[Previous Section | Next Section | Chapter Index | Main Index]

Section 2.4

Text Input and Output

Subsections

WE HAVE SEEN THAT IT IS VERY EASY to display text to the user with the functions `System.out.print` and `System.out.println`. But there is more to say on the topic of outputting text. Furthermore, most programs use data that is input to the program at run time, so you need to know how to do input as well as output. This section explains how to get data from the user, and it covers output in more detail than we have seen so far. It also has a section on using files for input and output.

2.4.1 Basic Output and Formatted Output

The most basic output function is `System.out.print(x)`, where `x` can be a value or expression of any type. If the parameter, `x`, is not already a string, it is converted to a value of type String, and the string is then output to the destination called standard output. (Generally, this means that the string is displayed to the user; however, in GUI programs, it outputs to a place where a typical user is unlikely to see it. Furthermore, standard output can be "redirected" to write to a different output destination. Nevertheless, for the type of program that we are working with now, the purpose of `System.out` is to display text to the user.)

`System.out.println(x)` outputs the same text as `System.out.print`, but it follows that text by a line feed, which means that any subsequent output will be on the next line. It is possible to use this function with no parameter, `System.out.println()`, which outputs nothing but a line feed. Note that `System.out.println(x)` is equivalent to

```
System.out.print(x);
System.out.println();
```

You might have noticed that `System.out.print` outputs real numbers with as many digits after the decimal point as necessary, so that for example π is output as 3.141592653589793, and numbers that are supposed to represent money might be output as 1050.0 or 43.575. You might prefer to have these numbers output as, for example, 3.14159, 1050.00, and 43.58. Java has a "formatted output" capability that makes it easy to control how real numbers and other values are printed. A lot of formatting options are available. I will cover just a few of the simplest and most commonly used possibilities here.

The function `System.out.printf` can be used to produce formatted output. (The name "printf," which stands for "print formatted," is copied from the C and C++ programming languages, where this type of output originated.) `System.out.printf` takes one 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 );
```

The output format of a value is specified by a format specifier. In this example, the format specifier is `%1.2f`. 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." A very large 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 "20" in `%20s`, can be added to specify the (minimum) number of characters. The "s" stands for "string," and it can be used for values of type String. It can also be used for values of other types; in that case the value is converted into a String value in the usual way.

The format specifiers for values of type double are more complicated. An "f", as in `%1.2f`, is used to output a number in "floating-point" form, that is with digits after a 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; a "1" in this position 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 exponential form for very small values and very large values and in floating-point form for other 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`. If you want the output to be left-justified instead of right justified, add a minus sign to the beginning of the format specifier: for example, `%-20s`.

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!).

To output a percent sign, use the format specifier `%%` in the format string. You can use `%n` to output a line feed. You can also use a backslash, `\`, as usual in strings to output special characters such as tabs and double quote characters.

2.4.2 A First Text Input Example

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 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 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 ideal for use here at the beginning of this course. Java 6 introduced the Console class for communicating with the user, but Console has its own problems. (It is not always available, and it can only read strings, not numbers.) 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. However, we start with my own version of text input.

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 named TextIO that defines subroutines for reading values typed by the user. 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 has a few other capabilities that I will discuss later in this section.

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 information about how to use TextIO.

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 user must type a number and press return before the program can continue. The value that the user typed 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 that number:

```
/**
 * 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.println();
            System.out.println("The number that you entered was " + userInput);
            System.out.println("The square of that number is " + square);
            System.out.println();

        } // 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. Note that it is good style to output a question or some other prompt to the user before reading input. Otherwise, the user will have no way of knowing exactly what the computer is waiting for, or even that it is waiting for the user to do something.

2.4.3 Basic TextIO Input Functions

TextIO includes a variety of functions for inputting values of various types. Here are the functions 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 `TextIO.getln()` 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.

`TextIO.getln()` does **not** skip blanks or end-of-lines before reading a value. But the input functions `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. When one of these functions skips over an end-of-line, it outputs a '?' to let the user know that more input is expected.

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, even if they had typed more than one value on the previous line. 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.

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! Note that this program uses formatted output to print out monetary values in their correct format.

```
/**
 * 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.

       System.out.print("Enter the initial investment: ");
       principal = TextIO.getlnDouble();

       System.out.print("Enter the annual interest rate (as a decimal): ");
       rate = TextIO.getlnDouble();

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

       System.out.printf("The amount of interest is $%1.2f%n", interest);
       System.out.printf("The value after one year is $%1.2f%n", principal);

   } // end of main()

} // end of class Interest2
```

(You might be wondering why there is only one output routine, `System.out.println`, which can output data values of any type, while there is a separate input routine for each data type. For the output function, the computer can tell what type of value is being output by looking at the parameter.

However, the input routines don't have parameters, so the different input routines can only be distinguished by having different names.)

2.4.4 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. Similarly, `System.in` has only one possible source for input data.

TextIO has the ability to write data to files and to read data from files. TextIO includes output functions `TextIO.put`, `TextIO.putln`, and `TextIO.putf`. Ordinarily, these functions work exactly like `System.out.print`, `System.out.println`, and `System.out.printf` and are interchangeable with them. However, they can also be used to output text to files and to other destinations.

