Formatted Output

Now that we are comfortable with defining methods, we can think about packaging useful operations into methods and working them into Java programs. In this section, we'll examine some custom methods to assist in generating formatted output.

The standard library for C programming language includes a function known as `printf()`, for "print formatted". `printf()` receives a series of variables and a format string as arguments. The format string uses a custom syntax to specify how each variable should appear (i.e., the field width, the alignment within the field, and the type of padding). Unfortunately, an equivalent method does not exist in the Java API. That's OK, because we can "roll our own". Although we'll not attempt a full implementation of a `printf`-equivalent method, we can craft a few simple methods to help generate nicely formatted output.

As a first cut, let's examine how to control the format for integers printed on the standard output. The program `TableOfCubes` outputs a table of cubes for the integers 1 through 10 (see Figure 1).

```
 1  import java.io.*;
 2
 3  public class TableOfCubes
 4  {
 5     public static void main(String[] args)
 6     {
 7        System.out.print("=============\n"
 8                       +  " x  Cube of x\n"
 9                       +  "--- ---------\n");
10        int x = 0;
11        String s1;
12        String s2;
13        while (x <= 10)
14        {
15           s1 = formatInt(x, 3);
16           s2 = formatInt(x * x * x, 10);
17           System.out.println(s1 + s2);
18           x++;
19        }
20        System.out.println("=============\n");
21     }
22
23     // format int 'i' as a string of length 'len' (pad left with spaces)
24     public static String formatInt(int i, int len)
25     {
26        String s = "" + i;
27        while (s.length() < len)
28           s = " " + s;
29        return s;
30     }
31  }
```

Figure 1. `TableOfCubes.java`

This program generates the following output:

```
=============
 x  Cube of x
--- ---------
  0         0
  1         1
  2         8
  3        27
  4        64
  5       125
  6       216
  7       343
  8       512
  9       729
 10      1000
=============
```

The output contains two nicely-formatted columns of integers. The method `formatInt()` helps. It is defined in lines 24-30 and is used twice, in lines 15 and 16.

The entire method definition contains only seven lines. As seen in the signature, `formatInt()` receives two integer arguments and returns a reference to a new `String` object. The two arguments are the integer to be formatted, and the minimum length of the string. The first task is to convert the integer to a string. This is performed easily by concatenating the integer with an empty (line 26). Then, a two-line `while` loop pads the string on the left with spaces until the desired length is achieved (lines 27-28). The result is returned to the calling program in line 29.

In the main program `formatInt()` is called twice. In line 15, the arguments `x` and 3 are passed and the `String` reference returned is assigned to `s1`. In line 16, the arguments `x * x * x` and 10 are passed and the `String` reference returned is assigned to `s2`. The two strings are concatenated in the print statement in line 17. The string printed is 13 characters wide, with the original integer right-aligned in the first three positions, and the cube of the integer right-aligned in the next 10 positions.

The situation is more awkward for floating point numbers, since we must deal with padding left with spaces and with truncating decimal places or padding right with zeros. The program `TableOfSquareRoots` illustrates one approach to generating formatted output of floating point numbers (see Figure 2).

```
 1  public class TableOfSquareRoots
 2  {
 3     public static void main(String[] args)
 4     {
 5        System.out.print("====================\n"
 6                         +  " x  Square Root of x\n"
 7                         +  "--- ----------------\n");
 8
 9        for (int x = 0; x <= 200; x += 25)
10        {
11           String s1 = formatInt(x, 3);
12           String s2 = formatDouble(Math.sqrt(x), 17, 4);
13           System.out.println(s1 + s2);
14        }
15        System.out.println("====================\n");
16     }
17
18     // format int 'i' as a string of length 'len' (pad left with spaces)
19     public static String formatInt(int i, int len)
20     {
21        String s = "" + i;
22        while (s.length() < len)
23           s = " " + s;
24        return s;
25     }
26
27     // format double 'd' as a string of length 'len' & 'dp' decimal places
28     public static String formatDouble(double d, int len, int dp)
29     {
30        String fx = (dp <= 0) ? "" + Math.round(d) : "" + trim(d, dp);
31        while (fx.substring(fx.indexOf(".") + 1, fx.length()).length() < dp)
32           fx += "0";
33        while (fx.length() < len)
34           fx = " " + fx;
35        return fx;
36     }
37
38     // trim double 'd' to have 'dp' decimal places
39     public static double trim(double d, int dp)
40     {
41        double factor = Math.pow(10.0, dp);
42        return (int)(d * factor) / factor;
43     }
44  }
```

Figure 2. `TableOfSquareRoots.java`

This program generates the following output:

```
====================
 x  Square Root of x
--- ----------------
  0           0.0000
 25           5.0000
 50           7.0710
 75           8.6602
100          10.0000
125          11.1803
150          12.2474
175          13.2287
200          14.1421
====================
```

Within 44 lines of code, `TableOfSquareRoots` contains a `main()` method and three custom methods: `formatInt()`, `formatDouble()`, and `trim()`. The left-hand column of output displays the integers from 0 to 200 in increments of 25. Each integer appears right-justified in a three-character column. The `formatInt()` method does the work, as in the previous example. The right-hand column displays the square roots of the integers. Each root contains four decimal places of precision and appears right-justified in a 17-character column. The formatted strings for the integers and the square roots of the integers are generated in the `main()` method in lines 11 and 12, respectively.

Note in the call to the `formatDouble()` method in line 12 that three arguments are passed. The first is the `double` variable to format into a string. The expression `Math.sqrt(x)` appears; so, the value to format is the square root of the integer. The second argument is 17, which is the minimum length of the string, and the third argument is 4, which is the number of decimal places of precision.

The definition of `formatDouble()`appears in lines 28-36. The string that is eventually returned, `fx`, is declared and initialized in line 30. The assignment uses a selection operator, expanded as follows:

```
if (dp <= 0)
   fx = "" + Math.round(d);
else
   fx = "" + trim(d, dp);
```

So, if the method is called specifying less than one decimal place of precision, `fx` is initialized with a string representation of the integer returned by `Math.round(d)`. This is the closest integer to the double argument `d`. If the method is called specifying one or more decimal places of precision, `fx` is initialized with a string representation of the value returned by `trim(d, dp)`. The `trim()` method returns a `double` equal to `d` with at most `dp` decimal places (see lines 39-43). Lines 31-32 pad `fx` to the right with zeros in the event fewer than `dp` decimal places are present. Lines 33-34 pad `fx` to the left with spaces to fill the field width as specified in the second argument. The final value of `fx` is returned in line 35.

There are a number of deficiencies in the way formatting is performed in `TableOfSquareRoots`. The use of the `pow()` method to calculate a trimming factor in line 41 is overkill, and the technique for timing in line 42 truncates rather than rounds. We'll leave it to you to explore ways to improve these methods.