06.02 Virtual Lecture Notes (Part 2)

Formatting Integer Output

The following example illustrates how to format a numeric integer value with **`printf()`**.

```
System.out.printf("%d%n", 16934);
System.out.printf("%d%n", 12);
System.out.printf("%d%n ", -301);
System.out.printf(" %d%n ", 7432);
```

Notice that quotation marks only surround the format string inside the parentheses, not the argument to be formatted.

As usual, the format string begins with a percent sign (%). The conversion character "d" (i.e., decimal integer) is used to format an argument of type int, byte, short, and long. This will be confusing, since you probably associate "d" with doubles.

Download the FormattingNumericValues.java file into a BlueJ project folder. Open the file and notice that most of the code has been commented out. As you study the examples in the lesson, uncomment the appropriate lines to see how each format string works with the **`printf( )`** method. A simple scale is provided to help you visualize justification and field width.

Run the program and ask yourself the following questions:

- What happens if you remove the escape characters from the format string?
- What happens if you forget to include the conversion character ("d")?
- Are the numbers right or left justified?
- What is the effect of adding a space within the format string (as in the last example above)?

The output of these four lines of code shows how easy it is to format integers, but it also illustrates a problem: the numbers do not line up nicely. This is easy to fix, because, as with Strings, the width of the output field for numeric values can be specified. Uncomment the following lines of code in the demo program. In this segment, variables are used for the argument instead of actual values. Run the program again and notice the difference.

```
System.out.printf("%10d%n", intArgument1);
System.out.printf("%10d%n", intArgument2);
System.out.printf("%10d%n", intArgument3);
System.out.printf("%10d%n", intArgument4);
System.out.printf("%+10d%n", intArgument4);
```

It is good programming practice to always include a field width when formatting numeric values, to avoid alignment problems. Adding a “10” to the format string indicates that each argument will be printed in a field 10 characters wide. This right justifies and correctly aligns the numbers. Notice that the minus sign is automatically handled and you can use the plus (+) sign as a flag to designate positive numbers if necessary.

Experiment with the **`printf( )`** statement by changing the field widths and values of arguments. For example, what happens if you try to print a number that is larger than the assigned field width? There are also numeric conversion flags for left justification, padding values with leading zeroes, and enclosing negative numbers in parentheses. You can find further information about conversion flags in the Java API.

Instead of printing values with separate print statements, you may want to format multiple values in the same statement. Uncomment the following line to see how easily this can be done.

```
  System.out.printf("%10d%7d%,15d\n",-25,4963,7894502);
```

Formatting Decimal Output

So far we have dealt with formatting integers, but what about decimal numbers? Double values are just as easy to handle. Uncomment the following lines of code, then run the program and observe the output in relation to each format string.

```
System.out.printf("%f\n",3.141592654);
System.out.printf("%9f\n",3.141592654);
```

As you have seen previously, all format strings are written within quotation marks and begin with a percent sign (%). The letter “f” is the conversion character for a double, float, or BigDecimal argument.

It would appear that these two lines print the value of pi to nine decimal places, but that is not what the output reveals. (Be sure to use the scale to help count the number of positions printed.)

The format string of the first statement does not contain a field width, while the second statement specifies a field width of nine. When the format string does not indicate a field width, only six decimal places are printed by default. Although the second statement specified a field width of nine, the entire number did not print. However, count how many positions were printed, including the leading space and the decimal point. The value was truncated to fit within the space allocated by the format string.

It is possible to control the number of decimal places printed, by specifying the precision of the output. This is illustrated in the following two lines by the decimal point and number following the field width:

```
System.out.printf("%12.9f\n",3.141592654);
System.out.printf("%15.4f\n",3.141592654);
```

In the first line, pi is printed with nine decimal places in a field width of 12. In the second line, pi is printed in a field width of 15, put with a precision of only four decimal positions. Uncomment these lines in the demo program and observe the resulting output. Be sure to experiment with other numbers, field widths, and precisions until you understand how to write format strings for decimal numbers.

You have covered a lot of detail in this lesson. Only with practice will you become comfortable with reading input from the keyboard and formatting output to the screen. Formatting is a detail that is easy to ignore, but poorly-formatted program output is difficult to read, and generation of sloppy output is regarded as bad programming practice. You may find that formatting is something you save to do once a program compiles with no errors and runs correctly; however, never submit a program that is poorly formatted. And finally, we have not covered formatting dates, time, or currency. Consult the Java API if you have a need to format these types of output.