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 very 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 *(online)*

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 made input from any source 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. Java 6 introduced the *Console* class, specifically for communicating with the user, but again, using *Console* requires more knowledge about objects than you have at this point. (Furthermore, in my opinion, *Scanner* and *Console* still don't get things quite right. Nevertheless, I will introduce *Scanner* briefly at the end of this section, in case you want to start using it now.)

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 quite well 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. For displaying output to the user, you can use either `System.out` or `TextIO`, 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. That value will then 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 subroutine 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 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 `getlnWord()` 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 had 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.putf("%1.2f",amount);` and you could say `TextIO.putf("%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"—that is, base-10—numbers. You can replace the "d" with 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.

For numeric output, the format specifier can include a comma (","), which will cause the digits of the number to be separated into groups, to make it easier to read big numbers. In the United States, groups of three digits are separated by commas. For example, if `x` is one billion, then `System.out.printf("%,d",x)` will output 1,000,000,000. In other countries, the separator character and the number of digits per group might be different. The comma should come at the beginning of the format specifier, before the field width; for example: `%,12.3f`.

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()`. 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 it 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, and the
                  // comma in %,1.2f adds separators between groups of digits.

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

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