves the Internet or not. The first version of Java was neither of these things, but it evolved into both of these things.

In 1991, James Gosling lead a team at Sun Microsystems that developed the first version of Java (which was not yet called *Java*). This first version of the language was designed for programming home appliances, such as washing machines and television sets. Although that may not be a very glamorous application area, it is no easy task to design such a language. Home appliances are controlled by a wide variety of computer processors (chips). The language that Gosling was designing needed to work on all these different processors. Moreover, a home appliance is typically an inexpensive item, so the manufacturer would be unwilling to invest large amounts of money into developing complicated compilers. (A compiler is a program that translates the program into a language the processor can understand.) To simplify the tasks of writing compilers (translation programs) for each class of appliances, the team used a two-step translation process. The programs are first translated into an **intermediate language** that is the same for all appliances (or all computers), then a small, easy-to-write, and hence inexpensive, program translates this intermediate language into the machine language for a particular appliance or computer. This intermediate language is called **Java byte-code** or simply **byte-code**. Since there is only one intermediate language, the hardest step of the two-step translation from program to intermediate language to machine language is the same for all appliances (or all computers), and hence, most of the cost of translating to multiple machine languages was saved. The language for programming appliances never caught on with appliance manufacturers, but the Java language into which it evolved has become a widely used programming language.

intermediate language

byte-code

Why call it *byte-code*? The word **code** is commonly used to mean a program or part of a program. A byte is a small unit of storage (eight bits to be precise). Computer-readable information is typically organized into bytes. So the term *byte-code* suggests a program that is readable by a computer as opposed to a person.

code

In 1994, Patrick Naughton and Jonathan Payne at Sun Microsystems developed a Web browser that could run (Java) programs over the Internet. That Web browser has evolved into the browser known as HotJava. This was the start of Java's connection to the Internet. In the fall of 1995, Netscape Incorporated made its Web browser capable of running Java programs. Other companies followed suit and have developed software that accommodates Java programs.

OBJECTS AND METHODS

Java is an **object-oriented programming** language, abbreviated **OOP**. What is OOP? The world around us is made up of objects, such as people, automobiles, buildings, streets, adding machines, papers, and so forth. Each of these objects has the ability to perform certain actions, and each of these actions has some effect on some of the other objects in the world. OOP is a programming methodology that views a program as similarly consisting of objects that interact with each other by means of actions.

OOP

Object-oriented programming has its own specialized terminology. The objects are called, appropriately enough, **objects**. The actions that an object can take are called

object

Why Is the Language Named "Java"?

The current custom is to name programming languages according to the whims of their designers. Java is no exception. There are conflicting explanations of the origin of the name "Java." Despite these conflicting stories, one thing is clear: The word "Java" does not refer to any property or serious history of the Java language. One believable story about where the name "Java" came from is that the name was thought of when, after a fruitless meeting trying to come up with a new name for the language, the development team went out for coffee, and hence the inspiration for the name "Java."

method
class

methods. Objects of the same kind are said to have the same *type* or, more often, are said to be in the same **class**. For example, in an airport simulation program, all the simulated airplanes might belong to the same class, probably called the `Airplane` class. All objects within a class have the same methods. Thus, in a simulation program, all airplanes have the same methods (or possible actions) such as taking off, flying to a specific location, landing, and so forth. However, all simulated airplanes are not identical. They can have different characteristics, which are indicated in the program by associating different data (that is, some different information) with each particular airplane object. For example, the data associated with an airplane object might be two numbers for its speed and altitude.

If you have used some other programming language, it might help to explain Java terminology in terms of the terminology used in other languages. Things that are called *procedures*, *methods*, *functions*, or *subprograms* in other languages are all called *methods* in Java. In Java, all methods (and for that matter, all programming constructs whatsoever) are part of a class. As we will see, a Java **application program** is a class with a method named `main`, and when you run the Java program, the run-time system automatically invokes the method named `main` (that is, it automatically initiates the `main` action). An application program is a "regular" Java program; as we are about to see, there is another kind of Java program known as an applet. Other Java terminology is pretty much the same as the terminology in most other programming languages and, in any case, will be explained when each concept is introduced.

application
program

APPLETS

applet
application

There are two kinds of Java programs, *applets* and *applications*. An application or application program is just a regular program. Although the name *applet* may sound like it has something to do with apples, the name really means a *little Java application*, not a little apple. Applets and applications are almost identical. The difference is that applications are meant to be run on your computer like any other program, whereas an **applet** is meant to be run from a Web browser, and so can be sent to another location on the Internet and run there. Applets always use a windowing interface, but not all programs with a windowing interface are applets, as you will see in Chapters 16–18.

Although applets were designed to be run from a Web browser, they can also be run with a program known as an **applet viewer**. The applet viewer is really meant as a debugging aid and not as the final environment to allow users to run applets. Nonetheless, applets are now often run as stand-alone programs using an applet viewer.[1] We find this to be a somewhat unfortunate accident of history. Java has multiple libraries of software for designing windowing interfaces that run with no connection to a browser. We prefer to use these libraries, rather than applets, to write windowing programs that will not be run from a Web browser. In this book we show you how to do windowing interfaces as applets and as programs with no connection to a Web browser. In fact, the two approaches have a large overlap of both techniques and the Java libraries that they use. Once you know how to design and write either applets or applications, it is easy to learn to write the other of these two kinds of programs.

An applet always has a windowing interface. An application program may have a windowing interface or use simple console I/O. So as not to detract from the code being studied, most of our example programs, particularly early in the book, use simple console I/O (that is, simple text I/O).

A SAMPLE JAVA APPLICATION PROGRAM

Display 1.1 contains a simple Java program and the screen displays produced when it is run. A Java program is really a class definition (whatever that is) with a method named `main`. When the program is run, the method named `main` is invoked; that is, the action specified by `main` is carried out. The body of the method `main` is enclosed in braces, `{}`, so that when the program is run, the statements in the braces are executed. (If you are not even vaguely familiar with the words *class* and *method*, they will be explained. Read on.)

The following line says that this program is a class called `FirstProgram`:

```
public class FirstProgram
{
```

The next two lines, shown below, begin the definition of the `main` method:

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

The details of exactly what a Java class is and what words like `public`, `static`, `void`, and so forth mean will be explained in the next few chapters. Until then, you can think of these opening lines, repeated below, as being a rather wordy way of saying "Begin the program named `FirstProgram`."

```
public class FirstProgram
{
```

[1] An applet viewer does indeed use a browser to run an applet, but the look and feel is that of a stand-alone program with no interaction with a browser.

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

The next two lines, shown in what follows, are the first actions the program performs:

println

```
System.out.println("Hello reader.");
System.out.println("Welcome to Java.");
```

Each of these lines begins with `System.out.println`. Each one causes the quoted string given within the parentheses to be output to the screen. For example, consider

```
System.out.println("Hello reader.");
```

This causes the line

```
Hello reader.
```

to be written to the screen. The output produced by the next line that begins with `System.out.println` will go on the following line. Thus, these two lines cause the following output:

```
Hello reader.
Welcome to Java.
```

Display 1.1 A Sample Java Program

Name of class (program)

The `main` method

```
1  public class FirstProgram
2  {
3      public static void main(String[] args)
4      {
5          System.out.println("Hello reader.");
6          System.out.println("Welcome to Java.");

7          System.out.println("Let's demonstrate a simple calculation.");
8          int answer;
9          answer = 2 + 2;
10         System.out.println("2 plus 2 is " + answer);
11     }
12 }
```

SAMPLE DIALOGUE 1

```
Hello reader.
Welcome to Java.
Let's demonstrate a simple calculation.
2 plus 2 is 4
```

These lines that begin with `System.out.println` are a way of saying "output what is shown in parentheses," and the details of why the instruction is written this way need not concern us yet. However, we can tell you a little about what is going on here.

`System.out.println`

Java programs work by having things called *objects* perform actions. The actions performed by an object are called *methods*. `System.out` is an object used for sending output to the screen; `println` is the method (that is, the action) that this object performs. The action is to send what is in parentheses to the screen. When an object performs an action using a method, that is called **invoking** the method (or **calling** the method). In a Java program, you write such a method invocation by writing the object followed by a dot (period), followed by the method name, and some parentheses that may or may not have something inside them. The thing (or things) inside the parentheses is called an **argument(s)** and provides information needed by the method to carry out its action. In each of these two lines and the similar line that follows them, the method is `println`. The method `println` writes something to the screen, and the argument (a string in quotes) tells it what it should write.

invoking

dot

argument

Invoking a method is also sometimes called **sending a message** to the object. With this view a message is sent to the object (by invoking a method) and in response the object performs some action (namely the action taken by the method invoked). We seldom use the terminology *sending a message*, but it is standard terminology used by some programmers and authors.

sending a message

Variable declarations in Java are similar to what they are in other programming languages. The following line from Display 1.1 declares the variable `answer`:

```
int answer;
```

variable `int`

The type `int` is one of the Java types for integers (whole numbers). So, this line says that `answer` is a variable that can hold a single integer (whole number).

The following line is the only real computing done by this first program:

```
answer = 2 + 2;
```

equal sign

In Java, the equal sign is used as the **assignment operator**, which is an instruction to set the value of the variable on the left-hand side of the equal sign. In the preceding program line, the equal sign does not mean that `answer` *is equal to* `2 + 2`. Instead, the equal sign is an instruction to the computer to *make* `answer` *equal to* `2 + 2`.

assignment operator

The last program action is

```
System.out.println("2 plus 2 is " + answer);
```

This is an output statement of the same kind as we discussed earlier, but there is something new in it. Note that the string `"2 plus 2 is "` is followed by a plus sign and the variable `answer`. In this case the plus sign is an operator to concatenate (connect) two strings. However, the variable `answer` is not a string. If one of the two operands to `+` is a string, Java will convert the other operand, such as the value of `answer`, to a string. In this program, `answer` has the value `4`, so `answer` is converted to the string `"4"` and then

concatenated to the string `"2 plus 2 is "`, so the output statement under discussion is equivalent to

```
System.out.println("2 plus 2 is 4");
```

The remainder of this first program consists of two closing braces. The first closing brace ends the definition of the method `main`. The last closing brace ends the definition of the class named `FirstProgram`.

Self-Test Exercises

1. If the following statement were used in a Java program, it would cause something to be written to the screen. What would it cause to be written to the screen?

   ```
   System.out.println("Java is not a drink.");
   ```

2. Give a statement or statements that can be used in a Java program to write the following to the screen:

   ```
   I like Java.
   You like tea.
   ```

3. Write a complete Java program that uses `System.out.println` to output the following to the screen when run:

   ```
   Hello World!
   ```

   Note that you do not need to fully understand all the details of the program in order to write the program. You can simply follow the model of the program in Display 1.1.

BYTE-CODE AND THE JAVA VIRTUAL MACHINE

high-level, low-level, and machine languages

compiler

Most modern programming languages are designed to be (relatively) easy for people to write and to understand. These languages that were designed for people to read are called **high-level languages**. The language that the computer can directly understand is called **machine language**. Machine language or any language similar to machine language is called a **low-level language**. A program written in a high-level language, like Java, must be translated into a program in machine language before the program can be run. The program that does the translating (or at least most of the translating) is called a **compiler** and the translation process is called **compiling**.

One disadvantage of most programming languages is that the compiler translates the high-level-language program directly into the machine language for your computer. Since different computers have different machine languages, this means you need a different compiler for each type of computer. Java, however, uses a slightly different and much more versatile approach to compiling.

