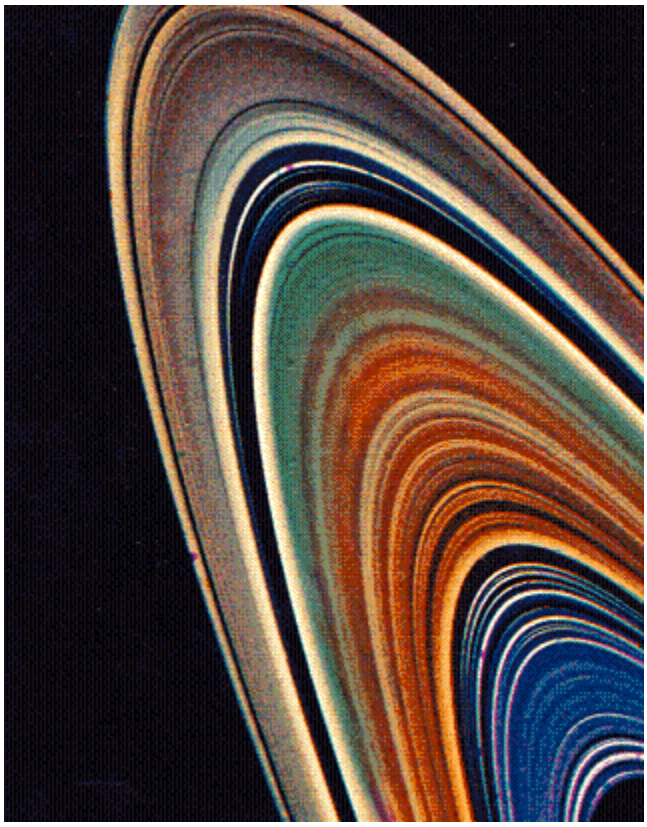
Section 1 - Introduction and Resources

2A.1.1 Overview



We learned a lot of new concepts last week, but the program we wrote was of very limited use. Ideally, we want the computer to do something useful, entertaining or ... at least destructive. Unless you are very easily entertained, printing "Hello World!" to the screen isn't going to satisfy you. Nor should it.

This week we look into the very basic rules of Java programs. We will learn to do numerical calculations, work with the fundamental data types and display the results on the screen. After a few hours of reading and experimentation, you should be able to write your own programs that do certain computational and textual tasks.

2A.1.2 Assigned Reading

As usual, read the week's modules and paste every code sample into your IDE. Then run it, make changes, re-run it, and so on, until you understand what's going on.

Caution: some code samples are just **"fragments"** and will only work if they are pasted inside a **main()** method within the **Foothill** main class.

Read the modules and try the example code *before looking at the assignment*.

After reading the modules, look up any new terms in your textbook index, and see what it has to say about these things.

Additional Module Readings

Don't forget to check the class schedule in page 2 of the syllabus to see what extra handout modules you have this week. I recommend re-reading last week's handouts, especially:

- **Syllabus** - Review sections on course rules and assignment rules until you feel you understand them well.

- **Resources 1R and 2R** - Read a little deeper each week, paying close attention to those rules that apply to this week's material.

2A.1.3 Resources for the Entire Course

Once again, I will remind you about optional resources:

- The ***Elements of Java Style*** listed in the syllabus.
- http://docs.oracle.com/javase/7/docs/index.html

The web page referenced above has many links and you can easily get lost. Once you are on that page, here are some of the most helpful places to look for answers:

- **For the Java Language Technical Specification:** http://docs.oracle.com/javase/specs/jls/se7/html/index.html
- **For Information On The Built-In Java Classes:** [search for **String**, **Math**, **Integer**, etc. on this page then follow link]: http://docs.oracle.com/javase/7/docs/api/index.html
- **For Information on Swing:** http://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html
- **For Non-Swing GUI Classes:** http://docs.oracle.com/javase/7/docs/api/java/awt/package-summary.html
- **For Common Utility Classes:** [search for Scanner, Data, Formatter, etc on this page then follow link]: http://docs.oracle.com/javase/7/docs/api/java/util/package-summary.html

