Chapter 2

Variables and Control Structures

What is in this Chapter?

This chapter explains the notion of a ***variable*** which is a fundamental part of mathematics, problem solving and computer programming. Variables are used throughout a program with various ***control structures*** such as **if** statements as well as **for** and **while** loops. These are discussed with various examples. The idea of writing ***procedures*** and ***functions*** is explained in the context of the Processing language.

2.1 Variables

When we use a function in a program, it is necessary to store the result somehow so that it can be used later in the program. Remember a "standard" computer only evaluates one instruction at a time from your program. It is similar to the idea of having only one hand to perform an operation. For example, recall the example from chapter 1 where we needed to solve the thirst-quenching problem. If you wanted to perform **pourDrink()** with one hand, you would probably need to expand on the instructions by making use of the counter top to put things down occasionally so your hands are free. Here is a comparison:

Two-handed Algorithm

```
void chooseADrink() {
    openRefrigerator();
    pourDrink();
    closeRefrigerator();
}

void pourDrink() {
    takeCarton();
    closeRefrigerator();
    getAGlass();
    pourLemonadeOrJuiceIntoGlass();
    goToRefrigerator();
    openRefrigerator();
    putCartonInRefrigerator();
}

void getAGlass() {
    goToTheCupboard();
    openCupboard();
    takeGlass();
    closeCupboard();
}
```

One-handed Algorithm

```
void chooseADrink() {
    openRefrigerator();
    pourDrink();
    closeRefrigerator();
    goToCounter();
    pickUpDrink();
}

void pourDrink() {
    takeCarton();
    goToCounter();
    putCartonDownOnCounter();
    goToRefrigerator();
    closeRefrigerator();
    getAGlass();
    pickUpCarton();
    pourLemonadeOrJuiceIntoGlass();
    putDownCarton();
    goToRefrigerator();
    openRefrigerator();
    goToCounter();
    pickUpCarton();
    goToRefrigerator();
    putCartonInRefrigerator();
}

void getAGlass() {
    goToTheCupboard();
    openCupboard();
    takeGlass();
    putGlassDownOnCounter();
    closeCupboard();
}
```

Notice all the highlighted changes that are necessary now because we have only one hand available. Since a typical computer program also has only one "hand" (i.e., processor) running your program, we will also have to make use of "counter tops" (i.e., ***storage space*** in the computer's memory) to "put down" (i.e., ***store***) the intermediate values/objects so that the program can continue performing other operations. The counter top is analogous to the computer's internal memory.

Notice in the algorithm that there were repeated trips back and forth from the counter to the refrigerator. This was because with only one hand, the glass and carton had to be put down in order for the refrigerator and cupboard doors to be opened and closed. When we go to the counter, that's like going to the computer's memory to store or retrieve something that we left there. Likewise, going back to the refrigerator is like going back to the task at hand (i.e., back from the computer's memory and ready to do something).

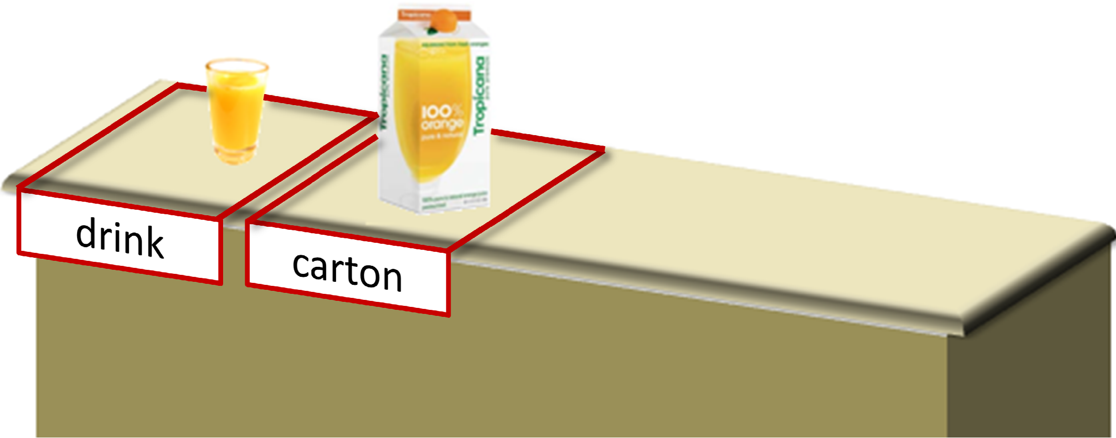
In reality, such details are usually an obvious part of the solution and need not be mentioned. However, it is important to indicate **what** information is being stored and **where** it is being stored so that we can use it later in the program. For example, what if we placed the carton of juice down somewhere but forgot where we put it ? We would not be able to fill up our glass later then. So, it is important in our algorithm to specify *when* and *where* we are storing information/data. A *variable* is used to store data:

A variable is a location in the computer's memory that stores a single piece of data.

A single algorithm or program may use many variables to store intermediate results. Each variable must be given a unique name so that it can be identified later.

Consider our one-handed algorithm for getting a drink from the refrigerator. Recall that we need to place both the **glass** (or soda) and possibly the **carton** on the counter top during the algorithm. In the **pourDrink()** function, for example, after the drink is poured, both the **glass** and the **carton** are sitting on the counter while we go to open the refrigerator. That should help us to see that we need two unique variables (i.e., two unique locations on the counter top) to store these objects.

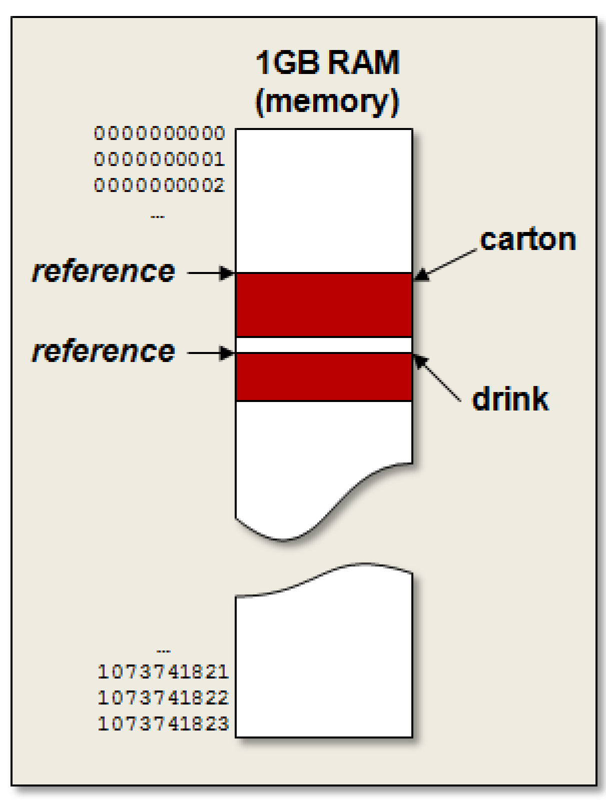
For each variable, we need to choose a meaningful ***variable name*** (i.e., a label) so that we can refer to it later. It makes sense to use the label "glass" to store the glass and "carton" to store the carton. However, the glass will eventually contain the drink that we want to ingest, so we could use the label "drink" instead:



So, each variable that we make will ***have its own counter space on the counter top***. When we create (or ***declare***) a variable, we are really ***reserving space*** for an object on the counter top. And as with any reservation, we need to have a name for the reservation … which corresponds to the label (i.e., variable name).

When a variable's space has been reserved, usually there is nothing yet in that space … just the label. The term ***null*** is often used to indicate that *nothing* has been stored in the variable yet (i.e., noting is at that spot on the counter top). Once we put something in the variable (i.e., on the counter top at that label), the item that we place there is called the ***value*** of the variable.

Each variable that we declare (i.e., each time we reserve space for something), we are actually taking up space in the computer's memory. You may already know that your phone, your ipod, your flash drive, your computer etc… all have limited memory (or storage) space. The computer's memory space is called ***RAM***, which stands for Random Access Memory. At the time that these notes were written, some phones had **512MB** (roughly 512 million bytes) of storage, while typical computers had **8GB** (roughly 8 billion bytes) of storage.



