2 Program Fundamentals

In this chapter we introduce programming fundamentals in Java. We write a Java program by using different elements of the Java language. The most basic elements are the tokens of the language. Tokens are the words that make up the sentences of the program. Programming is comparable to writing. To write an essay we write words that make up sentences, sentences that make up paragraphs, and paragraphs that make up essays. In this chapter we concentrate on how to use the "words" and combine them into the program equivalent of useful sentences and paragraphs.

2.1 "Hello, world!" in Java

A simple first Java program is the classic "Hello, world!" program, so named because it prints the message `Hello, world!` on the computer's screen.

```
/* HelloWorld.java
 * Purpose:
 *   The classic "Hello, world!" program.
 *   It prints a message to the screen.
 * Author: Jane Programmer
 *       as derived from Kernighan and Richie
 */

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

DISSECTION OF THE HelloWorld PROGRAM

❑
```
/* HelloWorld.java
 * Purpose:
 ...
 */
```

Everything between a `/*` and a `*/` is a comment. Comments are ignored by the Java compiler and are inserted to make the program more understandable to the human reader. Every program should begin with a comment such as the one in this example. In our examples, the name of the file appears in the comment. This example indicates that it is from the file *HelloWorld.java*. Other things to include in program comments are the function or purpose of the program, the author of the program, and a revision history with dates, indicating major modifications to the program.

❑
```
class HelloWorld {
```

The word `class` is a keyword preceding the name of the *class*. A *keyword* has a predefined special purpose. A *class* is a named collection of data and instructions. The name of the class being defined in this example is `HelloWorld`. The left brace "{" begins the definition of a class. A matching right brace "}" is needed to end the class definition. Forgetting to match braces is a common error.

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

This line declares that the class `HelloWorld` contains a *method* with the name `main`. A *method* is a named group of instructions within a class. In this example, only one method, named `main`, is defined for the class `HelloWorld`. In Chapter 4, Methods: Functional Abstraction, we create classes that contain several methods in a single class. When we use the name of a method, we add parentheses at the end to remind you that it is a method. This convention comes from actual Java code, wherein the name of a method is always followed by parentheses. The method defined by this line is thus referred to as `main()`.

There are two kinds of Java programs: *stand-alone applications* and *applets.* The method `main()` appears in every stand-alone Java program indicating where program execution will begin. Later we use a different line that serves a similar purpose for applets. An explanation of the words `public`, `static`, and `void` in this line is left until later.

❑
```
{
  System.out.println("Hello, world!");
}
```

The entire body of the method `main()`, the real instructions to the com-

puter, appears between the braces. In this example, just one instruction prints the desired message. You can memorize this instruction and use it as an incantation to get something printed out on your computer screen. What gets printed is between the quotation marks.

2.2 Compiling and Running Your Java Program

You will be dealing with two different representations of your programs: the part you write and a form more suitable for the computer to use when it finally runs your program. The text you write to give the computer instructions is called the *source code* or simply the *source*. This source code will be compiled by the Java *compiler* into a form more suitable as instructions to the computer called *object code*. The source code form of the program is represented in the language Java that you will be learning. Informally, you can think of the source code as the *raw* form of the program in contrast to the object code, which is the *cooked* or compiled form. In Java, all source code file names have the suffix *.java*, such as *HelloWorld.java*. The result of correctly compiling *HelloWorld.java* is the object code *HelloWorld.class*. In some situations, the name of the file without the *.java* and the name of the class defined in the file must be the same. Although this requirement does not apply to the programs in the first part of this book, we follow that practice even when it isn't required and suggest that you do the same.

There are many names and forms for the machine representation of a program after it has been processed somewhat by the computer. A common first step in processing source code is to compile it, which means to translate it into a form more suitable for the computer. For most common programming languages, this compiled form is called the *machine code, object code,* or *binary form*. When you buy a piece of software you usually get *binaries* or an *executable image*. For Java, this form is a bit different and is called *Java Bytecode*. These bytecodes are the same whether you are running on a Macintosh, on an Intel machine with Microsoft Windows, or on a machine from Sun running Unix. This sameness is an important advantage of Java programs, as they can be written to be platform independent, which is generally not true for most other programming languages, such as C or COBOL.

In Java, all bytecode files must have a name that ends in *.class* such as *HelloWorld.class.* The word *class* is used because, in Java, programs are broken into chunks called classes, much like a chapter is broken into sections. The following diagram illustrates the compilation process and the conventions just described.

There are two principal methods for compiling and running Java programs. One method uses an Integrated Development Environment (IDE), of which there are many for Java.

The actual details of compiling for each of the IDEs vary slightly but the basic steps are the same.

COMPILING AND RUNNING JAVA PROGRAMS

1. Use the editor that comes with the IDE to create the source file. These editors generally help with syntax by using special fonts or colors for keywords and helping you match braces, and so on.
2. Create some kind of project and add your source file to the project.
3. Select Run from a menu. The IDE will automatically determine that the source file needs to be compiled and compile it before trying to run the program.

The other principal method is the command line approach. In it you run the compiler yourself from the command line of either a DOS or Unix shell program. For both Unix and DOS shells the steps are the same.

COMMAND LINE APPROACH

1. Use your favorite text editor to create the source file.
2. Compile the program with the command `javac` followed by the source file name—for example, `javac HelloWorld.java`.
3. If the program compiles without errors, run the program with the command `java` followed by the name of the class. Do not append *.class*, as in *HelloWorld.class*; use only the class name— for example, `java HelloWorld`.

The last two steps are shown as follows for the program `HelloWorld`.

```
os-prompt>javac HelloWorld.java
os-prompt>java HelloWorld
Hello, world!
os-prompt>
```

2.3 LEXICAL ELEMENTS

As with any language there are certain rules about how words in the Java programming language are formed and what constitutes a legal sentence. In addition, Java has rules about how sentences can be put together to form a complete program, somewhat like the less formal rules about how to format a formal letter with the address of the recipi-

ent and opening salutation. In this section we begin by looking at the various words and symbols, called *lexical elements*, that are used to construct Java programs.

The most fundamental element in the structure of a program is a single character that can be displayed on a computer screen or typed at a computer keyboard. Prior to Java, most programming languages, such as C and C++, used the ASCII character set. It provides for 127 different characters, which is enough to represent all the characters on the conventional English language keyboard. This set may seem like a lot, but when you consider all the human languages in the world and the various symbols they use, it is inadequate. Because Java was designed to be used throughout the world, not just in English-speaking countries, Java developers adopted the Unicode character set. It provides for more than 64,000 different characters.

When a Java compiler first begins to analyze a Java program, it groups the individual characters into larger lexical elements, usually called *tokens*. Some tokens—such as the plus sign, +, which is the Java symbol used to add two numbers—are only one character long. Other tokens—such as the keywords `class` and `public`—are many characters long. These basic tokens are then combined into larger language forms, such as expressions.

There are five types of tokens: keywords, identifiers, literals, operators, and punctuation. White space and comments are two additional lexical elements that are discarded early in the compilation process.

2.3.1 White Space

White space in Java refers to the space character, which you get when you strike the space bar on the keyboard; the tab character, which is actually one character, although it may appear as several spaces on your screen; and the newline character which you get when you hit the Return or Enter key on the keyboard. White space is used primarily to make the program look nice and also serves to separate adjacent tokens that are not separated by any other punctuation and would otherwise be considered a single, longer token. For example, white space is used to separate the following three tokens:

```
public static void
```

in the `HelloWorld` program. In such situations, where one white space character is required, any number of white space characters can be used. For example, we could have put each of the words `public`, `static`, and `void` on separate lines or put lots of spaces between them as in

```
public      static      void      main(...
```

Except for string literals, which we discuss shortly, any number of adjacent white space characters—even mixing tab, space, and newline characters—is the same as just one white space character as far as the structure and meaning of the program are concerned. Stated another way, if you can legally put in one space, you can put in as many spaces, tabs, and newlines as you want. You can't put white space in the middle of a *keyword* or *identifier*, such as a *variable* or class name. We discuss keywords, identifiers, and variables later in this chapter.

2.3.2 Comments

Comments are very important in writing good code and are too often neglected. The primary purpose of a comment is to provide additional information to the person reading a program. It serves as a concise form of program documentation. As far as the computer is concerned, the comment does not result in a token; it separates other tokens or is ignored completely when it isn't needed to separate tokens. Java has three ways to specify comments.

A *single line* comment begins with // and causes the rest of the line—all characters to the next newline—to be treated as a comment and ignored by the compiler. It is called a single line comment because the comment can't be longer than a single line. By definition, the comment ends at the end of the line containing the //.

A *multiline* comment can extend across several lines in a program. The beginning of the comment is marked with /* and the end of the comment is marked with */. Everything between the marks and the marks themselves is a comment and is ignored. Here is the multiline comment from our first Java program.