Section 2 - Whitespace

2A.2.1 Whitespace

Everything I tell you about Java for the time being will concern syntax that takes place within the braces of a method, and for the first few lectures, the method will be the **main()** method:

```
public static void main(String[] args)
{
   System.out.println("Hello World");
}
```

In Java, ***whitespace***, means ***spaces***, ***tabs*** and ***returns***. Other than when text is inside quotation marks, " ", whitespace is unimportant to the compiler. All of the following are equivalent:

```
   public static void main(String[] args) {  System.out.println("Hello
World");  }

   public static void main(String[] args)
      {  System.out.println("Hello World");  }

   public static void
        main(String[] args) {
           System.out.println("Hello World");
               }

   public static void main(String[] args)
   {
      System.out.println("Hello World");
   }
```

However, even though they are the same to the compiler, they are *not the same to the programmer*. Proper style makes the program much easier to read than random or haphazardly written programs.

2A.2.2 Style Reminder

I prefer a style similar to the last example. As you read my sample program fragments, notice how I indent and try to use a similar, or equally rational style. Do not invent your own style. Use mine, the one in the text, or the style used in the recommended ***Elements of Java Style*** and adapt that style. Do not mix styles - stick to one. Be careful: if the book contains a typo, it should be obvious based on the style rules. Don't copy obviously incorrect indentation, even from the text, into your assignments.

I can't say this often enough, because I don't want it to happen:

You will lose points if you do not use correct style.

In week 1 you can lose 1 point. In week two you can lose 2 points. By the end of the quarter, you could lose 9 points if you do not indent correctly, even if your program works perfectly.

These are the facts of this class.

Section 3 - Program Statements

2A.3.1 Declarations and Executables

For your first programs, you can think of the statements within **main()**'s braces as organized into two parts:

1. the ***declarations/definitions*** - I sometimes use one or the other word, but technically they have different meanings. For simple programs, there will be no difference.
2. the ***executable*** statements.

Every statement, whether it is a ***declaration*** or an ***executable*** statement is set apart from the rest with either semicolons, ";", or braces, "{ ...}." We will only consider semicolons in this lecture.

The form of the executable statement part is:

```
<statement> ;
<statement> ;
<statement> ;
```

and so on. Remember that these statements might be on one line, like

```
<statement> ; <statement> ;
```

or there may be one statement on several lines:

```
<sta
   teme
     nt> ;
```

2A.3.2 Declarations and Executables

Typical statements are found inside the **main()** method:

```
public class Experiment_1
{
   public static void main(String[] args)
   {
      int x;
      double y;

      x = -234;
      y = 3.1345;

      System.out.println( "X is " + x + " and Y is " + y);
   }
}
```

The first two are ***type declarations***, or just ***declarations***. They identify the ***variables*** **x** and **y** to the compiler as ***objects*** that you will use to hold ***data***. The last three are ***executables***; they do something with the ***variable objects*** **x** and **y** that are defined above.

It is good programming to place your ***declarations*** first, then separate them by a blank line from your ***executables*** (like I did above). You can have additional blank lines as needed for clarity in later parts of the program, but don't forget this one important division.

The above example will generate the following output on the console screen:

```
X is -234 and Y is 3.1345
```

The exact meaning of each statement is something you won't completely understand yet, but it would be good for you to try to see why those Java statements might result in the above output.

2A.3.3 Comments

In addition to ***statements***, you can put ***comments*** in your program. ***Comments*** are not compiled into the executable file (the **.class** file) but are used by the programmer to document the code in the source file only (the **.java** file). ***Comments*** are set off from the rest of the program using the syntax, /* and */, which can enclose any comment, even ones that run across multiple lines:

```
   int x;    /* x will hold the temperature of the radiator */
   float y;  /* y will be used to describe the percentage
                accuracy of the gaming strategy */

   /* the next line initializes x to the lowest temperature */
   x = -234;
   y = 3.1234;   /* at first the strategy is very imperfect */
   dog =    /* 1.1  this doesn't work so I use: */       1.2;
```

Or you can use another type of comment: a double forward slash, //, which is used to begin a comment that automatically ends with the newline:

```
y = 3.1234;  // The next line is not in comment
z = 4;
```

To ***comment out*** several lines, it is more efficient to use /* */, but it is easier to see which lines are commented out if you put a // at the beginning of each line.