Consider, for example, a computer with 1GB of storage. Each time we declare a variable, we are reserving space in the RAM. That is, we are using up a portion of the computer's available memory. The bigger the object that we are storing, the more space that it takes up. So, here, we see that the carton would take up a little more space than a glass of orange juice.

The labels **carton** and **drink** are called ***references***, since they are used to ***refer to*** a particular object. The *reference* is actually just a number within the sequence of storage bytes in the RAM. It is also known as a ***memory address***, because it sort of represents the "home" of the object, as a real life address uniquely identifies a home in the real world.
As we will see later, there are simple kinds of objects (such as numbers and letters) called ***primitives*** that are stored in a simpler manner.

As with any real counter top, we can alter at any time what we place at that location on the counter. Similarly, we can change a variable's value at any time by putting a different value there. What happens to the old value? It simply disappears. In real life, the object at a specific spot on the counter does not disappear when we put a new object there at the exact same location. So variables in a computer are a little different that real life. When it comes to replacing a variable's value with a new value, it is easiest to understand that process as ***over-writing*** the variable's value. You may already have experience with over-writing information, perhaps from erasing mp3 songs from your ipod or phone, by putting new ones on the ipod/phone … the old songs are replaced (or overwritten) by the new ones. Variables are overwritten in the same manner.

So now, we should adjust our code to make use of the variables. Notice what the code now looks like with two variables **drink** and **carton**:

```
void MainAlgorithm() {
   getOffCouch();
   walkToKitchen();
   goToRefrigerator();
   drink = chooseADrink();
   drinkThis(drink);
}

void chooseADrink() {
   openRefrigerator();
   drink = pourDrink();
   closeRefrigerator();
   return drink;
}

void pourDrink() {
   carton = takeCarton();
   glass = getAGlass();
   pourContentsInto(carton, glass);
   goToRefrigerator();
   putCartonInRefrigerator(carton);
   return glass;
}

void getAGlass() {
   goToTheCupboard();
   openCupboard();
   aGlass = takeGlass();
   closeCupboard();
   return aGlass;
}
```

Notice that the **drink** variable represents either an empty glass or a glass with juice in it, depending on the line of the program. The **=** operator is used to indicate that something is to be stored in the variable (i.e., that we want to give the variable a new value).

Notice as well how **goToCounter()** and **putCartonDownOnCounter()** are both combined into one storage step as **carton** = **takeCarton()**. That's because leaving the refrigerator and heading over to the counter was done as a means to store the carton on the countertop so that the single hand is free again to close the refrigerator. The entire storage process is now specified with just one = operator.

Similarly, the **goToCounter()** and **pickUpDrink()** combination as well as the **goToCounter()** and **pickUpCarton()** combination is analogous to simply getting the value of the variable in that it accesses the object stored at that location on the countertop. The functions **chooseADrink()**, **pourDrink()** and **getAGlass()** all bring back (i.e., ***return***) a drink, whether it is a full glass or an empty glass. The **drink** is returned (from each function) back to the function that called it.

The return value is the value returned as a result of the function.

The idea of a *return value* becomes more obvious when the function is mathematical. For example, **sine(90)** is naturally understood to *return* the *value* **1** and **squareRoot(100)** would naturally *return* the *value* **10**.

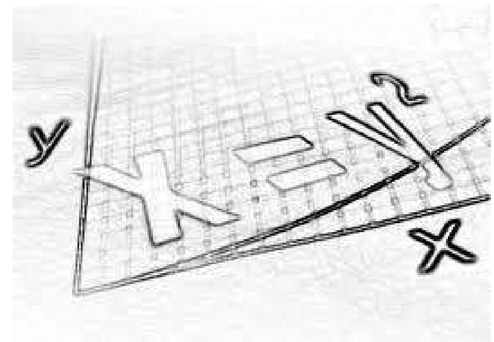
One more point to mention regarding the variables is with respect to where (and how often) they are used. You will notice that the **drink** variable is used throughout the algorithm, whereas the **carton** variable is used only within the **pourDrink()** function. Variables that are used only with a single function/procedure (e.g., **carton**) are called ***local variables*** because they are only used locally (i.e., within the vicinity of the function or procedure). In contrast, variables that are used across many functions/procedures (e.g., **drink**) are known as ***global variables***.

Notice as well that the **pourContentsInto(carton, drink)** and the **putInRefrigerator(carton)** procedures both take variables as their incoming parameters. Recall from chapter 1 that a *parameter* is a piece of data that is provided to a function. In these two cases, we are being specific by telling the procedures which variables to use (i.e., which carton to use and which glass to pour into) when pouring the drink and when returning the carton to the refrigerator.

Back in chapter 1, we used a parameter to represent the number of glasses/plates and utensils that we wanted to get from the cupboard: **getGlasses(8)**. We also used parameters to specify the values for drawing our houses such as the (**x**, **y**) coordinates and **width** X **height** dimensions.

A parameter is similar to a variable because it is a value that is used in your program. A parameter is different, however, in that it usually represents a value that remains constant within the context of where it is being used.

For example, when performing the **getGlasses(8)** function, the value of 8 is fixed (i.e., unchanging) while we are getting the glasses. Also, when we call **rect(100, 50, 100, 100)** in Processing to draw a rectangle, these 4 parameters are fixed/constant while the rectangle is being drawn. Since you (the algorithm designer and/or programmer) came up with these constant values, we can say that these values represent incoming ***algorithm parameters***.



In general, an algorithm may have many initial parameters and like variables they are usually given names. So inside an algorithm or computer program a parameter will usually look just like a variable. However, you should understand that these parameters will not change throughout the execution of the algorithm/program.

Later, you will create your own functions and procedures that may take incoming parameters. You will assign a name/label to these incoming values. You should understand though that the values of the parameters will NOT change throughout the function/procedure.

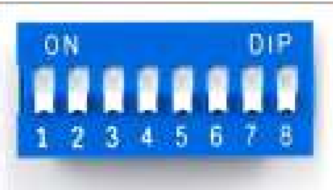
At this point, you should understand why variables are needed in a program. They allow you to store some data temporarily, while you deal with other aspects of the program. At any time you can come back and obtain those stored values. It is as useful as having a notepad with you to jot down someone's phone number for use later on. As we do some examples, you will get to understand where and why to use variables.

2.2 Variable Representation

All information in the computer is actually stored in the electronics as voltages … high and low voltages that can be thought of as billions of 1's and 0's that have some kind of meaning to them. That is, all user information (whether it is a name, phone number, picture, email, database, game, etc..) is stored as 1's and 0's which we call ***bits***.

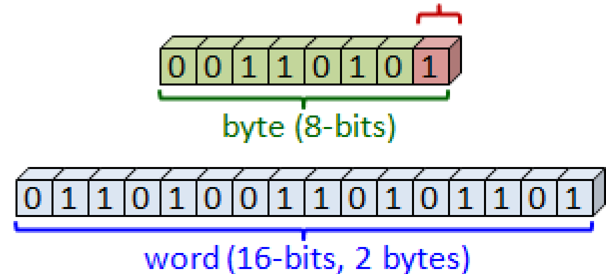
As humans, we have a hard time working at such a low level. We do better working with things like numbers, characters and real-world objects. So, rather than work with single bits, we group these bits into more abstract or higher-level packages.

A group of 8 bits is known as a ***byte***. A byte can represent 256 combinations of 1's and 0's. That is, if we think of each bit as being a switch which is either on or off, we can flip the 8 switches in 256 unique combinations. This allows a single byte to store a number from **0 to 255**.



When two bytes are required to represent a number, the pair of bytes is called a ***word***. A *word* can store a number in the range from **0 to 65535**. We can continue to group bytes together to store even larger integer numbers.

So, the *bit* and the *byte* are the two most primitive forms that a computer uses for number representation. Most computers will use the term ***boolean*** to represent a 0 or 1, but instead of saying "0" or "1" the terms "***false***" and "***true***" are used.