```
/* HelloWorld.java
 * Purpose:
 *   This is the classic "Hello, world!" program.
 *   It simply prints a message to the screen.
 * Author:
 *    Jane Programmer
 *       as derived from Kernighan and Richie
 */
```

The single asterisks on the intermediate lines are not required and are used merely to accent the extent of the comment. These comments are also called *block comments*.

The third style of comment is a minor variation on the multiline comment. The beginning marker has an additional asterisk; that is, the beginning of the comment is marked with /** and the end of the comment is marked with */. These comments are identical to the multiline comment except that they are recognized by a special program called *javadoc* that automatically extracts such comments and produces documentation for the program organized as an HTML document. See Section 4.13 Programming Style for more about *javadoc*.

2.3.3 Keywords

Keywords, also known as *reserved words*, have a predefined special purpose and can't be used for any but that purpose. Each of the 47 keywords in Java has a special meaning to the Java compiler. A keyword must be separated from other keywords or identifiers by white space, a comment, or some other punctuation symbol. The following table shows all the Java keywords.

```
abstract    default    if          private      throw
boolean     do         implements  protected    throws
break       double     import      public       transient
byte        else       instanceof  return       try
case        extends    int         short        void
catch       final      interface   static       volatile
char        finally    long        super        while
class       float      native      switch
const       for        new         synchronized
continue    goto       package     this
```

The keywords `const` and `goto` have no meaning in Java. They are keywords in C++, a language that was a precursor to Java. They are included as keywords to facilitate error reporting when programmers with C++ experience accidentally use them. In addition, the words `null`, `true`, and `false` look like keywords in that they have a predefined meaning, but they are in fact literals, as discussed later.

2.3.4 Identifiers

Identifiers are the names used to specify different elements of a Java program, such as a class, method, or variable. (We discuss variables in Section 2.4.1 Variables.) An identifier in our first program was `HelloWorld`, a name we picked for the `class`. Another identifier was the library method name `println`, a name picked by the Java developers. In both cases, the name gives a clue as to the use of that element of the program. An *identifier* is any sequence of Java letters and digits, the first of which must be a *Java letter*, with two exceptions. A keyword can't be an identifier, and the special literal terms `true`, `false`, and `null` can't be used as identifiers. The Java letters and digits include the letter and digit symbols for many modern written languages. The Java letters include the English language uppercase and lowercase letters, the `$`, and the `_` (underscore). The last two are included because of Java's close kinship to the programming language C, which included these symbols as legal characters in identifiers. The Java digits are 0 through 9. In our examples, we use only English letters and digits. The following are some examples of legal identifiers along with comments providing some explanation.

```
data                        //variable name conveying its use
HelloWorld                  //class name
youCanAlmostMakeASentence   //unlikely
readInt                     //method name from tio
x                           //simple variable usually double
__123                       //obscure name-a poor choice
```

The following are some illegal identifiers, along with comments indicating what the sequence of symbols really is.

```
3                   //a digit or integer literal
x+y                 //an expression where x and y are identifiers
some***name         //illegal internal characters
no Space            //intended was noSpace
1floor              //cannot start with a digit
class               //keyword - cannot be used as an identifier
```

2.3.5 Literals

Java has built-in *types* for numbers, characters, and booleans. Java also has a standard class type for strings. The *type* of a data value tells the computer how to interpret the data. These built-in types are called *primitive types* in Java. *Literals,* also called *constants,* are the literal program representations of values for the primitive numeric types, the primitive type `boolean`, the primitive character type `char`, and the standard class string type `String`. Without going into the details of these various types, the following are some examples of literal values.

Java Type	Explanation	Examples
`int`	Integers—numbers without fractional parts	`123  -999  0`
`double`	Double precision numbers with fractional parts	`1.23  -0.01`
`String`	Arbitrary strings of characters	`"Oh J" "123"`
`boolean`	Logical values true or false	`true  false`
`char`	Single characters	`'a'  '1'`

Like keywords, the symbols `true` and `false` can't be used as identifiers. They are reserved to represent the two possible boolean values. We explain later, in more detail, what constitutes an acceptable literal for each data type.

2.3.6 Operators and Punctuation

In addition to keywords, identifiers, and literals, a Java program contains operators and separators or punctuation. The operators are things like "+", and the separators are things like the ";" that terminates some statements and the braces "{ }" used to group things. To fully understand operators, you need to understand type, precedence, and associativity. *Type* determines the kind of value computed, such as `int` or `double`. *Precedence* determines among operators, such as + and / used in an expression, which is done first. *Associativity* is the order in which operators of the same precedence are evaluated and is usually left-most first—for example,

```
int n = 1;
n = n * 3 + 2;      //an assignment expression
```

The variable n is an integer variable initialized to 1. Next n is multiplied by the integer literal 3. This result is added to 2, and finally this value is assigned to n. The result is 5, which is then assigned to n. Precedence of the multiplication operator * is higher than +, so the multiplication is done before the addition. Precedence of the assignment operator = is lowest, so assignment occurs as the last action in this expression. We discuss precedence and associativity of operators further in Section 2.13 Precedence and Associativity of Operators.

2.4 Data Types and Variable Declarations

In order to do something useful, computer programs must store and manipulate data. Many programming languages, including Java, require that each data item have a declared type. That is, you must specify the kind of information represented by the data, or the data's *type*. The data's type determines how data is represented in the computer's memory and what operations can be performed on the data.

Different programming languages support different data types. A data type can be as fundamental as a type for representing integers or as complex as a type for representing a digital movie. Some examples of data types found in Java are shown below

JAVA DATA TYPES

- `int`—for representing integers or whole numbers;
- `double`—for representing numbers having a fraction;
- `String`—for representing text;
- `Button`—for representing a push button in a graphical user interface; and
- `Point`—for representing points in a plane.

The types of data that are created, stored, and manipulated by Java programs can be separated into two main groups: *primitive types* and class types, or simply *classes*. There are eight primitive types:

C++ PRIMITIVE TYPES

- the numeric types `byte`, `short`, `int`, `long`, `float`, and `double` for storing numeric values;
- the character type `char` for storing a single alphabetic character, digit, or symbol; and
- the type `boolean` for storing `true` or `false`.

Primitive data values can be created by using literals such as `100`, `-10.456`, `'a'`, and `true`. Primitive values can also be operated on by using built-in operators such as + for

addition and – for subtraction of two numeric values, producing a new primitive value. For example,

```
2 + 3
```

uses + (addition) to operate on the two numeric literal values, 2 and 3, to produce the new primitive value, 5.

Standard Java has more than 1500 classes. The `String`, `Button`, and `Point` types mentioned previously are standard Java classes. You will learn in Chapter 6, Objects: Data Abstraction, that you can create your own classes. Also, in Chapter 5, Arrays, we discuss arrays, which are a special case of class types.

The data values that are class types are called *objects*. You can create object data values by using the special operator `new` followed by the class name and possibly some additional values needed to create the new object. For example,

```
new Button("Quit")
```

creates a new object describing a button with the label "Quit."

You can create new objects of type `String`, as a special case, by using the string literal notation that surrounds the text of the string with double quotation marks. For example, `"Hello, world!"` creates a new string object.

In most cases, the operations supported for the particular class are given a name and invoked by placing the name after the object value, separated by a dot. For example,

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

operates on the literal string `"Hello, world!"` and evaluates to 13—the number of characters in this string, including any blanks. Operations such as `length()` defined for a particular class are called *methods.* We discuss methods in great detail in subsequent chapters.

2.4.1 Variables

In all but the most trivial programs, such as `HelloWorld`, you will declare *variables* that are identifiers used to refer to data values that are stored in the computer's memory. These are called variables because a variable actually refers to a particular place in the computer's memory, and the value stored in the computer's memory can vary as the program runs. A variable declaration always begins with a type and ends with a semicolon. The *type* is used to identify the kind of data that will be stored in the memory location associated with the variable being declared. Some examples of variable declarations are

```
int i, j;
String sentence;
boolean flag1, flag2, flag3;
Button clickToExit;
```

Note that you can declare several variables of the same type by separating the names with a comma. Good choice of variable names is important to writing clearly under-

standable code. Stylistically, choose variable names that are meaningful. Also, variable names usually start in lowercase, and if they are multiword, the internal words start with an uppercase character, as in `clickToExit`.

2.4.2 Variable Initialization

Variables can be given initial values. The preceding set of declarations given initial values for each variable becomes

```
int i = 2, j = 3;
String sentence = "I am a camera.";
boolean flag1 = true, flag2 = true, flag3 = false;
Button clickToExit = new Button("Exit");
```

Initializing variables with literals of their respective type is normal. In this example, the `int` variable `i` is initially given the value 2. The `boolean` variable `flag1` is initially `true`. The `String` variable `sentence` is initialized with a string literal.

With the exception of `String`, Java has no literals for creating object values. You initialize the `Button` variable by creating a `Button` object, using `new` as discussed briefly earlier. We discuss object creation in Chapter 6, Objects: Data Abstraction.

2.5 An Example: String Concatenation

The following example is a complete program that declares three variables: `word1`, `word2`, and `sentence`. Values are then assigned to the parts of the computer memory referred to by those variables.