This is the basic structure of a Java Program. In the next sections we learn the individual components that comprise statements and how to write meaningful statements that combine to form meaningful programs.

Section 4 - Simple Numeric Statements

2A.4.1 Integers

Programming is all about putting ***values*** into ***memory*** locations and methodically changing them, occasionally sending them to the screen or file so someone who lives outside the computer can take a look at them.

The simplest kinds of ***values*** are ***numbers***.

The simplest kind of ***numbers*** are ***integers***, or **ints**. These are the numbers

1, 2, 3, 4, ...

as well as

0, -1, -2, -3 ...

Integers don't include decimals or fractions. For decimals we need more accommodating data types, like ***floats*** or ***doubles***, which we shall introduce soon.

2A.4.2 Data Types

The **int** represents for us the first of several kinds of data that we can use in our programs. The different kinds of data are called ***data types***. **float, double, char** and **long** are also data types in Java, whose meanings we shall learn shortly.

Some ***data types*** are so simple they are called ***primitive*** types. They can't be broken down any further. The above types, **int, float, double**, etc. are all ***primitive*** types. Some data types are

more complicated and contain an internal structure. These are usually called ***classes***. Therefore, a ***class*** is nothing more than a fancy data type.

So far, the only data type we have formally introduced is the **int** type.

2A.4.3 Constants, Variables and Assignment Statements

I use the term ***variable*** or ***object*** to refer to the memory location that holds the data. You can think of a ***variable*** or ***object*** as an entity that can take on different values as the program progresses.

The word ***object*** will have a richer meaning as we proceed. ***Variables*** can be thought of as the simplest kinds of ***objects***, just as atoms can be thought of as the simplest kinds of molecules. ***Variables*** are normally used to refer to instances of ***primitive data types*** while ***objects*** are used to refer to instances of complex data types (instances of ***classes***).

To define an **int *variable*** (or ***object***) I do this:

```
int someNumber;
```

This causes **someNumber** to become an ***identifier***, or ***variable name***, used to label one location in the computer's memory. That location initially has a a zero value in it (but we should ignore that fact and assume it is some undesirable value we need to replace). We can improve that situation and place a useful value into the storage location known as ***someNumber*** like this:

```
someNumber = 34;
```
Assignment Statement

The above statement puts the ***int constant,* 34**, into the object location, **someNumber**. It is called an ***assignment statement*** because it ***assigns*** the value **34** to the variable **someNumber**. The = sign is also called the ***assignment operator***.

When I say **34** is a constant, I mean that no matter how hard I try, I can't change it to be **35**. It was born, and will die **34**. But I *can* change the value of ***someNumber*** to be something else. This is done with other **assignment statements**. We can have as many assignment statements as we wish, even if they all have the same variable on the ***LHS*** (left hand side) of the assignment operator:

```
someNumber = -10;          // overwrites the 34 with a -10
someNumber = someNumber + 3; // adds 3 to the -10 making
                             // it -7
someNumber++;              // special notation that adds 1 to
                           // someNumber making it -6
someNumber = (someNumber + 1) / 2;  // adds 1 to someNumber, making
                                    // a -5, then divides by 2
                                    // making it a ...
```

What about that last statement. If **someNumber** is -6 when we reach that statement, what will **someNumber** be after the statement is done executing? You might think, (-6 + 1) / 2 equals ...-2.5. But that's not correct.

The answer lies in the underlying data type, in this case, **int**. Integer arithmetic never uses decimal points or fractions. When you divide two **ints** the result is the quotient, as in quotient without the remainder. So -5 / 2 = -2 with a remainder of -1. We throw away the remainder, and the quotient is just -2. Another way, perhaps easier, is to get the complete answer, -2.5, then throw away, or ***truncate***, the fractional part, leaving -2.

A NOTE OF INTEREST: If you want the ***remainder*** of an integer division, use the ***modulo operator, %.*** In other words, in the integer world,

```
14 / 3 evaluates to 4
14 % 3 evaluates to 2
```

This is a very useful operator.