Bytes can also be used to represent letters, digits, punctuation, etc. which are called ***characters***. How so? Well, back in 1968 it was decided that computers conform to a numbering standard called ***ASCII*** (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange). That is, each combination of numbers in the range from 0 through 127 was mapped to (i.e., corresponds to) a particular keyboard character. Here to the right, is the ASCII table (provided as a reference only … DO NOT try to memorize it). So, for example, the letter "**A**" corresponds to number **65** in the ACSII table which is number **01000001** in binary bits (we will not discuss bit representation any further in this course).

ASCII value	Character(s)
0	null
1-31	various special characters
10	line feed
13	carriage return
32	space
33-47	!"#$%&'()*+,-./
48-57	0123456789
58-64	:;⇔?@
65-90	ABCDEFGHIJKLMNOPQRSTUVWXYZ
91-96	[\]^_`
97-122	abcdefghijklmnopqrstuvwxyz
123-127	{|}~□

There are also versions of extended ACSII tables covering the numbers from 128 to 255. In addition, since computers began to be used internationally, 256 combinations were not enough to represent the letters of various international languages. Therefore, a new standard called ***Unicode*** has been developed (and continues to be expanded) to account for the other characters. However, a single byte is no longer sufficient to represent the character … two or more bytes are required.

In addition, we can actually use bytes to represent real numbers (also called floating-point numbers) such as **3.14159265**. Also, by making assumptions on one particular bit in a byte (i.e., the ***most significant bit***, also called the ***sign bit***), we can allow the numbers to be either negative or positive. There are many details regarding number representation, but we will not discuss them further in this course.

The point is that *bits* are grouped to form *bytes* (or characters) which are also grouped to form larger numbers. Ultimately, this leads to what are known as ***primitive data types*** that are used in most programming languages. Here are the four basic primitives that are available in most programming languages (although the names may differ in each language):

- boolean – **true** or **false**
- integer – a positive or negative whole number
- floating-point number – a positive or negative real number with decimal places
- character – a letter, digit, punctuation or some other keyboard character

These are called primitive because they are the most **basic** types of data that we can store on the computer. Some languages will further distinguish between various types of integers or floats. For example, the following are the four official primitive data types in Processing (& JAVA) that can represent integers of various sizes:

Type	Bytes Used	Can Store an Integer Within this Range
byte	1	-128 **to** +127
short	2	-32 768 **to** +32 767
int	4	-2 147 483 648 **to** +2 147 483 647
long	8	-9 223 372 036 854 775 808 **to** +9 223 372 036 854 775 807

Notice that the various types take up a different amount of memory space.

Similarly, there are two official primitive data types in Processing (& JAVA) to store floating-point numbers:

Type	Bytes Used	Can Store a Real Number Within this Range
float	4	-10^{38} **to** $+10^{38}$
double	8	-10^{308} **to** $+10^{308}$

Regardless of how we group the bytes, all information/data can be represented through the 4 basic primitives of boolean, integer, floating point numbers and characters.

So why are we talking about this? Well, when programming, some languages (like Processing and JAVA) ***force you to specify the types*** for all of your variables. That means, for every variable that you create, you must indicate its type and its name.

In Processing and JAVA, for example, in order to use a variable to store a primitive kind of value (e.g., a boolean, integer, floating-point number or character), you must specify in your program the **type** followed by the **name** (must be unique) of the variable. Here are some examples:

```
boolean    hungry;
int        days;
byte       age;
short      years;
long       seconds;
char       gender;
float      amount;
double     weight;
```

The above examples show all 8 primitive possible types that you may use in Processing/JAVA programs. Note that these will differ from language to language. Notice as well that there is a **;** character after each line, as with any step of an algorithm.

Each line above is responsible for ***declaring*** a variable. That means that a space is reserved in the computer's memory with the given label (variable name) that can hold a value of the given type.

A note about variable names … make sure to pick **meaningful** names that are not too long!! The name must be unique and it is **case-sensitive** (i.e., `Hello` and `hello` would not be considered the same). Variable names may contain only **letters**, **digits** and the **'_'** character. As standard convention, multiple-word names should have every word capitalized except the first, (often referred to as 'camelCase'). Here are some good examples of variable names:

- count
- average
- insuranceRate
- timeOfDay
- poundsPerSquareInch
- aString
- latestAccountNumber
- weightInKilograms

There are some restrictions when it comes to naming variables. Variable names may **not** contain spaces or other special characters (with the exception of '_'), and the first character must **not** be a number. In addition there are several names that are restricted (known as reserved words). Here are some examples of invalid variable names:

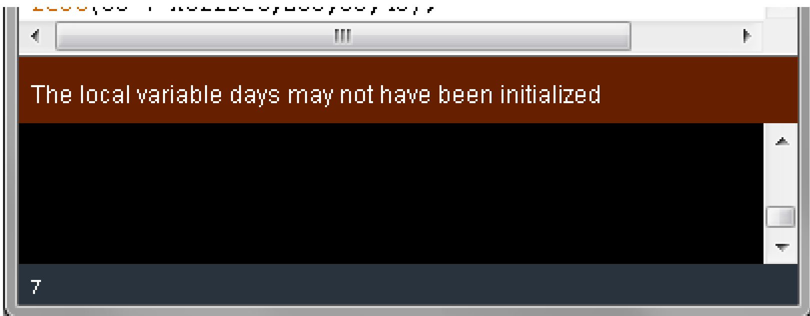
- × insurance Rate
- × time-of-day
- × 3rdVariableName
- × lbs/(inch*inch)
- × for
- × int

There is one more restriction when it comes to writing processing code. It is a good idea NOT to use **width** and **height** as variable names because these are variables that are already defined in Processing, which represent the width and height of the drawing area.

In processing, after you declare a variable, you MUST give it a value ***before*** you use it. Declaring a variable, DOES NOT assign it any value, it only reserves space for the variable. For example, suppose that tried to do this within one of your Processing functions:

```
int   days;
print(days);     // prints out the day variable's value
```

You would get the following error, preventing your program from running:



Interestingly, Processing allows you to create **global** variables (i.e., variables outside of a function which are available through your entire program). In this case, it will assign a value of **0** to your variable and it will not produce an error.

We use the term ***assign*** to represent the idea of "giving a value" to a variable. In Processing, the *assignment operator* is the **=** sign. So, we use **=** to put a value into a variable. Here are a few example of how we can do this with some of the variables that we declared earlier:

```
hungry = true;
days = 15;
gender = 'M';
amount = 21.3f;  // (in JAVA only) floats must have an 'f' after them
weight = 165.23;
```

Something VERY important to remember when learning to program is that the value of the variable **must be the same type** of object (or primitive) **as the variable's type** that was specified when you declared it earlier. So for example, in the following table, make sure that you understand why the examples on the left are wrong, while the right examples are correct:

✗	✓
`int days;` `days = 10.2789;`	`int days;` `days = 10;`
`boolean hungry;` `hungry = 'y';`	`boolean hungry;` `hungry = false;`
`char sex;` `sex = "F";`	`char sex;` `sex = 'F';`

To help cut down the number of lines of code in our program, we are allowed to both *declare* and *assign* a value to our variables all on one line. So, from our earlier examples, we can do the following:

```
boolean    hungry = true;
int        days = 15;
char       gender = 'M';
float      amount = 21.3f;
double     weight = 165.23;
```

A variable may be ***declared*** only once in the program, but we may ***assign*** a value to it multiple times. Can you determine the output of this piece of code:

```
int  days;
days = 43;
print(days);     // prints out 43
days = 15;
print(days);     // prints out 15
```

So, variables can be ***re-assigned*** a value, but cannot be ***declared*** again. Therefore, the following code will NOT compile:

```
int days = 365;
print(days);
int days = 7;    // cannot declare days again
print(days);
```

Here are some more pieces of code. Do you know what the output is?

```
int x;
int y;
x = 34;
y = 23;
print (x + y);
```

Here is a similar example. Notice in Processing (and JAVA) that we are allowed to declare multiple variables of the same type on the same line, each separated by a **','**:

```
int x, y;
x = 34;
y = x;
print(x + y);
```

Here is another one:

```
int x, y, z;
x = 3*2*1;
y = x + x;
z = x;
print(z);
```

Note that even though we use **x** a few times, it does not change its value.

