Chapter 2

Programming in the Small I: Names and Things

On a basic level (the level of machine language), a computer can perform only very simple operations. A computer performs complex tasks by stringing together large numbers of such operations. Such tasks must be "scripted" in complete and perfect detail by programs. Creating complex programs will never be really easy, but the difficulty can be handled to some extent by giving the program a clear overall ***structure***. The design of the overall structure of a program is what I call "programming in the large."

Programming in the small, which is sometimes called ***coding***, would then refer to filling in the details of that design. The details are the explicit, step-by-step instructions for performing fairly small-scale tasks. When you do coding, you are working fairly "close to the machine," with some of the same concepts that you might use in machine language: memory locations, arithmetic operations, loops and branches. In a high-level language such as Java, you get to work with these concepts on a level several steps above machine language. However, you still have to worry about getting all the details exactly right.

This chapter and the next examine the facilities for programming in the small in the Java programming language. Don't be misled by the term "programming in the small" into thinking that this material is easy or unimportant. This material is an essential foundation for all types of programming. If you don't understand it, you can't write programs, no matter how good you get at designing their large-scale structure.

2.1 The Basic Java Application

A program is a sequence of instructions that a computer can execute to perform some task. A simple enough idea, but for the computer to make any use of the instructions, they must be written in a form that the computer can use. This means that programs have to be written in ***programming languages***. Programming languages differ from ordinary human languages in being completely unambiguous and very strict about what is and is not allowed in a program. The rules that determine what is allowed are called the ***syntax*** of the language. Syntax rules specify the basic vocabulary of the language and how programs can be constructed using things like loops, branches, and subroutines. A syntactically correct program is one that can be successfully compiled or interpreted; programs that have syntax errors will be rejected (hopefully with a useful error message that will help you fix the problem).

So, to be a successful programmer, you have to develop a detailed knowledge of the syntax

of the programming language that you are using. However, syntax is only part of the story. It's not enough to write a program that will run—you want a program that will run and produce the correct result! That is, the **meaning** of the program has to be right. The meaning of a program is referred to as its ***semantics***. A semantically correct program is one that does what you want it to.

Furthermore, a program can be syntactically and semantically correct but still be a pretty bad program. Using the language correctly is not the same as using it **well**. For example, a good program has "style." It is written in a way that will make it easy for people to read and to understand. It follows conventions that will be familiar to other programmers. And it has an overall design that will make sense to human readers. The computer is completely oblivious to such things, but to a human reader, they are paramount. These aspects of programming are sometimes referred to as ***pragmatics.***

When I introduce a new language feature, I will explain the syntax, the semantics, and some of the pragmatics of that feature. You should memorize the syntax; that's the easy part. Then you should get a feeling for the semantics by following the examples given, making sure that you understand how they work, and maybe writing short programs of your own to test your understanding. And you should try to appreciate and absorb the pragmatics—this means learning how to use the language feature *well*, with style that will earn you the admiration of other programmers.

Of course, even when you've become familiar with all the individual features of the language, that doesn't make you a programmer. You still have to learn how to construct complex programs to solve particular problems. For that, you'll need both experience and taste. You'll find hints about software development throughout this textbook.

* * *

We begin our exploration of Java with the problem that has become traditional for such beginnings: to write a program that displays the message "Hello World!". This might seem like a trivial problem, but getting a computer to do this is really a big first step in learning a new programming language (especially if it's your first programming language). It means that you understand the basic process of:

1. getting the program text into the computer,

2. compiling the program, and

3. running the compiled program.

The first time through, each of these steps will probably take you a few tries to get right. I won't go into the details here of how you do each of these steps; it depends on the particular computer and Java programming environment that you are using. See Section 2.6 for information about creating and running Java programs in specific programming environments. But in general, you will type the program using some sort of text editor and save the program in a file. Then, you will use some command to try to compile the file. You'll either get a message that the program contains syntax errors, or you'll get a compiled version of the program. In the case of Java, the program is compiled into Java bytecode, not into machine language. Finally, you can run the compiled program by giving some appropriate command. For Java, you will actually use an interpreter to execute the Java bytecode. Your programming environment might automate some of the steps for you, but you can be sure that the same three steps are being done in the background.

Here is a Java program to display the message "Hello World!". Don't expect to understand what's going on here just yet—some of it you won't really understand until a few chapters from

now:

```
// A program to display the message
// "Hello World!" on standard output

public class HelloWorld {

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

}   // end of class HelloWorld
```

The command that actually displays the message is:

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

This command is an example of a ***subroutine call statement***. It uses a "built-in subroutine" named `System.out.println` to do the actual work. Recall that a subroutine consists of the instructions for performing some task, chunked together and given a name. That name can be used to "call" the subroutine whenever that task needs to be performed. A ***built-in subroutine*** is one that is already defined as part of the language and therefore automatically available for use in any program.

When you run this program, the message "Hello World!" (without the quotes) will be displayed on standard output. Unfortunately, I can't say exactly what that means! Java is meant to run on many different platforms, and standard output will mean different things on different platforms. However, you can expect the message to show up in some convenient place. (If you use a command-line interface, like that in Sun Microsystem's Java Development Kit, you type in a command to tell the computer to run the program. The computer will type the output from the program, Hello World!, on the next line.)

You must be curious about all the other stuff in the above program. Part of it consists of ***comments***. Comments in a program are entirely ignored by the computer; they are there for human readers only. This doesn't mean that they are unimportant. Programs are meant to be read by people as well as by computers, and without comments, a program can be very difficult to understand. Java has two types of comments. The first type, used in the above program, begins with // and extends to the end of a line. The computer ignores the // and everything that follows it on the same line. Java has another style of comment that can extend over many lines. That type of comment begins with /* and ends with */.

Everything else in the program is required by the rules of Java syntax. All programming in Java is done inside "classes." The first line in the above program (not counting the comments) says that this is a class named *HelloWorld*. "HelloWorld," the name of the class, also serves as the name of the program. Not every class is a program. In order to define a program, a class must include a subroutine named `main`, with a definition that takes the form:

```
public static void main(String[] args) {
      ⟨statements⟩
}
```

When you tell the Java interpreter to run the program, the interpreter calls the `main()` subroutine, and the statements that it contains are executed. These statements make up the script that tells the computer exactly what to do when the program is executed. The `main()` routine can call subroutines that are defined in the same class or even in other classes, but it is the `main()` routine that determines how and in what order the other subroutines are used.

The word "public" in the first line of `main()` means that this routine can be called from outside the program. This is essential because the `main()` routine is called by the Java interpreter, which is something external to the program itself. The remainder of the first line of the routine is harder to explain at the moment; for now, just think of it as part of the required syntax. The definition of the subroutine—that is, the instructions that say what it does—consists of the sequence of "statements" enclosed between braces, `{` and `}`. Here, I've used ⟨*statements*⟩ as a placeholder for the actual statements that make up the program. Throughout this textbook, I will always use a similar format: anything that you see in ⟨*this style of text*⟩ (italic in angle brackets) is a placeholder that describes something you need to type when you write an actual program.

As noted above, a subroutine can't exist by itself. It has to be part of a "class". A program is defined by a public class that takes the form:

```
public class ⟨program-name⟩ {

    ⟨optional-variable-declarations-and-subroutines⟩

    public static void main(String[] args) {
       ⟨statements⟩
    }

    ⟨optional-variable-declarations-and-subroutines⟩

}
```

The name on the first line is the name of the program, as well as the name of the class. If the name of the class is HelloWorld, then the class must be saved in a file called `HelloWorld.java`. When this file is compiled, another file named `HelloWorld.class` will be produced. This class file, `HelloWorld.class`, contains the Java bytecode that is executed by a Java interpreter. `HelloWorld.java` is called the ***source code*** for the program. To execute the program, you only need the compiled `class` file, not the source code.

The layout of the program on the page, such as the use of blank lines and indentation, is not part of the syntax or semantics of the language. The computer doesn't care about layout—you could run the entire program together on one line as far as it is concerned. However, layout is important to human readers, and there are certain style guidelines for layout that are followed by most programmers. These style guidelines are part of the pragmatics of the Java programming language.

Also note that according to the above syntax specification, a program can contain other subroutines besides `main()`, as well as things called "variable declarations." You'll learn more about these later, but not until Chapter 4.

2.2 Variables and the Primitive Types

NAMES ARE FUNDAMENTAL TO PROGRAMMING. In programs, names are used to refer to many different sorts of things. In order to use those things, a programmer must understand the rules for giving names to things and the rules for using the names to work with those things. That is, the programmer must understand the syntax and the semantics of names.

According to the syntax rules of Java, a name is a sequence of one or more characters. It must begin with a letter or underscore and must consist entirely of letters, digits, and underscores. ("Underscore" refers to the character '_'.) For example, here are some legal names:

```
N   n   rate  x15   quite_a_long_name   HelloWorld
```

No spaces are allowed in identifiers; `HelloWorld` is a legal identifier, but "Hello World" is not. Upper case and lower case letters are considered to be different, so that `HelloWorld`, `helloworld`, `HELLOWORLD`, and `hElloWorLD` are all distinct names. Certain names are reserved for special uses in Java, and cannot be used by the programmer for other purposes. These ***reserved words*** include: `class`, `public`, `static`, `if`, `else`, `while`, and several dozen other words.

Java is actually pretty liberal about what counts as a letter or a digit. Java uses the ***Unicode*** character set, which includes thousands of characters from many different languages and different alphabets, and many of these characters count as letters or digits. However, I will be sticking to what can be typed on a regular English keyboard.

The pragmatics of naming includes style guidelines about how to choose names for things. For example, it is customary for names of classes to begin with upper case letters, while names of variables and of subroutines begin with lower case letters; you can avoid a lot of confusion by following the same convention in your own programs. Most Java programmers do not use underscores in names, although some do use them at the beginning of the names of certain kinds of variables. When a name is made up of several words, such as `HelloWorld` or `interestRate`, it is customary to capitalize each word, except possibly the first; this is sometimes referred to as ***camel case***, since the upper case letters in the middle of a name are supposed to look something like the humps on a camel's back.

Finally, I'll note that things are often referred to by ***compound names*** which consist of several ordinary names separated by periods. (Compound names are also called ***qualified names***.) You've already seen an example: `System.out.println`. The idea here is that things in Java can contain other things. A compound name is a kind of path to an item through one or more levels of containment. The name `System.out.println` indicates that something called "System" contains something called "out" which in turn contains something called "println". Non-compound names are called ***simple identifiers***. I'll use the term ***identifier*** to refer to any name—simple or compound—that can be used to refer to something in Java. (Note that the reserved words are **not** identifiers, since they can't be used as names for things.)

2.2.1 Variables

Programs manipulate data that are stored in memory. In machine language, data can only be referred to by giving the numerical address of the location in memory where it is stored. In a high-level language such as Java, names are used instead of numbers to refer to data. It is the job of the computer to keep track of where in memory the data is actually stored; the programmer only has to remember the name. A name used in this way—to refer to data stored in memory—is called a ***variable***.

Variables are actually rather subtle. Properly speaking, a variable is not a name for the data itself but for a location in memory that can hold data. You should think of a variable as a container or box where you can store data that you will need to use later. The variable refers directly to the box and only indirectly to the data in the box. Since the data in the box can change, a variable can refer to different data values at different times during the execution of the program, but it always refers to the same box. Confusion can arise, especially for beginning programmers, because when a variable is used in a program in certain ways, it refers to the container, but when it is used in other ways, it refers to the data in the container. You'll see examples of both cases below.

(In this way, a variable is something like the title, "The President of the United States." This title can refer to different people at different times, but it always refers to the same office.

If I say "the President went fishing," I mean that George W. Bush went fishing. But if I say "Hillary Clinton wants to be President" I mean that she wants to fill the office, not that she wants to be George Bush.)

In Java, the **only** way to get data into a variable—that is, into the box that the variable names—is with an ***assignment statement***. An assignment statement takes the form:

⟨*variable*⟩ = ⟨*expression*⟩;

where ⟨*expression*⟩ represents anything that refers to or computes a data value. When the computer comes to an assignment statement in the course of executing a program, it evaluates the expression and puts the resulting data value into the variable. For example, consider the simple assignment statement

```
rate = 0.07;
```

The ⟨*variable*⟩ in this assignment statement is `rate`, and the ⟨*expression*⟩ is the number 0.07. The computer executes this assignment statement by putting the number 0.07 in the variable `rate`, replacing whatever was there before. Now, consider the following more complicated assignment statement, which might come later in the same program:

```
interest = rate * principal;
```

Here, the value of the expression "`rate * principal`" is being assigned to the variable `interest`. In the expression, the `*` is a "multiplication operator" that tells the computer to multiply `rate` times `principal`. The names `rate` and `principal` are themselves variables, and it is really the **values** stored in those variables that are to be multiplied. We see that when a variable is used in an expression, it is the value stored in the variable that matters; in this case, the variable seems to refer to the data in the box, rather than to the box itself. When the computer executes this assignment statement, it takes the **value** of `rate`, multiplies it by the **value** of `principal`, and stores the answer in the **box** referred to by `interest`. When a variable is used on the left-hand side of an assignment statement, it refers to the box that is named by the variable.