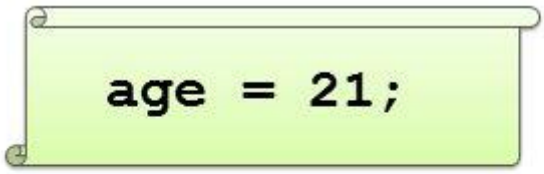
Section 5 - Playing Computer

2A.5.1 Picturing Memory With Assignment Statements

If you have never programmed before, you'll need a mental picture of what happens when you write an assignment statement like:

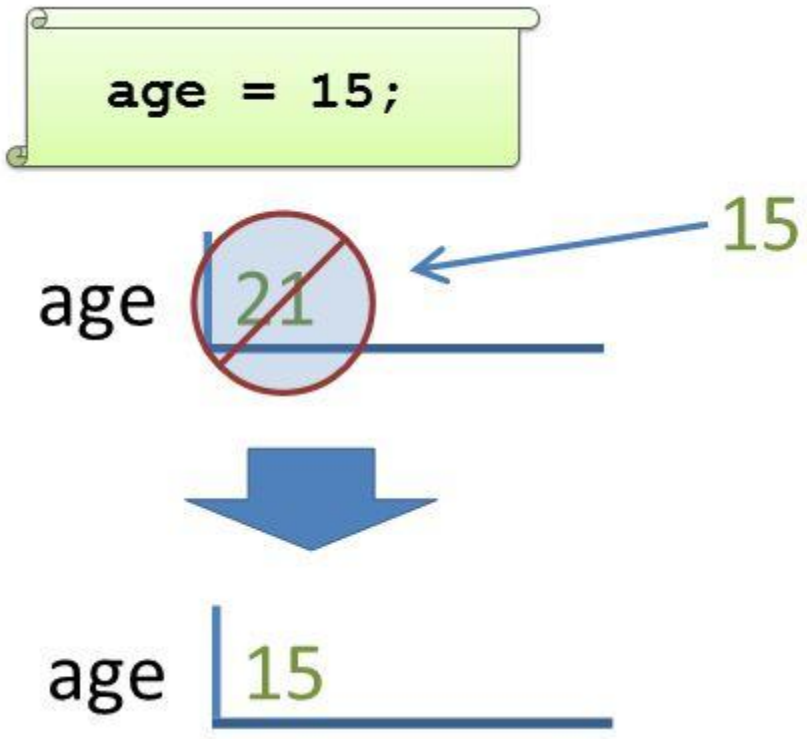
```
age = 21;
```

In the computer's memory, there is a location labeled **`age`**, and this statement causes a **`21`** to be placed inside the location. Here is a picture to help. The upper portion of the picture shows your line of code as it would appear in a program you might write. The lower portion of the picture is the mental image you should have, representing the computer memory after the statement is executed:



As you can see, the assignment of **21** to **age**, (in the statement **age = 21;**) should evoke a picture of **21** being placed in the **age** location. It is a bit like placing mail in a mailbox. **age** is the name on the outside of the mailbox, and **21** is a letter placed in the box. *The only problem with that metaphor is that you can place lots of letters in a mailbox but you can only place ONE ITEM in a memory location.*

Let's say, a little further down in your program, you have a new statement, **age = 15;** This assignment will completely overwrite what was in the **age** location and replace it with **15**. Here's the before and after picture:



Because **age** cannot contain more than one value, the **15** overwrites the **21**. This is important.

Every year, one or two students make the following mistake. They place two assignment statements, back-to-back, assigning two values to the same variable:

```
age = 21;
age = 15;   // horrible - makes no sense
```

Now that you have the picture in memory, you know that this makes absolutely no sense. Why would we put a **21** in age if we are going to immediately destroy it in the very next statement with a **15**? If there were intervening statements between these two assignments, then it would be perfectly fine:

```
age = 21;
System.out.println("My age is " + age);
age = 15;   // now this is okay
```

Because the programmer actually *did* something with the old **age** before overwriting it with the new **age**, it makes sense to have these statements.

