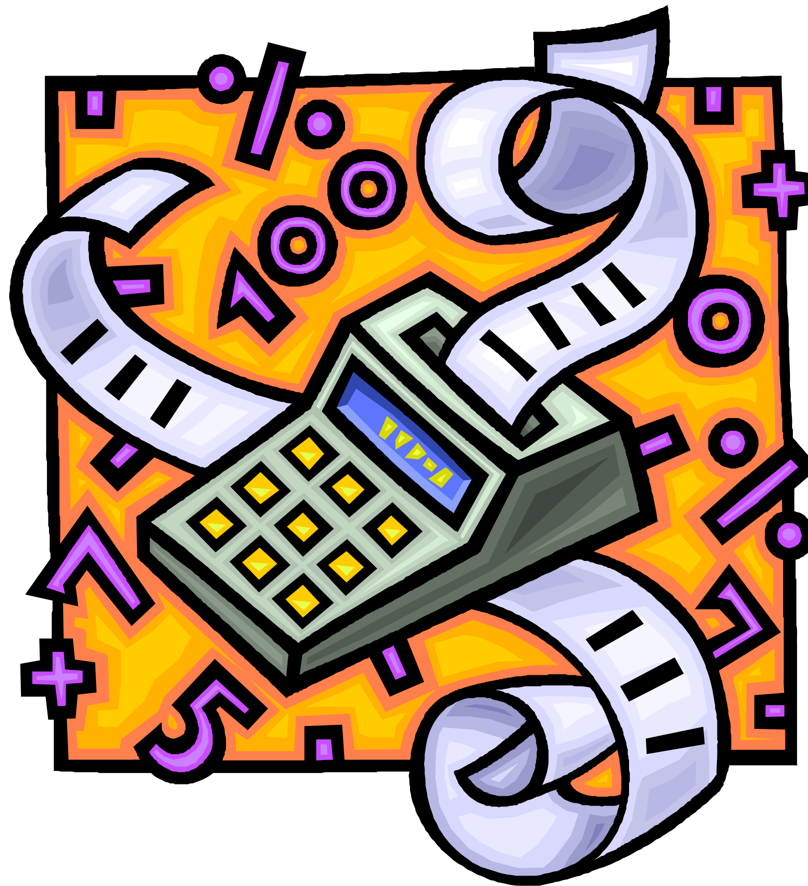
Chapter 5

Calculations, Formatting and Conversions

What is in This Chapter ?

In this chapter we discuss how to do basic math calculations as well as use some readily available **Math** functions in JAVA. We then look at the **String.format()** function as a means of formatting the output of our programs. Lastly, we look at some of the ways in which we can use **typecasting** to convert between various primitive data types as well as to/from Strings.



5.1 Calculations and Formulas

Obviously, a computer can compute solutions to mathematical expressions. We can actually perform simple math expressions such as:

```
30 + 5 * 2 - 18 / 2 – 2
```

In such a math expression, we need to understand the order that these calculations are done in. You may recall from high school the **BEDMAS** memory aid which tells you to perform **B**rackets first, then **E**xponents, then **D**ivision & **M**ultiplication, followed by **A**ddition and **S**ubtraction.

So, for example, in the above JAVA expression, the multiplication ***** operator has preference over the addition **+** operator. In fact, the ***** and **/** operators are evaluated first from left to right and then the **+** and **-**. Thus, the step-by-step evaluation of the expression is:

```
30 + 5 * 2 - 18 / 2 - 2
30 + 10 - 18 / 2 - 2
30 + 10 - 9 - 2
40 - 9 - 2
31 - 2
29
```

We can always add round brackets (called ***parentheses***) to the expression to force a different order of evaluation. Expressions in round brackets are evaluated first (left to right):

```
(30 + 5) * (2 - (18 / 2 - 2))
35 * (2 - (18 / 2 - 2))
35 * (2 - (9 - 2))
35 * (2 - 7)
35 * -5
-175
```

In JAVA, it is good to add round brackets around code when it helps the person reading the program to understand what calculations/operations are done first.