(Note, by the way, that an assignment statement is a command that is executed by the computer at a certain time. It is not a statement of fact. For example, suppose a program includes the statement "`rate = 0.07;`". If the statement "`interest = rate * principal;`" is executed later in the program, can we say that the `principal` is multiplied by 0.07? No! The value of `rate` might have been changed in the meantime by another statement. The meaning of an assignment statement is completely different from the meaning of an equation in mathematics, even though both use the symbol "=".)

2.2.2 Types and Literals

A variable in Java is designed to hold only one particular type of data; it can legally hold that type of data and no other. The compiler will consider it to be a syntax error if you try to violate this rule. We say that Java is a ***strongly typed*** language because it enforces this rule.

There are eight so-called ***primitive types*** built into Java. The primitive types are named **byte**, **short**, **int**, **long**, **float**, **double**, **char**, and **boolean**. The first four types hold integers (whole numbers such as 17, -38477, and 0). The four integer types are distinguished by the ranges of integers they can hold. The **float** and **double** types hold real numbers (such as 3.6 and -145.99). Again, the two real types are distinguished by their range and accuracy. A variable of type **char** holds a single character from the Unicode character set. And a variable of type **boolean** holds one of the two logical values `true` or `false`.

Any data value stored in the computer's memory must be represented as a binary number, that is as a string of zeros and ones. A single zero or one is called a ***bit***. A string of eight bits is called a ***byte***. Memory is usually measured in terms of bytes. Not surprisingly, the **byte** data type refers to a single byte of memory. A variable of type **byte** holds a string of eight bits, which can represent any of the integers between -128 and 127, inclusive. (There are 256 integers in that range; eight bits can represent 256—two raised to the power eight—different values.) As for the other integer types,

- **short** corresponds to two bytes (16 bits). Variables of type **short** have values in the range -32768 to 32767.
- **int** corresponds to four bytes (32 bits). Variables of type **int** have values in the range -2147483648 to 2147483647.
- **long** corresponds to eight bytes (64 bits). Variables of type **long** have values in the range -9223372036854775808 to 9223372036854775807.

You don't have to remember these numbers, but they do give you some idea of the size of integers that you can work with. Usually, you should just stick to the **int** data type, which is good enough for most purposes.

The **float** data type is represented in four bytes of memory, using a standard method for encoding real numbers. The maximum value for a **float** is about 10 raised to the power 38. A **float** can have about 7 significant digits. (So that 32.3989231134 and 32.3989234399 would both have to be rounded off to about 32.398923 in order to be stored in a variable of type **float**.) A **double** takes up 8 bytes, can range up to about 10 to the power 308, and has about 15 significant digits. Ordinarily, you should stick to the **double** type for real values.

A variable of type **char** occupies two bytes in memory. The value of a **char** variable is a single character such as A, *, x, or a space character. The value can also be a special character such a tab or a carriage return or one of the many Unicode characters that come from different languages. When a character is typed into a program, it must be surrounded by single quotes; for example: 'A', '*', or 'x'. Without the quotes, A would be an identifier and * would be a multiplication operator. The quotes are not part of the value and are not stored in the variable; they are just a convention for naming a particular character constant in a program.

A name for a constant value is called a ***literal***. A literal is what you have to type in a program to represent a value. 'A' and '*' are literals of type **char**, representing the character values A and *. Certain special characters have special literals that use a backslash, \, as an "escape character". In particular, a tab is represented as `'\t'`, a carriage return as `'\r'`, a linefeed as `'\n'`, the single quote character as `'\''`, and the backslash itself as `'\\'`. Note that even though you type two characters between the quotes in `'\t'`, the value represented by this literal is a single tab character.

Numeric literals are a little more complicated than you might expect. Of course, there are the obvious literals such as 317 and 17.42. But there are other possibilities for expressing numbers in a Java program. First of all, real numbers can be represented in an exponential form such as 1.3e12 or 12.3737e-108. The "e12" and "e-108" represent powers of 10, so that 1.3e12 means 1.3 times 10^{12} and 12.3737e-108 means 12.3737 times 10^{-108}. This format can be used to express very large and very small numbers. Any numerical literal that contains a decimal point or exponential is a literal of type **double**. To make a literal of type **float**, you have to append an "F" or "f" to the end of the number. For example, "1.2F" stands for 1.2 considered as a value of type **float**. (Occasionally, you need to know this because the rules of Java say that you can't assign a value of type **double** to a variable of type **float**, so you might be

confronted with a ridiculous-seeming error message if you try to do something like "`x = 1.2;`" when `x` is a variable of type **float**. You have to say "`x = 1.2F;`". This is one reason why I advise sticking to type **double** for real numbers.)

Even for integer literals, there are some complications. Ordinary integers such as 177777 and -32 are literals of type **byte**, **short**, or **int**, depending on their size. You can make a literal of type **long** by adding "L" as a suffix. For example: 17L or 728476874368L. As another complication, Java allows octal (base-8) and hexadecimal (base-16) literals. I don't want to cover base-8 and base-16 in detail, but in case you run into them in other people's programs, it's worth knowing a few things: Octal numbers use only the digits 0 through 7. In Java, a numeric literal that begins with a 0 is interpreted as an octal number; for example, the literal 045 represents the number 37, not the number 45. Hexadecimal numbers use 16 digits, the usual digits 0 through 9 and the letters A, B, C, D, E, and F. Upper case and lower case letters can be used interchangeably in this context. The letters represent the numbers 10 through 15. In Java, a hexadecimal literal begins with `0x` or `0X`, as in `0x45` or `0xFF7A`.

Hexadecimal numbers are also used in character literals to represent arbitrary Unicode characters. A Unicode literal consists of `\u` followed by four hexadecimal digits. For example, the character literal `'\u00E9'` represents the Unicode character that is an "e" with an acute accent.

For the type **boolean**, there are precisely two literals: `true` and `false`. These literals are typed just as I've written them here, without quotes, but they represent values, not variables. Boolean values occur most often as the values of conditional expressions. For example,

```
rate > 0.05
```

is a boolean-valued expression that evaluates to `true` if the value of the variable `rate` is greater than 0.05, and to `false` if the value of `rate` is not greater than 0.05. As you'll see in Chapter 3, boolean-valued expressions are used extensively in control structures. Of course, boolean values can also be assigned to variables of type **boolean**.

Java has other types in addition to the primitive types, but all the other types represent objects rather than "primitive" data values. For the most part, we are not concerned with objects for the time being. However, there is one predefined object type that is very important: the type *String*. A *String* is a sequence of characters. You've already seen a string literal: `"Hello World!"`. The double quotes are part of the literal; they have to be typed in the program. However, they are not part of the actual string value, which consists of just the characters between the quotes. Within a string, special characters can be represented using the backslash notation. Within this context, the double quote is itself a special character. For example, to represent the string **value**

```
I said, "Are you listening!"
```

with a linefeed at the end, you would have to type the string **literal**:

```
"I said, \"Are you listening!\"\n"
```

You can also use `\t`, `\r`, `\\`, and unicode sequences such as `\u00E9` to represent other special characters in string literals. Because strings are objects, their behavior in programs is peculiar in some respects (to someone who is not used to objects). I'll have more to say about them in the next section.

2.2.3 Variables in Programs

A variable can be used in a program only if it has first been ***declared***. A ***variable declaration statement*** is used to declare one or more variables and to give them names. When the computer executes a variable declaration, it sets aside memory for the variable and associates the variable's name with that memory. A simple variable declaration takes the form:

⟨*type-name*⟩ ⟨*variable-name-or-names*⟩;

The ⟨***variable-name-or-names***⟩ can be a single variable name or a list of variable names separated by commas. (We'll see later that variable declaration statements can actually be somewhat more complicated than this.) Good programming style is to declare only one variable in a declaration statement, unless the variables are closely related in some way. For example:

```
int numberOfStudents;
String name;
double x, y;
boolean isFinished;
char firstInitial, middleInitial, lastInitial;
```

It is also good style to include a comment with each variable declaration to explain its purpose in the program, or to give other information that might be useful to a human reader. For example:

```
double principal;    // Amount of money invested.
double interestRate; // Rate as a decimal, not percentage.
```

In this chapter, we will only use variables declared inside the `main()` subroutine of a program. Variables declared inside a subroutine are called ***local variables*** for that subroutine. They exist only inside the subroutine, while it is running, and are completely inaccessible from outside. Variable declarations can occur anywhere inside the subroutine, as long as each variable is declared before it is used in any expression. Some people like to declare all the variables at the beginning of the subroutine. Others like to wait to declare a variable until it is needed. My preference: Declare important variables at the beginning of the subroutine, and use a comment to explain the purpose of each variable. Declare "utility variables" which are not important to the overall logic of the subroutine at the point in the subroutine where they are first used. Here is a simple program using some variables and assignment statements:

```
/**
 * This class implements a simple program that
 * will compute the amount of interest that is
 * earned on $17,000 invested at an interest
 * rate of 0.07 for one year.  The interest and
 * the value of the investment after one year are
 * printed to standard output.
 */

public class Interest {

   public static void main(String[] args) {

       /* Declare the variables. */

       double principal;     // The value of the investment.
       double rate;          // The annual interest rate.
       double interest;      // Interest earned in one year.
```

```
        /* Do the computations. */

        principal = 17000;
        rate = 0.07;
        interest = principal * rate;   // Compute the interest.

        principal = principal + interest;
              // Compute value of investment after one year, with interest.
              // (Note: The new value replaces the old value of principal.)

        /* Output the results. */

        System.out.print("The interest earned is $");
        System.out.println(interest);
        System.out.print("The value of the investment after one year is $");
        System.out.println(principal);

    } // end of main()

} // end of class Interest
```

This program uses several subroutine call statements to display information to the user of the program. Two different subroutines are used: `System.out.print` and `System.out.println`. The difference between these is that `System.out.println` adds a linefeed after the end of the information that it displays, while `System.out.print` does not. Thus, the value of `interest`, which is displayed by the subroutine call "`System.out.println(interest);`", follows on the same line after the string displayed by the previous `System.out.print` statement. Note that the value to be displayed by `System.out.print` or `System.out.println` is provided in parentheses after the subroutine name. This value is called a ***parameter*** to the subroutine. A parameter provides a subroutine with information it needs to perform its task. In a subroutine call statement, any parameters are listed in parentheses after the subroutine name. Not all subroutines have parameters. If there are no parameters in a subroutine call statement, the subroutine name must be followed by an empty pair of parentheses.

All the sample programs for this textbook are available in separate source code files in the on-line version of this text at http://math.hws.edu/javanotes/source. They are also included in the downloadable archives of the web site. The source code for the `Interest` program, for example, can be found in the file *Interest.java*.

2.3 Strings, Objects, Enums, and Subroutines

The previous section introduced the eight primitive data types and the type *String*. There is a fundamental difference between the primitive types and the *String* type: Values of type *String* are objects. While we will not study objects in detail until Chapter 5, it will be useful for you to know a little about them and about a closely related topic: classes. This is not just because strings are useful but because objects and classes are essential to understanding another important programming concept, subroutines.

Another reason for considering classes and objects at this point is so that we can introduce ***enums***. An enum is a data type that can be created by a Java programmer to represent a small collection of possible values. Technically, an enum is a class and its possible values are objects. Enums will be our first example of adding a new type to the Java language. We will look at them later in this section.

2.3.1 Built-in Subroutines and Functions

Recall that a subroutine is a set of program instructions that have been chunked together and given a name. In Chapter 4, you'll learn how to write your own subroutines, but you can get a lot done in a program just by calling subroutines that have already been written for you. In Java, every subroutine is contained in a class or in an object. Some classes that are standard parts of the Java language contain predefined subroutines that you can use. A value of type *String*, which is an object, contains subroutines that can be used to manipulate that string. These subroutines are "built into" the Java language. You can call all these subroutines without understanding how they were written or how they work. Indeed, that's the whole point of subroutines: A subroutine is a "black box" which can be used without knowing what goes on inside.

Classes in Java have two very different functions. First of all, a class can group together variables and subroutines that are contained in that class. These variables and subroutines are called ***static members*** of the class. You've seen one example: In a class that defines a program, the `main()` routine is a static member of the class. The parts of a class definition that define static members are marked with the reserved word "`static`", just like the `main()` routine of a program. However, classes have a second function. They are used to describe objects. In this role, the class of an object specifies what subroutines and variables are contained in that object. The class is a **type**—in the technical sense of a specification of a certain type of data value—and the object is a value of that type. For example, *String* is actually the name of a class that is included as a standard part of the Java language. *String* is also a type, and literal strings such as `"Hello World"` represent values of type *String*.