2A.5.2 Variables on the LHS and RHS of the = Operator

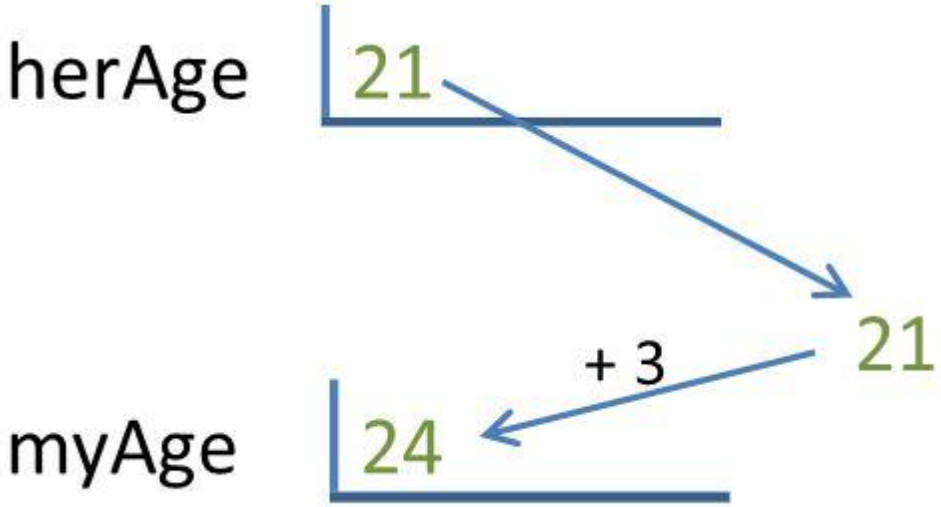
Often, one or more variables will appear on the ***Right Hand Side*** (**RHS**) of the assignment operator (=). For example, here is a typical assignment statement that does this:

```
myAge = herAge + 3;
```

We know we are going to put a number into the "mailbox" or memory location **myAge**. But before we do that, we have to figure out what value to assign. The answer is on the **RHS** of the assignment operator. The expression **herAge + 3** means to take the value stored in **herAge**, add **3** to it, and that's what you put into **myAge**.

Here is your picture. Make sure you understand it before going on:

```
myAge = herAge + 3;
```



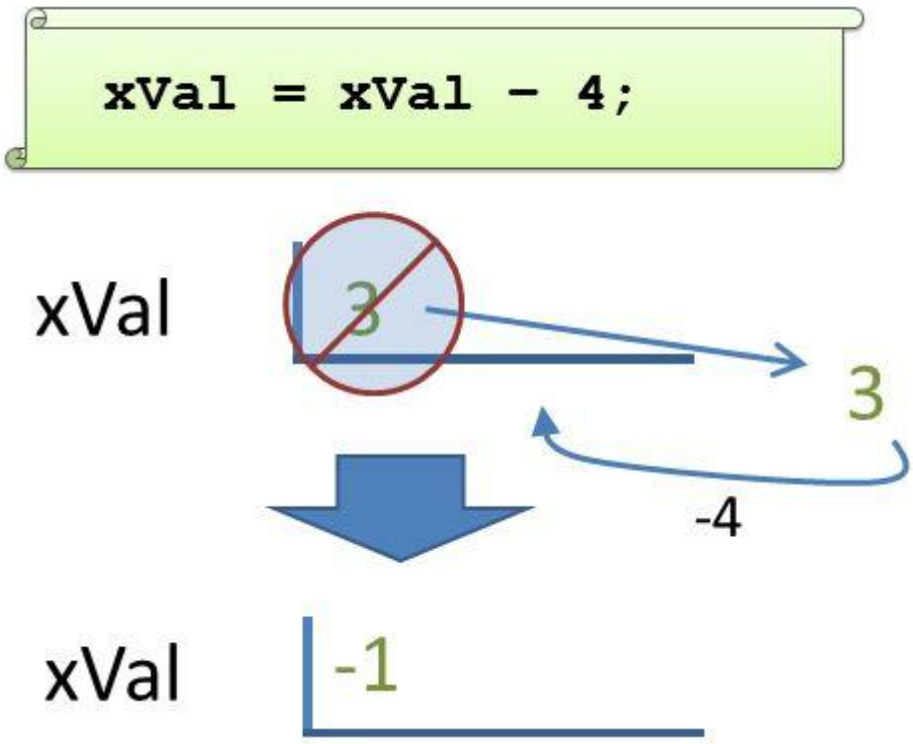
When a variable or object appears on the **RHS**, we take its current value and use that value. We are reading what is inside it. When a variable appears on the ***Left Hand Side*** (**LHS**) of the assignment operator, we are storing a new value into it, overwriting what was there.