Compiler

A **compiler** is a program that translates a high-level-language program, such as a Java program, into an equivalent low-level language program.

The Java compiler does not translate your program into the machine language for your particular computer. Instead, it translates your Java program into a language called **byte-code**. Byte-code is not the machine language for any particular computer. Byte-code is the machine language for a fictitious computer called the **Java Virtual Machine**. The Java Virtual Machine is very similar to all typical computers. Thus, it is easy to translate a program written in byte-code into a program in the machine language for any particular computer. The program that does this translation is called an **interpreter**. An interpreter combines the translation of the byte-code and the execution of the corresponding machine-language instructions. The interpreter works by translating an instruction of byte-code into instructions expressed in your computer's machine language and then executing those instructions on your computer. It does this one byte-code instruction at a time. Thus, an interpreter translates and executes the instructions in the byte-code one after the other, rather than translating the entire byte-code program at once.[2]

byte-code

Java Virtual Machine

interpreter

To run a Java program, you proceed as follows. First, you use the compiler to translate the Java program into byte-code. Then, you use the byte-code interpreter for your computer to translate each byte-code instruction to machine language and to run the machine-language instructions.

It sounds as though Java byte-code just adds an extra step in the process. Why not write compilers that translate directly from Java to the machine language for your particular computer? That is what is done for most other programming languages. However, Java byte-code makes your Java program very portable. After you compile your Java program into byte-code, you can use that byte-code on any computer. When you run your program on another type of computer, you do not need to recompile it. This means that you can send your byte-code over the Internet to another computer and have it easily run on that computer. This is one of the reasons Java is good for Internet applications. Of course, every kind of computer must have its own byte-code interpreter, but these interpreters are simple programs when compared to a compiler.

When compiling and running a Java program, you are usually not even aware of the fact that your program is translated into byte-code and not directly translated into machine-language code. You normally give two commands, one to compile your program

---

[2] Sometimes people use the term *Java Virtual Machine* (JVM) to refer to the Java byte-code interpreter (as well as to refer to the underlying hypothetical machine that the interpreter is based on).

Byte-Code

The Java compiler translates your Java program into a language called **byte-code**, which is the machine language for a fictitious computer. It is easy to translate this byte-code into the machine language of any particular computer. Each type of computer will have its own interpreter that translates and executes byte-code instructions.

run command

(into byte-code) and one to run your program. The **run command** executes the Java byte-code interpreter on the byte-code.

When you use a compiler, the terminology can get a bit confusing, because both the input to the compiler program and the output from the compiler program are also programs. Everything in sight is some kind of program. To make sure it is clear which program we mean, we call the input program, which in our case will be a Java program, the

source code
object code
code

source program, or **source code**, and call the translated low-level-language program that the compiler produces the **object program**, or **object code**. The word **code** just means a program or a part of a program.

CLASS LOADER

A Java program is divided into smaller parts called *classes*, and normally each class definition is in a separate file and is compiled separately. In order to run your program, the byte-code for these various classes needs to be connected together. The connecting is done by a program known as the **class loader**. This connecting is typically done automatically, so you normally need not be concerned with it. (In other programming languages, the program corresponding to the Java class loader is called a *linker*.)

COMPILING A JAVA PROGRAM OR CLASS

As we noted in the previous subsection, a Java program is divided into classes. Before you can run a Java program, you must compile these classes.

Before you can compile a Java program, each class definition used in the program (and written by you, the programmer) should be in a separate file. Moreover, the name of the file should be the same as the name of the class, except that the file name has

.java files

`.java` added to the end. The program in Display 1.1 is a class called `FirstProgram` and so it should be in a file named `FirstProgram.java`. This program has only one class, but a more typical Java program would consist of several classes.

If you are using an IDE (Integrated Development Environment), there will be a simple command to compile your Java program from the editor. You will have to check your local documentation to see exactly what this command is, but it is bound to be very simple. (In the TextPad environment the command is Compile Java on the Tools menu.)

If you want or need to compile your Java program or class with a one-line command given to the operating system, then that is easy to do. We will describe the commands for the Java system distributed by Sun Microsystems (usually called "the SDK" or "the JDK" or "Java 2").

Suppose you want to compile a class named `FirstProgram`. It will be in a file named `FirstProgram.java`. To compile it, you simply give the following command:

```
javac FirstProgram.java
```

javac

You should be in the same directory (folder) as the file `FirstProgram.java` when you give this `javac` command. To compile any Java class, whether it is a full program or not, the command is `javac` followed by the name of the file containing the class.

When you compile a Java class, the resulting byte-code for that class is placed in a file of the same name, except that the ending is changed from `.java` to `.class`. So, when you compile a class named `FirstProgram` in the file `FirstProgram.java`, the resulting byte-code is stored in a file named `FirstProgram.class`.

.class files

RUNNING A JAVA PROGRAM

running a Java program

A Java program can consist of a number of different classes, each in a different file. When you run a Java application program, you only run the class that you think of as the program; that is, the class that contains a `main` method. Look for the following line, which starts the `main` method:

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

The critical words to look for are `public static void main`. The remaining portion of the line might be spelled slightly differently in some cases.

If you are using an IDE, you will have a menu command that can be used to run a Java program. You will have to check your local documentation to see exactly what this command is. (In the TextPad environment the command is Run Java Application on the Tools menu.)

If you want or need to run your Java program with a one-line command given to the operating system, then (in most cases) you can run a Java program by giving the command `java` followed by the name of the class containing the `main` method. For example, for the program in Display 1.1, you would give the following one-line command:

```
java FirstProgram
```

Note that when you run a program, you use the class name, such as `FirstProgram`, without any `.java` or `.class` ending.

When you run a Java program, you are actually running the Java byte-code interpreter on the compiled version of your program. When you run your program, the system will automatically load in any classes you need and run the byte-code interpreter on those classes as well.

Syntax and Semantics

The description of a programming language, or any other kind of language, can be thought of as having two parts, called the *syntax* and *semantics* of the language.

The **syntax** tells what arrangement of words and punctuations are legal in the language. The syntax of the language is often called the *grammar rules* of the language. For Java, the syntax describes what arrangements of words and punctuations are allowed in a class or program definition.

The **semantics** of a language describes the meaning of things written while following the syntax rules of the language. For a Java program, the syntax describes how you write a program and the semantics describes what happens when you run the program.

When writing a program in Java, you are always using both the syntax and the semantics of the Java language.

We have been assuming that the Java compiler and related software were already set up for you. We were also assuming that all the files were in one directory. (Directories are also called *folders*.) If you need to set up the Java compiler and system software, consult the manuals that came with the software. If you wish to spread your class definitions across multiple directories, that is not difficult, but we will not concern ourselves with that detail until later.

Tip

Error Messages

bug
debugging

A mistake in a program is called a **bug**. For this reason, the process of eliminating mistakes in your program is called **debugging**. There are three commonly recognized types of bugs or errors, and they are known as *syntax errors*, *run-time errors*, and *logic errors*. Let's consider them in order.

syntax error

A **syntax error** is a grammatical mistake in your program; that is, a mistake in the allowed arrangement of words and punctuations. If you violate one of these rules, for example, by omitting a required punctuation, that is a syntax error. The compiler will catch syntax errors and output an error message telling you that it has found the error, where it thinks the error is, and what it thinks the error is. If the compiler says you have a syntax error, you undoubtedly do have an error. However, the compiler could be incorrect about where and what the error is.

run-time error

An error that is not detected until your program is run is called a **run-time error**. If the computer detects a run-time error when your program is run, then it will output an error message. The error message may not be easy to understand, but at least it lets you know that something is wrong.

A mistake in the underlying algorithm for your program is called a **logic error**. If your program has only logic errors, it will compile and run without any error message. You have written a valid Java program, but you have not written a program that does what you want. The program runs and gives output, but gives incorrect output. For example, if you were to mistakenly use the multiplication sign in place of the addition sign, that would be a logic error. Logic errors are the hardest kind of error to locate, because the computer does not give you any error messages.

logic error

Self-Test Exercises

4. What is a compiler?
5. What is a source program?
6. What is an object program?
7. What do you call a program that runs Java byte-code instructions?
8. Suppose you define a class named `NiceClass` in a file. What name should the file have?
9. Suppose you compile the class `NiceClass`. What will be the name of the file with the resulting byte-code?

1.2 Expressions and Assignment Statements

Once a person has understood the way variables are used in programming, he has understood the quintessence of programming.

E. W. Dijkstra, Notes on Structured Programming

Variables, expressions, and assignments in Java are similar to their counterparts in most other general-purpose languages. In this section we describe the details.

IDENTIFIERS

The name of a variable (or other item you might define in a program) is called an **identifier.** A Java identifier must not start with a digit and all the characters must be letters, digits, or the underscore symbol. (The symbol $ is also allowed, but it is reserved for special purposes, and so you should not use $ in your Java identifiers.) For example, the following are all valid identifiers:

identifier

```
x x1 x_1 _abc ABC123z7 sum RATE count data2 bigBonus
```

All of the preceding names are legal and would be accepted by the compiler, but the first five are poor choices for identifiers, since they are not descriptive of the identifier's use. None of the following are legal identifiers and all would be rejected by the compiler:

```
12  3X  %change  data-1  myfirst.java  PROG.CLASS
```

The first two are not allowed because they start with a digit. The remaining four are not identifiers because they contain symbols other than letters, digits, and the underscore symbol.

case-sensitive

Java is a case-sensitive language; that is, it distinguishes between uppercase and lowercase letters in the spelling of identifiers. Hence, the following are three distinct identifiers and could be used to name three distinct variables:

```
rate  RATE  Rate
```

However, it is usually not a good idea to use two such variants in the same program, since that might be confusing. Although it is not required by Java, variables are usually spelled with their first letter in lowercase. The convention that has now become universal in Java programming is to spell variable names with a mix of upper- and lowercase letters (and digits), to always start a variable name with a lowercase letter, and to indicate "word" boundaries with an uppercase letter, as illustrated by the following variable names:

```
topSpeed bankRate1 bankRate2 timeOfArrival
```

A Java identifier can theoretically be of any length, and the compiler will accept even unreasonably long identifiers.

Names (Identifiers)

The name of something in a Java program, such as a variable, class, method, or object name, must not start with a digit and may only contain letters, digits (0 through 9), and the underscore character (_). Uppercase and lowercase letters are considered to be different characters. (The symbol $ is also allowed, but it is reserved for special purposes, and so you should not use $ in a Java name.)

Names in a program are called **identifiers**.

Although it is not required by the Java language, the common practice, and the one followed in this book, is to start the names of classes with uppercase letters and to start the names of variables, objects, and methods with lowercase letters. These names are usually spelled using only letters and digits.

keyword

There is a special class of identifiers, called **keywords** or **reserved words**, that have a predefined meaning in Java and that you cannot use as names for variables or anything

else. In the code displays of this book, keywords are shown in a different color, as illustrated by the keyword `public`. A complete list of keywords is given in Appendix 1.

Some predefined words, such as `System` and `println`, are not keywords. These predefined words are not part of the core Java language and you are allowed to redefine them. These predefined words are not keywords. However, they are defined in libraries required by the Java language standard. Needless to say, using a predefined identifier for anything other than its standard meaning can be confusing and dangerous, and thus should be avoided. The safest and easiest practice is to treat all predefined identifiers as if they were keywords.