```
// HelloWorld2.java - simple variable declarations
class HelloWorld2 {
  public static void main (String[] args) {
    String word1;   // declare a String variable
    String word2, sentence;  // declare two more

    word1 = "Hello, ";
    word2 = "world!";
    sentence = word1.concat(word2);
    System.out.println(sentence);
  }
}
```

DISSECTION OF THE HelloWorld2 PROGRAM

❑
```
String word1;   // declare a String variable
String word2, sentence;  // declare two more
```

Whenever you introduce a new identifier you must declare it. You declare that an identifier is a variable by first writing the name of the kind of value the variable refers to, called the *type*. The type is `String` in this example. Insert the name of the new variable after the type. For variables in Java, the computer must always know the type of value to be stored in the memory location associated with that variable. As shown in the second line, you can declare more than one variable at a time by giving first the type and then a comma separated list of new identifiers.

❑
```
word1 = "Hello, ";
word2 = "world!";
```

The symbol = is called the *assignment operator* and is used to store values in variables. Read the first statement as "word1 *gets* the value `Hello, `" or "word1 is *assigned* the value `Hello, `". Here it is used to assign the `String` literals `"Hello, "` and `"world!"` to the newly declared variables `word1` and `word2`, respectively. The variable name will always be on the left and the new value to be assigned to the variable will always be on the right. Saying "assign the value `"Hello, "` to the variable `word1`" really means to store the `String` value `"Hello, "` in the computer memory location associated with the variable `word1`.

❑
```
sentence = word1.concat(word2);
```

This statement contains an expression used to create a third `String` value, which is then assigned to the variable `sentence`. This expression uses the method `concat()`, which is defined for values of type `String`. Recall that operations on objects are called methods. This particular operation requires a second `String` value placed between the parentheses. The method name `concat` is short for concatenation. The concatenation of two strings is a new string that contains the symbols from the first string followed by the symbols from the second string. Note that the first string, `word1`, contains a space at the end. Thus, when we concatenate the two strings, we get the new string `"Hello, world!"`, which is assigned to the variable `sentence`.

❑
```
System.out.println(sentence);
```

The program then prints out the string in the variable `sentence`, which now is `"Hello, world!"`.

As we showed earlier, when a variable is declared, it can be given an initial value. This approach essentially combines an assignment with the declaration. Using this notation, you can now write the body of `main()` in `HelloWorld2` as

```
String word1 = "Hello, ";
String word2 = "world!";
String sentence = word1.concat(word2);
System.out.println(sentence);
```

You could even combine two initializations in a single statement,

```
String word2 = "world!", sentence = word1.concat(word2);
```

although, as a general rule, multiple complex initializations such as this should be placed on separate lines.

2.5.1 Strings Versus Identifiers Versus Variables

A *string* is a particular data value that a program can manipulate. A *variable* is a place in the computer's memory with an associated *identifier*. The following example uses the identifier `hello` to refer to a variable of type `String`. The identifier `stringVary` refers to a second variable of type `String`. We first assign `stringVary` the value associated with the `String` variable `hello`. Later we reassign it the string value `"hello"`.

```
// StringVsId.java - contrast strings and identifiers
class StringVsId {
  public static void main(String[] args) {
    String hello = "Hello, world!";
    String stringVary;
    stringVary = hello;
    System.out.println(stringVary);
    stringVary = "hello";
    System.out.println(stringVary);
  }
}
```

The output of this program is

```
Hello, world!
hello
```

The program demonstrates two important points. First it shows the difference between the identifier `hello`, which in fact refers to the string `"Hello, world!"`, and the string `"hello"`, which is referred to at one point by the variable `stringVary`. This example also shows that a variable can *vary*. The variable `stringVary` first refers to the value `"Hello, world!"` but later refers to the value `"hello"`.

2.6 User Input

In the programs presented so far, we have generated output only by using `System.out.println()`. Most programs input some data as well as generate output. There are lots of ways to input data in Java, but the simplest is to use a class provided in the text input-output package `tio`, as shown in the following example. This package isn't a standard Java package but is provided for use with this text. The complete source for `tio` is listed in Appendix C, The Text I/O Package tio.

```
// SimpleInput.java - reading numbers from the keyboard
import tio.*;    // use the package tio

class SimpleInput {
  public static void main (String[] args) {
    int width, height, area;

    System.out.println("type two integers for" +
              " the width and height of a box");
    width = Console.in.readInt();
    height = Console.in.readInt();
    area = width * height;
    System.out.print("The area is ");
    System.out.println(area);
  }
}
```

Dissection of the SimpleInput Program

❑ `import tio.*;      // use the package tio`

This line tells the Java compiler that the program `SimpleInput` uses some of the classes defined in the package `tio`. The `*` indicates that you might use any of the classes in the package `tio`. The class used here is `Console`, allowing you to write `Console.in.readInt()`, which we explain shortly. Because only `Console` is used from `tio`, you could also write `import tio.Console;`, but common practice is to import the entire package.

❑ `int width, height, area;`

This program declares three integer variables. The width and height must be whole numbers; fractional values are not allowed.

❑
```
System.out.println("type two integers for" +
                   " the width and height of a box");
```

A string literal must be on one line. You can't type a newline between the quotation marks. When the string that you want to print is too long to fit on a single line, you can break it into pieces and then put them back together, using the + for string concatenation, as shown here. You can put newlines, or any amount of white space, around the + symbol. Use `println()` to *prompt* a user to do something—here to type two numbers. Whenever a program is expecting the user to do something, it should print out a prompt telling the user what to do.

❑
```
width = Console.in.readInt();
height = Console.in.readInt();
```

The statement `Console.in` is similar to `System.out`. For now just memorize `Console.in.readInt()` as an idiom for reading an integer from the keyboard. The expression `Console.in.readInt()` is replaced at execution time by the integer value corresponding to that typed by the user. That value is stored in the variable whose identifier appears on the left of the assignment operator. If the user types something other than a string of digits, then the program will print an error message and terminate.

❑
```
area = width * height;
```

The asterisk, `*`, is the Java symbol for multiplication. The result of multiplying the values stored in `width` and `height` is assigned to the variable `area`.

❑
```
System.out.print("The area is ");
```

Here `System.out.print()` is used. Note the absence of `ln` at the end. This line is just like `System.out.println()` except that it doesn't append a newline.

❑
```
System.out.println(area);
```

Note that you can use `System.out.println()` to print both an `int` and a `String`. The same applies to `System.out.print()`. The two outputs will appear on the same line on the screen. Be sure that the last line in a series contains the `ln`. Nothing will appear on your screen until a newline is printed.

2.7 Calling Predefined Methods

A *method* is a group of instructions having a name. In the programs introduced so far, we've defined a single method called `main()`. In addition, we've used some methods from other classes that were either standard Java classes or were part of the `tio` package provided with this book. The methods have names, which makes it possible for you to request the computer to perform the instructions that comprise the method. That's what we're doing by using the expression `System.out. println("Hello, world!")`. We're *calling* a method with the name `println` and asking that the instructions be executed. Just as two people can have the same name, two methods can have the same name. We must therefore, in general, tell Java where to look for the method. Using `System.out` tells Java that we're interested in the `println()` method associated with the object identified by `System.out`. (We must still postpone a full explanation of the meaning of `System.out`.) As we have shown, the same method can be called several times in one program. In `SimpleInput`, we called the method `println()` twice.

For many methods, we need to provide some data values in order for the method to perform its job. These values are provided to the method by placing them between parentheses following the name of the method. These values are called the *parameters* of the method, and we say that we are *passing* the parameters to the method. The `println()` method requires one parameter, which is the value to be printed. As we indicated previously, this parameter can be either a string or a numeric value. In the latter case the numeric value will be converted by the `println()` method to a string and then printed. If more than one parameter is required, we separate the parameter values by commas. For example, the predefined method `Math.min()` is used to determine the minimum of two numbers. The following program fragment will print out `3`.

```
int numberOne = 10, numberTwo = 3, smallest;
smallest = Math.min(numberOne, numberTwo);
System.out.println(smallest);
```

The method `min()` is contained in the standard Java class `Math`, which includes other common mathematical functions, such as `Math.sqrt()` that is used to find the square root of a number.

Here are some of the predefined methods that we've mentioned so far.

