FORMATTING OUTPUT IN JAVA

THE *DecimalFormat* CLASS

When you are outputting data, particularly numbers, to the console screen, it is often necessary to format that data. The **DecimalFormat** class (which is part of the **java.text** package) is one way to format numbers.

To format a number using the **DecimalFormat** class, a number of steps are required. In the following example, we will format a number to two decimal places.

1. Import the **java.text.DecimalFormat** class.

   ```
   import java.text.DecimalFormat;
   ```

2. Now you must create a **DecimalFormat** object to format the text according to a pattern that you specify.

   ```
   // Declare and initialize DecimalFormat object
   DecimalFormat df = new DecimalFormat(".00");
   ```

   df is the name given to the **DecimalFormat** object we have created
   # is a placeholder object that will be removed if there is not a digit at that location
   0 is a placeholder that will show up as zero if a digit is not found at that location

3. Now we can use the **DecimalFormat** object to format your number.

   ```
   System.out.println(df.format(234.5678));
   ```

   The above expression outputs 234.57.

   The format pattern #.00 asks that a number be converted into four characters – i.e. one digit to the left of the decimal separator, a decimal separator, and two digits on the right. Keep in mind, however, that all the digits that make up the integer part of the number are outputted regardless of the format pattern.

If I wanted to format the number to zero decimal places, my **DecimalFormat** object would be created as follows:

```
DecimalFormat df = new DecimalFormat("#");
System.out.println(df.format(234.5678));
```

The above program would output the number 234.5678 would be outputted as 235.

THOUSANDS SEPARATORS

The format pattern to the left of the decimal separator can include commas to show thousands-separators. In the following example, the number 123456.555 will be outputted as 123,456.56:

```
DecimalFormat df = new DecimalFormat(",###.00");
System.out.println(df.format(123456.555));
```

CURRENCY VALUES

To format numbers to currency, you would just need to simply add a $ dollar sign as follows:

```
DecimalFormat df = new DecimalFormat("$,###.00");
System.out.println(df.format(25000.789));
```

The above program formats the number 25000.789 to $25,000.79. If I wanted to format the number .79 using the above **DecimalFormat** object, I would end up with $.79. If, however, I wanted a zero to appear before the decimal in cases where there is no value before the zero, I would just have to place a zero before the decimal in the constructor as follows:

```
DecimalFormat df = new DecimalFormat("$,##0.00");
```

PERCENTAGE VALUES

The **DecimalFormat** object also allows you to format numbers to a percentage as follows:

```
DecimalFormat df = new DecimalFormat("#.0%");
System.out.println(df.format(0.8816));
```

The above program formats the number 0.8816 to 88.2% by multiplying the number by 100, formatting it to one decimal place and appending a percentage symbol at the end.

THE *NumberFormat* CLASS

The **NumberFormat** class, which is also part of the **java.text** package, can also be used to create objects that format numbers to currency, percentage, etc.

The following Java program formats currency values, integers, decimals and percentages.

```
import java.text.NumberFormat;

public class FormattingOutput {

   public static void main(String[] args) {

      // Declare variable
      double num = 1253.8945

      // Declare and initialize NumberFormat objects
      NumberFormat currency = NumberFormat.getCurrencyInstance();
      NumberFormat integer = NumberFormat.getIntegerInstance();
      NumberFormat number = NumberFormat.getNumberInstance();
      NumberFormat percent = NumberFormat.getPercentInstance();

      // Outputs $1,253.89
      System.out.println(currency.format(num));

      // Outputs 1,254
      System.out.println(integer.format(num));

      // Outputs 1,253.894
      System.out.println(number.format(num));

      // Outputs 59%
      System.out.println(percent.format(num));
   }
}
```

The following are a list of the methods included in the **NumberFormat** class:

METHOD	DESCRIPTION
getCurrencyInstance()	Formats numbers to currency values.
getIntegerInstance()	Formats numbers to integers (no decimal places).
getNumberInstance()	Formats numbers to the default number of decimal places (usually 3 decimal places but depends on the locale's default format settings).
getPercentInstance()	Formats numbers to a percentage value.

When using any of the above methods, you may want to specify the number of decimal places by using the **setMinimumFractionDigits()** method, which will set the minimum number of decimal places, or the **setMaximumFractionDigits()** method, which will set the maximum number of decimal places.