VARIABLES

Every variable in a Java program must be *declared* before it is used. When you **declare** a variable, you are telling the compiler—and, ultimately, the computer—what kind of data you will be storing in the variable. For example, the following are two declarations that might occur in a Java program:

declare

```
int numberOfBeans;
double oneWeight, totalWeight;
```

The first declares the variable `numberOfBeans` so that it can hold a value of type `int`, that is, a whole number. The name `int` is an abbreviation for "integer." The type `int` is the default type for whole numbers. The second definition declares `oneWeight` and `totalWeight` to be variables of type `double`, which is the default type for numbers with a decimal point (known as **floating-point numbers**). As illustrated here, when there is more than one variable in a declaration, the variables are separated by commas. Also, note that each declaration ends with a semicolon.

floating-point number

Every variable must be declared before it is used. A variable may be declared any place, so long as it is declared before it is used. Of course, variables should always be declared in a location that makes the program easier to read. Typically variables are declared either just before they are used or at the start of a block (indicted by an opening brace `{`). Any legal identifier, other than a keyword, may be used for a variable name.

Java has basic types for characters, different kinds of integers, and different kinds of floating-point numbers (numbers with a decimal point), as well as a type for the values `true` and `false`. These basic types are known as **primitive types**. Display 1.2 shows all of Java's primitive types. The preferred type for integers is `int`. The type `char` is the type for single characters. The preferred type for floating-point numbers is `double`. The type `boolean` has the two values `true` and `false`. (Unlike some other programming languages, the Java values `true` and `false` are not integers and will not be automatically converted to integers.) Objects of the predefined class `String` represent strings of characters. `String` is not a primitive type, but is often considered a basic type along with the primitive types. The class `String` is discussed later in this chapter.

primitive types

Variable Declarations

In Java, a variable must be declared before it is used. Variables are declared as follows:

Syntax:

Type Variable_1, Variable_2, . . .;

Examples

```
int count, numberOfDragons, numberOfTrolls;
char answer;
double speed, distance;
```

Syntactic Variables

Remember that when you see something such as *Type*, *Variable_1*, or *Variable_2*, these words do not literally appear in your Java code. They are **syntactic variables**, which means they are replaced by something of the category that they describe. For example, *Type* can be replaced by `int`, `double`, `char`, or any other type name. *Variable_1* and *Variable_2* can each be replaced by any variable name.

Display 1.2 Primitive Types

TYPE NAME	KIND OF VALUE	MEMORY USED	SIZE RANGE
`boolean`	`true` or `false`	1 byte	not applicable
`char`	single character (Unicode)	2 bytes	all Unicode characters
`byte`	integer	1 byte	−128 to 127
`short`	integer	2 bytes	−32768 to 32767
`int`	integer	4 bytes	−2147483648 to 2147483647
`long`	integer	8 bytes	−92233720368547758O8 to 9223372036854775807
`float`	floating-point number	4 bytes	$-3.40282347 \times 10^{+38}$ to $-1.40239846 \times 10^{-45}$
`double`	floating-point number	8 bytes	$\pm 1.76769313486231570 \times 10^{+308}$ to $\pm 4.94065645841246544 \times 10^{-324}$

ASSIGNMENT STATEMENTS

assignment statement

assignment operator

The most direct way to change the value of a variable is to use an **assignment statement**. In Java the equal sign is used as the **assignment operator**. An assignment statement always consists of a variable on the left-hand side of the assignment operator (the equal sign) and an expression on the right-hand side. An assignment statement ends with a semicolon. The expression on the right-hand side of the equal sign may be a variable, a number, or a more complicated expression made up of variables, numbers, operators, and method invocations. An assignment statement instructs the computer to evaluate (that is, to compute the value of) the expression on the right-hand side of the equal sign and to set the value of the variable on the left-hand side equal to the value of that expression. The following are examples of Java assignment statements:

```
totalWeight = oneWeight * numberOfBeans;
temperature = 98.6;
count = count + 2;
```

The first assignment statement sets the value of `totalWeight` equal to the number in the variable `oneWeight` multiplied by the number in `numberOfBeans`. (Multiplication is expressed using the asterisk `*` in Java.) The second assignment statement sets the value of `temperature` to `98.6`. The third assignment statement increases the value of the variable `count` by `2`.

Note that a variable may occur on both sides of the assignment operator (both sides of the equal sign). The assigned statement

```
count = count + 2;
```

sets the new value of `count` equal to the old value of `count` plus `2`.

The assignment operator when used with variables of a class type requires a bit more explanation, which we will give in Chapter 4.

Assignment Statements with Primitive Types

An assignment statement with a variable of a primitive type on the left-hand side of the equal sign causes the following action: First, the expression on the right-hand side of the equal sign is evaluated, and then the variable on the left-hand side of the equal sign is set equal to this value.

Syntax:

Variable = *Expression*;

Example:

```
distance = rate * time;
count = count + 2;
```

An assigned statement may be used as an expression that evaluates to a value. When used this way, the variable on the left-hand side of the equal sign is changed as we have described, and the new value of the variable is also the value of the assignment expression. For example,

```
number = 3;
```

both changes the value of `number` to `3` and evaluates to the value `3`. This allows you to chain assignment statements. The following changes the values of both the variables, `number1` and `number2`, to `3`:

```
number2 = (number1 = 3);
```

The assignment operator automatically is executed right-to-left if there are no parentheses, so this is normally written in the following equivalent way:

```
number2 = number1 = 3;
```

Tip

Initialize Variables

uninitialized variable

A variable that has been declared but that has not yet been given a value by some means, such as an assignment statement, is said to be **uninitialized**. In some cases an uninitialized variable may be given some default value, but this is not true in all cases. Moreover, it makes your program clearer to explicitly give the variable a value, even if you are simply reassigning it the default value. (The exact details on default values have been known to change and should not be counted on.)[3]

One easy way to ensure that you do not have an uninitialized variable is to initialize it within the declaration. Simply combine the declaration and an assignment statement, as in the following examples:

```
int count = 0;
double speed = 65.5;
char grade = 'A';
int initialCount = 50, finalCount;
```

Note that you can initialize some variables and not initialize other variables in a declaration.

Sometimes the compiler may say that you have failed to initialize a variable. In most cases, you will indeed have failed to initialize the variable. Occasionally, the compiler is mistaken in giving this advice. However, the compiler will not compile your program until you convince it that the variable in question is initialized. To make the compiler happy, initialize the variable when it is declared, even if the variable will be given a different value before the variable is used for anything. In such cases, you cannot argue with the compiler.

---

[3] The official rules are that the variables we are now using, which we will later call *local variables*, are not automatically initialized. Later in this book we will introduce variables called *static variables* and *instance variables* that are automatically initialized. However, we urge you to never rely on automatic initialization.

Initializing a Variable in a Declaration

You can combine the declaration of a variable with an assignment statement that gives the variable a value.

Syntax:

Type Variable_1 = Expression_1, Variable_2 = Expression_2, ...;

Some of the variables may have no equal sign and no expression, as in the first example.

Example:

```
int numberReceived = 0, lastNumber, numberOfStations = 5;
double speed = 98.9, distance = speed * 10;
char initial = 'J';
```

MORE ASSIGNMENT STATEMENTS

There is a shorthand notation that combines the assignment operator (=) and an arithmetic operator so that a given variable can have its value changed by adding, subtracting, multiplying, or dividing by a specified value. The general form is

Variable Op = Expression

which is equivalent to

Variable = Variable Op (Expression)

The *Expression* can be another variable, a constant, or a more complicated arithmetic expression. The *Op* can be any of +, –, *, /, %, as well as some operators we have not yet discussed. The operator % has not yet been discussed but is explained later in this chapter. (A full list of values for *Op* can be seen at the bottom of the precedence table in Appendix 2.) Below are examples:

EXAMPLE:	EQUIVALENT TO:
`count += 2;`	`count = count + 2;`
`total -= discount;`	`total = total - discount;`
`bonus *= 2;`	`bonus = bonus * 2;`
`time /= rushFactor;`	`time = time / rushFactor;`
`change %= 100;`	`change = change % 100;`
`amount *= count1 + count2;`	`amount = amount * (count1 + count2);`

Self-Test Exercises

10. Which of the following may be used as variable names in Java?

```
rate1, 1stPlayer, myprogram.java, long,
TimeLimit, numberOfWindows
```

11. Can a Java program have two different variables named `number` and `Number`?

12. Give the declaration for two variables called `feet` and `inches`. Both variables are of type `int` and both are to be initialized to zero in the declaration.

13. Give the declaration for two variables called `count` and `distance`. `count` is of type `int` and is initialized to zero. `distance` is of type `double` and is initialized to `1.5`.

14. Write a Java assignment statement that will set the value of the variable `distance` to the value of the variable `time` multiplied by `80`. All variables are of type `int`.

15. Write a Java assignment statement that will set the value of the variable `interest` to the value of the variable `balance` multiplied by the value of the variable `rate`. The variables are of type `double`.

16. What is the output produced by the following lines of program code?

```
char a, b;
a = 'b';
System.out.println(a);
b = 'c';
System.out.println(b);
a = b;
System.out.println(a);
```

ASSIGNMENT COMPATIBILITY

As a general rule, you cannot store a value of one type in a variable of another type. For example, the compilers will object to the following:

```
int intVariable;
intVariable = 2.99;
```

The problem is a type mismatch. The constant `2.99` is of type `double` and the variable `intVariable` is of type `int`.

assigning `int` values to `double` variables

There are some special cases where it is permitted to assign a value of one type to a variable of another type. It is acceptable to assign a value of an integer type, such as `int`, to a variable of a floating-point type, such as the type `double`. For example, the following is both legal and acceptable style:

```
double doubleVariable;
doubleVariable = 2;
```

The preceding will set the value of the variable named `doubleVariable` equal to `2.0`.

Similarly, assignments of integer type variables to floating-point type variables are also allowed. For example, the following is allowed:

```
int intVariable;
intVariable = 42;
double doubleVariable;
doubleVariable = intVariable;
```

More generally, you can assign a value of any type in the following list to a variable of any type that appears further down in the list:

```
byte -> short -> int -> long -> float -> double
```

For example, you can assign a value of type `int` to a variable of type `long`, `float`, or `double` (or of course to a variable of type `int`), but you cannot assign a value of type `int` to a variable of type `byte` or `short`. Note that this is not an arbitrary ordering of the types. As you move down the list from left to right, the range of allowed values for the types becomes larger.

You can assign a value of type `char` to a variable of type `int` or to any of the numeric types that follow `int` in our list of types (but not to those that precede `int`). However, in most cases it is not wise to assign a character to an `int` variable, because the result could be confusing.[4]

If you want to assign a value of type `double` to a variable of type `int`, then you must change the type of the value by using a *type cast*, as explained in the subsection later in this chapter that is entitled "Type Casting."

integers and booleans

In many languages other than Java, you can assign integers to variables of type `boolean` and assign `boolean` values to integer variables. You cannot do that in Java. In Java, the `boolean` values `true` and `false` are not integers nor will they be automatically converted to integers. (In fact, it is not even legal to do an explicit type cast from the type `boolean` to the type `int` or vice versa. Explicit type casts are discussed later in this chapter.)

---

[4] Readers who have used certain other languages, such as C or C++, may be surprised to learn that you cannot assign a value of type `char` to a variable of type `byte`. This is because Java uses the Unicode character set rather than the ASCII character set, and so Java reserves two bytes of memory for each value of type `char`, but naturally only reserves one byte of memory for values of type `byte`. This is one of the few cases where you might notice that Java uses the Unicode character set. Indeed, if you convert from an `int` to a `char` or vice versa, you can expect to get the usual correspondence of ASCII numbers and characters. It is also true that you cannot assign a value of type `char` to a variable of type `short`, even though they both use two bytes of memory.