Here is one that is a little more interesting:

```
int    total;
float  average;

total = 12 + 25 + 36 + 15;
average = total / 4;
print("The average is ");
print(average);
```

Here is the output:

```
The average is 22.0
```

Notice that the **print()** function also allows you to display a fixed set of characters defined within double quotes. This fixed set of characters is called a ***String***.

Each time we call **print()** , the information will appear on the same line, so it is important to have the extra space character at the end of the string above, otherwise the result would be crowded close to the text like this:

```
The average is22.0
```

We can also combine the two print statements into one line as follows:

```
print("The average is " + average);
```

This code will append the **average** variable's value to the string by using the **'+'** operator.

A similar function called **println()** is available that will allow you to stop printing on one line and start another:

```
println("The average is ");
print(average);
```

will produce:

```
The average is
22.0
```

What happens in this example of trying to swap two variables?

```
int x = 10;
int y = 20;
x = y;
y = x;
println(x+", "+y);
```

will output: `20, 20`

To understand what went wrong, consider the values of x and y before and after the third line. Before the assignment, x and y have different values (10 and 20), but after the assignment, the value of x is replaced by the value of y (20). The old value of x (10) is lost! Before reading ahead try to figure out how to fix the above code so that the values of x and y are actually exchanged.

In order to correctly swap two variables, you need a 3rd *temporary* variable as in the solution here:

```
int x = 10;
int y = 20;
int temp = x;
x = y;
y = temp;
println(x+”, “+y);
```

In this code the variable temp is used to backup the value of the variable x. This way when we assign the value of y to x the old value of x is not lost.

Let's consider another potential pitfall:

```
int x = 1;
int y = 2;
float result = x/y;
print(result);
```

will output:

```
0.0
```

What went wrong? This error is commonly known as *Integer Division*. Normally we would expect the result of 1/2 to equal 0.5. However, **integer values do not have decimals**! So, the .5 part is dropped (**note**: not rounded!) and the result is 0. Following the division in the above code, we store the result (0) in a variable of type **float**. Since float variables *do* have decimals, 0 is expanded to 0.0.

How can we fix the above code, so that it prints 0.5? The simplest way would be to change the type of either x or y or both from **int** to **float**. But let's say, for whatever reason, those variables must stay **int**'s, then what can we do?

Processing offers some pre-defined conversion functions that give the value of the variable as a different type (without changing the variable itself):

- `int(x)` // converts x into an **int**
- `float(x)` // converts x into a **float**
- `byte(x)` // converts x into a **byte**
- `char(x)` // converts x into a **char**

As a side note, JAVA does not have such conversion functions. Instead, it uses something called type-casting with different syntax as follows:

- `(int)x` `// converts x into an int`
- `(float)x` `// converts x into a float`
- `(byte)x` `// converts x into a byte`
- `(char)x` `// converts x into a char`

Both do the same thing (in different ways) and both are valid options in Processing.

Using type-conversion in our integer-division function above we can get the correct result with something like this:

```
int x = 1;
int y = 2;
float result = float(x)/y;
print(result);
```

If you have a value that will remain ***constant*** throughout your program you can use the keyword ***final*** (implying that it has its final value and will not change again) before the variable's type. In this case, you must assign the value to the constant when it is declared:

```
final int      DAYS = 365;
final float    INTEREST_RATE = 4.923;
final double   PI = 3.1415965;
```

By convention, constants use uppercase letters with underscores (i.e., _) separating words.

2.3 Functions & Procedures with Parameters

Recall from Chapter 1 we introduced procedures as a way to clean up our code. This is one reason for using procedures, known as **abstraction**. There are two other main reasons for using procedures: **code reuse** and **modularity**. As you will see, using procedures for all of these reasons increases your code's **readability**.

> *Abstraction is the process of reducing or factoring out details that are not necessary in order to describe an algorithm.*

An abstracted procedure uses the idea of a "black-box", in that we can use a procedure without knowing the details of how it works (e.g., **ellipse**(x,y,w,h), **println**(s)).

> *Reusability is the likelihood of code to be used again with little or no modification.*

Reusability greatly reduces the amount of code you have to write. A common phrase in computer science is "Don't reinvent the wheel." Meaning, don't rewrite the same code over and over when you could encapsulate that code in a method call (examples below).

> *Modularity is the degree to which a system's components may be separated and recombined.*

A modular procedure ideally does one single thing. For example, a modular procedure could be one that calculates an average, or one that draws a house. Modularity lets you mix and match parts of your code to create new things (just like Lego® blocks!)

These features of procedures/functions are not mutually exclusive. For example, abstraction is often achieved by breaking difficult problems into simple modules that can be combined and reused to create complex solutions.

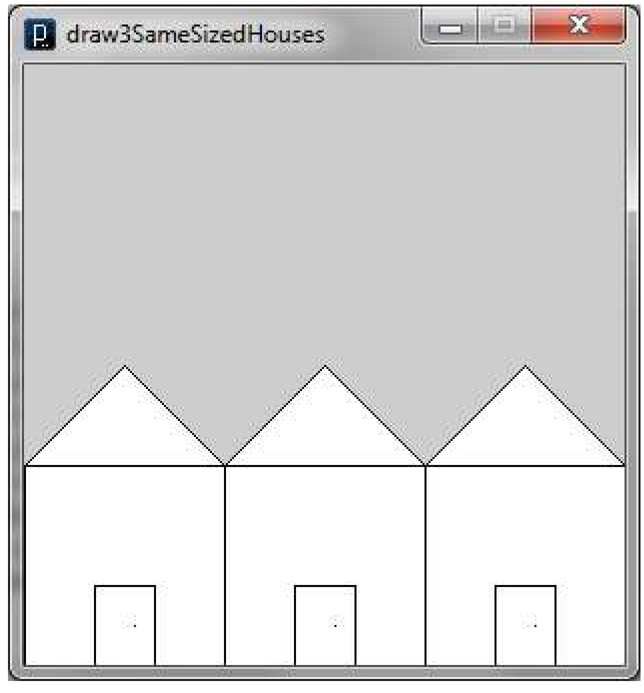
One way to get more use out of simple procedures is to use parameters to allow procedures to change their behaviour. We've seen this before when using the drawing procedures (e.g. **rect**(100,200,100,100), and **rect**(130,260,30,40)), but how do we write our own procedure that uses parameters? Here is the format for passing in parameters to a function:

```
void procedureName (type1  name1, type2  name2, …, typek  namek) {
        // Write your procedure's code here
}
```

Each parameter must be declared like a variable (i.e., with a type followed by a name), with commas in between. So, $type_1$, $type_2$, …, $type_k$ are the *types* of the parameters to the function and $name_1$, $name_2$, …, $name_k$ are the *names* of the parameters. Inside the procedure each parameter is available for us to use just as we would use an ordinary variable.

Example:

In order to more fully understand the benefits of creating functions and procedures consider how we can ***efficiently*** draw 3 houses side-by-side. Here is an *inefficient* way to do it:




```
void setup() {
    size(300,300);

    // 1st house
    rect(0,200,100,100);
    triangle(0,200,50,150,100,200);
    rect(35,260,30,40);
    point(55,280);

    // 2nd house
    rect(100,200,100,100);
    triangle(100,200,150,150,200,200);
    rect(135,260,30,40);
    point(155,280);

    // 3rd house
    rect(200,200,100,100);
    triangle(200,200,250,150,300,200);
    rect(235,260,30,40);
    point(255,280);
}
```

The program is inefficient because we are duplicating portions of code. Notice that the only differences in the code (shown underlined in red) are is with respect to the parameter values. The rest of the code remains *exactly* the same. Do you notice how these values differ from the 1st house to the 2nd house and from the 2nd house to the 3rd?