Another operator that is often useful is the ***modulus*** operator which returns the remainder after dividing by a certain value. In JAVA we use the **%** sign as the modulus operator:

```
10 % 2      // results in the remainder after dividing 10 by 2 which is 0
10 % 3      // results in the remainder after dividing 10 by 3 which is 1
10 % 4      // results in the remainder after dividing 10 by 4 which is 2
39 % 20     // results in the remainder after dividing 39 by 20 which is 19
```

Note that using a modulus of 2 will allow you to determine if a number is an odd number or an even number … which may be useful in some applications.

In addition to the standard math operations (i.e., **+**, **-**, *****, **/** and **%**), there are some math operators that can reduce the amount of code that you need to write. For example, the following code outputs the values 8 and 9 to the console window:

```
int     n = 8;

System.out.println(n);

n = n + 1;

System.out.println(n);
```

There is an increment operator called **++** which is a quick way to add 1 to a variable. The following code does the same thing:

```
int     n = 8;

System.out.println(n);

n++;         // same as n = n + 1

System.out.println(n);
```

In fact, this short form of incrementing a variable by 1 is often used within other JAVA expressions. For example, the following produces the same result:

```
int     n = 8;

System.out.println(n++);
System.out.println(n);
```

Here, the **++** is considered a ***post-operator*** in that it increments **n** *after* it is used in the expression (i.e., after it is printed). The ++ can also be used as a ***pre-operator*** by placing it in front of the variable so that the value of the variable is incremented *before* it is used in the expression. Hence, the following code produces the same result of 8 and 9:

```
int     n = 8;

System.out.println(n);
System.out.println(++n);
```

Similarly, there is a **--** post-operator/pre-operator that can be used to *decrement* a variable.

In addition to these operators, there are some binary assignment operators that perform an operation on some numbers and also assign the new value to the variable. For example, consider the following code which outputs 8 and 80 to the console:

```
int     n = 8;

System.out.println(n);

n = n * 10;

System.out.println(n);
```

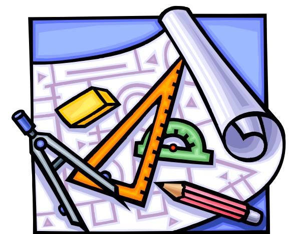
We can replace n = n * 10 with a shorter form which does the same thing as follows:

```
int     n = 8;

System.out.println(n);

n *= 10;         // same as n = n * 10

System.out.println(n);
```

This code multiples **n** by 10 and then puts the result back into **n** again. There are similar binary operators for the other standard operations: **`+=`**, **`-=`**, **`/=`** and **`%=`**.

If you want to do something beyond these simple operations, you may want to look at the **Math** library in JAVA. The following is a list of *some* of the more useful functions in the **Math** library that can be used on numbers:

Trigonometric:

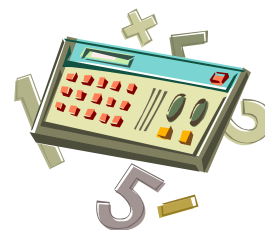


- `Math.sin(0)` `// returns 0.0 which is a double`
- `Math.cos(0)` `// returns 1.0`
- `Math.tan(0.5)` `// returns 0.5463024898437905`
- `Math.PI` `// returns 3.141592653589793`
  (note that **Math.PI** has no brackets … because it is a fixed constant value, not a function)

Conversion and Rounding:

- `Math.round(6.6)` `// returns 7`
- `Math.round(6.3)` `// returns 6`
- `Math.ceil(9.2)` `// returns 10`
- `Math.ceil(-9.8)` `// returns -9`
- `Math.floor(9.2)` `// returns 9`
- `Math.floor(-9.8)` `// returns -10`
- `Math.abs(-7.8)` `// returns 7.8`
- `Math.abs(7.8)` `// returns 7.8`

Powers and Exponents:



- `Math.sqrt(144)              // returns 12.0`
- `Math.pow(5,2)               // returns 25.0`
- `Math.exp(2)                 // returns 7.38905609893065`
- `Math.log(7.38905609893065) // returns 2.0`

Comparison:

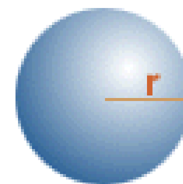
- `Math.max(560, 289)          // returns 560`
- `Math.min(560, 289)          // returns 289`

Generation of a Random Number:

- `Math.random()               // returns a double >=0.0 and <1.0`

These functions are used just as shown above. Notice that we need to write **Math.** in front of all of these functions in order to use them. This is the way we tell JAVA that we want to use the functions in the **Math** library. In fact, **Math** is just one of the many useful pre-defined classes in JAVA. We do not need to import the **Math** class because it is in the **java.lang** package and is thus automatically imported.