Assignment Compatibilities

You can assign a value of any type on the following list to a variable of any type that appears further down on the list:

```
byte -> short -> int -> long -> float -> double
```

In particular, note that you can assign a value of any integer type to a variable of any floating-point type. You can also assign a value of type `char` to a variable of type `int` or of any type that follows `int` in the above list.

CONSTANTS

literals
constants

Constants or **literals** are names for one specific value. For example, `2` and `3.1459` are two constants. We prefer the name *constants* because it contrasts nicely with the word *variables*. Constants do not change value; variables can change their values. Integer constants are written in the way you are used to writing numbers. Constants of type `int` (or any other integer type) must not contain a decimal point. Constants of floating-point types (`float` and `double`) may be written in either of two forms. The simple form for floating-point constants is like the everyday way of writing decimal fractions. When written in this form, a floating-point constant must contain a decimal point. No number constant (neither integer nor floating point) in Java may contain a comma.

e notation

A more complicated notation for floating-point constants, such as constants of type `double`, is called **scientific notation** or **floating-point notation** and is particularly handy for writing very large numbers and very small fractions. For instance,

3.67×10^5, which is the same as `367000.0`,

is best expressed in Java by the constant `3.67e5`. The number

5.89×10^{-4}, which is the same as `0.000589`,

is best expressed in Java by the constant `5.89e–4`. The `e` stands for *exponent* and means "multiply by 10 to the power that follows." The `e` may be either uppercase or lowercase.

Think of the number after the `e` as telling you the direction and number of digits to move the decimal point. For example, to change `3.49e4` to a numeral without an `e`, you move the decimal point `4` places to the right to obtain `34900.0`, which is another way of writing the same number. If the number after the `e` is negative, you move the decimal point the indicated number of spaces to the left, inserting extra zeros if need be. So, `3.49e–2` is the same as `0.0349`.

The number before the `e` may contain a decimal point, although that is not required. However, the exponent after the `e` definitely must *not* contain a decimal point.

What Is Doubled?

How did the floating-point type `double` get its name? Is there another type for floating-point numbers called "single" that is half as big? Something like that is true. There is a type that uses half as much storage, namely the type `float`. Many programming languages traditionally used two types for floating-point numbers. One type used less storage and was very imprecise (that is, it did not allow very many significant digits). The second type used *double* the amount of storage and so could be much more precise; it also allowed numbers that were larger (although programmers tend to care more about precision than about size). The kind of numbers that used twice as much storage were called *double precision* numbers; those that used less storage were called *single precision*. Following this tradition, the type that (more or less) corresponds to this double precision type in Java was named `double` in Java. The type that corresponds to single precision in Java was called `float`.

(Actually, the type name `double` was inherited from C++, but this explanation applies to why the type was named `double` in C++, and so ultimately it is the explanation of why the type is called `double` in Java.)

Constants of type `char` are expressed by placing the character in single quotes, as illustrated in what follows:

```
char symbol = 'Z';
```

Note that the left and right single quote symbol are the same symbol.

Constants for strings of characters are given in double quotes, as illustrated by the following line taken from Display 1.1:

```
System.out.println("Welcome to Java.");
```

quotes

Be sure to notice that string constants are placed inside of double quotes, while constants of type `char` are placed inside of single quotes. The two kinds of quotes mean different things. In particular, `'A'` and `"A"` mean different things. `'A'` is a value of type `char` and can be stored in a variable of type `char`. `"A"` is a string of characters. The fact that the string happens to contain only one character does *not* make the string `"A"` a value of type `char`. Also notice that, for both strings and characters, the left and right quotes are the same. We will have more to say about strings later in this chapter.

The type `boolean` has two constants, `true` and `false`. These two constants may be assigned to a variable of type `boolean` or used anyplace else an expression of type `boolean` is allowed. They must be spelled with all lowercase letters.

ARITHMETIC OPERATORS AND EXPRESSIONS

As in most other languages, Java allows you to form expressions using variables, constants, and the arithmetic operators: + (addition), – (subtraction), * (multiplication),

/ (division), % (modulo, remainder). These expressions can be used anyplace it is legal to use a value of the type produced by the expression.

mixing types

All of the arithmetic operators can be used with numbers of type int, numbers of type double, and even with one number of each type. However, the type of the value produced and the exact value of the result depends on the types of the numbers being combined. If both operands (that is, both numbers) are of type int, then the result of combining them with an arithmetic operator is of type int. If one, or both, of the operands is of type double, then the result is of type double. For example, if the variables baseAmount and increase are both of type int, then the number produced by the following expression is of type int:

```
baseAmount + increase
```

However, if one, or both, of the two variables is of type double, then the result is of type double. This is also true if you replace the operator + with any of the operators –, *, /, or %.

More generally, you can combine any of the arithmetic types in expressions. If all the types are integer types, the result will be the integer type. If at least one of the subexpressions is of a floating-point type, the result will be a floating-point type.

Knowing whether the value produced is of an integer type or a floating-point type is typically all that you need to know. However, if you need to know the exact type of the value produced by an arithmetic expression, it can be determined as follows: The type of the value produced is one of the types used in the expression. Of all the types used in the expression, it is, with rare exceptions, the last type (reading left to right) on the following list:

```
byte -> short -> int -> long -> float -> double
```

Here are the rare exceptions: Of all the types used in the expression, if the last type (reading left to right) is byte or short, then the type of the value produced is int. In other words, an expression never evaluates to either of the types byte or short. These exceptions have to do with an implementation detail that need not concern us, especially since we almost never use the types byte and short in this book.

Note that this sequence of types is the same sequence of types we saw when discussing assignment compatibility. As you go from left to right, the types increase in the range of values they allow.[5]

[5] Although we discourage the practice, you can use values and variables of type char in arithmetic expressions using operators such as +. If you do so, the char values and variables will contribute to the expression as if they were of type int.

PARENTHESES AND PRECEDENCE RULES

If you want to specify exactly what subexpressions are combined with each operator, you can fully parenthesize an expression. For example:

```
((base + (rate * hours))/(2 + rate))
```

If you omit some parentheses in an arithmetic expression, Java will, in effect, put in parentheses for you. When adding parentheses, Java follows rules called **precedence rules** that determine how the operators, such as + and *, are enclosed in parentheses. These precedence rules are similar to rules used in algebra. For example,

```
base + rate * hours
```

is evaluated by Java as if it were parenthesized as follows:

```
base + (rate * hours)
```

So, the multiplication will be done before the addition.

Except in some standard cases, such as a string of additions or a simple multiplication embedded inside an addition, it is usually best to include the parentheses, even if the intended groupings are the ones dictated by the precedence rules. The parentheses make the expression easier to read and less prone to programmer error.

A partial list of precedence rules is given in Display 1.3. A complete set of Java precedence rules is given in Appendix 2. Operators that are listed higher on the list are said to have **higher precedence**. When the computer is deciding which of two adjacent operations to group with parentheses, it groups the operation of higher precedence and its apparent arguments before the operation of lower precedence. Some operators have equal precedence, in which case the order of operations is determined by **associativity rules**. A brief summary of associativity rules is that binary operators of equal precedence are grouped in left-to-right order.[6] Unary operators of equal precedence are grouped in right-to-left order. So, for example,

```
base + rate + hours
```

is interpreted by Java to be the same as

```
(base + rate) + hours
```

And, for example,

```
+-+rate
```

is interpreted by Java to be the same as

```
+(-(+rate))
```

---

[6] There is one exception to this rule. A string of assignment operators, like `n1 = n2 = n3;`, is performed right to left, as we noted earlier in this chapter.

Display 1.3 Precedence Rules

Highest Precedence

First: the unary operators: +, –, ++, ––, and !

Second: the binary arithmetic operators: *, /, and %

Third: the binary arithmetic operators: + and –

Lowest Precedence

For now you can think of the explicit parentheses put in by the programmer and the implicit parentheses determined by precedence and associativity rules as determining the order in which operations are performed. For example, in

```
base + (rate * hours)
```

the multiplication is performed first and the addition is performed second.

The actual situation is a bit more complicated than what we have described for evaluating expressions, but we will not encounter any of these complications in this chapter. A complete discussion of evaluating expressions using precedence and associativity rules will be given in Chapter 3.

Pitfall

Round-Off Errors in Floating-Point Numbers

For all practical purposes, floating-point numbers are only approximate quantities. For example, in formal mathematics the floating-point number `1.0/3.0` is equal to

```
0.3333333...
```

where the three dots indicate that the 3s go on forever. The computer stores numbers in a format somewhat like this decimal representation, but it has room for only a limited number of digits. If it can store only 10 digits after the decimal, then `1.0/3.0` is stored as

```
0.3333333333
```

with only 10 threes. Thus, `1.0/3.0` is stored as a number that is slightly smaller than one-third. In other words, the value stored as `1.0/3.0` is only approximately equal to one-third.

In reality, the computer stores numbers in binary notation, rather than in base 10 notation, but the principles are the same and the consequences are the same. Some floating-point numbers lose accuracy when they are stored in the computer.

Floating-point numbers (like numbers of type `double`) and integers (like numbers of type `int`) are stored differently. Floating-point numbers are, in effect, stored as approximate quantities. Integers are stored as exact quantities. This difference sometimes can be subtle. For example, the numbers 42 and `42.0` are different in Java. The whole number 42 is of type `int` and is an exact quantity. The number `42.0` is of type `double` because it contains a fractional part (even though the fraction is `0`), and so `42.0` is stored with only limited accuracy.

As a result of this limited accuracy, arithmetic done on floating-point numbers only gives approximate results. Moreover, one can easily get results on floating-point numbers that are very far from the true result you would obtain if the numbers could have unlimited accuracy (unlimited number of digits after the decimal point). For example, if a banking program used numbers of type `double` to represent amounts of money and did not do sophisticated manipulations to preserve accuracy, it would quickly bring the bank to ruin since the computed amounts of money would frequently be very incorrect. Dealing with these inaccuracies in floating-point numbers is part of the field of Numerical Analysis, a topic we will not discuss in this book. But, there is an easy way to obtain accuracy when dealing with amounts of money: use integers instead of floating-point numbers (perhaps one integer for the dollar amount and another integer for the cents amount).

INTEGER AND FLOATING-POINT DIVISION

When used with one or both operands of type `double`, the division operator, `/`, behaves as you might expect. However, when used with two operands of type `int`, the division operator yields the integer part resulting from division. In other words, integer division discards the part after the decimal point. So, `10/3` is `3` (not `3.3333...`), `5/2` is `2` (not `2.5`), and `11/3` is `3` (not `3.6666...`). Notice that the number *is not rounded*; the part after the decimal point is discarded no matter how large it is.

integer division

The operator `%` can be used with operands of type `int` to recover the information lost when you use `/` to do division with numbers of type `int`. When used with values of type `int`, the two operators `/` and `%` yield the two numbers produced when you perform the long division algorithm you learned in grade school. For example, `14` divided by `3` is `4` with a remainder of `2`. The `/` operation yields the number of times one number "goes into" another (often called the *quotient*). The `%` operation gives the remainder. For example, the statements

the % operator

```
System.out.println("14 divided by 3 is " + (14/3));
System.out.println("with a remainder of " + (14%3));
```

yield the following output:

```
14 divided by 3 is 4
with a remainder of 2
```

The % operator can be used to count by 2's, 3's, or any other number. For example, if you want to do something to every other integer, you need to know if the integer is even or odd. Then, you can do it to every even integer (or alternatively every odd integer). An integer n is even if n%2 is equal to 0 and the integer is odd if n%2 is equal to 1. Similarly, to do something to every third integer, your program might step through all integers n but only do the action when n%3 is equal to 0.