We are actually adding **100** to these values each time that we draw a house. Notice as well that only the **x** coordinates of the shapes being drawn are changing … the y coordinate, width, and height values do not change. In other words, we are *offsetting* the **x** value of our house by a fixed amount (i.e., **100**) each time we re-draw it.

*An x-offset is the difference by which one graphical object is out of **horizontal** alignment from some fixed horizontal reference (e.g., origin or another object's position).*

*A y-offset is the difference by which one graphical object is out of **vertical** alignment from some fixed vertical reference (e.g., origin or another object's position).*

To simplify the above code, we can create a procedure for drawing the house and simply call it three times with different parameters as follows:

```
void setup() {
    int xOffset;// Make a variable to store the offset

    size(300,300);

    // draw 3 houses with different x offsets
    xOffset = 0;
    drawHouse(xOffset);

    xOffset = 100;
    drawHouse(xOffset);

    xOffset = 200;
    drawHouse(xOffset);
}

void drawHouse(int xOff) {
     rect((0+xOff),200,100,100);
     triangle((0+xOff),200,(50+xOff),150,(100+xOff),200);
     rect((35+xOff),260,30,40);
     point((55+xOff),280);
}
```

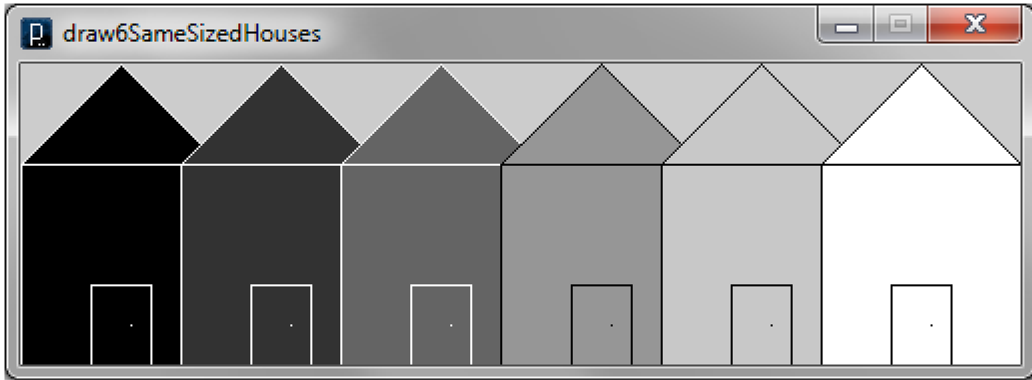
Notice that there are two steps involved in using a parameter. First, in order to call the **drawHouse()** procedure we need to pass in a value or variable (`xOffset`). Second, within the procedure definition there is a variable declaration (`int xOff`). These two parts are sometimes respectively called the argument and the parameter, or the actual parameter and the formal parameter.

As you have seen, the argument (in the procedure call) can be either a value (e.g. 100) or a variable with a value (e.g. `xOffset`).

You might have noticed that the parameter `int xOff` looks just like a normal variable declaration, but inside the brackets now. That's because it is! It tells the Processing/JAVA compiler to reserve space for this incoming integer value and gives it a new name. The `xOff` parameter is a new variable which can be used within the procedure. Its value is initialized to the value of the argument `xOffset`.

Example:

Can you write a program that would produce this picture:



What is different from the last program ?

- 6 houses instead of 3
- The size of the window is different … now **500 x 150**
- The offset is not 100 anymore, but less (since houses overlap)…**80** ?
- The color of gray changes as the houses are drawn
- The first 3 houses have black border while the last three have white

So, now that we understand the differences, how do we write the code that uses the same **drawHouse()** procedure that we wrote earlier ?

We need to vary the **stroke** color and the **fill** color for each house as follows:

```
void setup() {
    size(500,150);

    stroke(255); // use a white border on everything
    fill(0);  drawHouse(0);
    fill(50); drawHouse(80);
    fill(100);drawHouse(160);

    stroke(0); // use a white border on everything
    fill(150);drawHouse(240);
    fill(200);drawHouse(320);
    fill(255);drawHouse(400);
}

void drawHouse(int x) {
    rect(x,50,100,100);
    triangle(x,50,x+50,0,x+100,50);
    rect(x+35,110,30,40);
    point(x+55,130);
}
```

Notice how the **drawHouse()** procedure remains the same, but that the main algorithm differed. How would we change the **drawHouse()** function so that it takes two more parameters that specify the **stroke** and **fill** colors ? Here it is:

```
void drawHouse(int x, int s, int f) {
     stroke(s);
     fill(f);
     rect(x+0,50,100,100);
     triangle(x+0,50,x+50,0,x+100,50);
     rect(x+35,110,30,40);
     point(x+55,130);
}
```

Notice how we simply indicate two additional parameter types and names in the parameter list to the function and that we make use of these values on the first two lines of the function. How does the simplified setup code now look?

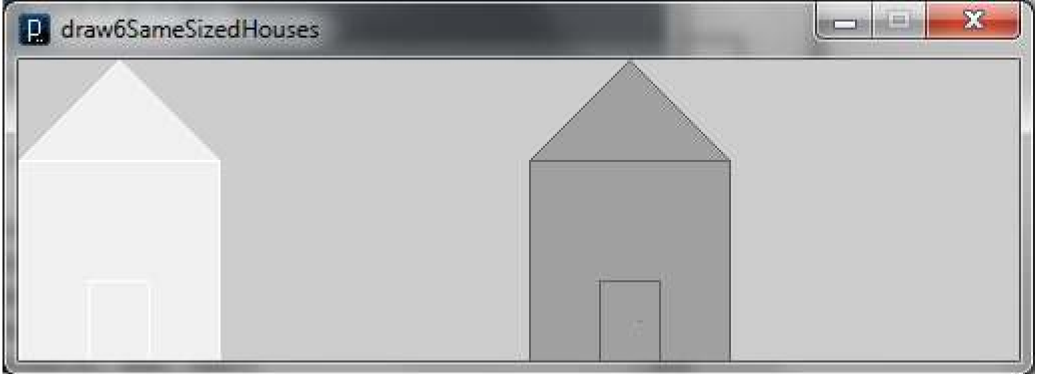
```
void setup() {
     size(500,150);

     drawHouse(0, 255, 0);
     drawHouse(80, 255, 50);
     drawHouse(160, 255, 100);
     drawHouse(240, 0, 150);
     drawHouse(320, 0, 200);
     drawHouse(400, 0, 255);
}
```

Wow, that looks like nice and clean code!

What would happen if we forgot the order of the parameters and mixed the order up between the stroke and the fill. What is we passed the parameters in this order (xOffset, fill, stroke) ?

Or even worse, if we did it in this order (stroke, fill, xOffset) ?



The point is…that lots can go wrong if you mix up the order of your parameters.

But what if we forget to pass in a parameter? What if you tried **drawHouse(100,0)**; unintentionally forgetting the fill color ? Well, this would be caught as a compile error indicating:

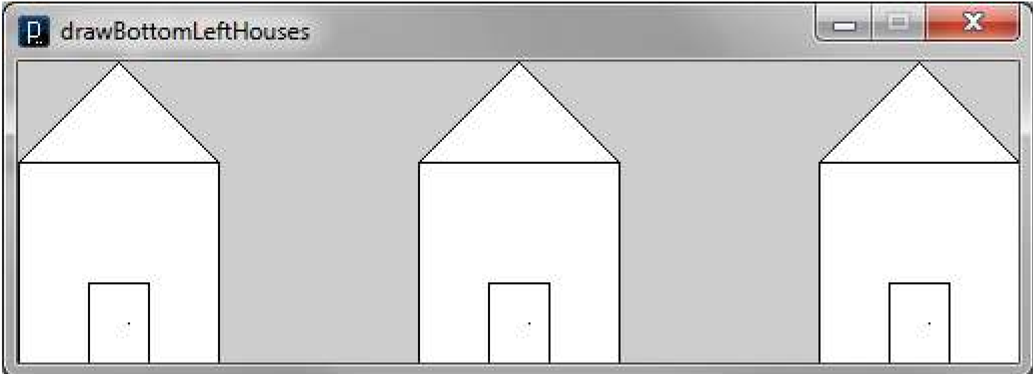
The method drawHouse(int, int, int) … is not applicable for the arguments (int, int)

A similar error would also occur if you passed in too many parameters to the procedure, or if you passed in the wrong type of values.