As an example, consider how to write a program that computes the volume of a ball (e.g., how much space a ball takes up).



How would we write a JAVA code that computes and displays the volume of such a ball with radius of **25cm** ?

We need to understand the operations. We need to do a division, some multiplications, raise the radius to the power of 3 and we need to know the value of **π** (i.e., pi).

$$\text{Volume} = \frac{4\pi r^3}{3}$$

Here is the simplest, most straight forward solution:

```
int r = 25;
System.out.println((4 * Math.PI * Math.pow(r, 3) / 3));
```

This would also have worked, but requires the radius **r** to be duplicated:

```
System.out.println((4 * Math.PI * (r*r*r) / 3));
```

We could even substitute our own value for **π** :

```
System.out.println((4 * 3.141592653589793 * (r*r*r) / 3));
```

Alternatively, we could have evaluated the 4/3 first:

```
System.out.println((4/3 * Math.PI * Math.pow(r, 3)));
```

Or even pre-compute 4 π /3 (which is roughly 4.188790204786) :

```
System.out.println((4.188790204786 * Math.pow(r, 3)));
```

The point is that there are often many ways to write out an expression. You will find in this course that there are many solutions to a problem and that everyone in the class will have their own unique solution to a problem (although much of the code will be similar because we will all usually follow the same guidelines when writing our programs).

What if the ball's radius was stored as an instance variable in a **Ball** class as follows:

```
class Ball {
      int   radius;

      Ball(int r) {
            this.radius = r;
      }
}
```

Then we could make some **Ball** objects and use their radius in the equations:

```
Ball b1, b2, b3;

b1 = new Ball(25);
b2 = new Ball(10);
b3 = new Ball(46);

System.out.println((4/3 * Math.PI * Math.pow(b1.radius, 3)));
System.out.println((4/3 * Math.PI * Math.pow(b2.radius, 3)));
System.out.println((4/3 * Math.PI * Math.pow(b3.radius, 3)));
```

Supplemental Information (Mathematical Operators)

JAVA also provides ***bitwise operators*** for integers and booleans:

- **~** bitwise complement (prefix unary operator)
- **&** bitwise and
- **|** bitwise or
- **^** bitwise exclusive-or
- **<<** shift bits left, filling in with zeros
- **>>** shift bits right, filling in with sign bit
- **>>>** shift bits right, filling in with zeros

To understand how these work, you must understand how the numbers are stored as bits in the computer. We will not discuss bit manipulation in this course.

Here is a table showing the operators in JAVA (some which we have not yet discussed) and their ***precedence*** (i.e., the order that they get evaluated in). The topmost elements of the table have higher precedence and are therefore evaluated first (in a left to right fashion). In the table, **<exp>** represents any JAVA expression. If however, you are writing code that depends highly on this table, then it is likely that your code is too complex.

<table>
<tr><td>postfix operators</td><td>[] . () ++ --</td></tr>
<tr><td>prefix operators</td><td>++ -- - ~ !</td></tr>
<tr><td>creation/cast</td><td>new (<typecast>)</td></tr>
<tr><td>multiplication/division/modulus</td><td>* / %</td></tr>
<tr><td>addition/subtraction</td><td>+ -</td></tr>
<tr><td>shift</td><td><< >> >>></td></tr>
<tr><td>comparison</td><td>< <= > >= instanceof</td></tr>
<tr><td>equality</td><td>= = ! =</td></tr>
<tr><td>bitwise-and</td><td>&</td></tr>
<tr><td>bitwise-xor</td><td>^</td></tr>
<tr><td>bitwise-or</td><td>|</td></tr>
<tr><td>logical and</td><td>&&</td></tr>
<tr><td>logical or</td><td>||</td></tr>
<tr><td>conditional</td><td><bool_exp>? <true_val>: <false_val></td></tr>
<tr><td>assignment</td><td>=</td></tr>
<tr><td>operation assignment</td><td>+= -= *= /= %=</td></tr>
<tr><td>bitwise assignment</td><td>>>= <<= >>>=</td></tr>
<tr><td>boolean assignment</td><td>&= ^= |=</td></tr>
</table>