So, every subroutine is contained either in a class or in an object. Classes **contain** subroutines called static member subroutines. Classes also **describe** objects and the subroutines that are contained in those objects.

This dual use can be confusing, and in practice most classes are designed to perform primarily or exclusively in only one of the two possible roles. For example, although the *String* class does contain a few rarely-used static member subroutines, it exists mainly to specify a large number of subroutines that are contained in objects of type *String*. Another standard class, named *Math*, exists entirely to group together a number of static member subroutines that compute various common mathematical functions.

* * *

To begin to get a handle on all of this complexity, let's look at the subroutine `System.out.print` as an example. As you have seen earlier in this chapter, this subroutine is used to display information to the user. For example, `System.out.print("Hello World")` displays the message, Hello World.

System is one of Java's standard classes. One of the static member variables in this class is named `out`. Since this variable is contained in the class *System*, its full name—which you have to use to refer to it in your programs—is `System.out`. The variable `System.out` refers to an object, and that object in turn contains a subroutine named `print`. The compound identifier `System.out.print` refers to the subroutine `print` in the object `out` in the class *System*.

(As an aside, I will note that the object referred to by `System.out` is an object of the class `PrintStream`. `PrintStream` is another class that is a standard part of Java. **Any** object of type `PrintStream` is a destination to which information can be printed; **any** object of type `PrintStream` has a `print` subroutine that can be used to send information to that destination. The object `System.out` is just one possible destination, and `System.out.print` is the subrou-

tine that sends information to that particular destination. Other objects of type `PrintStream` might send information to other destinations such as files or across a network to other computers. This is object-oriented programming: Many different things which have something in common—they can all be used as destinations for information—can all be used in the same way—through a `print` subroutine. The `PrintStream` class expresses the commonalities among all these objects.)

Since class names and variable names are used in similar ways, it might be hard to tell which is which. Remember that all the built-in, predefined names in Java follow the rule that class names begin with an upper case letter while variable names begin with a lower case letter. While this is not a formal syntax rule, I recommend that you follow it in your own programming. Subroutine names should also begin with lower case letters. There is no possibility of confusing a variable with a subroutine, since a subroutine name in a program is always followed by a left parenthesis.

(As one final general note, you should be aware that subroutines in Java are often referred to as ***methods***. Generally, the term "method" means a subroutine that is contained in a class or in an object. Since this is true of every subroutine in Java, every subroutine in Java is a method. The same is not true for other programming languages. Nevertheless, the term "method" is mostly used in the context of object-oriented programming, and until we start doing real object-oriented programming in Chapter 5, I will prefer to use the more general term, "subroutine.")

* * *

Classes can contain static member subroutines, as well as static member variables. For example, the *System* class contains a subroutine named `exit`. In a program, of course, this subroutine must be referred to as `System.exit`. Calling this subroutine will terminate the program. You could use it if you had some reason to terminate the program before the end of the `main` routine. For historical reasons, this subroutine takes an integer as a parameter, so the subroutine call statement might look like "`System.exit(0);`" or "`System.exit(1);`". (The parameter tells the computer why the program was terminated. A parameter value of 0 indicates that the program ended normally. Any other value indicates that the program was terminated because an error was detected. But in practice, the value of the parameter is usually ignored.)

Every subroutine performs some specific task. For some subroutines, that task is to compute or retrieve some data value. Subroutines of this type are called ***functions***. We say that a function ***returns*** a value. The returned value must then be used somehow in the program.

You are familiar with the mathematical function that computes the square root of a number. Java has a corresponding function called `Math.sqrt`. This function is a static member subroutine of the class named *Math*. If `x` is any numerical value, then `Math.sqrt(x)` computes and returns the square root of that value. Since `Math.sqrt(x)` represents a value, it doesn't make sense to put it on a line by itself in a subroutine call statement such as

```
Math.sqrt(x);   // This doesn't make sense!
```

What, after all, would the computer do with the value computed by the function in this case? You have to tell the computer to do something with the value. You might tell the computer to display it:

```
System.out.print( Math.sqrt(x) );  // Display the square root of x.
```

or you might use an assignment statement to tell the computer to store that value in a variable:

```
lengthOfSide = Math.sqrt(x);
```

The function call `Math.sqrt(x)` represents a value of type **double**, and it can be used anyplace where a numeric literal of type double could be used.

The *Math* class contains many static member functions. Here is a list of some of the more important of them:

- `Math.abs(x)`, which computes the absolute value of `x`.
- The usual trigonometric functions, `Math.sin(x)`, `Math.cos(x)`, and `Math.tan(x)`. (For all the trigonometric functions, angles are measured in radians, not degrees.)
- The inverse trigonometric functions arcsin, arccos, and arctan, which are written as: `Math.asin(x)`, `Math.acos(x)`, and `Math.atan(x)`. The return value is expressed in radians, not degrees.
- The exponential function `Math.exp(x)` for computing the number e raised to the power `x`, and the natural logarithm function `Math.log(x)` for computing the logarithm of `x` in the base e.
- `Math.pow(x,y)` for computing `x` raised to the power `y`.
- `Math.floor(x)`, which rounds `x` down to the nearest integer value that is less than or equal to `x`. Even though the return value is mathematically an integer, it is returned as a value of type **double**, rather than of type **int** as you might expect. For example, `Math.floor(3.76)` is 3.0. The function `Math.round(x)` returns the integer that is closest to `x`.
- `Math.random()`, which returns a randomly chosen **double** in the range `0.0 <= Math.random() < 1.0`. (The computer actually calculates so-called "pseudorandom" numbers, which are not truly random but are random enough for most purposes.)

For these functions, the type of the parameter—the `x` or `y` inside the parentheses—can be any value of any numeric type. For most of the functions, the value returned by the function is of type **double** no matter what the type of the parameter. However, for `Math.abs(x)`, the value returned will be the same type as `x`; if `x` is of type **int**, then so is `Math.abs(x)`. So, for example, while `Math.sqrt(9)` is the **double** value 3.0, `Math.abs(9)` is the **int** value 9.

Note that `Math.random()` does not have any parameter. You still need the parentheses, even though there's nothing between them. The parentheses let the computer know that this is a subroutine rather than a variable. Another example of a subroutine that has no parameters is the function `System.currentTimeMillis()`, from the *System* class. When this function is executed, it retrieves the current time, expressed as the number of milliseconds that have passed since a standardized base time (the start of the year 1970 in Greenwich Mean Time, if you care). One millisecond is one-thousandth of a second. The return value of `System.currentTimeMillis()` is of type **long**. This function can be used to measure the time that it takes the computer to perform a task. Just record the time at which the task is begun and the time at which it is finished and take the difference.

Here is a sample program that performs a few mathematical tasks and reports the time that it takes for the program to run. On some computers, the time reported might be zero, because it is too small to measure in milliseconds. Even if it's not zero, you can be sure that most of the time reported by the computer was spent doing output or working on tasks other than the program, since the calculations performed in this program occupy only a tiny fraction of a second of a computer's time.

```
/**
 * This program performs some mathematical computations and displays
 * the results.  It then reports the number of seconds that the
 * computer spent on this task.
 */

public class TimedComputation {

   public static void main(String[] args) {

      long startTime; // Starting time of program, in milliseconds.
      long endTime;   // Time when computations are done, in milliseconds.
      double time;    // Time difference, in seconds.

      startTime = System.currentTimeMillis();

      double width, height, hypotenuse;  // sides of a triangle
      width = 42.0;
      height = 17.0;
      hypotenuse = Math.sqrt( width*width + height*height );
      System.out.print("A triangle with sides 42 and 17 has hypotenuse ");
      System.out.println(hypotenuse);

      System.out.println("\nMathematically, sin(x)*sin(x) + "
                                     + "cos(x)*cos(x) - 1 should be 0.");
      System.out.println("Let's check this for x = 1:");
      System.out.print("      sin(1)*sin(1) + cos(1)*cos(1) - 1 is ");
      System.out.println( Math.sin(1)*Math.sin(1)
                                       + Math.cos(1)*Math.cos(1) - 1 );
      System.out.println("(There can be round-off errors when"
                                    + " computing with real numbers!)");

      System.out.print("\nHere is a random number:  ");
      System.out.println( Math.random() );

      endTime = System.currentTimeMillis();
      time = (endTime - startTime) / 1000.0;

      System.out.print("\nRun time in seconds was:  ");
      System.out.println(time);

   } // end main()

} // end class TimedComputation
```

2.3.2 Operations on Strings

A value of type *String* is an object. That object contains data, namely the sequence of characters that make up the string. It also contains subroutines. All of these subroutines are in fact functions. For example, every string object contains a function named `length` that computes the number of characters in that string. Suppose that `advice` is a variable that refers to a *String*. For example, `advice` might have been declared and assigned a value as follows:

```
String advice;
advice = "Seize the day!";
```

Then `advice.length()` is a function call that returns the number of characters in the string "Seize the day!". In this case, the return value would be 14. In general, for any string variable `str`, the value of `str.length()` is an **int** equal to the number of characters in the string that is the value of `str`. Note that this function has no parameter; the particular string whose length is being computed is the value of `str`. The `length` subroutine is defined by the class *String*, and it can be used with any value of type *String*. It can even be used with *String* literals, which are, after all, just constant values of type *String*. For example, you could have a program count the characters in "Hello World" for you by saying

```
System.out.print("The number of characters in ");
System.out.println("the string \"Hello World\" is ");
System.out.println( "Hello World".length() );
```

The *String* class defines a lot of functions. Here are some that you might find useful. Assume that `s1` and `s2` refer to values of type *String*:

- `s1.equals(s2)` is a function that returns a **boolean** value. It returns `true` if `s1` consists of exactly the same sequence of characters as `s2`, and returns `false` otherwise.
- `s1.equalsIgnoreCase(s2)` is another boolean-valued function that checks whether `s1` is the same string as `s2`, but this function considers upper and lower case letters to be equivalent. Thus, if `s1` is "cat", then `s1.equals("Cat")` is `false`, while `s1.equalsIgnoreCase("Cat")` is `true`.
- `s1.length()`, as mentioned above, is an integer-valued function that gives the number of characters in `s1`.
- `s1.charAt(N)`, where `N` is an integer, returns a value of type **char**. It returns the `N`-th character in the string. Positions are numbered starting with 0, so `s1.charAt(0)` is actually the first character, `s1.charAt(1)` is the second, and so on. The final position is `s1.length() - 1`. For example, the value of `"cat".charAt(1)` is 'a'. An error occurs if the value of the parameter is less than zero or greater than `s1.length() - 1`.
- `s1.substring(N,M)`, where `N` and `M` are integers, returns a value of type *String*. The returned value consists of the characters in `s1` in positions `N`, `N+1`,..., `M-1`. Note that the character in position `M` is not included. The returned value is called a substring of `s1`.
- `s1.indexOf(s2)` returns an integer. If `s2` occurs as a substring of `s1`, then the returned value is the starting position of that substring. Otherwise, the returned value is -1. You can also use `s1.indexOf(ch)` to search for a particular character, `ch`, in `s1`. To find the first occurrence of `x` at or after position `N`, you can use `s1.indexOf(x,N)`.
- `s1.compareTo(s2)` is an integer-valued function that compares the two strings. If the strings are equal, the value returned is zero. If `s1` is less than `s2`, the value returned is a number less than zero, and if `s1` is greater than `s2`, the value returned is some number greater than zero. (If both of the strings consist entirely of lower case letters, then "less than" and "greater than" refer to alphabetical order. Otherwise, the ordering is more complicated.)
- `s1.toUpperCase()` is a *String*-valued function that returns a new string that is equal to `s1`, except that any lower case letters in `s1` have been converted to upper case. For example, `"Cat".toUpperCase()` is the string `"CAT"`. There is also a function `s1.toLowerCase()`.
- `s1.trim()` is a *String*-valued function that returns a new string that is equal to `s1` except that any non-printing characters such as spaces and tabs have been trimmed from the

beginning and from the end of the string. Thus, if `s1` has the value `"fred "`, then `s1.trim()` is the string `"fred"`.

For the functions `s1.toUpperCase()`, `s1.toLowerCase()`, and `s1.trim()`, note that the value of `s1` is **not** modified. Instead a new string is created and returned as the value of the function. The returned value could be used, for example, in an assignment statement such as "`smallLetters = s1.toLowerCase();`". To change the value of `s1`, you could use an assignment "`s1 = s1.toLowerCase();`".

* * *

Here is another extremely useful fact about strings: You can use the plus operator, `+`, to ***concatenate*** two strings. The concatenation of two strings is a new string consisting of all the characters of the first string followed by all the characters of the second string. For example, `"Hello" + "World"` evaluates to `"HelloWorld"`. (Gotta watch those spaces, of course—if you want a space in the concatenated string, it has to be somewhere in the input data, as in `"Hello " + "World"`.)