Although the % operator is primarily used with integers, it can also be used with two floating-point numbers, such as two values of type double. However, we will not discuss nor use % with floating-point numbers.

Pitfall

Division with Whole Numbers

When you use the division operator / on two integers, the result is an integer. This can be a problem if you expect a fraction. Moreover, the problem can easily go unnoticed, resulting in a program that looks fine but is producing incorrect output without your even being aware of the problem. For example, suppose you are a landscape architect who charges $5,000 per mile to landscape a highway, and suppose you know the length in feet of the highway you are working on. The price you charge can easily be calculated by the following Java statement:

```
totalPrice = 5000 * (feet/5280.0);
```

This works because there are 5,280 feet in a mile. If the stretch of highway you are landscaping is 15,000 feet long, this formula will tell you that the total price is

```
5000 * (15000/5280.0)
```

Your Java program obtains the final value as follows: 15000/5280.0 is computed as 2.84. Then the program multiplies 5000 by 2.84 to produce the value 14200.00. With the aid of your Java program, you know that you should charge $14,200 for the project.

Now suppose the variable feet is of type int, and you forget to put in the decimal point and the zero, so that the assignment statement in your program reads:

```
totalPrice = 5000 * (feet/5280);
```

It still looks fine, but will cause serious problems. If you use this second form of the assignment statement, you are dividing two values of type int, so the result of the division feet/5280 is 15000/5280, which is the int value 2 (instead of the value 2.84, which you think you are getting). So the value assigned to totalPrice is 5000*2, or 10000.00. If you forget the decimal point, you will charge $10,000. However, as we have already seen, the correct value is $14,200. A missing decimal point has cost you $4,200. Note that this will be true whether the type of totalPrice is int or double; the damage is done before the value is assigned to totalPrice.

Self-Test Exercises

17. Convert each of the following mathematical formulas to a Java expression:

    $3x$ $\quad 3x + y$ $\quad \frac{x + y}{7}$ $\quad \frac{3x + y}{z + 2}$

18. What is the output of the following program lines?

```
double number = (1/3) * 3;
System.out.println("(1/3) * 3 is equal to " + number);
```

19. What is the output produced by the following lines of program code?

```
int quotient, remainder;
quotient = 7/3;
remainder = 7%3;
System.out.println("quotient = " + quotient);
System.out.println("remainder = " + remainder);
```

20. What is the output produced by the following code?

```
int result = 11;
result /= 2;
System.out.println("result is " + result);
```

21. Given the following fragment that purports to convert from degrees Celsius to degrees Fahrenheit, answer the following questions:

```
double celsius = 20;
double fahrenheit;
fahrenheit = (9/5) * celsius + 32.0;
```

    a. What value is assigned to `fahrenheit`?

    b. Explain what is actually happening, and what the programmer likely wanted.

    c. Rewrite the code as the programmer intended.

TYPE CASTING

A type cast takes a value of one type and produces a value of another type that is Java's best guess of an equivalent value. We will motivate type casts with a simple division example.

Consider the expression `9/2`. In Java this expression evaluates to `4`, because when both operands are of an integer type, Java performs integer division. In some situations, you might want the answer to be the `double` value `4.5`. You can get a result of `4.5` by using the "equivalent" floating-point value `2.0` in place of the integer value `2`, as in the expression `9/2.0`, which evaluates to `4.5`. But, what if the `9` and the `2` are the values of

variables of type `int` named `n` and `m`. Then, `n/m` yields `4`. If you want floating-point division in this case, you must do a type cast from `int` to `double` (or another floating-point type), such as in the following:

```
double ans = n/(double)m;
```

The expression

```
(double)m
```

is a type cast. The expression takes an `int` (in this example the value of `m`) and evaluates to an "equivalent" value of type `double`. So, if the value of `m` is `2`, the expression `(double)m` evaluates to the `double` value `2.0`.

Note that `(double)m` does not change the value of the variable `m`. If `m` has the value `2` before this expression is evaluated, then `m` still has the value `2` after the expression is evaluated.

You may use other type names in place of `double` to obtain a type cast to another type. We said this produces an "equivalent" value of the target type. The word "equivalent" is in quotes because there is no clear notion of equivalent that applies between any two types. In the case of a type cast from an integer type to a floating-point type, the effect is to add a decimal point and a zero. A type cast in the other direction, from a floating-point type to an integer type, simply deletes the decimal point and all digits after the decimal point. Note that when type casting from a floating-point type to an integer type, the number is truncated not rounded: `(int)2.9` is `2`; it is not `3`.

As we noted earlier, you can always assign a value of an integer type to a variable of a floating-point type, as in

```
double d = 5;
```

In such cases Java performs an automatic type cast, converting the `5` to `5.0` and placing `5.0` in the variable `d`. You cannot store the `5` as the value of `d` without a type cast, but sometimes Java does the type cast for you. Such an automatic type cast is sometimes called a **type coercion**.

type coercion

By contrast, you cannot place a `double` value in an `int` variable without an explicit type cast. The following is illegal:

```
int i = 5.5; //Illegal
```

Instead, you must add an explicit type cast, like so:

```
int i = (int)5.5;
```

INCREMENT AND DECREMENT OPERATORS

The **increment operator** ++ adds one to the value of a variable. The **decrement operator** –– subtracts one from the value of a variable. They are usually used with variables of

type `int`, but they can be used with any numeric type. If `n` is a variable of a numeric type, then `n++` increases the value of `n` by one and `n--` decreases the value of `n` by one. So, `n++` and `n--` (when followed by a semicolon) are executable statements. For example, the statements

```
int n = 1, m = 7;
n++;
System.out.println("The value of n is changed to " + n);
m--;
System.out.println("The value of m is changed to " + m);
```

yield the following output:

```
The value of n is changed to 2
The value of m is changed to 6
```

An expression like `n++` also evaluates to a number as well as changing the value of the variable `n`, so `n++` can be used in an arithmetic expression such as

```
2*(n++)
```

The expression `n++` changes the value of `n` by adding one to it, but it evaluates to the value `n` had *before* it was increased. For example, consider the following code:

```
int n = 2;
int valueProduced = 2*(n++);
System.out.println(valueProduced);
System.out.println(n);
```

This code produces the following output:

```
4
3
```

Notice the expression `2*(n++)`. When Java evaluates this expression, it uses the value that `number` has *before* it is incremented, not the value that it has after it is incremented. Thus, the value produced by the expression `n++` is `2`, even though the increment operator changes the value of `n` to `3`. This may seem strange, but sometimes it is just what you want. And, as you are about to see, if you want an expression that behaves differently, you can have it.

The expression `++n` also increments the value of the variable `n` by one, but it evaluates to the value `n` has after it is increased. For example, consider the following code:

```
int n = 2;
int valueProduced = 2*(++n);
System.out.println(valueProduced);
System.out.println(n);
```

This code is the same as the previous piece of code except that the ++ is before the variable, so this code will produce the following output:

```
6
3
```

v++ versus ++v

Notice that the two increment operators n++ and ++n have the exact same effect on a variable n: they both increase the value of n by one. But the two expressions evaluate to different values. Remember, if the ++ is *before* the variable, then the incrementing is done *before* the value is returned; if the ++ is *after* the variable, then the incrementing is done *after* the value is returned.

decrement operator

Everything we said about the increment operator applies to the decrement operator as well, except that the value of the variable is decreased by one rather than increased by one. For example, consider the following code:

```
int n = 8;
int valueProduced = n--;
System.out.println(valueProduced);
System.out.println(n);
```

This produces the output:

```
8
7
```

On the other hand, the code

```
int n = 8;
int valueProduced = --n;
System.out.println(valueProduced);
System.out.println(n);
```

produces the output:

```
7
7
```

Both n-- and --n change the value of n by subtracting one, but they evaluate to different values. n-- evaluates to the value n had before it was decremented; on the other hand, --n evaluates to the value n has after it is decremented.

You cannot apply the increment and decrement operators to anything other than a single variable. Expressions such as (x + y)++, --(x + y), 5++, and so forth are all illegal in Java.

The use of the increment and decrement operators can be confusing when used inside of more complicated expressions, and so, we prefer to not use increment or decrement operators inside of expressions, but to only use them as simple statements, such as:

```
n++;
```

Self-Test Exercises

22. What is the output produced by the following lines of program code?

```
int n = (int)3.9;
System.out.println("n == " + n);
```

23. What is the output produced by the following lines of program code?

```
int n = 3;
n++;
System.out.println("n == " + n);
n--;
System.out.println("n == " + n);
```

1.3 The Class String

Words, words, mere words, no matter from the heart.

William Shakespeare, *Troilus and Cressida*

There is no primitive type for strings in Java. However, there is a class called `String` that can be used to store and process strings of characters. This section introduces the class `String`. String

STRING CONSTANTS AND VARIABLES

You have already seen constants of type `String`. The quoted string

```
"Hello reader."
```

which appears in the following statement from Display 1.1, is a string constant:

```
System.out.println("Hello reader.");
```

A quoted string is a value of type `String`, although it is normally called an *object* of type `String` rather than a value of type `String`. An object of type `String` is a sequence of characters treated as a single item. A variable of type `String` can name one of these string objects.

For example, the following declares `blessing` to be the name for a `String` variable:

```
String blessing;
```

The following assignment statement sets the value of `blessing` so that `blessing` serves as another name for the `String` object `"Live long and prosper."`:

```
blessing = "Live long and prosper.";
```

The declaration and assignment can be combined into a single statement, as follows:

```
String blessing = "Live long and prosper.";
```

You can write the object named by the `String` variable `blessing` to the screen as follows:

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

which produces the screen output

```
Live long and prosper.
```

The String Class

The class `String` is a predefined class that is automatically made available to you when you are programming in Java. Objects of type `String` are strings of characters that are written within double quotes. For example, the following declares the variable `motto` to be of type `String` and makes `motto` a name for the `String` object `"We aim to please."`:

```
String motto = "We aim to please.";
```

CONCATENATION OF STRINGS

+ operator
concatenation

When you use the + operator on two strings, the result is the string obtained by connecting the two strings to get a longer string. This is called **concatenation**. So, when it is used with strings, the + is sometimes called the **concatenation operator**. For example, consider the following:

```
String noun = "Strings";
String sentence;
sentence = noun + "are cool.";
System.out.println(sentence);
```

This will set the variable `sentence` to `"Stringsare cool."` and will output the following to the screen:

```
Stringsare cool.
```

Note that no spaces are added when you concatenate two strings. If you wanted `sentence` set to `"Strings are cool."`, then you should change the assignment statement to add the extra space. For example, the following will add the desired space:

```
sentence = noun + " are cool.";
```

We added a space before the word "are".

Using the + Sign with Strings

If you connect two strings with the + operator, the result is the concatenation (pasting) of the two strings.

Example:

```
String name = "Chiana";
String farewell = "Good bye " + name;
System.out.println(farewell);
```

This sets farewell to the string "Good bye Chiana". So, it outputs the following to the screen:

```
Good bye Chiana
```

Note that we added a space at the end of "Good bye ".