```
System.out.print(x)          //print the value of x
System.out.println(x)        //print the value of x
                             //followed by a newline
Console.in.readInt()         //get an int from the keyboard
Math.min(x,y)                //find the smaller of x and y
Math.sqrt(x)                 //find the square root of x
w1.concat(w2)                //concatenate the strings w1 and w2
word.length()                //find the length of the string word
```

Java includes a rich set of many more predefined methods. They include methods for the most common mathematical functions (see Section B.2 The Standard Java Math Functions), for creating graphical user interfaces (see Chapter 8, Graphical User Inter-

faces: Part I and Chapter 9, Graphical User Interfaces: Part II), for reading and writing information from and to files (see Chapter 10, Reading and Writing Files), for communicating with programs on other computers using a network (see Chapter 13, Concurrent Programming with Java Threads), and many more. An important aspect of Java is the ability to use parts of programs created by others.

2.8 More on print() and println()

As the program SimpleInput shows, methods System.out.print() and System.out.println() can print more than just strings. The print() and println() methods can print all the primitive types.

We also showed in previous examples how two or more strings can be combined by using the string concatenation operator +. Using this operator, we can print out a long line of text without having to have a long line in our program, which would make the program hard to read. We demonstrated this capability in

```
System.out.println("type two integers for" +
                   " the width and height of a box");
```

Of course we could have achieved the same result by using a print() and then a println(), as in

```
System.out.print("type two integers for");
System.out.println(" the width and height of a box");
```

The earlier version would have allowed us to include even more actual text in the message to be printed. Recall, however, that you can't put a newline in the middle of a string, so the following would not be legal.

```
System.out.println("type two integers for
                   the width and height of a box");
```

In the same way that you can combine two strings with the string concatenation operator you can also combine one string and one value of any other type. In this case, the nonstring operand is first converted into a new string and then the two strings are concatenated. This allows rewriting the printing of the result in SimpleInput from Section 2.6 User Input, in the form

```
System.out.print("The area is " + area);
```

This version emphasizes that one message will appear on a single line of the output. The int value area is first converted to a String and then combined with the other string, using string concatenation.

What about the opposite? What if you wanted to have an output message that spanned several output lines but with a single println()? You can do so by putting the

symbols \n in a string literal to represent internally newlines within the string. Such a string will print on more than one output line. Thus for

```
System.out.println("One\nword\nper\nline.");
```

The output is

```
One
word
per
line.
```

The pair of symbols \n, when used in a string literal, mean to put a newline at this point in the string. This *escape sequence* allows you to escape from the normal meaning of the symbols \ and n when used separately. You can find out more about escape sequences in Section 2.9.3 The char Type.

The method System.out.println() is a convenient variant of System.out.print(). You can always achieve the same effect by adding \n to the end of what is being printed. For example, the following two lines are equivalent.

```
System.out.println("Hello, world.");
System.out.print("Hello, world.\n");
```

Care must be taken when you're using string concatenation to combine several numbers. Sometimes, parentheses are necessary to be sure that the + is interpreted as string concatenation, not numeric addition. For

```
int x = 1, y = 2;
System.out.println("x + y = " + x + y);
```

the output is x + y = 12 because the string "x + y =" is first concatenated with the string "1", and then "2" is concatenated onto the end of the string "x + y = 1". To print x + y = 3 you should use parentheses first to force the addition of x and y as integers and then concatenate the string representation of the result with the initial string, as in

```
System.out.println("x + y = " + (x + y));
```

2.9 Number Types

There are two basic representations for numbers in most modern programming languages: integer representations and floating point representations. The integer types are used to represent integers or whole numbers. The floating point types are used to

represent numbers that contain fractional parts or for numbers whose magnitude exceeds the capacity of the integer types.

2.9.1 The Integer Types

Some mechanism is needed to represent positive and negative numbers. One simple solution would be to set aside one bit to represent the sign of the number. However, this results in two different representations of zero, +0, and –0, which causes problems for computer arithmetic. Therefore an alternative system called two's complement is used.

An 8-bit value called a *byte* can represent 256 different values. Java supports five integral numeric types. The type `char`, although normally used to represent symbolic characters in a 16-bit format called Unicode, can be interpreted as an integer. As discussed shortly, the result of combining a `char` value with another numeric value will never be of type `char`. The `char` value is first converted to one of the other numeric types. These types are summarized in the following table.

Type	Number of Bits	Range of Values
`byte`	8	–128 to 127
`short`	16	–32768 to 32767
`char`	16	0 to 65536
`int`	32	–2147483648 to 2147483647
`long`	64	–9223372036854775808 to 9223372036854775807

The asymmetry in the most negative value and the most positive value of the integer types results from the two's complement system used in Java to represent negative values.

Literal integer values are represented by a sequence of digits. An integer literal is either an `int` or `long`. An explicit conversion, called a *cast*, of an `int` literal must be used to assign them to the smaller integer types `short` and `byte` (see Section 2.10.2 Type Conversion). Integer literals can be expressed in base 10 (decimal), base 8 (octal), and base 16 (hexadecimal) number systems. Decimal literals begin with a digit `1–9`, octal literals begin with the digit `0`, and hexadecimal literals begin with the two-character sequence `0x`. To specify a `long` literal instead of an `int` literal, append the letter *el*, uppercase or lowercase, to the sequence of digits.

Here are some examples:

Literal	Evaluation
`217`	the decimal value two hundred seventeen
`0217`	an octal number equivalent to 143 in the decimal system ($(2 \times 8^2 + 1 \times 8^1 + 7)$)
`0195`	would be illegal because 9 is not a valid octal digit (only 0–7 are octal digits)
`0x217`	a hexadecimal number equivalent to 535 in the decimal system ($2 \times 16^2 + 1 \times 16^1 + 7$); the hexadecimal digits include 0–9 plus $a = 10$, $b = 11$, $c = 12$, $d = 13$, $e = 14$, and $f = 15$.
`14084591234L`	a `long` decimal literal; without the `L` this would be an error because `14084591234` is too large to store in the 32 bits of an `int` type integer.

2.9.2 The Floating Point Types

Java supports two floating point numeric types. A floating point number consists of three parts: a sign, a magnitude, and an exponent. These two types are summarized in the following table.

Type	Number of Bits	Approximate Range of Values	Approximate Precision
`float`	32	$+/-10^{245}$ to $+/-10^{+38}$	7 decimal digits
`double`	64	$+/-10^{2324}$ to $+/-10^{+308}$	15 decimal digits

To represent floating point literals, you can simply insert a decimal point into a sequence of digits, as in `3.14159`. If the magnitude of the number is too large or too small to represent in this fashion, then a notation analogous to scientific notation is used. The letter `e` from *exponent* is used to indicate the exponent of 10 as shown in the following examples.

Java Representation	Value
`2.17e-27`	2.17×10^{-27}
`2.17e99`	2.17×10^{99}

Unless specified otherwise, a floating point literal is of type `double`. To specify a floating point literal of type `float`, append either `f` or `F` to the literal—for example, `2.17e-27f`.

2.9.3 The char Type

Java provides `char` variables to represent and manipulate characters. This type is an integer type and can be mixed with the other integer types. Each `char` is stored in memory in 2 bytes. This size is large enough to store the integer values 0 through 65535 as distinct character codes or nonnegative integers, and these codes are called Unicode. Unicode uses more storage per character than previous common character encodings because it was designed to represent all the world's alphabets, not just one particular alphabet.

For English, a subset of these values represents actual printing characters. These include the lowercase and uppercase letters, digits, punctuation, and special characters such as % and +. The character set also includes the white space characters blank, tab, and newline. This important subset is represented by the first 128 codes, which are also known as the ASCII codes. Earlier languages, such as C and C++, worked only with this more limited set of codes and stored them in 1 byte.

The following table illustrates the correspondence between some character literals and integer values. Character literals are written by placing a single character between single quotes, as in `'a'`.

Some Character Constants and Their Corresponding Integer Values

`'a'  'b'  'c' ... 'z'`	`97  98  99 ...  112`
`'A'  'B'  'C' ... 'Z'`	`65  66  67 ...  90`
`'0' '1' '2' ... '9'`	`48  49  50 ... 57`
`'&'  '*'  '+'`	`38  42  43`

There is no particular relationship between the value of the character constant representing a digit and the digit's intrinsic integer value. That is, the value of `'2'` is *not* 2. The property that the values for `'a'`, `'b'`, `'c'`, and so on occur in order is important. It makes the sorting of characters, words, and lines into lexicographical order convenient.

Note that character literals are different from string literals, which use double quotes, as in `"Hello"`. String literals can be only one character long, but they are still `String` values, not `char` values. For example, `"a"` is a string literal.

Some nonprinting and hard-to-print characters require an escape sequence. The horizontal tab character, for example, is written as `\t` in character constants and in strings. Even though it is being described by the two characters \ and `t`, it represents a single character. The backslash character \ is called the *escape character* and is used to escape the usual meaning of the character that follows it. Another way to write a character constant is by means of a hexadecimal-digit escape sequence, as in `'\u0007'`. This is the alert character, or the audible bell. These 4 hexadecimal digits are prefixed by the letter `u` to indicate their use as a Unicode literal. The 65,536 Unicode characters can be written in hexadecimal form from `'\u0000'` to `'\uFFFF'`.

The following table contains some nonprinting and hard-to-print characters.

Name of Character	Escape	int	hex
Backslash	\\	92	\u005C
Backspace	\b	8	\0008
Carriage return	\r	13	\u000D
Double quote	\"	34	\u0022
Formfeed	\f	12	\u000C
Horizontal tab	\t	9	\u0009
Newline	\n	10	\u000A
Single quote	\'	39	\u0027
Null character		0	\u0000
Alert		7	\u0007

The alert character is special; it causes the bell to ring. To hear the bell, try executing a program that contains the line