5.2 Formatting Your Output

Consider the following program which asks the user for the price of a product, then displays the cost with taxes included, then asks for the payment amount and finally prints out the change that would be returned:

```
import java.util.Scanner;

class ChangeCalculatorProgram {
    public static void main(String args[]) {
        // Declare the variables that we will be using
        double  price, total, payment, change;

        // Get the price from the user
        System.out.println("Enter product price:");
        price = new Scanner(System.in).nextFloat();

        // Compute and display the total with 13% tax
        total = price * 1.13;
        System.out.println("Total cost:$" + total);

        // Ask for the payment amount
        System.out.println("Enter payment amount:");
        payment = new Scanner(System.in).nextFloat();

        // Compute and display the resulting change
        change = payment - total;
        System.out.println("Change:$" + change);
    }
}
```

Here is the output from running this program with a price of **35.99** and payment of **50**:

```
Enter product price:
35.99
Total cost:$40.66870172505378
Enter payment amount:
50
Change:$9.33129827494622
```

Notice all of the decimal places. This is not pretty. Even worse …if you were to run the program and enter a price of **8.85** and payment of **10**, the output would be as follows:

```
Enter product price:
8.85
Total cost:$10.0005003888607
Enter payment amount:
10
Change:$-5.003888607006957E-4
```

The **E-4** indicates that the decimal place should be moved 4 units to the left…so the resulting change is actually -$0.00050038886070069957. While the above answers are correct, it would be nice to display the numbers properly as numbers with 2 decimal places.

JAVA's **String** class has a nice function called **format()** which will allow us to format a String in almost any way that we want to. Consider (from our code above) replacing the change output line to:

```
System.out.println("Change:$" + String.format("%,1.2f", change));
```

The **String.format()** always returns a **String** object with a format that we get to specify. In our example, this **String** will represent the formatted **change** which is then printed out. Notice that the function allows us to *pass-in* two parameters (i.e., two pieces of information separated by a comma **,** character). Recall that we discussed parameters when we created constructors and methods for our own objects.

The first parameter is itself a **String** object that specifies how we want to format the resulting String. The second parameter is the value that we want to format (usually a variable name). Pay careful attention to the brackets. Clearly, **change** is the variable we want to format. Notice the format string **"%,1.2f"**. These characters have special meaning to JAVA. The **%** character indicates that there will be a parameter after the format String (i.e., the **change** variable). The **1.2f** indicates to JAVA that we want it to display the **change** as a floating point number with at least **1** digit before the decimal and exactly **2** digits after the decimal. The **,** character indicates that we would like it to automatically display commas in the money amount when necessary (e.g., $1,500,320.28). Apply this formatting to the total amount as well:

```
import java.util.Scanner;

class ChangeCalculatorProgram2 {
    public static void main(String args[]) {
        // Declare the variables that we will be using
        double  price, total, payment, change;

        // Get the price from the user
        System.out.println("Enter product price:");
        price = new Scanner(System.in).nextFloat();

        // Compute and display the total with 13% tax
        total = price * 1.13;
        System.out.println("Total cost:$"+ String.format("%,1.2f", total));

        // Ask for the payment amount
        System.out.println("Enter payment amount:");
        payment = new Scanner(System.in).nextFloat();

        // Compute and display the resulting change
        change = payment - total;
        System.out.println("Change:$" + String.format("%,1.2f", change));
    }
}
```

Here is the resulting output for both test cases:

```
Enter product price:
35.99
Total cost:$40.67
Enter payment amount:
50
Change:$9.33
```

```
Enter product price:
8.85
Total cost:$10.00
Enter payment amount:
10
Change:$-0.00
```

It is a bit weird to see a value of **-0.00**, but that is a result of the calculation. Can you think of a way to adjust the **change** calculation of **payment - total** so that it eliminates the **-** sign ? Try it.