You can concatenate any number of Strings using the + operator. Moreover, you can use the + operator to concatenate a String to almost any other type of item. The result is always a String. In most situations, Java will convert an item of any type to a string when you connect it to a string with the + operator. For numbers, it does the obvious thing. For example,

```
String solution = "The answer is " + 42;
```

will set the String variable solution to "The answer is 42". Java converts the integer constant 42 to the string "42" and then concatenates the two strings "The answer is " and "42" to obtain the longer string "The answer is 42".

Notice that a number or other value is converted to a string object only when it is connected to a string with a plus sign. If it is connected to another number with a plus sign, it is not converted to a string. For example,

```
System.out.println("100" + 42);
```

outputs

```
10042
```

but

```
System.out.println(100 + 42);
```

outputs

```
142
```

CLASSES

class
object

Classes are central to Java and you will soon be defining and using your own classes. The class `String`, which we discuss in this section, gives us an opportunity to introduce some of the notation and terminology used for classes. A **class** is the name for a type whose values are objects. **Objects** are entities that store data and can take actions. For example, objects of the class `String` store data consisting of strings of characters, such as `"Hello"`. The actions that an object can take are called **methods**. Most of the methods for the class `String` return some value—that is, produce some value. For example, the method `length()` returns the number of characters in a `String` object. So, `"Hello".length()` returns the integer `5` and this returned value can be stored in an `int` variable as follows:

method

```
int n = "Hello".length();
```

As indicated by the example `"Hello".length()`, a method is called into action by writing a name for the object followed by a dot followed by the method name with parentheses. When you call a method into action, you are (or your code is) said to **invoke** the method or **call** the method, and the object before the dot is known as the **calling object**.

method call or
method invocation
calling object

Although you can call a method with a constant object, as in `"Hello".length()`, it is more common to use a variable as the calling object, as illustrated by the following:

```
String greeting = "Hello";
int n = greeting.length();
```

Information needed for the method invocation is given in the parentheses. In some cases, like the method `length`, no information is needed (other than the data in the calling object) and the parentheses are empty. In other cases, which we see soon, there is some information that must be provided inside the parentheses. The information in parentheses is known as an **argument** (or arguments).

argument

sending a
message

Invoking a method is also sometimes called **sending a message** to the object. With this view a message is sent to the object (by invoking a method) and in response the object performs some action. For example, in response to the message

```
greeting.length()
```

the object `greeting` answers with the value `5`.

All objects within a class have the same methods, but each object can have different data. For example, the two `String` objects `"Hello"` and `"Good-Bye"` have different data—that is, different strings of characters. However, they have the same methods. Thus, since we know that the `String` object `"Hello"` has the method `length()`, we know that the `String` object `"Good-Bye"` must also have the method `length()`.

You now have seen two kinds of types in Java: primitive types and class types. The main difference you have seen between these two kinds of types is that classes have

Classes, Objects, and Methods

A Java program works by having things called **objects** perform actions. The actions are known as **methods** and typically involve data contained in the object. All objects of the same kind are said to be of the same class. So, a **class** is a category of objects. When the object performs the action of a given method, that is called **invoking** the method (or **calling** the method). Information provided to the method in parentheses is called the **argument** (or arguments).

For example, in Display 1.1, `System.out` is an object, `println` is a method, and the following is an invocation of the method by this object using the argument `"Hello reader."`:

```
System.out.println("Hello reader.");
```

methods and primitive types do not have methods. We will later see more differences between classes and primitive types. A smaller difference between primitive types and class types is that all the primitive types are spelled using only lowercase letters but, by convention, class types are spelled with their first letter in uppercase, as in `String`.

STRING METHODS

The class `String` has a number of useful methods that can be used for string-processing applications. A sample of these `String` methods is presented in Display 1.4. Some of the notation and terminology used in Display 1.4 is described in the box entitled "Returned Value." A more complete list of `String` methods is given in Appendix 4.

As with any method, a `String` method is called (invoked) by writing a `String` object, a dot, the name of the method, and finally a pair of parentheses that enclose any arguments to the method. Let's look at some examples.

As we've already noted, the method `length` can be used to find out the number of characters in a string. You can use a call to the method `length` anywhere that you can use a value of type `int`. For example, all of the following are legal Java statements: length

```
String greeting = "Hello";
int count = greeting.length();
System.out.println("Length is " + greeting.length());
```

Some methods for the class `String` depend on counting **positions** in the string. Positions are counted starting with 0 not with 1. So, in the string `"Surf time"`, `'S'` is in position `0`, `'u'` is in position `1`, and so forth. A position is usually referred to as an **index**. So, it would be preferable to say: `'S'` is at index `0`, `'u'` is at index `1`, and so forth. position index

The method `indexOf` can be used to find the index of a substring of the calling objects. For example, consider

```
String phrase = "Java is fun.";
```

Returned Value

An expression like `numberOfGirls + numberOfBoys` produces a value. If `numberOfGirls` has the value 2 and `numberOfBoys` has the value 10, then the number produced is 12. The number 12 is the result of evaluating the expression.

Some method invocations are simple kinds of expression, and any such method invocation evaluates to some value. If a method invocation produces a value, we say that the method *returns* the value. For example, suppose your program executes

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

After that, if you evaluate `greeting.length()`, the value returned will be 6. So the following code outputs the integer 6:

```
String greeting = "Hello!";
System.out.println(greeting.length());
```

A method can return different values depending on what happens in your program. However, each method can return values of only one type. For example, the method `length` of the class `String` always returns an `int` value. In Display 1.4, The type given before the method name is the type of the values returned by that method. Since `length` always returns an `int` value, the entry for `length` begins

```
int length()
```

Display 1.4 Some Methods in the Class `String` *(Part 1 of 4)*

`int length()`

Returns the length of the calling object (which is a string) as a value of type `int`.

Example:

After program executes `String greeting = "Hello!";`
`greeting.length()` returns 6.

`boolean equals(`*Other_String*`)`

Returns `true` if the calling object string and the *Other_String* are equal. Otherwise, returns `false`.

Example:

After program executes `String greeting = "Hello";`
`greeting.equals("Hello")` returns `true`
`greeting.equals("Good-Bye")` returns `false`
`greeting.equals("hello")` returns `false`

Note that case matters. `"Hello"` and `"hello"` are not equal because one starts with an uppercase letter and the other starts with a lowercase letter.

Display 1.4 Some Methods in the Class String *(Part 2 of 4)*

`boolean equalsIgnoreCase(`*Other_String*`)`

Returns `true` if the calling object string and the *Other_String* are equal, considering uppercase and lowercase versions of a letter to be the same. Otherwise, returns `false`.

EXAMPLE:

After program executes `String name = "mary";`
`greeting.equalsIgnoreCase("Mary!")` returns `true`

`String toLowerCase()`

Returns a string with the same characters as the calling object string, but with all letter characters converted to lowercase.

EXAMPLE:

After program executes `String greeting = "Hi Mary!";`
`greeting.toLowerCase()` returns `"hi mary!"`.

`String toUpperCase()`

Returns a string with the same characters as the calling object string, but with all letter characters converted to uppercase.

EXAMPLE:

After program executes `String greeting = "Hi Mary!";`
`greeting.toUpperCase()` returns `"HI MARY!"`.

`String trim()`

Returns a string with the same characters as the calling object string, but with leading and trailing white space removed. Whitespace characters are the characters that print as white space on paper, such as the blank (space) character, the tab character, and the new-line character `'\n'`.

EXAMPLE:

After program executes `String pause = "   Hmm    ";`
`pause.trim()` returns `"Hmm"`.

`char charAt(`*Position*`)`

Returns the character in the calling object string at the *Position*. Positions are counted 0, 1, 2, etc.

EXAMPLE:

After program executes `String greeting = "Hello!";`
`greeting.charAt(0)` returns `'H'`, and
`greeting.charAt(1)` returns `'e'`.

Display 1.4 Some Methods in the Class `String` *(Part 3 of 4)*

`String substring(`*Start*`)`

Returns the substring of the calling object string starting from *Start* through to the end of the calling object. Positions are counted 0, 1, 2, etc. Be sure to notice that the character at position *Start* is included in the value returned.

EXAMPLE:

After program executes `String sample = "AbcdefG";`
`sample.substring(2)` returns `"cdefG"`.

`String substring(`*Start*`, `*End*`)`

Returns the substring of the calling object string starting from position *Start* through, but not including, position *End* of the calling object. Positions are counted 0, 1, 2, etc. Be sure to notice that the character at position *Start* is included in the value returned, but the character at position *End* is not included.

EXAMPLE:

After program executes `String sample = "AbcdefG";`
`sample.substring(2, 5)` returns `"cde"`.

`int indexOf(`*A_String*`)`

Returns the index (position) of the first occurrence of the string *A_String* in the calling object string. Positions are counted 0, 1, 2, etc. Returns −1 if *A_String* is not found.

EXAMPLE:

After program executes `String greeting = "Hi Mary!";`
`greeting.indexOf("Mary")` returns 3, and
`greeting.indexOf("Sally")` returns −1.

`int indexOf(`*A_String*`, `*Start*`)`

Returns the index (position) of the first occurrence of the string *A_String* in the calling object string that occurs at or after position *Start*. Positions are counted 0, 1, 2, etc. Returns −1 if *A_String* is not found.

EXAMPLE:

After program executes `String name = "Mary, Mary quite contrary";`
`name.indexOf("Mary", 1)` returns 6.
The same value is returned if 1 is replaced by any number up to and including 6.
`name.indexOf("Mary", 0)` returns 0.
`name.indexOf("Mary", 8)` returns −1.

Display 1.4 Some Methods in the Class String *(Part 4 of 4)*

`int lastIndexOf(A_String)`

Returns the index (position) of the last occurrence of the string *A_String* in the calling object string. Positions are counted 0, 1, 2, etc. Returns −1, if *A_String* is not found.

EXAMPLE:

After program executes `String name = "Mary, Mary, Mary quite so";`
`greeting.indexOf("Mary")` returns 0, and
`name.lastIndexOf("Mary")` returns 12.

`int compareTo(A_String)`

Compares the calling object string and the string argument to see which comes first in the lexicographic ordering. Lexicographic order is the same as alphabetical order but with the characters ordered as in Appendix 3. Note that in Appendix 3 all the uppercase letters are in regular alphabetical order and all the lowercase letters are in alphabetical order, but all the uppercase letters precede all the lowercase letters. So, lexicographic ordering is the same as alphabetical ordering provided both strings are either all uppercase letters or both strings are all lowercase letters. If the calling string is first, it returns a negative value. If the two strings are equal, it returns zero. If the argument is first, it returns a positive number.

EXAMPLE:

After program executes `String entry = "adventure";`
`entry.compareTo("zoo")` returns a negative number,
`entry.compareTo("adventure")` returns 0, and
`entry.compareTo("above")` returns a positive number.

`int compareToIgnoreCase(A_String)`

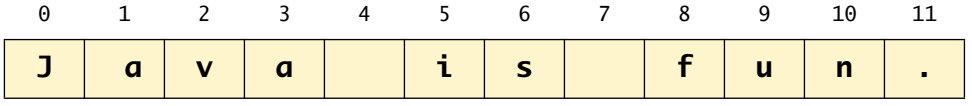
Compares the calling object string and the string argument to see which comes first in the lexicographic ordering, treating uppercase and lowercase letters as being the same. (To be precise, all uppercase letters are treated as if they were their lowercase versions in doing the comparison.) Thus, if both strings consist entirely of letters, the comparison is for ordinary alphabetical order. If the calling string is first, it returns a negative value. If the two strings are equal ignoring case, it returns zero. If the argument is first, it returns a positive number.