Let's suppose that `name` is a variable of type *String* and that it already refers to the name of the person using the program. Then, the program could greet the user by executing the statement:

```
System.out.println("Hello, " + name + ".  Pleased to meet you!");
```

Even more surprising is that you can actually concatenate values of **any** type onto a *String* using the `+` operator. The value is converted to a string, just as it would be if you printed it to the standard output, and then it is concatenated onto the string. For example, the expression `"Number" + 42` evaluates to the string `"Number42"`. And the statements

```
System.out.print("After ");
System.out.print(years);
System.out.print(" years, the value is ");
System.out.print(principal);
```

can be replaced by the single statement:

```
System.out.print("After " + years +
                    " years, the value is " + principal);
```

Obviously, this is very convenient. It would have shortened some of the examples presented earlier in this chapter.

2.3.3 Introduction to Enums

Java comes with eight built-in primitive types and a large set of types that are defined by classes, such as *String*. But even this large collection of types is not sufficient to cover all the possible situations that a programmer might have to deal with. So, an essential part of Java, just like almost any other programming language, is the ability to create **new** types. For the most part, this is done by defining new classes; you will learn how to do that in Chapter 5. But we will look here at one particular case: the ability to define ***enums*** (short for ***enumerated types***). Enums are a recent addition to Java. They were only added in Version 5.0. Many programming languages have something similar, and many people believe that enums should have been part of Java from the beginning.

Technically, an enum is considered to be a special kind of class, but that is not important for now. In this section, we will look at enums in a simplified form. In practice, most uses of enums will only need the simplified form that is presented here.

An enum is a type that has a fixed list of possible values, which is specified when the enum is created. In some ways, an enum is similar to the **boolean** data type, which has `true` and `false` as its only possible values. However, **boolean** is a primitive type, while an enum is not.

The definition of an enum types has the (simplified) form:

```
enum ⟨enum-type-name⟩ { ⟨list-of-enum-values⟩ }
```

This definition cannot be inside a subroutine. You can place it **outside** the `main()` routine of the program. The ⟨*enum-type-name*⟩ can be any simple identifier. This identifier becomes the name of the enum type, in the same way that "boolean" is the name of the **boolean** type and "String" is the name of the *String* type. Each value in the ⟨*list-of-enum-values*⟩ must be a simple identifier, and the identifiers in the list are separated by commas. For example, here is the definition of an enum type named `Season` whose values are the names of the four seasons of the year:

```
enum Season { SPRING, SUMMER, FALL, WINTER }
```

By convention, enum values are given names that are made up of upper case letters, but that is a style guideline and not a syntax rule. Enum values are not variables. Each value is a ***constant*** that always has the same value. In fact, the possible values of an enum type are usually referred to as ***enum constants***.

Note that the enum constants of type `Season` are considered to be "contained in" `Season`, which means—following the convention that compound identifiers are used for things that are contained in other things—the names that you actually use in your program to refer to them are `Season.SPRING`, `Season.SUMMER`, `Season.FALL`, and `Season.WINTER`.

Once an enum type has been created, it can be used to declare variables in exactly the same ways that other types are used. For example, you can declare a variable named `vacation` of type `Season` with the statement:

```
Season vacation;
```

After declaring the variable, you can assign a value to it using an assignment statement. The value on the right-hand side of the assignment can be one of the enum constants of type `Season`. Remember to use the full name of the constant, including "Season"! For example:

```
vacation = Season.SUMMER;
```

You can print out an enum value with an output statement such as `System.out.print(vacation)`. The output value will be the name of the enum constant (without the "Season."). In this case, the output would be "SUMMER".

Because an enum is technically a class, the enum values are technically objects. As objects, they can contain subroutines. One of the subroutines in every enum value is named `ordinal()`. When used with an enum value, it returns the ***ordinal number*** of the value in the list of values of the enum. The ordinal number simply tells the position of the value in the list. That is, `Season.SPRING.ordinal()` is the **int** value 0, `Season.SUMMER.ordinal()` is 1, `Season.FALL.ordinal()` is 2, and `Season.WINTER.ordinal()` is 3. (You will see over and over again that computer scientists like to start counting at zero!) You can, of course, use the `ordinal()` method with a variable of type `Season`, such as `vacation.ordinal()` in our example.

Right now, it might not seem to you that enums are all that useful. As you work though the rest of the book, you should be convinced that they are. For now, you should at least appreciate them as the first example of an important concept: creating new types. Here is a little example that shows enums being used in a complete program:

```
public class EnumDemo {

       // Define two enum types -- remember that the definitions
       // go OUTSIDE The main() routine!

    enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY }

    enum Month { JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC }

    public static void main(String[] args) {

         Day tgif;     // Declare a variable of type Day.
         Month libra;  // Declare a variable of type Month.

         tgif = Day.FRIDAY;    // Assign a value of type Day to tgif.
         libra = Month.OCT;    // Assign a value of type Month to libra.

         System.out.print("My sign is libra, since I was born in ");
         System.out.println(libra);   // Output value will be:  OCT
         System.out.print("That's the ");
         System.out.print( libra.ordinal() );
         System.out.println("-th month of the year.");
         System.out.println("   (Counting from 0, of course!)");

         System.out.print("Isn't it nice to get to ");
         System.out.println(tgif);   // Output value will be:  FRIDAY

         System.out.println( tgif + " is the " + tgif.ordinal()
                                              + "-th day of the week.");
              // You can concatenate enum values onto Strings!

    }

}
```

2.4 Text Input and Output

For some unfathomable reason, Java has never made it easy to read data typed in by the user of a program. You've already seen that output can be displayed to the user using the subroutine `System.out.print`. This subroutine is part of a pre-defined object called `System.out`. The purpose of this object is precisely to display output to the user. There is a corresponding object called `System.in` that exists to read data input by the user, but it provides only very primitive input facilities, and it requires some advanced Java programming skills to use it effectively.

Java 5.0 finally makes input a little easier with a new *Scanner* class. However, it requires some knowledge of object-oriented programming to use this class, so it's not appropriate for use here at the beginning of this course. (Furthermore, in my opinion, *Scanner* still does not get things quite right.)

There is some excuse for this lack of concern with input, since Java is meant mainly to write programs for Graphical User Interfaces, and those programs have their own style of input/output, which **is** implemented in Java. However, basic support is needed for input/output in old-fashioned non-GUI programs. Fortunately, it is possible to **extend** Java by creating new classes that provide subroutines that are not available in the standard part of the language. As soon as a new class is available, the subroutines that it contains can be used in exactly the same way as built-in routines.

Along these lines, I've written a class called *TextIO* that defines subroutines for reading values typed by the user of a non-GUI program. The subroutines in this class make it possible to get input from the standard input object, `System.in`, without knowing about the advanced aspects of Java that are needed to use *Scanner* or to use `System.in` directly. *TextIO* also contains a set of output subroutines. The output subroutines are similar to those provided in `System.out`, but they provide a few additional features. You can use whichever set of output subroutines you prefer, and you can even mix them in the same program.

To use the *TextIO* class, you must make sure that the class is available to your program. What this means depends on the Java programming environment that you are using. In general, you just have to add the source code file, *TextIO.java*, to the same directory that contains your main program. See Section 2.6 for more information about how to use *TextIO*.

2.4.1 A First Text Input Example

The input routines in the *TextIO* class are static member functions. (Static member functions were introduced in the previous section.) Let's suppose that you want your program to read an integer typed in by the user. The *TextIO* class contains a static member function named `getlnInt` that you can use for this purpose. Since this function is contained in the *TextIO* class, you have to refer to it in your program as `TextIO.getlnInt`. The function has no parameters, so a complete call to the function takes the form "`TextIO.getlnInt()`". This function call represents the **int** value typed by the user, and you have to do something with the returned value, such as assign it to a variable. For example, if `userInput` is a variable of type **int** (created with a declaration statement "`int userInput;`"), then you could use the assignment statement

```
userInput = TextIO.getlnInt();
```

When the computer executes this statement, it will wait for the user to type in an integer value. The value typed will be returned by the function, and it will be stored in the variable, `userInput`. Here is a complete program that uses `TextIO.getlnInt` to read a number typed by the user and then prints out the square of the number that the user types:

```
/**
 * A program that reads an integer that is typed in by the
 * user and computes and prints the square of that integer.
 */

public class PrintSquare {

     public static void main(String[] args) {

        int userInput;  // The number input by the user.
        int square;     // The userInput, multiplied by itself.

        System.out.print("Please type a number: ");
        userInput = TextIO.getlnInt();
        square = userInput * userInput;
        System.out.print("The square of that number is ");
        System.out.println(square);

     } // end of main()

} //end of class PrintSquare
```

When you run this program, it will display the message "Please type a number:" and will pause until you type a response, including a carriage return after the number.

2.4.2 Text Output

The *TextIO* class contains static member subroutines `TextIO.put` and `TextIO.putln` that can be used in the same way as `System.out.print` and `System.out.println`. For example, although there is no particular advantage in doing so in this case, you could replace the two lines

```
System.out.print("The square of that number is ");
System.out.println(square);
```

with

```
TextIO.put("The square of that number is ");
TextIO.putln(square);
```

For the next few chapters, I will use *TextIO* for input in all my examples, and I will often use it for output. Keep in mind that *TextIO* can only be used in a program if it is available to that program. It is not built into Java in the way that the *System* class is.

Let's look a little more closely at the built-in output subroutines `System.out.print` and `System.out.println`. Each of these subroutines can be used with one parameter, where the parameter can be a value of any of the primitive types **byte**, **short**, **int**, **long**, **float**, **double**, **char**, or **boolean**. The parameter can also be a *String*, a value belonging to an enum type, or indeed any object. That is, you can say "`System.out.print(x);`" or "`System.out.println(x);`", where `x` is any expression whose value is of any type whatsoever. The expression can be a constant, a variable, or even something more complicated such as `2*distance*time`. Now, in fact, the *System* class actually includes several different subroutines to handle different parameter types. There is one `System.out.print` for printing values of type **double**, one for values of type **int**, another for values that are objects, and so on. These subroutines can have the same name since the computer can tell which one you mean in a given subroutine call statement, depending on the type of parameter that you supply. Having several subroutines of the same name that differ in the types of their parameters is called ***overloading***. Many programming languages do not permit overloading, but it is common in Java programs.

The difference between `System.out.print` and `System.out.println` is that the `println` version outputs a carriage return after it outputs the specified parameter value. There is a version of `System.out.println` that has no parameters. This version simply outputs a carriage return, and nothing else. A subroutine call statement for this version of the program looks like "`System.out.println();`", with empty parentheses. Note that "`System.out.println(x);`" is exactly equivalent to "`System.out.print(x); System.out.println();`"; the carriage return comes **after** the value of `x`. (There is no version of `System.out.print` without parameters. Do you see why?)

As mentioned above, the *TextIO* subroutines `TextIO.put` and `TextIO.putln` can be used as replacements for `System.out.print` and `System.out.println`. The *TextIO* functions work in exactly the same way as the *System* functions, except that, as we will see below, *TextIO* can also be used to write to other destinations.

2.4.3 TextIO Input Functions

The *TextIO* class is a little more versatile at doing output than is `System.out`. However, it's input for which we really need it.

With *TextIO*, input is done using functions. For example, `TextIO.getlnInt()`, which was discussed above, makes the user type in a value of type **int** and returns that input value so that you can use it in your program. *TextIO* includes several functions for reading different types of input values. Here are examples of the ones that you are most likely to use:

```
j = TextIO.getlnInt();     // Reads a value of type int.
y = TextIO.getlnDouble();  // Reads a value of type double.
a = TextIO.getlnBoolean(); // Reads a value of type boolean.
c = TextIO.getlnChar();    // Reads a value of type char.
w = TextIO.getlnWord();    // Reads one "word" as a value of type String.
s = TextIO.getln();        // Reads an entire input line as a String.
```

For these statements to be legal, the variables on the left side of each assignment statement must already be declared and must be of the same type as that returned by the function on the right side. Note carefully that these functions do not have parameters. The values that they return come from outside the program, typed in by the user as the program is running. To "capture" that data so that you can use it in your program, you have to assign the return value of the function to a variable. You will then be able to refer to the user's input value by using the name of the variable.

When you call one of these functions, you are guaranteed that it will return a legal value of the correct type. If the user types in an illegal value as input—for example, if you ask for an **int** and the user types in a non-numeric character or a number that is outside the legal range of values that can be stored in in a variable of type **int**—then the computer will ask the user to re-enter the value, and your program never sees the first, illegal value that the user entered. For `TextIO.getlnBoolean()`, the user is allowed to type in any of the following: true, false, t, f, yes, no, y, n, 1, or 0. Furthermore, they can use either upper or lower case letters. In any case, the user's input is interpreted as a true/false value. It's convenient to use `TextIO.getlnBoolean()` to read the user's response to a Yes/No question.

