FORMATTING OUTPUT WITH printf

Starting with version 5.0, Java includes a method named printf that can be used to give output in a specific format. The method printf is used the same way as the method print but allows you to add formatting instructions that specify such things as the number of digits to include after a decimal point. For example, consider the following:

```java
double price = 19.8;
System.out.print("$");
System.out.printf("%.2f", price);
System.out.println(" each");
```

This code outputs the following line:

```
$ 19.80 each
```

The line

```java
System.out.printf("%.2f", price);
```

outputs the string " 19.80" (one blank followed by 19.80), which is the value of the variable price written in the format %.2f. In these simple examples, the first argument to printf is a string known as the format specifier, and the second argument is the number or other value to be output in that format. Let's explain this first sample format specifier.

The format specifier %.2f says to output a floating-point number in a field (number of spaces) of width 6 (room for six characters) and to show exactly two digits after the decimal point. So, 19.8 is expressed as "19.80" in a field of width 6. Because "19.80" has only five characters, a blank character is added to obtain the 6-character string " 19.80". Any extra blank space is added to the front (left-hand end) of the value output. That explains the 6.2 in the format specifier %.2f. The f means the output is a floating-point number, that is, a number with a decimal point. We will have more to say about the character % shortly, but among other things, it indicates that a format specification (in this case, 6.2f) follows.

Before we go on, let's note a few details about the method printf. Note that the first argument is a string that gives a format specifier. Also, note that printf, like print, does not advance the output to the next line. The method printf is like print, not like println, in this regard.
The first argument to printf can include text as well as a format specifier. For example, consider the following variant on our example:

```java
double price = 19.8;
System.out.printf("%6.2f each", price);
System.out.println();
```

This code also outputs the following line:

```
$ 19.88 each
```

The text before and after the format specifier %6.2f is output along with the formatted number. The character % signals the end of text to output and the start of the format specifier. The end of a format specifier is indicted by a conversion character (f in our example).

Other possible format specifiers are described in Display 2.1. (A more complete list of specifiers is given in Appendix 4.) The conversion character specifies the type of value that is output in the specified format. Note that the first number specifies the total number of spaces used to output the value. If that number is larger than the number of spaces needed for the output, extra blanks are added to the beginning of the value output. If that number is smaller than the number of spaces needed for the output, enough extra space is added to allow the value to be output; no matter what field width is specified, printf uses enough space to fit in the entire value output. Both of the numbers in a format specifier such as %6.2f are optional. You may omit either or both numbers, in which case Java chooses an appropriate default value or values. For example, %6f and %.2f. Note that the dot goes with the second number. You can use a format specifier that is just a % followed by a conversion character, such as %f or %g, in which case Java decides on the format details for you. For example, the format specifier %f is equivalent to %.6f, meaning six spaces after the decimal point and no extra space around the output.

The e and g format specifiers are partially explained in Display 2.1. We still need to explain the meaning of the number after the decimals point in e and g format specifiers, such as %8.3e and %8.3g. The first number, 8 in the examples, is the total field width for the value output. The second number (the one after the decimal point) specifies the number of digits after the decimal point of the output. So the numbers, such as 8.3, have the same meaning in the f, e, and g formats.

The s and c formats, for strings and characters, may include one number that specifies the field width for outputting the value, such as %15s and %2c. If no number is given, the value is output with no leading or trailing blank space.

When the value output does not fill the field width specified, blanks are added in front of the value output. The output is then said to be right justified. If you add a
Hyphen after the %, any extra blank space is placed after the value output, and the output is said to be left justified. For example, the lines

```java
double value = 12.123;
System.out.printf("Start%8.2fEnd", value);
System.out.println();
System.out.printf("Start%8.2fEnd", value);
System.out.println();
```

produce the following output. The first line has three spaces before the 12.12 and the second has three spaces after the 12.12.

```
Start  12.12End
Start12.12   End
```

So far we have used printf to output only one value. However, printf can output any number of values. The first argument always is a string known as the format string, which can be followed with any number of additional arguments, each of which is a value to output. The format string should include one format specifier, such as %6.2f or %s, for each value output, and they should be in the same order as the values to be output. For example:

```java
double price = 19.8;
String name = "magic apple";
System.out.printf("%6.2f for each %s.", price, name);
System.out.println();
System.out.println("Wow");
```