Now, this is nice, and we understand how to picture variables that appear on the right (read it) or left (write to it) of the assignment operator. But what if the same variable appears on both sides?

```
xVal = xVal - 4;
```

This is a very important statement to understand, even now, in the first few days of the class. We must get 100% comprehension, so let's have a mental image. The **xVal** is playing a different role on the **RHS** than it is playing on the **LHS**. On the **RHS** we are reading the old value and using that for the computation. That happens first. After that, on the **LHS** we are writing the result, erasing the old value.

Let's assume that **xVal** happened to have the value **3** stored in it (from earlier in the program) when we encounter this statement. Here's the statement followed by the mental picture:



We took the **3**out of the location so we could work with it. It's the *old value*, but we need to use that old value to compute with. Next, we subtract **4** from the value, which leaves us with **-1**. Finally, we store the **-1** back into the same location, **xVal**. **xVal** now has **-1**. The old value, **3**, is gone forever.

2A.5.3 Playing Computer

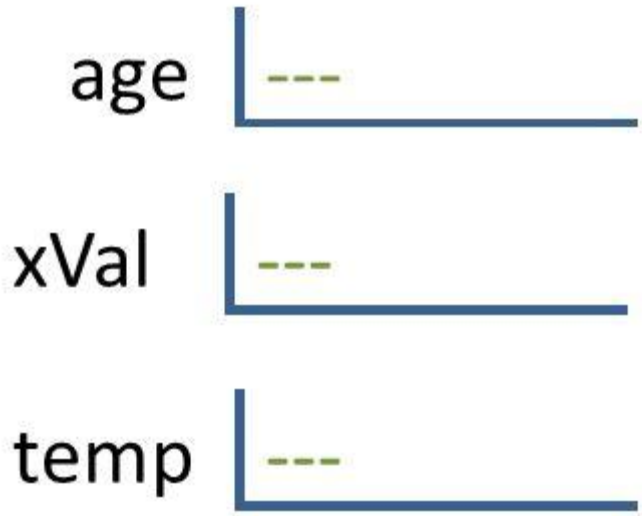
What we did above is a simple technique anyone can do with a pencil and paper. It gives us a way to look at computer programs and predict what is happening at each line of the code. We can draw out all the variables on a piece of paper as soon as we are presented with a program. Let's consider the start of a very simplistic program:

```
public static void main(String[] args)
{
   int age, xVal, temp;

   age = 91;

   temp = age - 8;
   age = temp + 19;

   // etc.
}
```

When trying to simulate what happens internally in memory as this program runs, we look at the first line of a program source:

```
int age, xVal, temp;
```

This is the declaration of the three variables, **age**, **xVal** and **temp**,. We spring into action. We grab a pencil and paper and write the variable locations without putting anything into them yet:



Next, we move to the first statement.

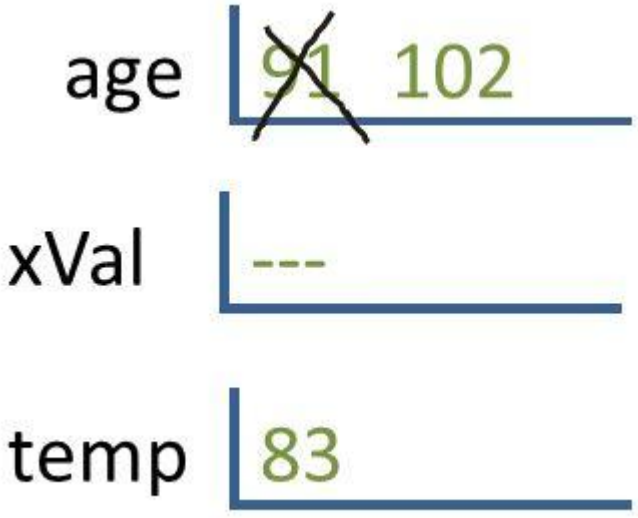
```
age = 91;
```

This tells us to write a **91** into the location **age**. So we do it.