You'll notice that there are two input functions that return Strings. The first, `getlnWord()`, returns a string consisting of non-blank characters only. When it is called, it skips over any spaces and carriage returns typed in by the user. Then it reads non-blank characters until it gets to the next space or carriage return. It returns a *String* consisting of all the non-blank characters that it has read. The second input function, `getln()`, simply returns a string consisting of all the characters typed in by the user, including spaces, up to the next carriage return. It gets an entire line of input text. The carriage return itself is not returned as part of the input string, but it is read and discarded by the computer. Note that the String returned by this function might be the ***empty string***, `""`, which contains no characters at all. You will get this return value if the user simply presses return, without typing anything else first.

All the other input functions listed—`getlnInt()`, `getlnDouble()`, `getlnBoolean()`, and `getlnChar()`—behave like `getWord()` in that they will skip past any blanks and carriage returns in the input before reading a value.

Furthermore, if the user types extra characters on the line after the input value, **all the extra characters will be discarded, along with the carriage return at the end of the line**. If the program executes another input function, the user will have to type in another line of input. It might not sound like a good idea to discard any of the user's input, but it turns out to be the safest thing to do in most programs. Sometimes, however, you do want to read more

than one value from the same line of input. *TextIO* provides the following alternative input functions to allow you to do this:

```
j = TextIO.getInt();     // Reads a value of type int.
y = TextIO.getDouble();  // Reads a value of type double.
a = TextIO.getBoolean(); // Reads a value of type boolean.
c = TextIO.getChar();    // Reads a value of type char.
w = TextIO.getWord();    // Reads one "word" as a value of type String.
```

The names of these functions start with "get" instead of "getln". "Getln" is short for "get line" and should remind you that the functions whose names begin with "getln" will get an entire line of data. A function without the "ln" will read an input value in the same way, but will then save the rest of the input line in a chunk of internal memory called the ***input buffer***. The next time the computer wants to read an input value, it will look in the input buffer before prompting the user for input. This allows the computer to read several values from one line of the user's input. Strictly speaking, the computer actually reads **only** from the input buffer. The first time the program tries to read input from the user, the computer will wait while the user types in an entire line of input. *TextIO* stores that line in the input buffer until the data on the line has been read or discarded (by one of the "getln" functions). The user only gets to type when the buffer is empty.

Clearly, the semantics of input is much more complicated than the semantics of output! Fortunately, for the majority of applications, it's pretty straightforward in practice. You only need to follow the details if you want to do something fancy. In particular, I **strongly** advise you to use the "getln" versions of the input routines, rather than the "get" versions, unless you really want to read several items from the same line of input, precisely because the semantics of the "getln" versions is much simpler.

Note, by the way, that although the *TextIO* input functions will skip past blank spaces and carriage returns while looking for input, they will **not** skip past other characters. For example, if you try to read two **int**s and the user types "2,3", the computer will read the first number correctly, but when it tries to read the second number, it will see the comma. It will regard this as an error and will force the user to retype the number. If you want to input several numbers from one line, you should make sure that the user knows to separate them with spaces, not commas. Alternatively, if you want to require a comma between the numbers, use `getChar()` to read the comma before reading the second number.

There is another character input function, `TextIO.getAnyChar()`, which does not skip past blanks or carriage returns. It simply reads and returns the next character typed by the user, even if it's a blank or carriage return. If the user typed a carriage return, then the **char** returned by `getAnyChar()` is the special linefeed character '\n'. There is also a function, `TextIO.peek()`, that lets you look ahead at the next character in the input without actually reading it. After you "peek" at the next character, it will still be there when you read the next item from input. This allows you to look ahead and see what's coming up in the input, so that you can take different actions depending on what's there.

The *TextIO* class provides a number of other functions. To learn more about them, you can look at the comments in the source code file, *TextIO.java*.

(You might be wondering why there are only two output routines, `print` and `println`, which can output data values of any type, while there is a separate input routine for each data type. As noted above, in reality there are many `print` and `println` routines, one for each data type. The computer can tell them apart based on the type of the parameter that you provide. However, the input routines don't have parameters, so the different input routines can only be

distinguished by having different names.)

* * *

Using *TextIO* for input and output, we can now improve the program from Section 2.2 for computing the value of an investment. We can have the user type in the initial value of the investment and the interest rate. The result is a much more useful program—for one thing, it makes sense to run it more than once!

```
/**
 * This class implements a simple program that will compute
 * the amount of interest that is earned on an investment over
 * a period of one year.  The initial amount of the investment
 * and the interest rate are input by the user.  The value of
 * the investment at the end of the year is output.  The
 * rate must be input as a decimal, not a percentage (for
 * example, 0.05 rather than 5).
 */

public class Interest2 {

   public static void main(String[] args) {

       double principal;  // The value of the investment.
       double rate;       // The annual interest rate.
       double interest;   // The interest earned during the year.

       TextIO.put("Enter the initial investment: ");
       principal = TextIO.getlnDouble();

       TextIO.put("Enter the annual interest rate (decimal, not percentage!): ");
       rate = TextIO.getlnDouble();

       interest = principal * rate;       // Compute this year's interest.
       principal = principal + interest;  // Add it to principal.

       TextIO.put("The value of the investment after one year is $");
       TextIO.putln(principal);

   } // end of main()

} // end of class Interest2
```

2.4.4 Formatted Output

If you ran the preceding `Interest2` example, you might have noticed that the answer is not always written in the format that is usually used for dollar amounts. In general, dollar amounts are written with two digits after the decimal point. But the program's output can be a number like 1050.0 or 43.575. It would be better if these numbers were printed as 1050.00 and 43.58.

Java 5.0 introduced a formatted output capability that makes it much easier than it used to be to control the format of output numbers. A lot of formatting options are available. I will cover just a few of the simplest and most commonly used possibilities here.

You can use the function `System.out.printf` to produce formatted output. (The name "printf," which stands for "print formatted," is copied from the C and C++ programming languages, which have always have a similar formatting capability). `System.out.printf` takes two or more parameters. The first parameter is a *String* that specifies the format of the output. This parameter is called the ***format string***. The remaining parameters specify the values that

are to be output. Here is a statement that will print a number in the proper format for a dollar amount, where `amount` is a variable of type **double**:

```
System.out.printf( "%1.2f", amount );
```

TextIO can also do formatted output. The function `TextIO.putf` has the same functionality as `System.out.printf`. Using *TextIO*, the above example would be: `TextIO.printf("%1.2",amount);` and you could say `TextIO.putln("%1.2f",principal);` instead of `TextIO.putln(principal);` in the `Interest2` program to get the output in the right format.

The output format of a value is specified by a ***format specifier***. The format string (in the simple cases that I cover here) contains one format specifier for each of the values that is to be output. Some typical format specifiers are `%d`, `%12d`, `%10s`, `%1.2f`, `%15.8e` and `%1.8g`. Every format specifier begins with a percent sign (%) and ends with a letter, possibly with some extra formatting information in between. The letter specifies the type of output that is to be produced. For example, in `%d` and `%12d`, the "d" specifies that an integer is to be written. The "12" in `%12d` specifies the minimum number of spaces that should be used for the output. If the integer that is being output takes up fewer than 12 spaces, extra blank spaces are added in front of the integer to bring the total up to 12. We say that the output is "right-justified in a field of length 12." The value is not forced into 12 spaces; if the value has more than 12 digits, all the digits will be printed, with no extra spaces. The specifier `%d` means the same as `%1d`; that is an integer will be printed using just as many spaces as necessary. (The "d," by the way, stands for "decimal" (base-10) numbers. You can use an "x" to output an integer value in hexadecimal form.)

The letter "s" at the end of a format specifier can be used with any type of value. It means that the value should be output in its default format, just as it would be in unformatted output. A number, such as the "10" in `%10s` can be added to specify the (minimum) number of characters. The "s" stands for "string," meaning that the value is converted into a *String* value in the usual way.

The format specifiers for values of type **double** are even more complicated. An "f", as in `%1.2f`, is used to output a number in "floating-point" form, that is with digits after the decimal point. In `%1.2f`, the "2" specifies the number of digits to use after the decimal point. The "1" specifies the (minimum) number of characters to output, which effectively means that just as many characters as are necessary should be used. Similarly, `%12.3f` would specify a floating-point format with 3 digits after the decimal point, right-justified in a field of length 12.

Very large and very small numbers should be written in exponential format, such as 6.00221415e23, representing "6.00221415 times 10 raised to the power 23." A format specifier such as `%15.8e` specifies an output in exponential form, with the "8" telling how many digits to use after the decimal point. If you use "g" instead of "e", the output will be in floating-point form for small values and in exponential form for large values. In `%1.8g`, the 8 gives the total number of digits in the answer, including both the digits before the decimal point and the digits after the decimal point.

In addition to format specifiers, the format string in a `printf` statement can include other characters. These extra characters are just copied to the output. This can be a convenient way to insert values into the middle of an output string. For example, if `x` and `y` are variables of type **int**, you could say

```
System.out.printf("The product of %d and %d is %d", x, y, x*y);
```

When this statement is executed, the value of `x` is substituted for the first `%d` in the string, the

value of `y` for the second `%d`, and the value of the expression `x*y` for the third, so the output would be something like "The product of 17 and 42 is 714" (quotation marks not included in output!).

2.4.5 Introduction to File I/O

`System.out` sends its output to the output destination known as "standard output." But standard output is just one possible output destination. For example, data can be written to a ***file*** that is stored on the user's hard drive. The advantage to this, of course, is that the data is saved in the file even after the program ends, and the user can print the file, email it to someone else, edit it with another program, and so on.

TextIO has the ability to write data to files and to read data from files. When you write output using the `put`, `putln`, or `putf` method in *TextIO*, the output is sent to the ***current output destination.*** By default, the current output destination is standard output. However, *TextIO* has some subroutines that can be used to change the current output destination. To write to a file named "result.txt", for example, you would use the statement:

```
TextIO.writeFile("result.txt");
```

After this statement is executed, any output from *TextIO* output statements will be sent to the file named "result.txt" instead of to standard output. The file should be created in the same directory that contains the program. Note that if a file with the same name already exists, its previous contents will be erased! In many cases, you want to let the user select the file that will be used for output. The statement

```
TextIO.writeUserSelectedFile();
```

will open a typical graphical-user-interface file selection dialog where the user can specify the output file. If you want to go back to sending output to standard output, you can say

```
TextIO.writeStandardOutput();
```

You can also specify the input source for `TextIO's` various "get" functions. The default input source is standard input. You can use the statement `TextIO.readFile("data.txt")` to read from a file named "data.txt" instead, or you can let the user select the input file by saying `TextIO.readUserSelectedFile()`, and you can go back to reading from standard input with `TextIO.readStandardInput()`.

When your program is reading from standard input, the user gets a chance to correct any errors in the input. This is not possible when the program is reading from a file. If illegal data is found when a program tries to read from a file, an error occurs that will crash the program. (Later, we will see that is is possible to "catch" such errors and recover from them.) Errors can also occur, though more rarely, when writing to files.

A complete understanding of file input/output in Java requires a knowledge of object oriented programming. We will return to the topic later, in Chapter 11. The file I/O capabilities in *TextIO* are rather primitive by comparison. Nevertheless, they are sufficient for many applications, and they will allow you to get some experience with files sooner rather than later.

As a simple example, here is a program that asks the user some questions and outputs the user's responses to a file named "profile.txt":

```
public class CreateProfile {

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

```
        String name;     // The user's name.
        String email;    // The user's email address.
        double salary;   // the user's yearly salary.
        String favColor; // The user's favorite color.

        TextIO.putln("Good Afternoon!  This program will create");
        TextIO.putln("your profile file, if you will just answer");
        TextIO.putln("a few simple questions.");
        TextIO.putln();

        /* Gather responses from the user. */

        TextIO.put("What is your name?           ");
        name = TextIO.getln();
        TextIO.put("What is your email address?  ");
        email = TextIO.getln();
        TextIO.put("What is your yearly income?  ");
        salary = TextIO.getlnDouble();
        TextIO.put("What is your favorite color? ");
        favColor = TextIO.getln();

        /* Write the user's information to the file named profile.txt. */

        TextIO.writeFile("profile.txt");  // subsequent output goes to the file
        TextIO.putln("Name:            " + name);
        TextIO.putln("Email:           " + email);
        TextIO.putln("Favorite Color:  " + favColor);
        TextIO.putf( "Yearly Income:   %1.2f\n", salary);
                   // The "/n" in the previous line is a carriage return.