EXAMPLE:

After program executes `String entry = "adventure";`
`entry.compareToIgnoreCase("Zoo")` returns a negative number,
`entry.compareToIgnoreCase("Adventure")` returns 0, and
`"Zoo".compareToIgnoreCase(entry)` returns a positive number.

Display 1.5 String Indexes

The 12 characters in the string `"Java is fun."` have indexes `0` through `11`.



Notice that the blanks and the period count as characters in the string.

After this declaration, the invocation `phrase.indexOf("is")` will return `5` because the `'i'` of `"is"` is at index `5`. (Remember, the first index is `0`, not `1`.) This is illustrated in Display 1.5.

ESCAPE SEQUENCES

backslash \
escape sequence

A backslash, \, preceding a character tells the compiler that the character following the \ does not have its usual meaning. Such a sequence is called an **escape sequence** or an **escape character**. The sequence is typed in as two characters with no space between the symbols. Several escape sequences are defined in Java.

If you want to put a backslash, \, or a quote symbol, ", into a string constant, you must escape the ability of the " to terminate a string constant by using \", or the ability of the \ to escape, by using \\. The \\ tells the compiler you mean a real backslash, \, not an escape sequence, and \" means a quote character, not the end of a string constant. A list of escape sequences is given in Display 1.6.

It is important to note that each escape sequence is a single character, even though it is spelled with two symbols. So, the string `"Say \"Hi\"!"` contains 9 characters (`'S'`, `'a'`, `'y'`, the blank character, `'\"'`, `'H'`, `'i'`, `'\"'`, and `'!'`), not 11 characters.

Display 1.6 Escape Sequences

```
\" Double quote.
\' Single quote.
\\ Backslash.
\n New line. Go to the beginning of the next line.
\r Carriage return. Go to the beginning of the current line.
\t Tab. White space up to the next tab stop.
```

Including a backslash in a quoted string is a little tricky. For example, the string "abc\def" is likely to produce the error message "Invalid escape character." To include a backslash in a string, you need to use two backslashes. The string "abc\\def", if output to the screen, would produce

```
abc\def
```

The escape sequence \n indicates the start of a new line. For example, the statement

```
System.out.println("To be or\nNot to be.");
```

will write the following to the screen:

```
To be or
Not to be.
```

You do not need to use the escape sequence \' to include a single quote inside a quoted string. For example, "Time's up!" is a valid quoted string. However, you do need \' if you want to indicate the constant for the single-quote character, as in

```
char singleQuote = '\'';
```

STRING PROCESSING

In Java an object of type String is an **immutable object**, meaning that the characters in the String object cannot be changed. This will eventually prove to be important to us, but at this stage of our exploration of Java, it is a misleading statement. To see that an object of type String cannot be changed, note that none of the methods in Display 1.4 changes the value of the String calling object. There are more String methods than those shown in Display 1.4, but none of them lets you write statements that say things like "Change the fifth character in the calling object string to 'x'". This was done intentionally to make the implementation of the String class more efficient and for other reasons that we will discuss later in this book. There is another string class, called StringBuffer, that has methods for altering its string object. We will not discuss the class StringBuffer in this text, but a table explaining many of the methods of the class StringBuffer is included in Appendix 4.

immutable object

Although there is no method that allows you to change the value of a String object, such as "Hello", you can still write programs that change the value of a String variable, which is probably all you want anyway. To perform the change, you simply use an assignment statement, as in the following example:

```
String name = "Soprano";
name = "Anthony " + name;
```

The assignment statement in the second line changes the value of the name variable so that the string it names changes from "Soprano" to "Anthony Soprano". Display 1.7 contains a demonstration of some simple string processing.

THE UNICODE CHARACTER SET

ASCII

Until recently, most programming languages used the *ASCII* character set, which is given in Appendix 3. The **ASCII** character set is simply a list of all the characters normally used on an English-language keyboard plus a few special characters. In this list, each character has been assigned a number so that characters can be stored by storing the corresponding number. Java, and now many other programming languages, uses the *Unicode* character set. The **Unicode** character set includes the ASCII character set plus many of the characters used in languages with a different alphabet from English. This is not likely to be a big issue if you are using an English-language keyboard. Normally, you can just program as if Java were using the ASCII character set. The ASCII character set is a subset of the Unicode character set, and the subset you are likely to use. Thus, Appendix 3, which lists the ASCII character set, in fact lists the subset of the Unicode character set that we will use in this book. The advantage of the Unicode char-

Unicode

Display 1.7 Using the `String` Class


```
1  public class StringProcessingDemo
2  {
3      public static void main(String[] args)
4      {
5          String sentence = "I hate text processing!";
6          int position = sentence.indexOf("hate");
7          String ending =
8                  sentence.substring(position + "hate".length());
9
10         System.out.println("012345678901234567890123");
11         System.out.println(sentence);
12         System.out.println("The word \"hate\" starts at index "
13                                                  + position);
14         sentence = sentence.substring(0, position) + "adore"
15                                                  + ending;
16         System.out.println("The changed string is:");
17         System.out.println(sentence);
18     }
19 }
```

You could just use 4 here, but if you had a `String` variable instead of `"hate"`, you would have to use `length` as shown.

SAMPLE DIALOGUE

```
012345678901234567890123
I hate text processing!
The word "hate" starts at index 2
The changed string is:
I adore text processing!
```

acter set is that it makes it possible to easily handle languages other than English. For example, it is legal to spell a Java identifier using the letters of the Greek alphabet (although you may want a Greek-language keyboard and monitor to do this). The disadvantage of the Unicode character set is that it sometimes requires more computer memory to store each character than it would if Java used only the ASCII character set.

Self-Test Exercises

24. What is the output produced by the following?

```
String verbPhrase = "is money";
System.out.println("Time" + verbPhrase);
```

25. What is the output produced by the following?

```
String test = "abcdefg";
System.out.println(test.length());
System.out.println(test.charAt(1));
```

26. What is the output produced by the following?

```
String test = "abcdefg";
System.out.println(test.substring(3));
```

27. What is the output produced by the following?

```
System.out.println("abc\ndef");
```

28. What is the output produced by the following?

```
System.out.println("abc\\ndef");
```

29. What is the output produced by the following?

```
String test = "Hello Tony";
test = test.toUpperCase();
System.out.println(test);
```

30. What is the output of the following two lines of Java code?

```
System.out.println("2 + 2 = " + (2 + 2));
System.out.println("2 + 2 = " + 2 + 2);
```

31. Suppose `sam` is an object of a class named `Person` and suppose `increaseAge` is a method for the class `Person` that takes one argument that is an integer. How do you write an invocation of the method `increaseAge` using `sam` as the calling object and using the argument `10`? The method `increaseAge` will change the data in `sam` so that it simulates `sam` aging by 10 years.

1.4 Program Style

In matters of grave importance,
style, not sincerity, is the vital thing.

Oscar Wilde, The Importance of Being Earnest

Java programming style is similar to that used in other languages. The goal is to make your code easy to read and easy to modify. This section gives some basic points on good programming style in general and some information on the conventions normally followed by Java programmers.

NAMING CONSTANTS

There are two problems with numbers in a computer program. The first is that they carry no mnemonic value. For example, when the number `10` is encountered in a program, it gives no hint of its significance. If the program is a banking program, it might be the number of branch offices or the number of teller windows at the main office. To understand the program, you need to know the significance of each constant. The second problem is that when a program needs to have some numbers changed, the changing tends to introduce errors. Suppose that `10` occurs 12 times in a banking program, that 4 of the times it represents the number of branch offices, and that 8 of the times it represents the number of teller windows at the main office. When the bank opens a new branch and the program needs to be updated, there is a good chance that some of the `10`'s that should be changed to `11` will not be, or some that should not be changed will be. The way to avoid these problems is to name each number and use the name instead of the number within your program. For example, a banking program might have two constants with the names `BRANCH_COUNT` and `WINDOW_COUNT`. Both of these numbers might have a value of `10`, but when the bank opens a new branch, all you need to do to update the program is to change the definition of `BRANCH_COUNT`.

One way to name a number is to initialize a variable to that number value, as in the following example:

```
int BRANCH_COUNT = 10;
int WINDOW_COUNT = 10;
```

There is, however, one problem with this method of naming number constants: You might inadvertently change the value of one of these variables. Java provides a way of marking an initialized variable so that it cannot be changed. The syntax is

```
public static final Type Variable = Constant;
```

For example, the names BRANCH_COUNT and WINDOW_COUNT can be given values that cannot be changed by your code as follows:

```
public static final int BRANCH_COUNT = 10;
public static final int WINDOW_COUNT = 10;
```

These constant definitions must be placed outside of the main method and, when we start having more methods, outside of any other methods. This is illustrated in Display 1.8. When we start writing programs and classes with multiple methods, you will see that the defined constants can be used in all the methods of a class.

Display 1.8 Comments and a Named Constant


```
1  /**
2   Program to show interest on a sample account balance.
3   Author: Jane Q. Programmer.
4   E-mail Address: janeq@somemachine.etc.etc.
5   Last Changed: September 21, 2004.
6  */
7  public class ShowInterest
8  {
9      public static final double INTEREST_RATE = 2.5;

10     public static void main(String[] args)
11     {
12         double balance = 100;
13         double interest; //as a percent

14         interest = balance * (INTEREST_RATE/100.0);
15         System.out.println("On a balance of $" + balance);
16         System.out.println("you will earn interest of $"
17                                                 + interest);
18         System.out.println("All in just one short year.");
19     }
20
21 }
```

Although it would not be as clear, it is legal to place the definition of INTEREST_RATE here instead.

SAMPLE DIALOGUE

```
On a balance of $100.0
you will earn interest of $2.5
All in just one short year.
```

We will fully explain the modifiers `public static final` later in this book, but we can now explain most of what they mean. The part

```
int BRANCH_COUNT = 10;
```

simply declares `BRANCH_COUNT` as a variable and initializes it to `10`. The words that precede this modify the variable `BRANCH_COUNT` in various ways. The word `public` says there are no restrictions on where you can use the name `BRANCH_COUNT`. The word `static` will have to wait until Chapter 5 for an explanation, but be sure to include it. The word `final` means that the value `10` is the *final* value assignment to `BRANCH_COUNT`, or, to phrase it another way, that the program is not allowed to change the value of `BRANCH_COUNT`.

Naming Constants

The syntax for defining a name for a constant, such as a name for a number, is as follows:

Syntax:

```
public static final Type Variable = Constant;
```

Example:

```
public static final int MAX_SPEED = 65;
public static final double MIN_SIZE = 0.5;
public static final String GREETING = "Hello friend!";
public static final char GOAL = 'A';
```

Although it is not required, it is the normal practice of programmers to spell named constants using all uppercase letters with the underscore symbol used to separate "words."

JAVA SPELLING CONVENTIONS

In Java, as in all programming languages, identifiers for variables, methods, and other items should always be meaningful names that are suggestive of the identifiers' meanings. Although it is not required by the Java language, the common practice of Java programmers is to start the names of classes with uppercase letters and to start the names of variables, objects, and methods with lowercase letters. Defined constants are normally spelled with all uppercase letters and underscore symbols for "punctuation," as we did in the previous subsection.

For example, `String`, `FirstProgram`, and `JOptionPane` are classes, although we have not yet discussed the last one. The identifiers `println`, `balance`, and `readLine` should each be either a variable, object, or method.