We continue this, line-by-line. After the next two lines,

```
temp = age - 8;
age = temp + 19;
```

our paper would look like this:



This is called "playing computer." You'll have to do this when you get stuck. It will reveal the errors in your logic, and this leads to ideas on how to fix those errors.

Section 6 - Numeric Expressions and Operators

2A.6.1 Expressions

All of the stuff on the ***RHS*** (right hand side) of the = sign is called an ***expression*** - a conglomeration of numbers, operators and other things that evaluates to a single number. The number to which that ***expression*** evaluates is what gets stored into the variable location named on the ***LHS*** (left hand side):

```
n = 3 * (2 + b) * (tree - 1);
```

Here, **b** and **tree** are assumed to be ***variables*** (or ***objects***). Their values are retrieved from memory and plugged into the expression. If **b** happened to have a **5** stored in it, and **tree**, a **0**, then the expression would evaluate to

```
3 * (2 + 5) * (0 - 1)  →  3 * 7 * (-1)  →  -21.
```

Notice two things:

1. The spaces don't matter in expressions. Remember, ***whitespace*** that lives outside quotes is irrelevant to the program, even if it is important for human readability.
2. The parentheses are sometimes needed, other times just put in for clarity of reading. This depends on what is called ***operator precedence.***

SPECIAL NOTE.

You will lose points if you use a single letter variable name in most situations. You see me using single letter names here because I am trying not to confuse you with words in these early examples. However, in actual practice and in your assignments you would use names like **basePrice** instead of **b**, or **age** instead of **n**. The name of the variable should reflect what role it plays in your program. In other words, **someNumber** is probably not a good variable name.

2A.6.2 Numeric Operators and Precedence

There are many ***numeric operators*** in Java. **Addition** (+), **subtraction** (-), **multiplication** (*), **division** (/) and **modulo** (%) are just a few of the obvious ones. *There is no exponentiation operator in Java*. You have to perform exponentiation using a method in the math class, but I don't want you doing that yet. So, if you need an exponent, you'll have to do it with repeated multiplications.

If you have a number with an operator on each side, like

```
dog + tree * cat
```

then the two operators are vying for the same variable object, **tree** in this case. So which operator gets tree's attention first? Since multiplication, *, has a ***higher precedence*** than addition, +, it is computed first.

If two operators have ***equal precedence***, usually the one on the left is executed before the one on the right. I will mention exceptions when they come up.

So if we don't use parentheses, then

```
2+3 * 5
```

will evaluate to 17, not 25. If you want it otherwise, write it as such:

```
(2+3) * 5
```

Besides remembering that * and / have equal precedence and both have greater precedence than + and -, I personally don't remember the precedence of most operators. That's what the parentheses are for. When in doubt, use them to group your expressions.

For the moment, we will only consider expressions that have **int** ***variables*** and ***constants*** in them. Soon we will deal with "mixed mode" arithmetic: expressions containing varying data types, i.e. other types of data besides **int**.

Section 7 - A Console Example

2A.7.1 System.out.println()

In the ***Hello World!*** program we saw the statement

```
System.out.println("Hello World!");
```

which sent to the screen the words inside the quotation marks. There is a name for these *words inside quotation* marks. They are called ***Strings***.

String Terminology

A ***character String***, or just ***String***, is a natural language sequence. It is a series of letters, numerals or special symbols like "This is the ** 3rd ** best day of my life!!!". Sometimes we refer to constructs like this as ***String literals*** to distinguish them from ***String variables*** which we will talk about a little later. For now, you can use the term ***String*** or ***String literal*** to refer to anything inside double quotes "like this".