        /* Print a final message to standard output. */

        TextIO.writeStandardOutput();
        TextIO.putln("Thank you.  Your profile has been written to profile.txt.");

    }

}
```

2.5 Details of Expressions

This section takes a closer look at expressions. Recall that an expression is a piece of program code that represents or computes a value. An expression can be a literal, a variable, a function call, or several of these things combined with operators such as + and >. The value of an expression can be assigned to a variable, used as a parameter in a subroutine call, or combined with other values into a more complicated expression. (The value can even, in some cases, be ignored, if that's what you want to do; this is more common than you might think.) Expressions are an essential part of programming. So far, these notes have dealt only informally with expressions. This section tells you the more-or-less complete story (leaving out some of the less commonly used operators).

The basic building blocks of expressions are literals (such as `674`, `3.14`, `true`, and `'X'`), variables, and function calls. Recall that a function is a subroutine that returns a value. You've already seen some examples of functions, such as the input routines from the *TextIO* class and the mathematical functions from the *Math* class.

The *Math* class also contains a couple of mathematical constants that are useful in mathematical expressions: `Math.PI` represents π (the ratio of the circumference of a circle to its diameter), and `Math.E` represents e (the base of the natural logarithms). These "constants" are actually member variables in *Math* of type **double**. They are only approximations for the mathematical constants, which would require an infinite number of digits to specify exactly.

Literals, variables, and function calls are simple expressions. More complex expressions can be built up by using ***operators*** to combine simpler expressions. Operators include `+` for adding two numbers, `>` for comparing two values, and so on. When several operators appear in an expression, there is a question of ***precedence***, which determines how the operators are grouped for evaluation. For example, in the expression "`A + B * C`", `B*C` is computed first and then the result is added to `A`. We say that multiplication (`*`) has ***higher precedence*** than addition (`+`). If the default precedence is not what you want, you can use parentheses to explicitly specify the grouping you want. For example, you could use "`(A + B) * C`" if you want to add `A` to `B` first and then multiply the result by `C`.

The rest of this section gives details of operators in Java. The number of operators in Java is quite large, and I will not cover them all here. Most of the important ones are here; a few will be covered in later chapters as they become relevant.

2.5.1 Arithmetic Operators

Arithmetic operators include addition, subtraction, multiplication, and division. They are indicated by `+`, `-`, `*`, and `/`. These operations can be used on values of any numeric type: **byte**, **short**, **int**, **long**, **float**, or **double**. When the computer actually calculates one of these operations, the two values that it combines must be of the same type. If your program tells the computer to combine two values of different types, the computer will convert one of the values from one type to another. For example, to compute 37.4 + 10, the computer will convert the integer 10 to a real number 10.0 and will then compute 37.4 + 10.0. This is called a ***type conversion***. Ordinarily, you don't have to worry about type conversion in expressions, because the computer does it automatically.

When two numerical values are combined (after doing type conversion on one of them, if necessary), the answer will be of the same type. If you multiply two **int**s, you get an **int**; if you multiply two **double**s, you get a **double**. This is what you would expect, but you have to be very careful when you use the division operator `/`. When you divide two integers, the answer will always be an integer; if the quotient has a fractional part, it is discarded. For example, the value of `7/2` is `3`, not `3.5`. If `N` is an integer variable, then `N/100` is an integer, and `1/N` is equal to zero for any `N` greater than one! This fact is a common source of programming errors. You can force the computer to compute a real number as the answer by making one of the operands real: For example, when the computer evaluates `1.0/N`, it first converts `N` to a real number in order to match the type of `1.0`, so you get a real number as the answer.

Java also has an operator for computing the remainder when one integer is divided by another. This operator is indicated by `%`. If `A` and `B` are integers, then `A % B` represents the remainder when `A` is divided by `B`. (However, for negative operands, `%` is not quite the same as the usual mathematical "modulus" operator, since if one of `A` or `B` is negative, then the value of `A % B` will be negative.) For example, `7 % 2` is `1`, while `34577 % 100` is `77`, and `50 % 8` is `2`. A common use of `%` is to test whether a given integer is even or odd. `N` is even if `N % 2` is zero, and it is odd if `N % 2` is `1`. More generally, you can check whether an integer `N` is evenly divisible by an integer `M` by checking whether `N % M` is zero.

Finally, you might need the ***unary minus*** operator, which takes the negative of a number.

For example, `-X` has the same value as `(-1)*X`. For completeness, Java also has a unary plus operator, as in `+X`, even though it doesn't really do anything.

By the way, recall that the `+` operator can also be used to concatenate a value of any type onto a *String*. This is another example of type conversion. In Java, any type can be automatically converted into type *String*.

2.5.2 Increment and Decrement

You'll find that adding `1` to a variable is an extremely common operation in programming. Subtracting `1` from a variable is also pretty common. You might perform the operation of adding `1` to a variable with assignment statements such as:

```
counter = counter + 1;
goalsScored = goalsScored + 1;
```

The effect of the assignment statement `x = x + 1` is to take the old value of the variable `x`, compute the result of adding `1` to that value, and store the answer as the new value of `x`. The same operation can be accomplished by writing `x++` (or, if you prefer, `++x`). This actually changes the value of `x`, so that it has the same effect as writing "`x = x + 1`". The two statements above could be written

```
counter++;
goalsScored++;
```

Similarly, you could write `x--` (or `--x`) to subtract `1` from `x`. That is, `x--` performs the same computation as `x = x - 1`. Adding `1` to a variable is called ***incrementing*** that variable, and subtracting `1` is called ***decrementing***. The operators `++` and `--` are called the increment operator and the decrement operator, respectively. These operators can be used on variables belonging to any of the numerical types and also on variables of type **char**.

Usually, the operators `++` or `--` are used in statements like "`x++;`" or "`x--;`". These statements are commands to change the value of `x`. However, it is also legal to use `x++`, `++x`, `x--`, or `--x` as expressions, or as parts of larger expressions. That is, you can write things like:

```
y = x++;
y = ++x;
TextIO.putln(--x);
z = (++x) * (y--);
```

The statement "`y = x++;`" has the effects of adding `1` to the value of `x` and, in addition, assigning some value to `y`. The value assigned to `y` is the value of the expression `x++`, which is defined to be the **old** value of `x`, before the `1` is added. Thus, if the value of `x` is `6`, the statement "`y = x++;`" will change the value of `x` to `7`, but it will change the value of `y` to `6` since the value assigned to `y` is the **old** value of `x`. On the other hand, the value of `++x` is defined to be the **new** value of `x`, after the `1` is added. So if `x` is `6`, then the statement "`y = ++x;`" changes the values of both `x` and `y` to `7`. The decrement operator, `--`, works in a similar way.

This can be confusing. My advice is: Don't be confused. Use `++` and `--` only in stand-alone statements, not in expressions. I will follow this advice in all the examples in these notes.

2.5.3 Relational Operators

Java has boolean variables and boolean-valued expressions that can be used to express conditions that can be either `true` or `false`. One way to form a boolean-valued expression is

to compare two values using a ***relational operator***. Relational operators are used to test whether two values are equal, whether one value is greater than another, and so forth. The relational operators in Java are: ==, !=, <, >, <=, and >=. The meanings of these operators are:

```
A == B       Is A "equal to" B?
A != B       Is A "not equal to" B?
A < B        Is A "less than" B?
A > B        Is A "greater than" B?
A <= B       Is A "less than or equal to" B?
A >= B       Is A "greater than or equal to" B?
```

These operators can be used to compare values of any of the numeric types. They can also be used to compare values of type **char**. For characters, < and > are defined according the numeric Unicode values of the characters. (This might not always be what you want. It is not the same as alphabetical order because all the upper case letters come before all the lower case letters.)

When using boolean expressions, you should remember that as far as the computer is concerned, there is nothing special about boolean values. In the next chapter, you will see how to use them in loop and branch statements. But you can also assign boolean-valued expressions to boolean variables, just as you can assign numeric values to numeric variables.

By the way, the operators == and != can be used to compare boolean values. This is occasionally useful. For example, can you figure out what this does:

```
boolean sameSign;
sameSign = ((x > 0) == (y > 0));
```

One thing that you **cannot** do with the relational operators <, >, <=, and <= is to use them to compare values of type *String*. You can legally use == and != to compare `Strings`, but because of peculiarities in the way objects behave, they might not give the results you want. (The == operator checks whether two objects are stored in the same memory location, rather than whether they contain the same value. Occasionally, for some objects, you do want to make such a check—but rarely for strings. I'll get back to this in a later chapter.) Instead, you should use the subroutines `equals()`, `equalsIgnoreCase()`, and `compareTo()`, which were described in Section 2.3, to compare two `Strings`.

2.5.4 Boolean Operators

In English, complicated conditions can be formed using the words "and", "or", and "not." For example, "If there is a test **and** you did **not** study for it...". "And", "or", and "not" are ***boolean operators***, and they exist in Java as well as in English.

In Java, the boolean operator "and" is represented by `&&`. The `&&` operator is used to combine two boolean values. The result is also a boolean value. The result is `true` if **both** of the combined values are `true`, and the result is `false` if **either** of the combined values is `false`. For example, "`(x == 0) && (y == 0)`" is `true` if and only if both `x` is equal to 0 and `y` is equal to 0.

The boolean operator "or" is represented by `||`. (That's supposed to be two of the vertical line characters, |.) The expression "`A || B`" is `true` if either `A` is `true` or `B` is `true`, or if both are true. "`A || B`" is `false` only if both `A` and `B` are false.

The operators `&&` and `||` are said to be ***short-circuited*** versions of the boolean operators. This means that the second operand of `&&` or `||` is not necessarily evaluated. Consider the test

```
(x != 0) && (y/x > 1)
```

Suppose that the value of `x` is in fact zero. In that case, the division `y/x` is undefined mathmatically. However, the computer will never perform the division, since when the computer evaluates `(x != 0)`, it finds that the result is `false`, and so it knows that (`(x != 0) &&` **anything**) has to be false. Therefore, it doesn't bother to evaluate the second operand, `(y/x > 1)`. The evaluation has been short-circuited and the division by zero is avoided. Without the short-circuiting, there would have been a division by zero. (This may seem like a technicality, and it is. But at times, it will make your programming life a little easier.)

The boolean operator "not" is a unary operator. In Java, it is indicated by `!` and is written in front of its single operand. For example, if `test` is a boolean variable, then

```
test = ! test;
```

will reverse the value of test, changing it from `true` to `false`, or from `false` to `true`.

2.5.5 Conditional Operator

Any good programming language has some nifty little features that aren't really necessary but that let you feel cool when you use them. Java has the conditional operator. It's a ternary operator—that is, it has three operands—and it comes in two pieces, ? and :, that have to be used together. It takes the form

⟨*boolean-expression*⟩ `?` ⟨*expression1*⟩ `:` ⟨*expression2*⟩

The computer tests the value of ⟨***boolean-expression***⟩. If the value is `true`, it evaluates ⟨***expression1***⟩; otherwise, it evaluates ⟨***expression2***⟩. For example:

```
next = (N % 2 == 0) ? (N/2) : (3*N+1);
```

will assign the value `N/2` to `next` if `N` is even (that is, if `N % 2 == 0` is `true`), and it will assign the value `(3*N+1)` to `next` if `N` is odd. (The parentheses in this example are not required, but they do make the expression easier to read.)

2.5.6 Assignment Operators and Type-Casts

You are already familiar with the assignment statement, which uses the symbol "=" to assign the value of an expression to a variable. In fact, = is really an operator in the sense that an assignment can itself be used as an expression or as part of a more complex expression. The value of an assignment such as `A=B` is the same as the value that is assigned to `A`. So, if you want to assign the value of `B` to `A` and test at the same time whether that value is zero, you could say:

```
if ( (A=B) == 0 )...
```

Usually, I would say, **don't do things like that!**

In general, the type of the expression on the right-hand side of an assignment statement must be the same as the type of the variable on the left-hand side. However, in some cases, the computer will automatically convert the value computed by the expression to match the type of the variable. Consider the list of numeric types: **byte**, **short**, **int**, **long**, **float**, **double**. A value of a type that occurs earlier in this list can be converted automatically to a value that occurs later. For example:

```
int A;
double X;
short B;
```

```
A = 17;
X = A;    // OK; A is converted to a double
B = A;    // illegal; no automatic conversion
          //       from int to short
```

The idea is that conversion should only be done automatically when it can be done without changing the semantics of the value. Any **int** can be converted to a **double** with the same numeric value. However, there are **int** values that lie outside the legal range of **short**s. There is simply no way to represent the **int** 100000 as a **short**, for example, since the largest value of type **short** is 32767.

In some cases, you might want to force a conversion that wouldn't be done automatically. For this, you can use what is called a ***type cast***. A type cast is indicated by putting a type name, in parentheses, in front of the value you want to convert. For example,