The **String.format()** can also be used to align text as well. For example, suppose that we wanted our program to display a receipt instead of just the change. How could we display a receipt in this format:

```
  Product Price     35.99
            Tax      4.68
-------------------------
       Subtotal     40.67
Amount Tendered     50.00
=========================
     Change Due      9.33
```

If you notice, the largest line of text is the "Amount Tendered" line which requires 15 characters. After that, the remaining spaces and money value take up 10 characters. We can therefore see that each line of the receipt takes up 25 characters. We can then use the following format string to print out a line of text:

```
System.out.println(String.format("%15s%10.2f", aString, aFloat));
```

Here, the **%15s** indicates that we want to display a string which we want to take up exactly **15** characters. The **%10.2f** then indicates that we want to display a float value with **2** decimal places that takes up exactly 10 characters in total (including the decimal character). Notice that we then pass in two parameters: which must be a **String** and a **float** value in that order (these would likely be some variables from our program). We can then adjust our program to use this new String format as follows …

```
import java.util.Scanner;

class ChangeCalculatorProgram3 {
    public static void main(String args[]) {
        // Declare the variables that we will be using
        double    price, tax, total, payment, change;

        // Get the price from the user
        System.out.println("Enter product price:");
        price = new Scanner(System.in).nextFloat();

        // Ask for the payment amount
        System.out.println("Enter payment amount:");
        payment = new Scanner(System.in).nextFloat();

        // Compute the total with 13% tax as well as the change due
        tax = price * 0.13;
        total = price + tax;
        change = payment - total;

        // Display the whole receipt
        System.out.println(String.format("%15s%10.2f","Product Price", price));
        System.out.println(String.format("%15s%10.2f","Tax", tax));
        System.out.println("-------------------------");
        System.out.println(String.format("%15s%10.2f","Subtotal", total));
        System.out.println(String.format("%15s%10.2f","Amount Tendered", payment));
        System.out.println("=========================");
        System.out.println(String.format("%15s%10.2f","Change Due", change));
    }
}
```

The result is the correct formatting that we wanted. Realize though that in the above code, we could have also left out the formatting for the 15 character strings by manually entering the necessary spaces:

```
System.out.println(String.format("  Product Price%10.2f",price));
System.out.println(String.format("            Tax%10.2f", tax));
System.out.println(              "-------------------------");
System.out.println(String.format("       Subtotal%10.2f", total));
System.out.println(String.format("Amount Tendered%10.2f", payment));
System.out.println(              "=========================");
System.out.println(String.format("     Change Due%10.2f", change));
```

However, the **String.format** function provides much more flexibility. For example, if we used **%-15S** instead of **%15s**, we would get a left justified result (due to the **-**) and capitalized letters (due to the capital **S**) as follows:

```
PRODUCT PRICE       34.99
TAX                  4.55
-------------------------
SUBTOTAL            39.54
AMOUNT TENDERED     50.00
=========================
CHANGE DUE          10.46
```

There are many more format options that you can experiment with. Just make sure that you supply the required number of parameters. That is, you need as many parameters as you have **%** signs in your format string.

For example, the following code will produce a **MissingFormatArgumentException** since one of the arguments (i.e., values) is missing (i.e., 4 % signs in the format string, but only 3 supplied values:

```
System.out.println(String.format("$%.2f + $%.2f + $%.2f = $%.2f", x, y, z));
```

Also, you should be careful not to miss-match types, otherwise an error may occur (i.e., **IllegalFormatConversionException**).

Supplemental Information (Other String.format Flags)

There are a few other format types that may be used in the format string:

Type	Description of What it Displays	Example Output
%d	a general integer	4096
%x	an integer in lowercase hexadecimal	ff
%X	an integer in uppercase hexadecimal	FF
%o	an integer in octal	377
%f	a floating point number with a fixed number of spaces	83.43
%e	an exponential floating point number	7.869877e-03
%g	a general floating point number with a fixed number of significant digits	0.008
%s	a string as given	"Hello"
%S	a string in uppercase	"HELLO"
%n	a platform-independent line end	<CR><LF>
%b	a boolean in lowercase	true
%B	a boolean in uppercase	FALSE

There are also various format flags that can be added after the % sign:

Format Flag	Description of What It Does	Example Output
-	numbers are to be left justified	2378.348 followed by any necessary spaces
0	leading zeros should be shown	000244.87
+	plus sign should be shown if positive number	+67.34
(	enclose number in round brackets if negative	(439.67)
,	show decimal group separators	2,347,892.99

There are many options for specifying various formats including the formatting of Dates and Times, but they will not be discussed any further here. Please look at the java documentation.