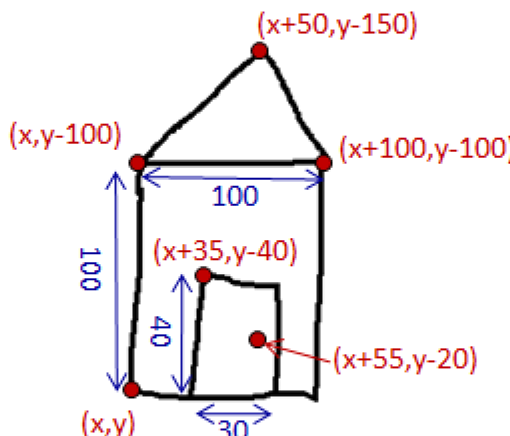
Example:

Adjust the **drawHouse()** procedure to take both an **x** and **y** value representing the bottom-left corner of the house and have the house drawn with respect to that coordinate. For example, if this was the code in the **setup()** method, then the picture shown would result:

```
void setup() {
    size(500,150);
    drawHouse(0, 150);
    drawHouse(200, 150);
    drawHouse(400, 150);
}
```



To do this, we will need to re-compute the coordinate values for our house points with respect to the (x,y) being the bottom left:



Now we can re-write our procedure accordingly to take the extra parameter and adjust the points:

```
void drawHouse(int x, int y) {
     rect(x,(y-100),100,100);
     triangle(x,(y-100),(x+50),(y-150),(x+100),(y-100));
     rect((x+35),(y-40),30,40);
     point((x+55),(y-20));
}
```

That was not too difficult, but it did require some computations.

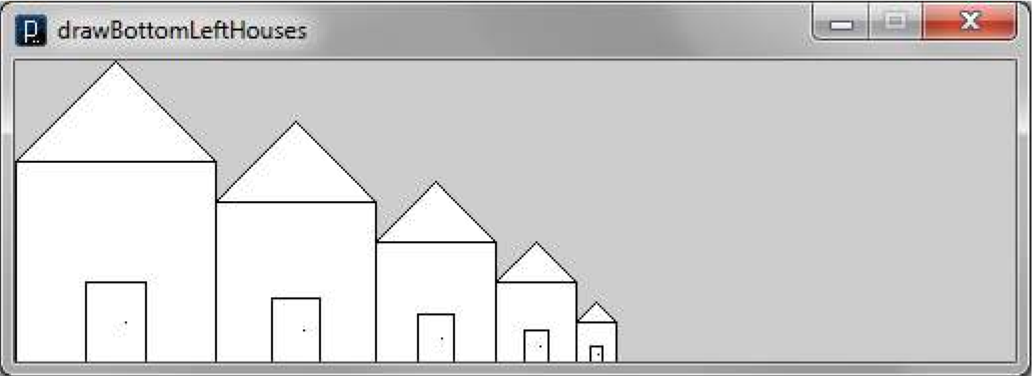
Example:

Add a **scale** parameter (i.e., a **float** between 1 and 0) as a third parameter to the **drawHouse()** procedure from the previous example.

Use the scale parameter so that the following code produces the image shown:

```
void setup() {
     size(500,150);

     drawHouse(0, 150, 1);
     drawHouse(100, 150, 0.8);
     drawHouse(180, 150, 0.6);
     drawHouse(240, 150, 0.4);
     drawHouse(280, 150, 0.2);
}
```



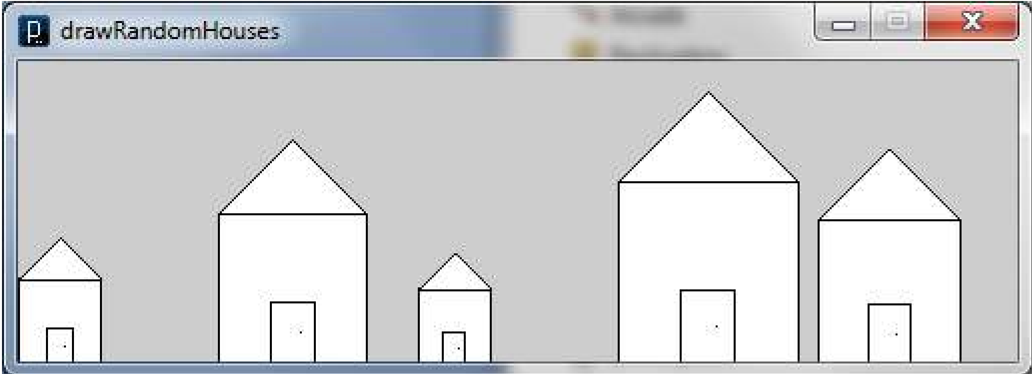
The code is not too difficult, but we must understand how the scale works. The **setup()** method has already adjusted the arguments for the position of the bottom-left corner of the houses. All that remains is to ensure that the dimensions are all somehow adjusted by the scale value:

```
void drawHouse(int x, int y, float s) {
     rect(x, y-100*s,100*s,100*s);
     triangle(x,y-100*s,(x+50*s),y-150*s,(x+100*s),y-100*s);
     rect((x+35*s),y-40*s,30*s,40*s);
     point((x+55*s),y-20*s);
}
```

Notice that we simply multiply all dimensions and offsets (i.e., any constant numbers) by the scalar value of **s**.

Example:

What if we wanted to have a random value for the scale so that our houses had different sizes each time we ran the program:



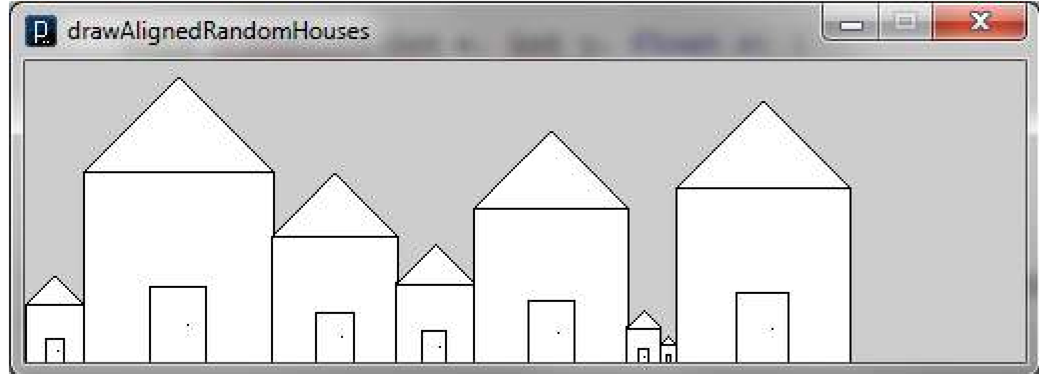
This can be done simply by making use of the **random()** function in Processing. The `random(r)` function will return a random float value in the range from **0** to **r**. (Note: the random function is described in more detail at the end of this Chapter.)