```
print('\u0007');
```

Character values are small integers, and, conversely, small integer values can be characters. Consider the declaration

```
char   c = 'a';
```

The variable `c` can be printed either as a character or as an integer:

```
print(c);          // a is printed
print((int)c);     // 97 is printed
```

2.9.4 Numbers Versus Strings

The sequence of 0s and 1s inside a computer used to store the `String` value `"1234"` is different from the sequence of 0s and 1s used to store the `int` value `1234`, which is different from the sequence of 0s and 1s used to store the `double` value `1234.0`. The string form is more convenient for some types of manipulation, such as outputing to screen or combining with other strings. The `int` form is better for some numeric calculations, and `double` is better for others. How does the computer know how to interpret a particular sequence of 0s and 1s? The answer is: Look at the type of the variable used to store the value. This answer is precisely why you must specify a type for each variable. Without the type information the sequence of 0s and 1s could be misinterpreted.

We do something similar with words all the time. Just as computers interpret sequences of 0s and 1s, human beings interpret sequences of alphabetic symbols. How people interpret those symbols depends on the context in which the symbols appear.

For example, at times the same word can be a noun, and at other times it can be a verb. Also, certain sequences of letters mean different things in different languages. Take the word *pie* for example. What does it mean? If it is English, it is something good to eat. If it is Spanish, it means foot.

From the context of the surrounding words we can usually figure out what the type of the word is: verb or noun, English or Spanish. Some programming languages do something similar and "figure out" the type of a variable. These programming languages are generally considered to be more error prone than languages such as Java, which require the programmer to specify the type of each variable. Languages such as Java are called *strongly typed* languages.

2.10 Arithmetic Expressions

The basic arithmetic operators in Java are addition +, subtraction –, multiplication *, division /, and modulus %. You can use all arithmetic operators with all primitive numeric types: `char`, `byte`, `short`, `int`, `long`, `float`, and `double`. In addition, you can combine any two numeric types by using these operators in what is known as *mixed mode arithmetic*. Although you can use the operators with any numeric type, Java actually does arithmetic only with the types `int`, `long`, `float`, and `double`. Therefore the following rules are used first to convert both operands into one of four types.

TYPE CONVERSIONS

1. If either operand is a `double`, then the other is converted to `double`.
2. Otherwise, if either operand is a `float`, then the other is converted to `float`.
3. Otherwise, if either operand is a `long`, then the other is converted to a `long`.
4. Otherwise, both are converted to `int`.

This conversion is called *binary numeric promotion* and is also used with the binary relational operators discussed in Section 3.2.1 Relational and Equality Operators.

When both operands are integer types, the operations of addition, subtraction, and multiplication are self-evident except when the result is too large to be represented by the type of the operands.

Integer values can't represent fractions. In Java, integer division truncates toward 0. For example, `6 / 4` is `1` and `6 / (-4)` is `-1`. A common mistake is to forget that integer division of nonzero values can result in 0. To obtain fractional results you must force one of the operands to be a floating point type. In expressions involving literals you can do so by adding a decimal point, as in `6.0 / 4`, which results in the floating point value `1.5`. In addition, integer division by zero will result in an error called an `ArithmeticException`. An *exception*, as the name implies, is something unexpected. Java provides a way for you to tell the computer what to do when exceptions occur. If

you don't do anything and such an error occurs, the program will print an appropriate error message and terminate. We discuss exceptions in Chapter 11, Exceptions.

Unlike some programming languages, Java doesn't generate an exception when integer arithmetic results in a value that is too large. Instead, the extra bits of the true result are lost, and in some cases pollute the bit used for the sign. For example, adding two very large positive numbers could generate a negative result. Likewise, subtracting a very large positive number from a negative number could generate a positive result. If values are expected to be near the limit for a particular type, you should either use a larger type or check the result to determine whether such an overflow has occurred.

When one of the operands is a floating point type, but both operands are not of the same type, one of them is converted to the other as described earlier. Unlike some programming languages, floating point arithmetic operations in Java will never generate an exception. Instead three special values can result: positive infinity, negative infinity, and "Not a Number." See Section A.1.2 Floating Point Representations for more details.

The modulus operator % returns the remainder from integer division. For example, `16 % 3` is 1, because 16 divided by 3 is 5 with a remainder of 1. The modulus operator is mostly used with integer operands; however, in Java it can be used with floating point operands. For floating point values `x % y` is `n`, where `n` is the largest integer such that `y * n` is less than or equal to `x`.

2.10.1 An Integer Arithmetic Example: MakeChange.java

The computation in Section 1.2 Algorithms—Being Precise whereby we made change for a dollar, is a perfect illustration of the use of the two integer division operators, / and %.

```
// MakeChange.java - change in dimes and pennies
import tio.*;     // use the package tio

class MakeChange {
  public static void main (String[] args) {
    int price, change, dimes, pennies;

    System.out.println("type price (0:100):");
    price = Console.in.readInt();
    change = 100 - price;         //how much change
    dimes = change / 10;          //number of dimes
    pennies = change % 10;        //number of pennies
    System.out.print("The change is : ");
    System.out.println(dimes + " dimes " + pennies + " pennies");
  }
}
```

DISSECTION OF THE MakeChange PROGRAM

❑ `int price, change, dimes, pennies;`

The program declares four integer variables. The type determines the range of values that can be used with the variables. They also dictate that the program use integer arithmetic operations, not floating point operations.

- `price = Console.in.readInt();`

  The `Console.in.readInt()` is used to obtain the input from the keyboard. The `readInt()` method is found in the `tio` package. At this point we must type in an integer price. For example, we would type 77 and hit Enter.

- `change = 100 - price; //how much change`

  This line computes the amount of change. This is the integer subtraction operator.

- ```
  dimes = change / 10;          //number of dimes
  pennies = change % 10;        //number of pennies
  ```

  To compute the number of dimes, we compute the integer result of dividing change by 10 and throw away any remainder. So if `change` is 23, then the integer result of 23/10 is 2. The remainder 3 is discarded. The number of pennies is the integer remainder of `change` divided by 10. The `%` operator is the integer remainder or modulo operator in Java. So if `change` is 23, then `23 % 10` is 3. In Exercise 18, we ask you to use `double` for the variables and to report on your results.

2.10.2 Type Conversion

You may want or need to convert from one primitive numeric type to another. As mentioned in the preceding section, Java will sometimes automatically convert the operands of a numeric operator. These automatic conversions are also called *widening primitive conversions* and always convert to a type that requires at least as many bits as the type being converted, hence the term *widening*. In most but not all cases, a widening primitive conversion doesn't result in loss of information. An example that can lose some precision is the conversion of the `int` value 123456789 to a `float` value, which results in 123456792. To understand this loss of information, see Section A.1.2 Floating Point Representations. The following are the possible widening primitive conversions:

From	To
byte	short, int, long, float, or double
short	int, long, float, or double
char	int, long, float, or double
int	long, float, or double
long	float or double
float	double
  ```

In addition to performing widening conversions automatically as part of mixed mode arithmetic, widening primitive conversions are also used to convert automatically the right-hand side of an assignment operator to the type of the variable on the left. For example, the following assignment will automatically convert the integer result to a floating point value.

```
int x = 1, y = 2;
float z;
z = x + y;    // automatic widening from int to float
```

A *narrowing primitive conversion* is a conversion between primitive numeric types that may result in significant information loss. The following are narrowing primitive conversions.

From	To
byte	char
short	byte or char
char	byte or short
int	byte, short, or char
long	byte, short, char, or int
float	byte, short, char, int, or long
double	byte, short, char, int, long, or float

Narrowing primitive conversions generally result only from an explicit type conversion called a *cast*. A cast is written as (*type*) *expression*, where the expression to be converted is preceded by the new type in parentheses. A cast is an operator and, as the table in Section 2.13 Precedence and Associativity of Operators, indicates, has higher precedence than the five basic arithmetic operators. For example, if you are interested only in the integer portion of the floating point variable `someFloat`, then you can store it in `someInteger`, as in

```
int someInteger;
float someFloat = 3.14159;
someInteger = (int)someFloat;
System.out.println(someInteger);
```

The output is

```
3
```

If the cast is between two integral types, the most significant bits are simply discarded in order to fit the resulting format. This discarding can cause the result to have a

different sign from the original value. The following example shows how a narrowing conversion can cause a change of sign.

```
int i = 127, j = 128;
byte iAsByte = (byte)i, jAsByte = (byte)j;
System.out.println(iAsByte);
System.out.println(jAsByte);
```

The output is

```
127

-128
```

The largest positive value that can be stored in a `byte` is 127. Attempting to force a narrowing conversion on a value greater than 127 will result in the loss of significant information. In this case the sign is reversed. To understand exactly what happens in this example, see Section A.1 Integer Representation.