```
int A;
short B;
A = 17;
B = (short)A;  // OK; A is explicitly type cast
               //      to a value of type short
```

You can do type casts from any numeric type to any other numeric type. However, you should note that you might change the numeric value of a number by type-casting it. For example, `(short)100000` is -31072. (The -31072 is obtained by taking the 4-byte **int** 100000 and throwing away two of those bytes to obtain a **short**—you've lost the real information that was in those two bytes.)

As another example of type casts, consider the problem of getting a random integer between 1 and 6. The function `Math.random()` gives a real number between 0.0 and 0.9999..., and so `6*Math.random()` is between 0.0 and 5.999.... The type-cast operator, `(int)`, can be used to convert this to an integer: `(int)(6*Math.random())`. A real number is cast to an integer by discarding the fractional part. Thus, `(int)(6*Math.random())` is one of the integers 0, 1, 2, 3, 4, and 5. To get a number between 1 and 6, we can add 1: "`(int)(6*Math.random()) + 1`".

You can also type-cast between the type **char** and the numeric types. The numeric value of a **char** is its Unicode code number. For example, `(char)97` is `'a'`, and `(int)'+'` is `43`. (However, a type conversion from **char** to **int** is automatic and does not have to be indicated with an explicit type cast.)

Java has several variations on the assignment operator, which exist to save typing. For example, "`A += B`" is defined to be the same as "`A = A + B`". Every operator in Java that applies to two operands gives rise to a similar assignment operator. For example:

```
x -= y;     // same as:   x = x - y;
x *= y;     // same as:   x = x * y;
x /= y;     // same as:   x = x / y;
x %= y;     // same as:   x = x % y;   (for integers x and y)
q &&= p;    // same as:   q = q && p;  (for booleans q and p)
```

The combined assignment operator `+=` even works with strings. Recall that when the `+` operator is used with a string as one of the operands, it represents concatenation. Since `str += x` is equivalent to `str = str + x`, when `+=` is used with a string on the left-hand side, it appends the value on the right-hand side onto the string. For example, if `str` has the value "tire", then the statement `str += 'd';` changes the value of `str` to "tired".

2.5.7 Type Conversion of Strings

In addition to automatic type conversions and explicit type casts, there are some other cases where you might want to convert a value of one type into a value of a different type. One common example is the conversion of a *String* value into some other type, such as converting the string `"10"` into the **int** value 10 or the string `"17.42e-2"` into the **double** value 0.1742. In Java, these conversions are handled by built-in functions.

There is a standard class named *Integer* that contains several subroutines and variables related to the **int** data type. (Recall that since **int** is not a class, **int** itself can't contain any subroutines or variables.) In particular, if `str` is any expression of type *String*, then `Integer.parseInt(str)` is a function call that attempts to convert the value of `str` into a value of type **int**. For example, the value of `Integer.parseInt("10")` is the **int** value 10. If the parameter to `Integer.parseInt` does not represent a legal **int** value, then an error occurs.

Similarly, the standard class named *Double* includes a function `Double.parseDouble` that tries to convert a parameter of type *String* into a value of type **double**. For example, the value of the function call `Double.parseDouble("3.14")` is the **double** value 3.14. (Of course, in practice, the parameter used in `Double.parseDouble` or `Integer.parseInt` would be a variable or expression rather than a constant string.)

Type conversion functions also exist for converting strings into enumerated type values. (Enumerated types, or enums, were introduced in Subsection 2.3.3.) For any enum type, a predefined function named `valueOf` is automatically defined for that type. This is a function that takes a string as parameter and tries to convert it to a value belonging to the enum. The `valueOf` function is part of the enum type, so the name of the enum is part of the full name of the function. For example, if an enum *Suit* is defined as

```
enum Suit { SPADE, DIAMOND, CLUB, HEART }
```

then the name of the type conversion function would be `Suit.valueOf`. The value of the function call `Suit.valueOf("CLUB")` would be the enumerated type value `Suit.CLUB`. For the conversion to succeed, the string must exactly match the simple name of one of the enumerated type constants (**without** the "Suit." in front).

2.5.8 Precedence Rules

If you use several operators in one expression, and if you don't use parentheses to explicitly indicate the order of evaluation, then you have to worry about the precedence rules that determine the order of evaluation. (Advice: don't confuse yourself or the reader of your program; use parentheses liberally.)

Here is a listing of the operators discussed in this section, listed in order from highest precedence (evaluated first) to lowest precedence (evaluated last):

```
Unary operators:              ++, --, !, unary - and +, type-cast
Multiplication and division:  *,  /,  %
Addition and subtraction:     +,  -
Relational operators:         <,  >,  <=,  >=
Equality and inequality:      ==,  !=
Boolean and:                  &&
Boolean or:                   ||
Conditional operator:         ?:
Assignment operators:         =,  +=,  -=,  *=,  /=,  %=
```

Operators on the same line have the same precedence. When operators of the same precedence are strung together in the absence of parentheses, unary operators and assignment operators are evaluated right-to-left, while the remaining operators are evaluated left-to-right. For example, `A*B/C` means `(A*B)/C`, while `A=B=C` means `A=(B=C)`. (Can you see how the expression `A=B=C` might be useful, given that the value of `B=C` as an expression is the same as the value that is assigned to `B`?)

2.6 Programming Environments

ALTHOUGH THE JAVA LANGUAGE is highly standardized, the procedures for creating, compiling, and editing Java programs vary widely from one programming environment to another. There are two basic approaches: a ***command line environment***, where the user types commands and the computer responds, and an ***integrated development environment*** (IDE), where the user uses the keyboard and mouse to interact with a graphical user interface. While there is just one common command line environment for Java programming, there is a wide variety of IDEs.

I cannot give complete or definitive information on Java programming environments in this section, but I will try to give enough information to let you compile and run the examples from this textbook, at least in a command line environment. There are many IDEs, and I can't cover them all here. I will concentrate on ***Eclipse***, one of the most popular IDEs for Java programming, but some of the information that is presented will apply to other IDEs as well.

One thing to keep in mind is that you do not have to pay any money to do Java programming (aside from buying a computer, of course). Everything that you need can be downloaded for free on the Internet.

2.6.1 Java Development Kit

The basic development system for Java programming is usually referred to as the ***JDK*** (Java Development Kit). It is a part of J2SE, the Java 2 Platform Standard Edition. This book requires J2SE version 5.0 (or higher). Confusingly, the JDK that is part of J2SE version 5.0 is sometimes referred to as JDK 1.5 instead of 5.0. Note that J2SE comes in two versions, a Development Kit version and a Runtime version. The Runtime can be used to run Java programs and to view Java applets in Web pages, but it does not allow you to compile your own Java programs. The Development Kit includes the Runtime and adds to it the JDK which lets you compile programs. You need a JDK for use with this textbook.

Java was developed by Sun Microsystems, Inc., which makes its JDK for Windows and Linux available for free download at its Java Web site, java.sun.com. If you have a Windows computer, it might have come with a Java Runtime, but you might still need to download the JDK. Some versions of Linux come with the JDK either installed by default or on the installation media. If you need to download and install the JDK, be sure to get JDK 5.0 (or higher). As of June, 2006, the download page for JDK 5.0 can be found at http://java.sun.com/j2se/1.5.0/download.jsp.

Mac OS comes with Java. The version included with Mac OS 10.4 is 1.4.2, the version previous to Java 5.0. However, JDK Version 5.0 is available for Mac OS 10.4 on Apple's Web site and can also be installed through the standard Mac OS Software Update application.

If a JDK is installed on your computer, you can use the command line environment to compile and run Java programs. Some IDEs depend on the JDK, so even if you plan to use an IDE for programming, you might still need a JDK.

2.6.2 Command Line Environment

Many modern computer users find the command line environment to be pretty alien and unintuitive. It is certainly very different from the graphical user interfaces that most people are used to. However, it takes only a little practice to learn the basics of the command line environment and to become productive using it.

To use a command line programming environment, you will have to open a window where you can type in commands. In Windows, you can open such a command window by running the program named ***cmd***. One way to run `cmd` is to use the "Run Program" feature in the Start menu, and enter "cmd" as the name of the program. In Mac OS, you want to run the ***Terminal*** program, which can be be found in the Utilities folder inside the Applications folder. In Linux, there are several possibilities, including ***Konsole***, ***gterm***, and ***xterm***.

No matter what type of computer you are using, when you open a command window, it will display a prompt of some sort. Type in a command at the prompt and press return. The computer will carry out the command, displaying any output in the command window, and will then redisplay the prompt so that you can type another command. One of the central concepts in the command line environment is the ***current directory*** which contains the files to which commands that you type apply. (The words "directory" and "folder" mean the same thing.) Often, the name of the current directory is part of the command prompt. You can get a list of the files in the current directory by typing in the command ***dir*** (on Windows) or ***ls*** (on Linux and Mac OS). When the window first opens, the current directory is your ***home directory***, where all your files are stored. You can change the current directory using the ***cd*** command with the name of the directory that you want to use. For example, to change into your Desktop directory, type in the command `cd Desktop` and press return.

You should create a directory (that is, a folder) to hold your Java work. For example, create a directory named `javawork` in your home directory. You can do this using your computer's GUI; another way to do it is to open a command window and enter the command `mkdir javawork`. When you want to work on programming, open a command window and enter the command `cd javawork` to change into your work directory. Of course, you can have more than one working directory for your Java work; you can organize your files any way you like.

* * *

The most basic commands for using Java on the command line are ***javac*** and ***java***; `javac` is used to compile Java source code, and `java` is used to run Java stand-alone applications. If a JDK is correctly installed on your computer, it should recognize these commands when you type them in on the command line. Try typing the commands `java -version` and `javac -version` which should tell you which version of Java is installed. If you get a message such as "Command not found," then Java is not correctly installed. If the "java" command works, but "javac" does not, it means that a Java Runtime is installed rather than a Development Kit.

To test the `javac` command, place a copy of *TextIO.java* into your working directory. (If you downloaded the Web site of this book, you can find it in the directory named `source`; you can use your computer's GUI to copy-and-paste this file into your working directory. Alternatively, you can navigate to *TextIO.java* on the book's Web site and use the "Save As" command in your Web browser to save a copy of the file into your working directory.) Type the command:

```
javac  TextIO.java
```

This will compile `TextIO.java` and will create a bytecode file named `TextIO.class` in the same directory. Note that if the command succeeds, you will not get any response from the computer; it will just redisplay the command prompt to tell you it's ready for another command.

To test the `java` command, copy sample program `Interest2.java` from this book's source directory into your working directory. First, compile the program with the command

```
javac  Interest2.java
```

Remember that for this to succeed, *TextIO* must already be in the same directory. Then you can execute the program using the command