5.3 Type Conversion

When programming, we often find ourselves working with different kinds of data. For example, even when performing simple calculations, we may end up using a variety of primitive data types.

```
float    price;
int      payment;
double   taxes, change;

price = 34.56f;
taxes = price * 0.13;
payment = 50;

change = payment - price - taxes;
```

Notice that the above code performs calculations using **ints**, **floats** and **doubles**. When performing such calculations, JAVA performs some automatic ***type-conversion***. That is, it converts one type of data into another when performing the calculation.

During computations, JAVA will always produce its calculated result as being the same type as the ***more precise*** data type that was used in the calculation. In the above example, doubles are more precise than **floats** and **ints**. Therefore, when we do **price * 0.13**, JAVA notices that this is a calculation using a **float** (less precise) and a **double** (more precise). Therefore the **float** is converted to a **double** during the computation and the resulting answer is returned as a **double**, and stored in the **taxes** variable.

Consider the difference in output of the following code:

```
float     price1 = 34.56f;
double    price2 = 34.56;
System.out.println(price1 * 0.13f);   // displays 4.4928
System.out.println(price2 * 0.13f);   // displays 4.492799835205078
```

Notice that the same calculation is performed in both cases but that one uses a **float** price amount while the other uses a **double** price amount. The 1st calculation uses two **floats**, and so the result is a less precise **float** value. The 2nd calculation uses a **double**, so the entire calculation is performed using **doubles**, generating a more precise result.

What if we changed the code to store the results as follows:

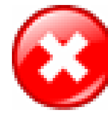
```
float     price1 = 34.56f;
double    price2 = 34.56;
float     result;

result = price1 * 0.13f;
result = price2 * 0.13f;   // gives "possible loss of precision" error
```

The above code will not compile. JAVA notices that in the last line, it is performing a calculation that will result in a **double**. However, **result** is of type **float**. Since **floats** are less

precise that **doubles**, the JAVA compiler informs us that there would be a loss of precision if we tried to take the **double** answer and "squeeze" it into a smaller **float** variable.

When assigning calculation results to a variable, JAVA always checks to make sure that the resulting type of the calculation will "fit" into the variable. We **CANNOT** store a …

- **double** result in a variable of type **float**, **int**, **long**, **byte**, **short**
- **float** result in a variable of type **int**, **long**, **byte**, **short**
- **long** result in a variable of type **int**, **byte**, **short**
- **int** result in a variable of type **byte**, **short**
- **short** result in a variable of type **byte**



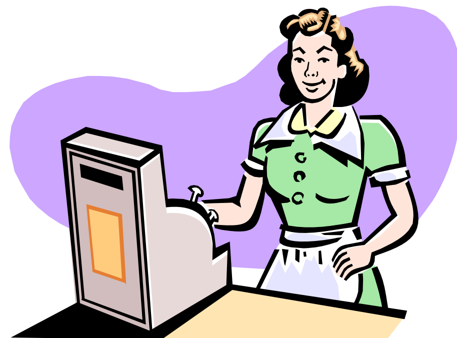
If we attempt to assign a result into a variable of a less precise type (as above), then we will ALWAYS get a compiler error stating "possible loss of precision".

However, we **CAN** store a:

- **double** result in a variable of type **double**
- **float** result in a variable of type **float, double**
- **long** result in a variable of type **long, float, double**
- **int** result in a variable of type **int, long, float, double**
- **short** result in a variable of type **short, int, long, float, double**
- **byte** result in a variable of type **byte, short, int, long, float, double**

Sometimes, however, we may want to take a more precise calculated value and store it into a less precise variable, perhaps for later use. For example, we may want to perform a money-based calculation precisely but then we may only be interested in the whole number portion, or maybe only 2 decimal places. This example calculates the change owed to a person, extracts and stores the whole portion (as whole monetary bills to be returned to the customer … ignore the fact that toonies and loonies are not bills) and the remaining change as a separate value:

```
float    price, changeDue;
int      payment, billsDue;
double   taxes, change;

price = 34.56f;
taxes = price * 0.13;
payment = 50;

change = payment - price - taxes;
billsDue = (int)change;
changeDue = (float)change - billsDue;
System.out.println(change);       // displays 10.947198448181151
System.out.println(billsDue);     // displays 10
System.out.println(changeDue);    // displays 0.94719887
```



Notice that we used **(int)** and **(float)**. These are called ***explicit type-casts***. In English, the term ***typecasting*** means *to identify as belonging to a certain type*. What we are doing is telling the JAVA compiler that we would like to "convert" a particular value into the type that we specified between parentheses **( )**.