Here is how we could adjust the code:

```
void drawHouse(int x, int y) {
    float s;

    s = random(1);
    rect(x, y-100*s,100*s,100*s);
    triangle(x,y-100*s,(x+50*s),y-150*s,(x+100*s),y-100*s);
    rect((x+35*s),y-40*s,30*s,40*s);
    point((x+55*s),y-20*s);
}
```

Sometimes we need to return a value from our procedure so that we can make use of it in our main program. For example, what would we have to change in order to adjust our previous code so that it packs 8 houses close together according to their scale as follows:



Think of what has changed and develop the algorithm. It is only the x-offset for each house that must vary each time. How much does it vary each time? It varies according to how much we have scaled the house. For example, in the above image, perhaps the first house had a width of **30**. In that case the 2nd house would have an offset of **30** as its bottom-left corner. Assuming then that the 2nd house had a width of **95**, then the third house would have an offset of **95** from the 2nd house's corner (or **30+95=125** from the left side of the screen). So we can piece this together into an algorithm:

You may notice from our previous code that the width of the house is actually **100*s**, where **s** is the randomly chosen scale:

```
void drawHouse(int x, int y) {
    float s;

    s = random(1);
    rect(x, y-100*s,(100*s),100*s);
    ...
}
```

In order to calculate the offset for the next house we need to know the width of the previous house. This means we need to get information that is created inside the drawHouse() procedure back out to the setup() procedure that called it. Our drawHouse() procedure must become a ***function***.

Functions are created much the same way that procedures are with a few small changes. First, instead of **void**, a function must declare the *type* of the value returned. Second, the end of the function must contain a return statement that states what value is returned. Thirdly, a function returns a value, so a variable is used to hold the value that is returned. The format of a function is as follows:

type$_r$ myVar **=** *functionName(…);* `//returned value is saved in a variable`

type$_r$ *functionName* (t_1 n_1, t_2 n_2, …, t_k n_k) { `// defined with a return type`
    // Write your function's code here
    return someValue; `//ended with a return statement`
}

where **type$_r$** is the ***return type*** of the function (e.g. int, float, etc):

In our example, the width of the house (i.e., **100*s**) is the value that must be returned. This is an **int** type. So, here is the function that we need:

```
int drawHouse(int x, int y) {
    float s;

    s = random(1);
    rect(x, y-100*s,100*s,100*s);
    triangle(x,y-100*s,(x+50*s),y-150*s,(x+100*s),y-100*s);
    rect((x+35*s),y-40*s,30*s,40*s);
    point((x+55*s),y-20*s);

    return int(100*s);
}
```

Notice the return type (shown in blue) and that we use a ***return statement*** at the bottom to indicate what value will be returned as a result of the function call.

So then, how do we make use of this new drawHouse() **function**? Well, we need to use the **width** from the previous **drawHouse()** function call as the **xOffset** for the next house:

Notice that since the **drawHouse()** call returns the width of the drawn house, we simply keep adding these widths to the **xOffset** to draw each successive house. Here is the Processing code:

```
void setup() {
     size(500,150);

     int xOffset = 0;
     xOffset = xOffset + drawHouse(xOffset, 150);
     xOffset = xOffset + drawHouse(xOffset, 150);
     xOffset = xOffset + drawHouse(xOffset, 150);
     xOffset = xOffset + drawHouse(xOffset, 150);
     xOffset = xOffset + drawHouse(xOffset, 150);
}

int drawHouse(int x, int y) {
     float s;

     s = random(1);
     rect(x, y-100*s,100*s,100*s);
     triangle(x,y-100*s,(x+50*s),y-150*s,(x+100*s),y-100*s);
     rect((x+35*s),y-40*s,30*s,40*s);
     point((x+55*s),y-20*s);

     return int(100*s);
}
```

You will be creating many functions and procedures throughout the course. You'll get lots of practice mastering all the little details!

2.4 Conditional Statements

When describing the flow of an algorithm, one of the most fundamental ***control structures*** that you will need is the conditional statement (also known as the **if** statement). The **if** statement will allow you to only evaluate a portion of code if some **condition** is met.

The structure of the **if** statement is as follows:

```
if (condition){
     // code here is only executed if condition is true
}
else{
     // code here is only executed if condition is false
}
```

The **condition** can be any expression as long as it evaluates to a Boolean result (i.e., true or false). The section of code immediately following the if statement will execute if the condition is true. The **else** is an optional part of the **if** statement that will only execute if the condition is false. In order for an else statement to work it must always appear after an **if** statement.

Here is an example of an **if** statement being used with a Processing/JAVA program:

```
int grade = ...;        // Assume this is set to some value
...
if (grade >= 50)
    print("Congratulations! ");
```

If the **grade** variable has a value which is 50 or above, then the code prints out a nice message, otherwise, nothing is printed.

The **>=** is called a ***logical operator*** and it determines whether one number is greater than or equal to another. It takes the two numerical values, compares them, and then determines a **boolean** result of **true** or **false**. Here is the list of all the available logical operators that we can use:

- **<** less than
- **<=** less than or equal to
- **==** equal to
- **!=** not equal to
- **>=** greater than or equal to
- **>** greater than

Notice that when we want to ask if something "is equal to" another thing we use two equal signs **==**, not one. The single equal sign **=** is **only** used to assign a value to a variable.

If you want to have more than one line of code evaluated within the **if** statement, you can use braces **{ }** to specify all the code that you want evaluated when the condition is true:

```
if (grade >= 50) {
    print("Congratulations! ");
    print(grade);
    println(" is a passing grade.");
}
```

If there is only one line within the **if** body, then the braces are not needed. It is often a good idea to use the braces anyway, even if you have only one line of code because it may prevent you from making some mistakes. For example, the following code is **not** the same as above:

```
if (grade >= 50)
    print("Congratulations! ");
    print(grade);
    println(" is a passing grade.");
```

The code above will compile, however since the brackets are missing, the code is assumed to have only one line in the **if** body as follows:

```
if (grade >= 50)
    print("Congratulations! ");
print(grade);
println(" is a passing grade.");
```

If the grade is zero, this code would print out:

```
0 is a passing grade.
```

Clearly this is wrong but the program continues as if nothing bad has happened. Also, be careful not to place a semi-colon **;** after the **if** statement brackets:

```
if (grade >= 50);
    println("Congratulations! " + grade + " is a passing grade.");
```

In the above code, here is the output:

```
Congratulations! 0 is a passing grade.
```

Why? Because the semi-colon **;** at the end of the first line counts as the end of an empty expression. This empty expression counts as the body of the **if** statement. Thus, the **println(…)** line is outside the **if** statement altogether and is therefore always evaluated.

Here's an example that uses the **else** clause:

```
int myNum = makeUpANumber(); // Assume this is written elsewhere

if (myNum % 2 == 0)
    println("Even!");
else
    println("Odd!");
```

Here, the condition uses the modulus (mod) operator. It returns the remainder of division (e.g., 5 / 2 = 2 with 1 remainder). The code within the **if** body is evaluated when **myNum** is a multiple of 2 (i.e., is even). Alternatively, the **else** body is evaluated when **myNum** is not a multiple of 2 (i.e., is odd). Either way only one of the two bodies is run, and the other is skipped.

2.5 Counting and Iteration

Another way to get the most out of a little bit of code is to use repetition. The first form of repetition we'll use in Processing/Java is the ***for loop***. It lets us repeat a section of code a set number of times. The structure of the for loop is as follows:

```
for (initializer ; loopTest ; countExpression){
        // repeated code goes here
}
```

A for loop has three parts separated by semi-colons. Each of these parts is explained here:

- **initializer** – this is usually used to declare and initialize (i.e., set the starting value for) a variable (called the ***loop variable***) which will be used as a counter within the loop. The loop variable can be used anywhere within the **FOR** loop but not outside of it. In most situations, this counter starts off at 0 or 1, but there are some times when you will use other values.

- **loopTest** – this is any coding expression that results in a boolean result. It is used to determine whether or not to go back into the loop again for another round. Usually, this loop will check if the loop variable has reached some kind of limit. As long as the boolean expression results in **true**, the loop will repeat again.

- **countExpression** – this is a portion of code that is evaluated AFTER each time the loop has completed one iteration (i.e., one round). It is usually used to increase or decrease the value of the loop variable by some value (such as 1), although this not always the case.

The code within the braces is called the ***loop body***, and it can be any chunk of code which will be evaluated over and over again depending on the code within the parentheses.

Example:

Assume that we want to stack concrete slabs on top of each other to form a staircase. Develop an algorithm that will determine how many slabs would be needed to create a staircase **n** stairs high?

