Lesson 29: Formatting (rounding-off)

One method of rounding off is to use the *NumberFormat* (requires *import java.text.*;*) class to create a *String*, and then convert that *String* back into the desired primitive number type.

A "rounding-off" example:

For example, to round 34.982665 to the nearest thousandths:

```
double d = 34.982665;

NumberFormat fmt = NumberFormat.getNumberInstance( );
fmt.setMaximumFractionDigits(3);
fmt.setMinimumFractionDigits(3);

String s = fmt.format(d);
System.out.println(s); //34.983
double d3 = Double.parseDouble(s); //Convert to a double for demo purposes
System.out.println(d3); //34.983 …Perfect, just the answer we expected!
```

Analyzing the details:

Let's examine four important details about the above code:

1. *fmt.setMaximumFractionDigits(3)* gives us **no more** than 3 decimal places… i.e. it rounds off to **3** decimal places.

2. *fmt.setMinimumFractionDigits(3)* **guarantees** at least 3 decimal places. For example, if we round off 34.9997 to 3 decimal places we would ordinarily get 35.0; however, with *setMinimumFractionDigits(3)* we would get 35.000.

3. *fmt.format(d)* returns a ***String***, not a numeric.

4. You may ultimately want a numeric instead of a *String*, hence the *Double.parseDouble(s)* part of the code.

More *NumberFormat* objects:

Notice above that the way we create a *NumberFormat* object is by calling the *getNumberInstance static* method. It returns a *NumberFormat* object. There are two other similar methods that return *NumberFormat* objects. These are detailed below along with sample usage.

1. The object returned by *getCurrencyInstance* is used for formatting money.

```
NumberFormat nf = NumberFormat.getCurrencyInstance( );
String str = nf.format(81.09745);
System.out.println(str); //$81.10
```

```
str = nf.format(.358);
System.out.println(str); //$.36
```

2. The object returned by *getPercentInstance* is used for formatting percents. The number to be formatted is multiplied by 100 and then a percent sign is appended. The settings determined by *setMinimumFractionDigits( )* and *setMaximumFractionDigits( )* are applied **after** multiplication by 100. If these methods are not specifically called, then their settings are automatically 0.

```
NumberFormat nf = NumberFormat.getPercentInstance( );
nf.setMinimumFractionDigits(2);
nf.setMaximumFractionDigits(2);
String str = nf.format(.35838);
System.out.println(str); //35.84%
```

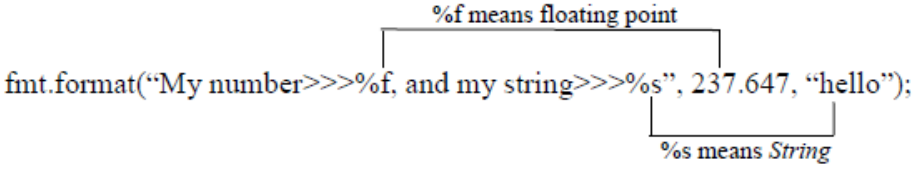
For even more formatting flexibility use the *DecimalFormat* class. Its usage is detailed in Appendix Z. There, you will learn, for example, to specify a pattern like "#,###.000", and then format a number like 3847.2 as 3,847.200.

The Formatter Class:

The *Formatter* class (new to Java 5.0) is used to format numbers (and other data, too) and to produce *String*s containing the formatted data. Begin the process by creating a *Formatter* object:

```
Formatter fmt = new Formatter( );
```

Actual formatting is done with the *format* method. It has several parameters, the first of which is a *String* with embedded **format specifiers**. This is followed by a corresponding sequence of data to be formatted. The **sequence** of embedded specifiers matches the **sequence** of data parameters as illustrated by the following example:



Finally, produce the formatted *String* with *fmt.toString( )* and get:

```
My number>>>237.647000, and my string>>>hello
```

In this example we formatted a floating-point number using **%f** and a *String* using **%s**. See Appendix AD for other format specifiers.

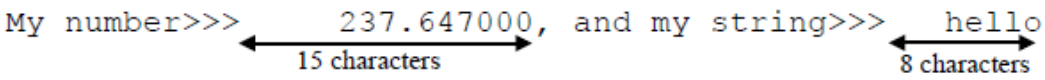
Minimum Field Width:

The output above is especially useful when we are able to specify a field width as in the next example where we allocate a width of 15 characters to the number and a width of 8

characters to the *String*. Each field is padded with spaces to insure it occupies the specified number of characters. If the *String* or number is longer than the setting, it will still be printed in its entirety.

fmt.format("My number>>>%15f, and my string>>>%8s", 237.647, "hello");

A subsequent application of *fmt.toString( )* will yield (notice the padding with spaces):



The ability to set field widths is especially useful in the printing of tables since it helps keep columns aligned.

Precision:

Next, we examine the notion of precision (typically, the number of decimal places). The precision specifier follows the minimum field width specifier (if there is one) and consists of a period followed by an integer. It can be applied to **%f**, **%e**, **%g**, or **%s**. The default precision for numerics is 6 decimal places.

The following examples show how to use the precision specifier:

1. **Example 1**… **%9.3f**… a decimal floating point number in a field 9 characters wide and having 3 decimal places… 187.9207 formats as " `187.921`"

2. **Example 2**… **%9.2e**… a scientific notation number in a field 9 characters wide and having 2 decimal places… 46238.123 formats as " `4.62e+04`"

3. **Example 3**… **%.4g**… either a decimal floating number or scientific notation (whichever is shorter) having no minimum field width and having 4 **significant digits**…. 187.0853211 formats as "`187.1`"

4. **Example 4**… **%6.8s**… displays a *String* of at least 6 but not exceeding 8 characters long. If the *String* is longer than the maximum, characters toward the end of the *String* will be truncated… "abc" formats as " `abc`"; "123456789A" formats as "`12345678`"

Format Flags:

It is possible to use special format flags to control various aspects of formatting. These flags **immediately** follow the **%**. Some of the more often used flags are detailed here (see Appendix AD for a more complete list):

- - Left justification… **%-9.2f**… 72.45822 formats as "`72.46 `"

- 0 Pad with zeros instead of the default spaces… **%09.2f**… 72.45822 formats as "`000072.46`"

- , Numeric values include grouping separators… **%-,10.2f**…1726.46 formats as "`1,726.46 `"

It is possible to pass a *Formatter* object as an argument to the *println* method where its *toString* method is **automatically** called with *System.out.println(fmt);*

An even handier shortcut is to dispense with the *Formatter* object entirely and use the *printf* method (new to Java 5.0). The parameters in the example below are **exactly** the same as for *Formatter*.

> System.out.printf("One number, %0,10.2f, followed by another, %-9e, %s", 1267.657, 56.71, "number.");

The output is:

```
One number, 001,267.66, followed by another, 5.671000e+01, number.
```