We can typecast any numeric type (i.e., **double, float**, **int**, **long**, **byte**, **short**, **char**) to any other numeric type at any time. We just need to remember what happens each time as follows:

- if **x** is a **double** or **float** then **(long)x, (int)x, (short)x, (byte)x** and **(char)x** will discard the decimal places … it will NOT round off, just truncate.
- if **x** is a **long**, **int**, **short**, **byte** or **char**, then **(double)x** and **(float)x** will set the decimal places to be **.0**.
- if a more precise value (e.g., **long** or **double**) is type-casted to a less precise value (e.g., **int** or **double**) then some data WILL be lost.

Here are some examples in which the conversion results in data loss:

```
(float)34.56767867    ==>    34.56768  // rounded off
(int)2.4              ==>    2          // decimal places lost
(int)2.9              ==>    2          // does not round off
```

Here are some examples in which the conversion results in data loss:

```
(char)947384                ==>    '?'
(int)12345678901234567 8L   ==>    -1506741426
```

Conversions may be intermediate and unintentional:

```
int        sum = 30;
double     avg = sum / 4;   // result is 7.0, not 7.5 !!!
```

Perhaps a more common type of conversion is that of converting numbers to Strings and vice-versa. In JAVA, there are some pre-defined functions to do this for us.

In order to convert a given **String** to a particular numeric data type, there are various class/static functions available:

```
String     s = ...;

Integer.parseInt(s);  // returns int value of s
Double.parseDouble(s) // returns double value of s
Float.parseFloat(s)   // returns float value of s
```

Here are some examples:

```
Integer.parseInt("7438");        // returns int value 7438
Double.parseDouble("234.65")     // returns double value 234.65
Float.parseFloat("234.65")       // returns float value 234.65f
```

We sometimes need to use these functions if we are given a **String** from the user and would like to convert the input string to a numeric value for calculation purposes. For example, the

following code asks the user for his/her age and then uses the input to determine the number of years until their retirement:

```
String      input;
Int         age;

input = JOptionPane.showInputDialog("What is your age ?");
age = Integer.parseInt(input);

JOptionPane.showMessageDialog(null, "You have " + (65 - age) +
                              " years until retirement");
```

Unfortunately, the technique for converting a **String** into a **boolean** or **char** is different.

```
String      input;
boolean     retired;

input = JOptionPane.showInputDialog("You are retired ... true or false ?");
retired = Boolean.valueOf(input).booleanValue();

if (retired)
     JOptionPane.showMessageDialog(null, "You get a discount");
else
     JOptionPane.showMessageDialog(null, "You pay full price");
```

Here is an example of getting a single character from the user:

```
String      input;
char        registered;

input = JOptionPane.showInputDialog("Are you a registered user ?");
registered = input.charAt(0);     // gets the 1st character

if ((registered == 'y') || (registered == 'Y'))
     JOptionPane.showMessageDialog(null, "OK.  Sign in please");
else
     JOptionPane.showMessageDialog(null, "Sorry, you must register first");
```

The advantage of the above code is that the user may type in any of the following strings which will acknowledge that he/she is a registered user: "**y**", "**Y**", "**Yes**", "**YES**", "**yes**", etc.. Unfortunately, it will allow them to use "**Yellow**", "**yarn**" and "**you confuse me**" as "**yes**" answers too ;).

Finally, we would also like to be able to convert in the other direction. That is, perhaps we would like to convert a number to a String. This is often necessary in order to place numeric information into a text field on a window.

The simplest way to do this is to take the **int**/**float**/**double**/**long** value and simply add it to an empty String. JAVA will then convert it:

```
int      age;
String   s;

age = 21;
s = age;        // compile error:   incompatible types
s = "" + age;   // this will work
```

Another option is to use any of the following functions:

```
Integer.toString(225)          ==> "225"
Double.toString(225.56)        ==> "225.56"
Float.toString(225.56f)        ==> "225.56"

Integer.toBinaryString(225)   ==> "11100001"
Integer.toHexString(34728)    ==> "87a8"
Integer.toOctalString(34728) ==> "103650"
```

Notice that the last 3 are quite useful because they actually change the appearance of the integer value within the string according to the desired number system.