Since blanks are not allowed in Java identifiers, "word" boundaries are indicated by an uppercase letter, as in `numberOfPods`. Since defined constants are spelled with all uppercase letters, the underscore symbol is used for "word" boundaries, as in `MAX_SPEED`.

The identifier `System.out` seems to violate this convention, since it names an object but yet begins with an uppercase letter. It does not violate the convention, but the explanation hinges on a topic we have not yet covered. `System` is the name of a class. Within the class named `System` there is a definition of an object named `out`. So, the identifier `System.out` is used to indicate the object `out` (starting with a lowercase letter for an object) that is defined in the class `System` (starting with an uppercase letter for a class). This sort of dot notation will be explained later in the book.

There is one Java convention that people new to Java often find strange. Java programmers normally do not use abbreviations in identifiers, but rather spell things out in full. A Java programmer would not use `numStars`. He or she would use `numberOfStars`. A Java programmer would not use `FirstProg`. He or she would use `FirstProgram`. This can produce long identifiers and sometimes exceedingly long identifiers. For example, `BufferedReader` and `ArrayIndexOutOfBoundsException` are the names of two standard Java classes. The first will be used in the next chapter and the second will be used later in this book. These long names cause you to do more typing and they make program lines quickly get too long. However, there is a very good reason for using these long names: With the long names, there is seldom any confusion on how the identifiers are spelled. With abbreviations, you often cannot recall how the identifier was abbreviated. Was it `BufReader` or `BuffReader` or `BufferedR` or `BR` or something else? Since all the words are spelled out, you know it must be `BufferedReader`. Once they get used to using these long names, most programmers learn to prefer them.

COMMENTS

There are two ways to insert **comments** in a Java program. In Java the symbols `//` are used to indicate the start of a comment. All of the text between the `//` and the end of the line is a comment. The compiler simply ignores anything that follows `//` on a line. If you want a comment that covers more than one line, place a `//` on each line of the comment. The symbols `//` are two slashes (without a space between them). Comments indicated with `//` are often called **line comments**.

// comments

line comments

There is another way to insert comments in a Java program. Anything between the symbol pair `/*` and the symbol pair `*/` is considered a comment and is ignored by the compiler. Unlike the `//` comments, which require an additional `//` on each line, the `/*` to `*/` comments can span several lines like so:

/*comments*/

```
/*This is a multi-line comment.
Note that there is no comment symbol
of any kind on the second line.*/
```

Comments of the `/* */` type are often called **block comments**. These block comments may be inserted anywhere in a program that a space or line break is allowed. However, they should not be inserted anywhere except where they do not distract from

block comments

the layout of the program. Usually comments are only placed at the ends of lines or on separate lines by themselves.

Java comes with a program called `javadoc` that will automatically extract documentation from the classes you define. The workings of the `javadoc` program dictate when you normally use each kind of comment.

The `javadoc` program will extract a `/* */` comment in certain situations, but `javadoc` will not extract a `//` comment. We will say more about `javadoc` and comments after we discuss defining classes. In the meantime you may notice the following conventions in our code:

We use line comments (that is, the `//` kind) for comments meant only for the code writer or for a programmer who modifies the code and not for any other programmer who merely uses the code.

For comments that would become part of the documentation for users of our code, we use block comments (that is, the `/* */` kind). The `javadoc` program allows you to indicate whether or not a block comment is eligible to be extracted for documentation. If the opening `/*` has an extra asterisk, like so `/**`, then the comment is eligible to be extracted. If there is only one asterisk, `javadoc` will not extract the comment. For this reason, our block comments invariably open with `/**`.

when to comment

It is difficult to say just how many comments a program should contain. The only correct answer is "just enough," which of course conveys little to the novice programmer. It will take some experience to get a feel for when it is best to include a comment. Whenever something is important and not obvious, it merits a comment. However, providing too many comments is as bad as providing too few. A program that has a comment on each line is so buried in comments that the structure of the program is hidden in a sea of obvious observations. Comments like the following contribute nothing to understanding and should not appear in a program:

```
interest = balance * rate; //Computes the interest.
```

self-documenting

A well-written program is what is called **self-documenting**, which means that the structure of the program is clear from the choice of identifier names and the indenting pattern. A completely self-documenting program would need none of these `//` comments that are only for the programmer who reads or modifies the code. That may be an ideal that is not always realizable, but if your code is full of `//` comments and you follow our convention on when to use `//` comments, then either you simply have too many comments or your code is poorly designed.

A very simple example of the two kinds of comments is given in Display 1.8.

INDENTING

We will say more about indenting as we introduce more Java. However, the general rule is easy to understand and easy to follow. When one structure is nested inside another structure, the inside structure is indented one more level. For example, in our pro-

grams, the `main` method is indented one level, and the statements inside the main method are indented two levels. We prefer to use four spaces for each level of indenting. More than four spaces eats up too much line length. It is possible to get by with indenting only two or three spaces for each level so long as you are consistent. One space for a level of indenting is not enough to be clearly visible.

Self-Test Exercises

32. What are the two kinds of comments in Java?
33. What is the output produced by the following Java code?

```
/**
 Code for Exercise.
*/
System.out.println("Hello");
//System.out.print("Mr. or Ms. ");
System.out.println("Student");
```

34. What is the normal spelling convention for named constants?
35. Write a line of Java code that will give the name `ANSWER` to the `int` value `42`. In other words, make `ANSWER` a named constant for `42`.

Chapter Summary

- Compiling a Java class or program produces byte-code, which is the machine language for a fictitious computer. When you run the byte-code, a program called an *interpreter* translates and executes the byte-code instructions on your computer one instruction at a time.
- A variable can be used to hold values, like numbers. The type of the variable must match the type of the value stored in the variable. All variables must be declared before they are used.
- The equal sign, `=`, is used as the assignment operator in Java. An assignment statement is an instruction to change the value of a variable.
- Each variable should be initialized before the program uses its value.
- Parentheses in arithmetic expressions indicate which arguments are given to an operator. When parentheses are omitted, Java adds implicit parentheses using precedence rules and associativity rules.
- You can have variables and constants of type `String`. `String` is a class type, not a primitive type.
- You can use the plus sign to concatenate two strings.
- There are methods in the class `String` that can be used for string processing.

- Variables (and all other items in a program) should be given names that indicate how they are used.
- You should define names for number constants in a program and use these names rather than writing out the numbers within your program.
- Programs should be self-documenting to the extent possible. However, you should also insert comments to explain any unclear points.

Answers to Self-Test Exercises

1. `Java is not a drink.`

2. ```
   System.out.println("I like Java.");
   System.out.println("You like tea.");
   ```

3. ```
   public class HelloWorld
   {
       public static void main(String[] args)
       {
          System.out.println("Hello World!");
       }
   }
   ```

4. A compiler translates a program written in a programming language such as Java into a program in a low-level language. When you compile a Java program, the compiler translates your Java program into a program expressed in Java byte-code.

5. The program that is input to a compiler is called the *source program*.

6. The translated program that is produced by a compiler is called the *object program* or *object code*.

7. A program that runs Java byte-code instructions is called an *interpreter*. It is also often called the *Java Virtual Machine*.

8. `NiceClass.java`

9. `NiceClass.class`

10. `1stPlayer` may not be used because it starts with a digit; `myprogram.java` may not be used because it contains an illegal symbol, the dot; `long` may not be used because it is a keyword. All the others may be used as variable names. However, `TimeLimit`, while legal, violates the style rule that all variable names begin with a lowercase letter.

11. Yes, a Java program can have two different variables named `number` and `Number`. However, it would not be good style to do so.

12. ```
    int feet = 0, inches = 0;
    ```

13. ```
    int count = 0;
    double distance = 1.5;
    ```

14. ```
    distance = time * 80;
    ```
    ```

15. `interest = balance * rate;`

16. `b`
    `c`
    `c`

17. `3*x`
    `3*x + y`
    `(x + y)/7` Note that `x + y/7` is not correct.
    `(3*x + y)/(z + 2)`

18. `(1/3) * 3 is equal to 0.0`

    Since `1` and `3` are of type `int`, the `/` operator performs integer division, which discards the remainder, so the value of `1/3` is `0`, not `0.3333`…. This makes the value of the entire expression `0 * 3`, which of course is `0`.

19. `quotient = 2`
    `remainder = 1`

20. `result is 5`

21. a. `52.0`

    b. `9/5` has `int` value `1`; because the numerator and denominator are both of type `int`, integer division is done; the fractional part is discarded. The programmer probably wanted floating-point division, which does not discard the part after the decimal point.

    c. `fahrenheit = (9.0/5) * celsius + 32.0;`
    or this
    `fahrenheit = 1.8 * celsius + 32.0;`

22. `n == 3`

23. `n == 4`
    `n == 3`

24. `Timeis money`

25. `7`
    `b`

26. `defg`

27. `abc`
    `def`

28. `abc\ndef`

29. `HELLO TONY`

30. The output is

    `2 + 2 = 4`

```
2 + 2 = 22
```

In the expression `"2 + 2 = " + (2 + 2)` the integers 2 and 2 in `(2 + 2)` are added to obtain the integer 4. When 4 is connected to the string `"2 + 2"` with a plus sign, the integer 4 is converted to the string `"4"` and the result is the string `"2 + 2 = 4"`. However `"2 + 2 = " + 2 + 2` is interpreted by Java to mean

```
("2 + 2 = " + 2) + 2
```

The first integer 2 is changed to the string `"2"` because it is being combined with the string `"2 + 2"`. The result is the string `"2 + 2 = 2"`. The last integer 2 is combined with the string `"2 + 2 = 2"`. So, the last 2 is converted to the string `"2"`. So the final result is

```
"2 + 2 = 2" + "2"
```

which is `"2 + 2 = 22"`.

31. `sam.increaseAge(10);`
32. The two kinds of comments are `//` comments and `/* */` comments. Everything following a `//` on the same line is a comment. Everything between a `/*` and a matching `*/` is a comment.
33.
```
Hello
Student
```
34. The normal spelling convention is to spell named constants using all uppercase letters with the underscore symbol used to separate words.
35. `public static final int ANSWER = 42;`

Programming Projects

1. A government research lab has concluded that an artificial sweetener commonly used in diet soda pop will cause death in laboratory mice. A friend of yours is desperate to lose weight but cannot give up soda pop. Your friend wants to know how much diet soda pop it is possible to drink without dying as a result. Write a program to supply the answer. The program has no input but does have defined constants for the following items: the amount of artificial sweetener needed to kill a mouse, the weight of the mouse, the starting weight of the dieter, and the desired weight of the dieter. To ensure the safety of your friend, be sure the program uses the weight at which the dieter will stop dieting, rather than the dieter's current weight, to calculate how much soda pop the dieter can safely drink. You may use any reasonable values for these defined constants. Assume that diet soda contains 1/10th of one percent artificial sweetener. Use another named constant for this fraction. You may want to express the percent as the `double` value `0.001`. (If your program turns out not to use a defined constant, you may remove that defined constant from your program.)

2. Write a program that starts with a line of text and then outputs that line of text with the first occurrence of "hate" changed to "love". For example, a possible sample output might be

```
The line of text to be changed is:
I hate you.
I have rephrased that line to read:
I love you.
```

You can assume that the word "hate" occurs in the input. If the word "hate" occurs more than once in the line, your program will replace only the first occurrence of "hate". Since we will not discuss input until Chapter 2, use a defined constant for the string to be changed. To make your program work for another string, you should only need to change the definition of this defined constant.

For additional online Programming Projects, click the CodeMate icons below.

1.3 

1.4