The following line of code formats the number 1234.5678 to 2 decimal places:

```
NumberFormat nf = NumberFormat.getNumberInstance();
nf.setMaximumFractionDigits(2);
System.out.println(nf.format(1234.5678));
```

This will result in the number 1,234.57.

THE *Math* CLASS

The **Math** class is another way you can format numbers by rounding down to the nearest integer, rounding up, or just a straight rounding off to the nearest integer.

ROUNDING UP

E.G. `System.out.println(Math.ceil(1234.567));` **`// Outputs 1235.0`**

ROUNDING DOWN

E.G. `System.out.println(Math.floor(1234.567));` **`// Outputs 1234.0`**

ROUNDING

E.G. `System.out.println(Math.round(1234.567));` **`// Outputs 1235`**

Notice that **Math.ceil** and **Math.floor** returns a floating-point value whereas **Math.round** returns an integer value.

THE *BigDecimal* CLASS

Although the **Math** class allows you to round, round up and round down, it only does so to the nearest integer. But what if we want to round up or round down a number to specific number of decimal places? For example, what if we wanted to round 5.6789 to two decimal places but we want to round down? The **Math** class does not provide us with that option.

The **BigDecimal** class, on the other hand, does provide us with the ability to round floating-point numbers to whatever number of decimals while providing us with the ability to set the rounding behavior.

When using the **BigDecimal** class, we need to do two things:

1. Specify a scale, which represents the number of digits after the decimal place.
2. Specify a rounding method

So let's say, for example we have double variable that equals 5.6789:

```
double num = 5.6789;
```

If I want to format this number to two decimal places and round down, we could create a **BigDecimal** object and use the **setScale()** method included in that class to specify the scale and the rounding method:

```
BigDecimal bd = new BigDecimal(num).setScale(2,
     BigDecimal.ROUND_DOWN);
```

What the above line of code does is it creates a **BigDecimal** object that is made up of our number stored in our variable called **num** (i.e. 5.6789) rounded down to two decimal places.

So, if we were to output our **BigDecimal** number, it would look like this: **5.67**.

There are four (4) common choices for rounding mode:

BigDecimal.ROUND_CEILING:	Rounds up
BigDecimal.ROUND_FLOOR:	Rounds down
BigDecimal.ROUND_HALF_UP:	Rounds up if decimal >= 5
BigDecimal.ROUND_HALF_DOWN:	Rounds up if decimal > 5

If you want to store a **BigDecimal** object in a double variable, you can use the **doubleValue()** method included in the **BigDecimal** class. In the following example, we will use a **BigDecimal** object to round a decimal value to two decimal places and store it in a double variable:

```
double num = new BigDecimal(25.67534).setScale(2,
    BigDecimal.ROUND_HALF_UP).doubleValue();
```

This line of code will store 25.68 in the variable **num**.

CONVERTING DATA TYPES

Sometimes you might find it necessary to change information stored in one format to another format. The following are some of the ways in which you can convert information from one data type to another:

ORIGINAL DATA TYPE	CONVERTED DATA TYPE	CODE
int	String	String x = Integer.toString(num);
double	String	String x = Double.toString(num);
long	String	String x = Long.toString(num);
String	int	int x = Integer.parseInt(str);
String	double	double x = Double.parseDouble(str);
String	long	long x = Long.parseLong(str);

CASTING NUMERIC DATA

The above methods convert Strings to numeric data and vice-versa. But sometimes you may need to convert numeric data of one type to another. For example, you may need to convert a floating-point number (12.5) into an integer (12) or vice-versa. This is called **casting**.

To do this you need to place the data type you want to convert to in parentheses in front of the original data.

E.G.
```
double num = 15.92;
System.out.println((int) num);      // Outputs 15
```

Be mindful of the fact that casting a **double** to an **int** truncates, or removes, the decimal portion of the number. It does not ROUND! Therefore, when decreasing the precision of a number, it is better to round the number (up when the decimal portion is greater than or equal to 0.5, and down when it is less than 0.5).

Here is another example where an integer value is converted into an ASCII character by casting the **int** into a **char**.

E.G.
```
int num = 65;
System.out.println((char) num);     // Outputs 'A'
```