Display 2.1 Format Specifiers for System.out.printf

<table>
<thead>
<tr>
<th>CONVERSION CHARACTER</th>
<th>TYPE OF OUTPUT</th>
<th>EXAMPLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>Decimal (ordinary) integer</td>
<td>%5d  %d</td>
</tr>
<tr>
<td>f</td>
<td>Fixed-point (everyday notation) floating point</td>
<td>%6.2f %f</td>
</tr>
<tr>
<td>e</td>
<td>E-notation floating point</td>
<td>%8.3e %e</td>
</tr>
<tr>
<td>g</td>
<td>General floating point (Java decides whether to use E-notation or not)</td>
<td>%8.3g %g</td>
</tr>
<tr>
<td>s</td>
<td>String</td>
<td>%12s  %s</td>
</tr>
<tr>
<td>c</td>
<td>Character</td>
<td>%2c  %c</td>
</tr>
</tbody>
</table>
This code outputs the following:

\$19.80 for each magic apple.

Wow

Note that the format string may contain text as well as format specifiers, and this text is output along with the values of the other arguments to printf.

You can include line breaks in a format string. For example, the two lines

\begin{lstlisting}
System.out.printf("$%.2f for each %s.\n", price, name);
System.out.println();
\end{lstlisting}

can be replaced by the single line below, which uses the escape sequence \n:

\begin{lstlisting}
System.out.printf("$%.2f for each %s.\n", price, name);
\end{lstlisting}

Although it is legal to use the escape sequence \n to indicate a line break in a format string, it is preferable to use %n. Exactly what happens when a \n is output can be system-dependent, whereas %n should always mean a simple new line on any system. So our last line of code would be a little more robust if rewritten using %n as follows:

\begin{lstlisting}
System.out.printf("$%.2f for each %s.%n", price, name);
\end{lstlisting}

Many of the details we have discussed about printf are illustrated in the program given in Display 2.2.

\begin{tframe}[TIP]
\begin{description}
\item[\textbf{Formatting Money Amounts with printf}] \end{description}
\end{tframe}

A good format specifier for outputting an amount of money stored as a value of type double (or other floating-point value) is %.2f. It says to include exactly two digits after the decimal point and to use the smallest field width that the value will fit into. For example,

\begin{lstlisting}
  double price = 19.99;
  System.out.printf("The price is $%.2f each.", price);
\end{lstlisting}

produces the output:

The price is $19.99 each.
public class PrintfDemo
{
    public static void main(String[] args)
    {
        String aString = "abc";
        System.out.println("String output: ");
        System.out.println("START1234567890 ");
        System.out.printf("START%5sEND \%n", aString);
        System.out.printf("START%4sEND \%n", aString);
        System.out.printf("START%2sEND \%n", aString);
        System.out.println();

        char oneCharacter = 'Z';
        System.out.println("Character output: ");
        System.out.println("START1234567890 ");
        System.out.printf("START%5cEND \%n", oneCharacter);
        System.out.printf("START%4cEND \%n", oneCharacter);
        System.out.println();

        double d = 12345.123456789;
        System.out.println("Floating-point output: ");
        System.out.println("START1234567890 ");
        System.out.printf("START%5fEND \%n", d);
        System.out.printf("START%4fEND \%n", d);
        System.out.printf("START%2fEND \%n", d);
        System.out.printf("START%12.4fEND \%n", d);
        System.out.printf("START%12.2fEND \%n", d);
        System.out.printf("START%12.5fEND \%n", d);
    }
}
Display 2.2  **The printf Method** (part 2 of 2)

**SAMPLE DIALOGUE**

String output:
START1234567890
STARTabcEnd
STARTabcEnd

The value is always output. If the specified field width is too small, extra space is taken.

Character output:
START1234567890
STARTZEND
START ZEND

Floating-point output:
START1234567890
START12345.123457END
START12345.1235END
START12345.122END
START 12345.1235END
START12345.12e+04END
START 1.23451e+04END

Note that the output is rounded, not truncated, when digits are discarded.

**TIP**

**Legacy Code**

Some code is so expensive to replace that it is used even if it is "old fashioned" or otherwise less than ideal. This sort of code is called legacy code. One approach to legacy code is to translate it into a more modern language. The Java method `printf` is essentially the same as a function\(^1\) in the C language that is also named `printf`. This was done intentionally so that it would be easier to translate C code into Java code.

---

\(^1\) Methods are called *functions* in the C language.