COMMON PROGRAMMING ERROR

Remember that integer division truncates toward zero. For example, the value of the expression 3/4 is 0.m Both the numerator and the denominator are integer literals, so this is an integer division. If what you want is the rounded result, you must first force this to be a floating point division and then use the routine `Math.round()` to round the floating point result to an integer. To force a floating point division you can either make one of the literals a floating point literal or use a cast. In the following example first recall that floating point literals of type `float` are specified by appending the letter `f`. Then for

```
int x = Math.round(3.0f/4);
```

the variable `x` will get the value 1. Forcing the division to be a floating point divide is not enough. In the following example, `z` will be 0.

```
int z = (int)3.0f/4;
```

The conversion of 0.75 to an `int` truncates any fractional part.

2.11 Assignment Operators

To change the value of a variable, we have already made use of assignment statements such as

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

Assignment is an operator, and its precedence is lower than all the operators we've discussed so far. The associativity for the assignment operator is right to left. In this section we explain in detail its significance.

To understand = as an operator, let's first consider + for the sake of comparison. The binary operator + takes two operands, as in the expression `a + b`. The value of the expression is the sum of the values of `a` and `b`. By comparison, a simple assignment expression is of the form

variable = *rightHandSide*

where *rightHandSide* is itself an expression. A semicolon placed at the end would make this an assignment statement. The assignment operator = has the two operands *variable* and *rightHandSide*. The value of *rightHandSide* is assigned to *variable,* and that value becomes the value of the assignment expression as a whole. To illustrate, let's consider the statements

```
b = 2;
c = 3;
a = b + c;
```

where the variables are all of type `int`. By making use of assignment expressions, we can condense these statements to

```
a = (b = 2) + (c = 3);
```

The assignment expression `b = 2` assigns the value `2` to the variable `b`, and the assignment expression itself takes on this value. Similarly, the assignment expression `c = 3` assigns the value `3` to the variable `c`, and the assignment expression itself takes on this value. Finally, the values of the two assignment expressions are added, and the resulting value is assigned to `a`.

Although this example is artificial, in many situations assignment occurs naturally as part of an expression. A frequently occurring situation is *multiple assignment*. Consider the statement

```
a = b = c = 0;
```

Because the operator = associates from right to left, an equivalent statement is

```
a = (b = (c = 0));
```

First, `c` is assigned the value `0`, and the expression `c = 0` has value `0`. Then `b` is assigned the value `0`, and the expression `b = (c = 0)` has value `0`. Finally, `a` is assigned the value `0`, and the expression `a = (b = (c = 0))` has value `0`.

In addition to =, there are other assignment operators, such as += and -=. An expression such as

```
k = k + 2
```

will add 2 to the old value of `k` and assign the result to `k`, and the expression as a whole will have that value. The expression

```
k += 2
```

accomplishes the same task. The following list contains all the assignment operators.

Assignment Operators						
=	+=	-=	*=	/		
=	%=	>>=	<<=	&=	^=	|=

All these operators have the same precedence, and all have right-to-left associativity. The meaning is specified by

variable *op*= *expression*

which is equivalent to

variable = *variable* *op* (*expression*)

with the exception that if *variable* is itself an expression, it is evaluated only once. Note carefully that an assignment expression such as

`j *= k + 3` is equivalent to `j = j * (k + 3)`

rather than

```
j = j * k + 3
```

The following table illustrates how assignment expressions are evaluated.

Declarations and Initializations			
`int i = 1, j = 2, k = 3, m = 4;`			
Expression	**Equivalent Expression**	**Equivalent Expression**	**Value**
`i += j + k`	`i += (j + k)`	`i = (i + (j + k))`	6
`j *= k = m + 5`	`j *= (k = (m + 5))`	`j = (j * (k = (m + 5)))`	18

2.12 The Increment and Decrement Operators

Computers are very good at counting. As a result, many programs involve having an integer variable that takes on the values 0, 1, 2, . . . One way to add 1 to a variable is

```
i = i + 1;
```

which changes the value stored in the variable `i` to be 1 more than it was before this statement was executed. This procedure is called *incrementing* a variable. Because it is so common, Java, like its predecessor C, includes a shorthand notation for incrementing a variable. The following statement gives the identical result.

```
i++;
```

The operator ++ is known as the increment operator. Similarly, there is a decrement operator, --, so that the following two statements are equivalent:

```
i = i - 1;
i--;
```

Here is a simple program that demonstrates the increment operator.

```
// Increment.java - demonstrate incrementing
class Increment {
  public static void main(String[] args) {
    int i = 0;
    System.out.println("i = " + i);
    i = i + 1;
    System.out.println("i = " + i);
    i++;
    System.out.println("i = " + i);
    i++;
    System.out.println("i = " + i);
  }
}
```

The output of this program is

```
i = 0
i = 1
i = 2
```

Note that both increment and decrement operators are placed after the variable to be incremented. When placed after its argument they are called the *postfix* increment and *postfix* decrement operators. These operators also can be used before the variable. They are then called the *prefix* increment and *prefix* decrement operators. Each of the expressions `++i` *(prefix)* and `i++` *(postfix)* has a value; moreover, each causes the stored

value of `i` in memory to be incremented by 1. The expression `++i` causes the stored value of `i` to be incremented first, with the expression then taking as its value the new stored value of `i`. In contrast, the expression `i++` has as its value the current value of `i`; then the expression causes the stored value of `i` to be incremented. The following code illustrates the situation.

```
int   a, b, c = 0;
a = ++c;
b = c++;
System.out.println("a = " + a);   //a = 1 is printed
System.out.println("b = " + b);   //b = 1 is printed
System.out.println("c = " + ++c); //c = 3 is printed
```

Similarly, `--i` causes the stored value of `i` in memory to be decremented by 1 first, with the expression then taking this new stored value as its value. With `i--` the value of the expression is the current value of `i`; then the expression causes the stored value of `i` in memory to be decremented by 1. Note that `++` and `--` cause the value of a variable in memory to be changed. Other operators do not do so. For example, an expression such as `a + b` leaves the values of the variables `a` and `b` unchanged. These ideas are expressed by saying that the operators `++` and `--` have a *side effect*; not only do these operators yield a value, but they also change the stored value of a variable in memory.

In some cases we can use `++` in either prefix or postfix position, with both uses producing equivalent results. For example, each of the two statements

```
++i;   and   i++;
```

is equivalent to

```
i = i + 1;
```

In simple situations you can consider `++` and `--` as operators that provide concise notation for the incrementing and decrementing of a variable. In other situations, you must pay careful attention as to whether prefix or postfix position is used.

2.13 Precedence and Associativity of Operators

When evaluating expressions with several operators, you need to understand the order of evaluation of each operator and its arguments. *Operator precedence* gives a hierarchy that helps determine the order in which operations are evaluated. For example, precedence determines which arithmetic operations are evaluated first in

```
x = -b + Math.sqrt(b * b - 4 * a * c)/(2 * a)
```

which you may recognize as the expression to compute one of the roots of a quadratic equation, written like this in your mathematics class:

$$x = -b + \frac{\sqrt{b^2 - 4ac}}{2a}$$

To take a simpler example, what is the value of integer variable `x` after the assignment `x = 7 + 5 * 3`? The answer is `22`, not $(7 + 5) \times 3$ which is 36. The reason is that multiplication has *higher precedence* than addition; therefore `5 * 3` is evaluated before `7` is added to the result.

In addition to some operators having higher precedence than others, Java specifies the *associativity* or the order in which operators of equal precedence are to be evaluated. For example, the value of `100 / 5 * 2` is 40, not 10. This is because `/` and `*` have equal precedence and arithmetic operators of equal precedence are evaluated from left to right. If you wanted to do the multiplication before the division, you would write the expression as `100 / (5 * 2)`. Parentheses can be used to override the normal operator precedence rules.

This left to right ordering is important in some cases that might not appear obvious. Consider the expression `x + y + z`. From simple algebra, the associativity of addition tells us that `(x + y) + z` is the same as `x + (y + z)`. Unfortunately, this is true only when you have numbers with infinite precision. Suppose that `y` is the largest positive integer that Java can represent, `x` is `-100`, and `z` is `50`. Evaluating `(y + z)` first will result in integer overflow, which in Java will be equivalent to some very large negative number, clearly not the expected result. If instead, `(x + y)` is evaluated first, then adding `z` will result in an integer that is still in range and the result will be the correct value.