We will now consider a more elaborate form of the **println**() statement.

First let's look at an entire program:

```
public class Foothill
{
   public static void main(String[] args)
   {
      int someNumber;

      someNumber = -7;
      someNumber = (someNumber - 2) / 3;
      System.out.println( "someNumber is " + someNumber );

      // just some more stuff to demo / and %
      System.out.println( "\n-7 divided by 2 is "
         + (-7 / 2)  );
      System.out.println( "\n-7 mod 2 is "
         + (-7 % 2)  );
   }
}
```

The output of this program is:

```
someNumber is -3

-7 divided by 2 is -3

-7 mod 2 is -1
```

I am hearing several questions:

QUESTION 1) In the **Hello World!** program the **println()** method was easy. But here there are several occurrences of the **+** operator on each line, and some items on the line are in quotes, while others are not. Also a **'\n'** is inside the quotes. What is all this stuff?!

ANSWER) Let's look at the first **println()** statement:

```
System.out.println( "someNumber is " + someNumber );
```

Inside a **println()** method call, you can use the + operator to *glue* together several items as you see above. When a ***String literal*** is present, the + operator does not mean addition, it means ***String Concatenation***, that is, the gluing together of two ***Strings***. If the process of concatenating two strings using the + operator results in a long statement in your source program, you can always spread the statement over two or more lines. I did this in the example above. Recall that we already learned that we can place a long statement on several lines because Java doesn't care about ***whitespace***.

Second, you mentioned that some items inside the **println()** call are in quotes while others are not. As we just learned, items in quotes are ***character string literals***, or simply ***Strings,*** and are printed out exactly as you see them. The items not in quotes, like the object **someNumber**, are ***numeric variables***, and they are first replaced with their ***values*** before they are printed. So if **someNumber** is not in quotes, you would not see the word "someNumber" printed, but rather the ***value*** that it contains, **-3**. All this relies on the fact that the + operator converts the numeric data (in this case the int -3) to a **String** (the String "-3" so that it can then be ***concatenated*** with another **String** (in this case the String "someNumber is ").

A '\n' can be put in anywhere inside a String literal, and it will result in a ***newline*** at that location in the string.

QUESTION 2) Okay, so you can output the value of an object by putting it outside the quotes in the **println()**. But what's going on in the second **println()** statement? I see **-7/2** where I expect to see an object name or a literal string.

ANSWER) In addition to single variable object identifiers, we can write ***entire numeric expressions*** right there in the ***output stream***. This is done if we want to compute a value and send it to the screen immediately without storing the result in some variable.

This expression could be made up of constants like

```
-7/2
```

or a combination of constants and variables like

```
(someNumber + 3) / (someNumber - 3)
```

Even though you can do this, don't do it too often. It is better to waste variables on intermediate results because it makes debugging the program easier. If you put too much computation unnecessarily into your output statements, I will take off points.

Don't lose points:

In fact, I will take off points if you EVER do a computation in the **println()** statement.

I will not take off a point if you first store the value into a variable, then output that variable.

QUESTION 3) What does the % mean in the expression -**7 % 2**?

ANSWER) The % is an operator, like +, -, *, and /. It's called the ***modulo operator***. This operator computes the remainder of the left number divided by the right number, i.e. ,

```
27 % 4  equals 3
-3 % 2  equals -1
 5 % 10 equals 5
```

We read this "27 mod 4."

The ***modulo operator*** is very useful in all kinds of non-mathematical applications. You'll see.

Notes:

1. The **println**() method always adds a ***newline*** to the screen output when it is done printing to the screen. If you don't want a newline, you can use the **print**() method, instead:

```
2.    System.out.print( "The value of  ");
3.    System.out.println( "someNumber is " + someNumber);
```

   will print:

```
The value of  someNumber is -3
```

4. If a **String literal** is long, don't try to spread it over multiple lines unless you first break it up into smaller Strings. Do it like this:

```
5.    System.out.print( "Now is the time "
6.       + "for all good students to "
7.       + "come to the aid of their "
8.       + "community college.");
```

   Also note, that the above example will print the entire sentence *on one line*. (Why?)