When you write output using `TextIO.put`, `TextIO.putln`, or `TextIO.putf`, the output is sent to the current output destination. By default, the current output destination is standard output. However, TextIO has 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 will be created if it does not already exist. Note that if a file with the same name already exists, its previous contents will be erased without any warning!

When you call `TextIO.writeFile`, TextIO remembers the file and automatically sends any output from `TextIO.put` or other output functions to that file. If you want to go back to writing to standard output, you can call

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

Here is a simple program that asks the user some questions and outputs the user's responses to a file named "profile.txt." As an example, it uses TextIO for output to standard output as well as to the file, but `System.out` could also have been used for the output to standard output.

```
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 file
            TextIO.putln("Name:            " + name);
            TextIO.putln("Email:           " + email);
            TextIO.putln("Favorite Color:  " + favColor);
            TextIO.putf( "Yearly Income:   %,1.2f%n", salary);

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

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

        }

    }
```

In many cases, you want to let the user select the file that will be used for output. You could ask the user to type in the file name, but that is error-prone, and users are more familiar with selecting a file from a file dialog box. The statement

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

will open a typical graphical-user-interface file selection dialog where the user can specify the output file. This also has the advantage of alerting the user if they are about to replace an existing file. It is possible for the user to cancel the dialog box without selecting a file. `TextIO.writeUserSelectedFile` is a function that returns a boolean value. The return value is `true` if the user selected a file, and is `false` if the user canceled the dialog box. Your program can check the return value if it needs to know whether it is actually going to write to a file or not.

TextIO can also read from files, as an alternative to reading from standard input. You can specify an 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 with a GUI-style dialog box by saying `TextIO.readUserSelectedFile()`. After you have done this, any input will come from the file instead of being typed by the user. You can go back to reading the user's 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 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.

2.4.5 Other TextIO Features

The TextIO input functions that we have seen so far can only read one value from a line of input. Sometimes, however, you do want to read more than one value from the same line of input. For example, you might want the user to be able to type something like "42 17" to input the two numbers 42 and 17 on the same line. 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 consume 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.

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 ints and the user types "42,17", 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*.

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.

2.4.6 Using Scanner for Input

TextIO makes it easy to get input from the user. However, since it is not a standard class, you have to remember to make `TextIO.java` available to any program that uses it. Another option for input is the

Scanner class. One advantage of using Scanner is that it's a standard part of Java and so is always there when you want it.

It's not that hard to use a Scanner for user input, and it has some nice features, but using it requires some syntax that will not be introduced until Chapter 4 and Chapter 5. I'll tell you how to do it here, without explaining why it works. You won't understand all the syntax at this point. (Scanners will be covered in more detail in Subsection 11.1.5.)

First, you should add the following line to your program at the beginning of the source code file, **before** the "public class...":

```
import java.util.Scanner;
```

Then include the following statement at the beginning of your `main()` routine:

```
Scanner stdin = new Scanner( System.in );
```

This creates a variable named `stdin` of type Scanner. (You can use a different name for the variable if you want; "stdin" stands for "standard input.") You can then use `stdin` in your program to access a variety of subroutines for reading user input. For example, the function `stdin.nextInt()` reads one value of type int from the user and returns it. It is almost the same as `TextIO.getInt()` except for two things: If the value entered by the user is not a legal int, then `stdin.nextInt()` will crash rather than prompt the user to re-enter the value. And the integer entered by the user must be followed by a blank space or by an end-of-line, whereas `TextIO.getInt()` will stop reading at any character that is not a digit.

There are corresponding methods for reading other types of data, including `stdin.nextDouble()`, `stdin.nextLong()`, and `stdin.nextBoolean()`. (`stdin.nextBoolean()` will only accept "true" or "false" as input.) These subroutines can read more than one value from a line, so they are more similar to the "get" versions of TextIO subroutines rather than the "getln" versions. The method `stdin.nextLine()` is equivalent to `TextIO.getln()`, and `stdin.next()`, like `TextIO.getWord()`, returns a string of non-blank characters.

As a simple example, here is a version of the sample program *Interest2.java* that uses Scanner instead of TextIO for user input:

```
import java.util.Scanner;  // Make the Scanner class available.

public class Interest2WithScanner {

   public static void main(String[] args) {

      Scanner stdin = new Scanner( System.in );  // Create the Scanner.

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

      System.out.print("Enter the initial investment: ");
      principal = stdin.nextDouble();

      System.out.print("Enter the annual interest rate (as a decimal): ");
      rate = stdin.nextDouble();

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

      System.out.printf("The amount of interest is $%1.2f%n", interest);
      System.out.printf("The value after one year is $%1.2f%n", principal);
```

```
        } // end of main()

    } // end of class Interest2With Scanner
```

Note the inclusion of the two lines given above to import `Scanner` and create `stdin`. Also note the substitution of `stdin.nextDouble()` for `TextIO.getlnDouble()`. (In fact, `stdin.nextDouble()` is really equivalent to `TextIO.getDouble()` rather than to the "getln" version, but this will not affect the behavior of the program as long as the user types just one number on each line of input.)

I will continue to use TextIO for input for the time being, but I will give a few more examples of using Scanner in the on-line solutions to the end-of-chapter exercises. There will be more detailed coverage of Scanner later in the book.

[Previous Section | Next Section | Chapter Index | Main Index]