The precedence and associativity of all Java operators is given in Section B.1 Operator Precedence Table. The following table gives the rules of precedence and associativity for the operators of Java that we have used so far.

Operator Precedence and Associativity

Operator	Associativity
`( )` `++` (postfix) `--` (postfix)	Left to right
`+` (unary) `-` (unary) `++` (prefix) `--` (prefix)	Right to left
`new` (*type*)*expr*	Right to left
`*` `/` `%`	Left to right
`+` `-`	Left to right
`=` `+=` `-=` `*=` `/=` *etc.*	Right to left

All the operators on the same line, such as `*`, `/`, and `%`, have equal precedence with respect to each other but have higher precedence than all the operators that occur on the lines below them. The associativity rule for all the operators on each line appears in the right-hand column.

In addition to the binary `+`, which represents addition, there is a unary `+`, and both operators are represented by a plus sign. The minus sign also has binary and unary

meanings. The following table gives some additional examples of precedence and associativity of operators.

Declarations and Initializations		
`int a = 1, b = 2, c = 3, d = 4;`		
Expression	**Equivalent Expression**	**Value**
`a * b / c`	`(a * b) / c`	`0`
`a * b % c + 1`	`((a * b) % c) + 1`	`3`
`++a * b - c--`	`((++a) * b) - (c --)`	`1`
`7 - - b * ++d`	`7 - ((- b) * (++d))`	`17`

2.14 Programming Style

A clear, consistent style is important to writing good code. We use a style that is largely adapted from the Java professional programming community. Having a style that is readily understandable by the rest of the programming community is important.

We've already mentioned the importance of comments for documenting a program. Anything that aids in explaining what is otherwise not clear in the program should be placed in a comment. Comments help the programmer keep track of decisions made while writing the code. Without good documentation, you may return to some code you have written, only to discover that you have forgotten why you did some particular thing. The documentation should enable someone other than the original programmer to pick up, use, and modify the code. All but the most trivial methods should have comments at the beginning, clearly stating the purpose of the method. Also, complicated blocks of statements should be preceded by comments summarizing the function of the block. Comments should add to the clarity of the code, not simply restate the program, statement by statement. Here is an example of a useless comment.

```
area = width * height;      // compute the area
```

Good documentation includes proper choice of identifiers. Identifiers should have meaningful names. Certain simple one-character names are used to indicate auxiliary variables, such as `i`, `j`, or `k`, as integer variables.

The code should be easy to read. Visibility is enhanced by the use of white space. In general, we present only one statement to a line and in all expressions separate operators from arguments by a space. As we progress to more complex programs, we shall present, by example or explicit mention, accepted layout rules for program elements.

NAMING CONVENTIONS USED BY MANY JAVA PROGRAMMERS

- Class names start with uppercase and embedded words, as in `HelloWorld`, are capitalized.
- Methods and variables start with lowercase and embedded words, as in `readInt`, `data`, `toString`, and `loopIndex`, are capitalized.
- Although legal, the dollar sign, `$`, should not be used except in machine-generated Java programs.

SUMMARY

- To create a Java program first define a class. Give the class a method called `main()`. Put whatever instructions you want the computer to execute inside the body of the method `main()`.
- A program stores data in variables. Each variable is given a type, such as `int` for storing integers or `String` for storing strings.
- You can use literals to embed constant values of various types in a program, such as in the constant string `"Hello, world!"` or the integer constant `123`.
- You can combine literals and variables in expressions by using operators such as + for addition or string concatenation and * for numeric multiplication.
- You can store the result of evaluating an expression in a variable by using the assignment operator =. The variable is always on the left, and the expression being assigned to the variable is always on the right.
- You can call a method from another class by writing the name of the class followed by a dot and the name of the method—for example, `Math.sqrt()`.
- The lexical elements of a Java program are keywords, identifiers, literals, operator symbols, punctuation, comments, and white space.
- You can print strings and numbers to the screen by using the method `System.out.print()` or `System.out.println()`. The latter appends a newline to whatever is printed. These methods are part of the standard Java classes.
- You can input integers (whole numbers) from the keyboard by using the method `Console.in.readInt()`. This method isn't a standard Java class but is provided in the package `tio`. The full text of the package appears in Appendix C, The Text I/O Package tio.
- Java supports the primitive integer types `char`, `byte`, `short`, `int`, and `long`. It also supports two floating point types, `float` and `double`.
- Integer division truncates toward zero—it doesn't round to the nearest whole number. You can use `Math.round()` if rounding is what you want.

Review Questions

1. What line appears in every complete Java program indicating where to begin executing the program?
2. What one-line instruction would you use to have a Java program print `Goodbye`?
3. What affect do strings such as `/* what is this */` have on the execution of a Java program?
4. What is a variable?
5. What is a method?
6. Complete the following table.

Text	Legal ID	Why or Why Not
`3xyz`	No	Digit is first character.
`xy3z`		
`a = b`		
`main`		
`Count`		
`class`		

7. What does the symbol = do in Java?
8. Programmers say _____ a method when they mean go and execute the instructions for the method.
9. True or false? A multiline comment can be placed anywhere white space could be placed.
10. True or false? Keywords can also be used as variables, but then the special meaning of the keyword is overridden.
11. What convention for identifiers given in this chapter is used in `whatAmI`, `howAboutThis`, `someName`?
12. What primitive types are used to store whole numbers?
13. What is the difference between `x = 'a'` and `x = a`?
14. What is the difference between `s = "hello"` and `s = hello`?
15. Which version of the Java program `HelloWorld` is the one you can view and edit with a text editor, *HelloWorld.java* or *HelloWorld.class*? What does the other one contain? What program created the one you do not edit?
16. What is Unicode?
17. List the primitive types in Java.

18. Before it can be used, every variable must be declared and given a _____.
19. What is the value of the Java expression `"10"+"20"`? Don't ignore the quotation marks; they are crucial.
20. Write a Java statement that could be used to read an integer value from the keyboard and store it in the variable `someNumber`.
21. What is wrong with the following Java statement?

```
System.out.println("This statement is supposed
     to print a message. What is wrong?");
```

22. Every group of input statements should be preceded by what?
23. How do you write `x` times `y` in Java?
24. What is the difference between `System.out.print("message")` and `System.out.println("message")`?
25. Write a *single* Java statement that will produce the following output.

```
X
XX
XXX
```

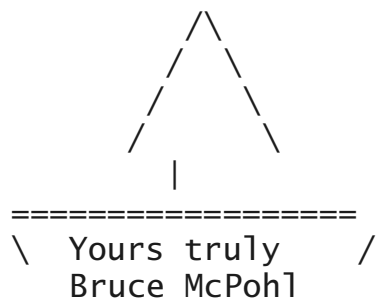
26. Approximately, what is the largest value that can be stored in the primitive type `int`? One thousand? One million? One billion? One trillion? Even larger?
27. What primitive Java type can store the largest numbers?
28. What is the value of the following Java expressions?

```
20 / 40
6 / 4
6.4 / 2
```

EXERCISES

1. Write a Java program that prints "Hello *your name*." You can do this by a simple modification to the `HelloWorld` program. Compile and run this program on your computer.
2. Write a Java program that prints a favorite poem of at least eight lines. Be sure to print it out neatly aligned. At the end of the poem, print two blank lines and then the author's name.

3. Design your own signature logo, such as a sailboat icon if you like sailing, and print it followed by "yours truly—*your name.*" A sailboat signature logo might look like



4. Write a Java program to read in two numbers and print the sum. Be sure to include a message to prompt the user for input and a message identifying the output. See what happens if you type in something that is not a number when the program is run. See how large a number you can type in and still have the program work correctly.
5. The following code contains three syntax errors and produces two syntax error messages from *javac*. Fix the problems.

```
// Ch2e1.java - fixing syntax errors
Class Ch2e1 {
  public static void main(String[] args) {
    System.out.println(hello, world);
  }
```

The *javac* compiler's message reads:

```
Ch2e1.java:2: Class or interface
declaration expected.
 Class ch2e1 {
 ^
Ch2e1.java:7: Unbalanced parentheses.
^
2 errors
```

6. The following code produces one syntax error message from *javac*. Fix the problem.

```
// Ch2e2.java - more syntax errors
class Ch2e2 {
  public static void main(String[] args) {
    int count = 1, i = 3,
    System.out.println("count + i = ",  count + i);
  }
}
```

The *javac* compiler's message reads

```
Ch2e2.java:6: Invalid declaration.
System.out.println("count + i = ", count + i);
                   ^
1 error
```

Here, unlike the previous exercise, the compiler doesn't as clearly point to the errors. Frequently, errors in punctuation lead to syntax error messages that are hard to decipher. After you fix the first syntax error in the code, a second error will be identified.