```
java  Interest2
```

Be careful to use **just the name** of the program, *Interest2*, not the name of the Java source code file or the name of the compiled class file. When you give this command, the program will run. You will be asked to enter some information, and you will respond by typing your answers into the command window, pressing return at the end of the line. When the program ends, you will see the command prompt, and you can enter another command.

You can follow the same procedure to run all of the examples in the early sections of this book. When you start work with applets, you will need a different command to execute the applets. That command will be introduced later in the book.

* * *

To create your own programs, you will need a ***text editor***. A text editor is a computer program that allows you to create and save documents that contain plain text. It is important that the documents be saved as plain text, that is without any special encoding or formatting information. Word processor documents are not appropriate, unless you can get your word processor to save as plain text. A good text editor can make programming a lot more pleasant. Linux comes with several text editors. On Windows, you can use notepad in a pinch, but you will probably want something better. For Mac OS, you might download the free ***TextWrangler*** application. One possibility that will work on any platform is to use ***jedit***, a good programmer's text editor that is itself written in Java and that can be downloaded for free from www.jedit.org.

To create your own programs, you should open a command line window and `cd` into the working directory where you will store your source code files. Start up your text editor program, such as by double-clicking its icon or selecting it from a Start menu. Type your code into the editor window, or open an existing source code file that you want to modify. Save the file. Remember that the name of a Java source code file must end in ".java", and the rest of the file name must match the name of the class that is defined in the file. Once the file is saved in your working directory, go to the command window and use the `javac` command to compile it, as discussed above. If there are syntax errors in the code, they will be listed in the command window. Each error message contains the line number in the file where the computer found the error. Go back to the editor and try to fix the errors, **save your changes**, and they try the `javac` command again. (It's usually a good idea to just work on the first few errors; sometimes fixing those will make other errors go away.) Remember that when the `javac` command finally succeeds, you will get no message at all. Then you can use the `java` command to run your program, as described above. Once you've compiled the program, you can run it as many times as you like without recompiling it.

That's really all there is to it: Keep both editor and command-line window open. Edit, save, and compile until you have eliminated all the syntax errors. (Always remember to save the file before compiling it—the compiler only sees the saved file, not the version in the editor window.) When you run the program, you might find that it has semantic errors that cause it to run incorrectly. It that case, you have to go back to the edit/save/compile loop to try to find and fix the problem.

2.6.3 IDEs and Eclipse

In an Integrated Development Environment, everything you need to create, compile, and run programs is integrated into a single package, with a graphical user interface that will be familiar to most computer users. There are many different IDEs for Java program development, ranging from fairly simple wrappers around the JDK to highly complex applications with a multitude of features. For a beginning programmer, there is a danger in using an IDE, since the difficulty of learning to use the IDE, on top of the difficulty of learning to program, can be overwhelming. Recently, however, I have begun using one IDE, ***Eclipse***, in my introductory programming courses. Eclipse has a variety of features that are very useful for a beginning programmer. And even though it has many advanced features, its design makes it possible to use Eclipse without understanding its full complexity. It is likely that other modern IDEs have similar properties, but my only in-depth experience is with Eclipse. Eclipse is used by many professional programmers and is probably the most commonly used Java IDE. (In fact, Eclipse is actually a general development platform that can be used for other purposes besides Java development, but its most common use is Java.)

Eclipse is itself written in Java. It requires Java 1.4 (or higher) to run, so it works on any computer platform that supports Java 1.4, including Linux, Windows, and recent versions of Mac OS. If you want to use Eclipse to compile and run Java 5.0 programs, you need Eclipse version 3.1 (or higher). Furthermore, Eclipse requires a JDK. You should make sure that JDK 5.0 (or higher) is installed on your computer, as described above, **before** you install Eclipse. Eclipse can be downloaded for free from www.eclipse.org.

The first time you start Eclipse, you will be asked to specify a ***workspace***, which is the directory where all your work will be stored. You can accept the default name, or provide one of your own. When startup is complete, the Eclipse window will be filled by a large "Welcome" screen that includes links to extensive documentation and tutorials. You can close this screen, by clicking the "X" next to the word "Welcome"; you can get back to it later by choosing "Welcome" from the "Help" menu.

The Eclipse GUI consists of one large window that is divided into several sections. Each section contains one or more ***views***. If there are several views in one section, there there will be tabs at the top of the section to select the view that is displayed in that section. Each view displays a different type of information. The whole set of views is called a ***perspective***. Eclipse uses different perspectives, that is different sets of views of different types of information, for different tasks. The only perspective that you will need is the "Java Perspective." Select the "Java Perspective" from the "Open Perspective" submenu of the "Window" menu. (You will only have to do this once, the first time you start Eclipse.) The Java perspective includes a large area in the center of the window where you will create and edit your Java programs. To the left of this is the Package Explorer view, which will contain a list of your Java projects and source code files. To the right is an "Outline" view which shows an outline of the file that you are currently editing; I don't find this very useful, and I suggest that you close the Outline view by clicking the "X" next to the word Outline. Several other views that will be useful while you are compiling and running programs appear in a section of the window below the editing area. If you accidently close one of the important views, such as the Package Explorer, you can get it back by selecting it form the "Show View" submenu of the "Window" menu.

* * *

To do any work in Eclipse, you need a ***project***. To start a Java project, go to the "New" submenu in the "File" menu, and select the "Project" command. In the window that pops

up, make sure "Java Project" is selected, and click the "Next" button. In the next window, it should only be necessary to fill in a "Project Name" for the project and click the "Finish" button. The project should appear in the "Package Explorer" view. Click on the small triangle next to the project name to see the contents of the project. At the beginning, it contains only the "JRE System Library"; this is the collection of standard built-in classes that come with Java.

To run the *TextIO* based examples from this textbook, you must add the source code file *TextIO.java* to your project. If you have downloaded the Web site of this book, you can find a copy of *TextIO.java* in the source directory. Alternatively, you can navigate to the file on-line and use the "Save As" command of your Web browser to save a copy of the file onto your computer. The easiest way to get *TextIO* into your project is to locate the source code file on your computer and drag the file icon onto the project name in the Eclipse window. If that doesn't work, you can try using copy-and-paste: Right-click the file icon (or control-click on Mac OS), select "Copy" from the pop-up menu, right-click the project name in the Eclipse window, and select "Paste". If you also have trouble with that, you can try using the "Import" command in the "File" menu; select "File system" in the window that pops up, click "Next", and provide the necessary information in the next window. (Unfortunately, using the file import window is rather complicated. If you find that you have to use it, you should consult the Eclipse documentation about it.) In any case, *TextIO* should appear in your project, inside a ***package*** named "default package". You will need to click the small triangle next to "default package" to see the file. Once a file is in this list, you can open it by double-clicking it; it will appear in the editing area of the Eclipse window.

To run any of the Java programs from this textbook, copy the source code file into your Eclipse Java project. To run the program, right-click the file name in the Package Explorer view (or control-click in Mac OS). In the menu that pops up, go to the "Run As" submenu, and select "Java Application". The program will be executed. If the program writes to standard output, the output will appear in the "Console" view, under the editing area. If the program uses *TextIO* for input, you will have to type the required input into the "Console" view—click the "Console" view before you start typing, so that the characters that you type will be sent to the correct part of the window. (Note that if you don't like doing I/O in the "Console" view, you can use an alternative version of *TextIO.java* that opens a separate window for I/O. You can find this "GUI" version of TextIO in a directory named `TextIO-GUI` inside this textbook's source directory.)

You can have more than one program in the same Eclipse project, or you can create additional projects to organize your work better. Remember to place a copy of *TextIO.java* in any project that requires it.

* * *

To create your own Java program, you must create a new Java class. To do this, right-click the Java project name in the "Project Explorer" view. Go to the "New" submenu of the popup menu, and select "Class". In the window that opens, type in the name of the class, and click the "Finish" button. Note that you want the name of the class, not the name of the source code file, so don't add ".java" at the end of the name. The class should appear inside the "default package," and it should automatically open in the editing area so that you can start typing in your program.

Eclipse has several features that aid you as you type your code. It will underline any syntax error with a jagged red line, and in some cases will place an error marker in the left border of the edit window. If you hover the mouse cursor over the error marker, a description of the

error will appear. Note that you do not have to get rid of every error immediately as you type; some errors will go away as you type in more of the program. If an error marker displays a small "light bulb," Eclipse is offering to try to fix the error for you. Click the light bulb to get a list of possible fixes, then double click the fix that you want to apply. For example, if you use an undeclared variable in your program, Eclipse will offer to declare it for you. You can actually use this error-correcting feature to get Eclipse to write certain types of code for you! Unfortunately, you'll find that you won't understand a lot of the proposed fixes until you learn more about the Java language.

Another nice Eclipse feature is ***code assist***. Code assist can be invoked by typing Control-Space. It will offer possible completions of whatever you are typing at the moment. For example, if you type part of an identifier and hit Control-Space, you will get a list of identifiers that start with the characters that you have typed; use the up and down arrow keys to select one of the items in the list, and press Return or Enter. (Or hit Escape to dismiss the list.) If there is only one possible completion when you hit Control-Space, it will be inserted automatically. By default, Code Assist will also pop up automatically, after a short delay, when you type a period or certain other characters. For example, if you type "`TextIO.`" and pause for just a fraction of a second, you will get a list of all the subroutines in the *TextIO* class. Personally, I find this auto-activation annoying. You can disable it in the Eclipse Preferences. (Look under Java / Editor / Code Assist, and turn off the "Enable auto activation" option.) You can still call up Code Assist manually with Control-Space.

Once you have an error-free program, you can run it as described above, by right-clicking its name in the Package Explorer and using "Run As / Java Application". If you find a problem when you run it, it's very easy to go back to the editor, make changes, and run it again. Note that using Eclipse, there is no explicit "compile" command. The source code files in your project are automatically compiled, and are re-compiled whenever you modify them.

Although I have only talked about Eclipse here, if you are using a different IDE, you will probably find a lot of similarities. Most IDEs use the concept of a "project" to which you have to add your source code files, and most of them have menu commands for running a program. All of them, of course, come with built-in text editors.

2.6.4 The Problem of Packages

Every class in Java is contained in something called a ***package***. Classes that are not explicitly put into a different package are in the "default" package. Almost all the examples in this textbook are in the default package, and I will not even discuss packages in any depth until Section 4.5. However, some IDEs might force you to pay attention to packages.

When you create a class in Eclipse, you might notice a message that says that "The use of the default package is discouraged." Although this is true, I have chosen to use it anyway, since it seems easier for beginning programmers to avoid the whole issue of packages, at least at first. Some IDEs might be even less willing than Eclipse to use the default package. If you create a class in a package, the source code starts with a line that specifies which package the class is in. For example, if the class is in a package named `testpkg`, then the first line of the source code will be

```
package testpkg;
```

In an IDE, this will not cause any problem unless the program you are writing depends on *TextIO*. You will not be able to use *TextIO* in a program unless *TextIO* is placed into the same package as the program. This means that you have to modify the source code file *TextIO.java*

to specify the package; just add a `package` statement using the same package name as the program. Then add the modified *TextIO.java* to the same folder that contains the program source code. Once you've done this, the example should run in the same way as if it were in the default package.

By the way, if you use packages in a command-line environment, other complications arise. For example, if a class is in a package named `testpkg`, then the source code file must be in a subdirectory named `testpkg` that is inside your main Java working directory. Nevertheless, when you compile or execute the program, you should be in the main directory, not in the subdirectory. When you compile the source code file, you have to include the name of the directory in the command: Use "`javac testpkg/ClassName.java`" on Linux or Mac OS, or "`javac testpkg\ClassName.java`" on Windows. The command for executing the program is then "`java testpkg.ClassName`", with a period separating the package name from the class name. Since packages can contain subpackages, it can get even worse than this! However, you will not need to worry about any of that when using the examples in this book.

Exercises for Chapter 2

1. Write a program that will print your initials to standard output in letters that are nine lines tall. Each big letter should be made up of a bunch of *'s. For example, if your initials were "DJE", then the output would look something like:

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

2. Write a program that simulates rolling a pair of dice. You can simulate rolling one die by choosing one of the integers 1, 2, 3, 4, 5, or 6 at random. The number you pick represents the number on the die after it is rolled. As pointed out in Section 2.5, The expression

```
(int)(Math.random()*6) + 1
```

   does the computation you need to select a random integer between 1 and 6. You can assign this value to a variable to represent one of the dice that are being rolled. Do this twice and add the results together to get the total roll. Your program should report the number showing on each die as well as the total roll. For example:

```
The first die comes up 3
The second die comes up 5
Your total roll is 8
```

3. Write a program that asks the user's name, and then greets the user by name. Before outputting the user's name, convert it to upper case letters. For example, if the user's name is Fred, then the program should respond "Hello, FRED, nice to meet you!".

4. Write a program that helps the user count his change. The program should ask how many quarters the user has, then how many dimes, then how many nickels, then how many pennies. Then the program should tell the user how much money he has, expressed in dollars.

5. If you have `N` eggs, then you have `N/12` dozen eggs, with `N%12` eggs left over. (This is essentially the definition of the `/` and `%` operators for integers.) Write a program that asks the user how many eggs she has and then tells the user how many dozen eggs she has and how many extra eggs are left over.

   A gross of eggs is equal to 144 eggs. Extend your program so that it will tell the user how many gross, how many dozen, and how many left over eggs she has. For example, if the user says that she has 1342 eggs, then your program would respond with

```
Your number of eggs is 9 gross, 3 dozen, and 10
```

since 1342 is equal to 9*144 + 3*12 + 10.

6. Suppose that a file named "testdata.txt" contains the following information: The first line of the file is the name of a student. Each of the next three lines contains an integer. The integers are the student's scores on three exams. Write a program that will read the information in the file and display (on standard output) a message the contains the name of the student and the student's average grade on the three exams. The average is obtained by adding up the individual exam grades and then dividing by the number of exams.

Quiz on Chapter 2

1. Briefly explain what is meant by the *syntax* and the *semantics* of a programming language. Give an example to illustrate the difference between a syntax error and a semantics error.

2. What does the computer do when it executes a variable declaration statement. Give an example.

3. What is a *type,* as this term relates to programming?

4. One of the primitive types in Java is *boolean.* What is the **boolean** type? Where are boolean values used? What are its possible values?

5. Give the meaning of each of the following Java operators:

   a) `++`

   b) `&&`

   c) `!=`

6. Explain what is meant by an *assignment statement,* and give an example. What are assignment statements used for?

7. What is meant by *precedence* of operators?

8. What is a *literal*?

9. In Java, classes have two fundamentally different purposes. What are they?

10. What is the difference between the statement "`x = TextIO.getDouble();`" and the statement "`x = TextIO.getlnDouble();`"

11. Explain why the value of the expression `2 + 3 + "test"` is the string `"5test"` while the value of the expression `"test" + 2 + 3` is the string `"test23"`. What is the value of `"test" + 2 * 3` ?

12. Integrated Development Environments such as Eclipse often use ***syntax coloring***, which assigns various colors to the characters in a program to reflect the syntax of the language. A student notices that Eclipse colors the word *String* differently from **int**, **double**, and **boolean**. The student asks why *String* should be a different color, since all these words are names of types. What's the answer to the student's question?