7. Continue with the code class `Ch2e2`. If you fixed just the syntax errors, you may get a running program that still has a run-time bug. Namely, the output is not the sum of `count + i`. Fixing run-time or semantic bugs is harder than fixing syntax bugs because something is wrong with your understanding of how to program the solution. Without introducing any other variables fix the run-time bug.

8. Write a program that draws a box like the one shown, using a *single* `println()` statement.

```
**********
*        *
*        *
**********
```

9. Use `Console.in.readDouble()` to read in one double precision floating point number and then print the results of calling `Math.sin()`, `Math.cos()`, `Math.asin()`, `Math.exp()`, `Math.log()`, `Math.floor()`, and `Math.round()` with the input value as a parameter. Be sure to prompt the user for input and label the output.

10. Write a program to read two double precision floating point numbers, using `Console.in.readDouble()`. Print the sum, difference, product, and quotient of the two numbers. Try two very small numbers, two very large numbers, and one very small number with one very large number. You can use the same notation used for literals to enter the numbers. For example, `0.123e-310` is a very small number.

11. Write a program to compute the area of a circle given its radius. Let `radius` be a variable of type `double` and use `Console.in.readDouble()` to read in its value. Be sure that the output is understandable. The Java class `Math` contains definitions for the constants `E` and `PI`, so you can use `Math.PI` in your program.

12. Extend the previous program to write out the circumference of a circle and the volume of a sphere given the radius as input. Recall that the volume of a sphere is

$$V = \frac{4 \times \pi r^3}{3}$$

13. Write a program that asks for a `double` and then prints it out. Then ask for a second `double`, this time printing out the sum and average of the two `doubles`. Then ask for a third double and again print out the accumulated sum and the average of the three doubles. Use variables `data1`, `data2`, `data3`, and `sum`. Later, when we

discuss loops, you will see how this is easily done for an arbitrary number of input values.

14. Write a program that reads in an integer and prints it as a character. Remember that character codes can be nonprinting.
15. Write a program using `Console.in.readChar()` to read in a character and print its integer value. This `tio` method returns an integer value, so it need not be cast from a `char` to an `int`. By the way, the reason is that the end-of-file character translates to 21. This method allows you to detect when you have reached the end of the input when reading from the keyboard or from a disk file (see Section 10.7 Detecting the End of an Input Stream).
16. Write a program that asks for the number of quarters, dimes, nickels, and pennies you have. Then compute the total value of your change and print the number of dollars and the remaining cents. The preferred output form would be `$X.YY`, but this is surprisingly difficult in Java and requires techniques not yet introduced. To get a handle on the problem, try storing $2.50 as a `float` then print it. Then try storing $2.05 as two numbers, one for the dollars and one for the remaining cents. Try to print these two numbers in the preferred format.
17. Write a program capable of converting one currency to another. For example, given U.S. dollars it should print out the equivalent number of French francs. Look up the exchange rate and use it as input.
18. Change the `MakeChange` program to use variables that are `doubles`. Run the program and see what goes wrong.

```
class MakeChange {
  public static void main (String[] args) {
    double price, change, dimes, pennies;
    ...
```

19. The following is a C program for printing “Hello, world!”.

```
/* Hello World In C
 * Purpose:
 *   The classic "Hello, world!" program.
 *   It simply prints a message to the screen.
 * Author:
 *    Jane Programmer
 *       as derived from Kernighan and Richie
 */

#include <stdio.h>     /* needed for IO */
int main(void) {
   printf("Hello, world!\n");
   return 0;           /* unneeded in Java */
}
```

Note how similar this program is to the Java version. A key difference is the lack of class encapsulation of `main()`. As in Java, `main()` starts the program’s execution. In C, methods are known as functions. The `printf()` function is found in the standard input–output library imported by the C compiler for use in this program. The

`return 0` ends program execution and is not used in Java. Convert the following C program to Java.

```
/* yada.c */
#include <stdio.h>
int main(void) {
   printf("Hello, world!\n");
   printf("My name is George.\n");
   printf("Yada Yada Yada ...\n");
   return 0;
}
```

20. In C, the `printf()` function can also be used to print primitive values, such as integer values and floating point values as in the following program.

```
/* cube.c */
#include <stdio.h>
int main(void) {
  double side = 3.5;    /* side of a cube */
  double volume;

  printf("The side of my cube is %f feet.\n", side);
  volume = side * side * side;
  printf("The cubes volume is %f cubic feet.\n",
              volume);
  return 0;
}
```

If you have a C compiler, such as *gcc,* compile and run this program. Then write the equivalent program in Java. The `%f` is a format control that tells the `printf()` function where to place the value of the corresponding variable, such as `side` or `volume`, in the program. Getting this format wrong causes many programming errors in C. This is one of the places that Java, a type-safe language, provides better support than C.

Applet Exercise

The following program is an example of a Java applet. This program uses several features of Java that we explain later. Note that there is no method `main()`; instead there is the method `paint()`. For now just concentrate on the body of the method `paint()`, treating the surrounding code as a template to be copied verbatim. By invoking the appropriate drawing operations on the `Graphics` object `g`, you can draw on the applet.

```
/* To place this applet in a web page, add the
   following two lines to the html document for the
   page.
  <applet code="FirstApplet.class"
  width=500 height=200></applet>
*/

// FirstApplet.java
// AWT and Swing together comprise the collection of
// classes used for building graphical Java programs
import java.awt.*; //required for programs that draw
import javax.swing.*; //required for Swing applets

public class FirstApplet extends JApplet {
  public void paint(Graphics g) {
    // draw a line from the upper left corner to
    // 100 pixels below the top center of the Applet
    g.drawLine(0,0,250,100);
    // draw a line from the end of the previous line
    // up to the top center of the Applet
    g.drawLine(250,100,250,0);
    // draw an oval inscribed in an invisible
    // rectangle with its upper left corner at the
    // intersection of the two lines drawn above
    g.drawOval(250,100,200,100);
  }
}
```

The class `Graphics` is used for simple drawing, and many drawing operations are defined for it. In this example we use the method `drawLine()` to draw a line. The first two numbers in parentheses for the `drawline()` operation are the *xy* coordinates of one end of the line, and the last two numbers are the *xy* coordinates of the other end. As you can see from the output of the program, the location (0, 0) is in the upper left corner, with increasing *x* moving to the right and increasing *y* moving down. To draw an oval, give the coordinates of the upper left corner and the width and height on an invisible rectangle. The oval will be inscribed inside the rectangle.

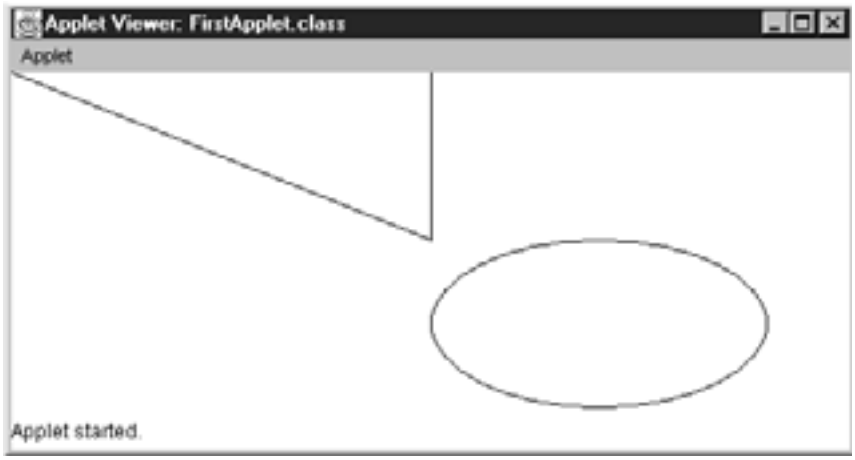
To execute an applet, first compile it like you do other Java programs. Then you can either run the program *appletviewer* or use a web browser to view the applet. To view the applet `FirstApplet` using the *appletviewer* on Unix and Windows machines, type the following at a command line prompt.

```
appletviewer FirstApplet.java
```

Notice that we are passing `FirstApplet.java`—not `FirstApplet` or `FirstApplet.class`—to the appletviewer. This procedure is different from running regular Java programs. In fact `appletviewer` just looks in the text file passed to it for an applet element. An applet element begins with `<applet` and ends with `</applet>`. Any text file containing the applet element shown in the opening comment for `FirstApplet.java` would work just as well.

To view the applet in a Web browser, create a file—for example, `FirstApplet.html`. Put the applet tag in the html file. Put the html file in the same directory as your applet and then open the html file with a Web browser.

The applet looks like the following when run with an `appletviewer`.



Modify this applet to draw a simple picture. Look up the documentation for `Graphics` on the Web at http://java.sun.com/products/jdk/1.2/docs/api/java.awt.Graphics.html#_top_and use at least one method/operation of `Graphics` not used in `FirstApplet`.