To begin, you should realize that **n** may be a very large number. What is our mathematical model? This is how many numbers we need:

1 + 2 + 3 + 4 + … + n

Some of you may realize that this value can be computed as **n(n+1)/2**. However, assume that we are unaware of that nifty formula. How would you go about solving this problem? You might realize that some kind of counter is required (i.e., a variable) and that we need to keep adding an increasingly large integer to the count. Here is one possible solution:

```
int total = 0;
for (int currentHeight = 1; currentHeight <= n; currentHeight++) {
    total = total + currentHeight;
}
print(total);
```

Here the **currentHeight** represents the number of stairs to stack at each level. The counter goes through the values of 1, 2, 3, 4, …, **n**. These are exactly the values that we want to add together … and we do so with the **total** variable.

This **for** loop contains a loop variable called **currentHeight**, that is initialized to the value **1** in the **initializer**. According to the **loopTest**, the **for** loop continues so long as **currentHeight <= 10**. Each *iteration*, the value of **currentHeight** is added to sum in the ***loop body***, then **currentHeight** is increased by 1 in the **countExpression**. This process repeats until **currentHeight** == 11, at which point the **loopTest** is false, and processing continues with any lines that follow the for loop's closing brace. This simple form of counter will be commonly used throughout your programs.

The **countExpression** has a **++** at the end of the **personNumber** variable. This is called the ***increment operator***. It has the same result as doing:

personNumber = **personNumber** + 1;

This is evaluated AFTER each iteration of the loop and BEFORE the **loopTest** is performed again. There is also a ***decrement operator*** **--** which has the same result as subtracting **1** from the variable. It is very useful when you are counting down from a number.

Whenever the body of a **for** loop has only one line of code in it, then the brace **{ }** characters are not needed. Also, in order to keep code simple to read, a programmer will often use the letter **i** to represent the loop variable (**i** being short for "*index*"). Here is the short version:

```
int total = 0;
for (int i=1; i<=n; i++)
    total = total + i;
print(total);
```

Example:

Returning to our draw house examples, how could we use **for** loops to make our code even more compact? Consider our algorithm as written below:

```
void setup() {
    size(500,150);

    drawHouse(0, 150);
    drawHouse(100, 150);
    drawHouse(200, 150);
    drawHouse(300, 150);
    drawHouse(400, 150);
}

void drawHouse(int x, int y) {

    rect(x, y-100,100,100);
    triangle(x,y-100,(x+50),y-150,(x+100),y-100);
    rect((x+35),y-40,30,40);
    point((x+55),y-20);
}
```

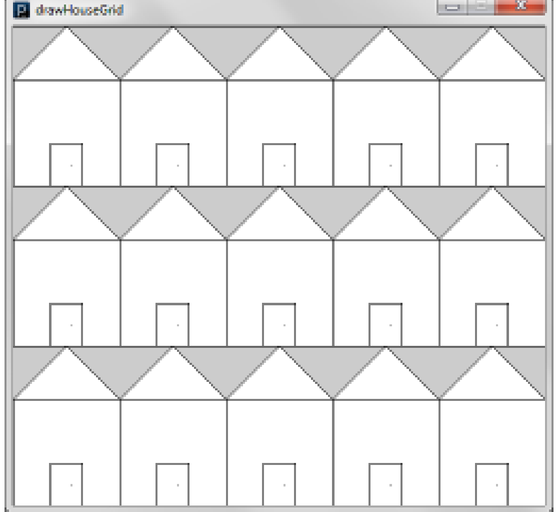
Notice all of the repetition in the setup method? Surely we can clean that up with a loop… To do so, take note of what is repeating, and what is changing. In the above code, there are five repeated calls to the drawHouse() procedure, and the only thing changing is the value of the first argument. Converting that into a loop is actually quite easy, as seen here:

```
void setup() {
    size(500,150);
    for(int i=0; i<=400; i=i+100)
        drawHouse(i, 150);
}
```

The values that our loop variable **i** takes on in the **for** loop are exactly those values from our original code, so the exact same row of houses will be drawn!

Example:

Using what we've learned about **for** loops so far, how could we draw an image like the one below?



The key to solving this problem is understanding that the **loop body** of a **for** loop can contain any normal code – *including other **for** loops!* We already know how to draw one row of houses, all we need to do now is repeat that 3 times at different heights. Let's see what this looks like in processing:

```
void setup() {
   size(500,450);
   for(int j=150; j<=450; j=j+150) // Outer loop
     for(int i=0; i<=400; i=i+100) // Inner loop
        drawHouse(i, j);
}
```

Here we have one **for** loop inside another **for** loop. This is a process called ***nested loops*** or ***nesting***. (Note that the variable name '**j**' does not stand for anything, but is nonetheless a common loop variable because it follows '**i**'.)

Nested loops are quite common in programming. They often appear in situations involving data arranged in a grid pattern such as applications that manipulate data tables, pictures or images, and graphics involving x/y coordinate systems.

You might be concerned that the above code has an error: there are two lines in the **loop body** of the outer loop, but we didn't use braces! However, this is not an error since we are using **nested loops.** In this code there is only one line in the outer loop, and one line in the inner loop. That is to say, the outer loop (the one with '**j**') only does one thing, it repeatedly runs the inner loop. Similarly, the inner loop (the one with '**i**') only does one thing, it repeatedly calls the drawHouse() procedure.

How many times is the drawHouse() procedure called? We can see from the above image that there are 15 houses, so it must be called 15 times, but why is that? It is important to understand that using nested loops involves multiplication. The variable '**i**' takes on 5 values (0, 100, 200, 300, 400), and with each new value draws a house. The variable '**j**' takes on 3 values (150, 300, 450), but with each new value, it starts a ***new*** inner loop. So, if the inner loop draws 5 houses, and each *iteration* of the outer loop runs one inner loop… Then with 3 outer loops we get: 5 houses + 5 houses + 5 houses = 3 x 5 houses = 15 houses!

2.6 Conditional Iteration

In the above examples, we used variables and loops to show how we can count within an algorithm. In each case there were a fixed number of items to iterate through. However, the situation often arises if real life when the total number of items to iterate through is unknown.

For example, a cashier must be able to repeat the scanning of items from a customer without knowing exactly how many items there will be (it could be 1, 10, 50, etc…). In fact, in this case the number of items is not important, only the final cost of the items being purchased is required. In such a situation, the repeated scanning of items will occur until some kind of ***condition*** is satisfied.

If we want to perform some repetitive action, but don't know how many times to repeat it, we require ***conditional looping*** (i.e., looping that requires some kind of stopping condition). To do this, Processing/Java provides the **while** loop:

```
while (condition){
    // repeated code goes here
}
```

This will repeat whatever code is in the **loop body** forever, so long as the ***condition*** evaluates to **true**.

Example:

Returning to the cashier example, an algorithm to describe a cashier's behaviour might look something like this:

```
float cost = 0.00;
while (itemsOnTheConveyor()) {
   float price = scanNextItem();
   cost = cost + price;
}
println("Your bill comes to: " + (cost*1.13));
```

In this code, the function called **itemsOnTheConveyor()** will return a boolean value. It is evaluated before every iteration, and if it evaluates to **true**, then the loop body is executed. If **itemsOnTheConveyor()** returns false, the loop is terminated and execution resumes on the next line after the **while** loop (in this case, the println() statement). Note that if **itemsOnTheConveyor()** always returns true that the loop will continue forever!

Just as with **for** loops, you should be careful not to put a semi-colon **;** after the parentheses, otherwise your *loop body* will not be evaluated. Usually your code will loop forever because the *stopping condition* may never change to **false**:

```
while (...); {                    // This code will loop forever
      ...
}
```

As with the **IF** statements and **FOR** loops, the braces **{ }** are not necessary when the *loop body* contains a single coding expression:

```
while (...)
      println(...);
```

Some students tend to confuse the **WHILE** loop with **IF** statements and try to replace an **IF** statement with a **WHILE** loop. Do you understand the difference in the two pieces of code below ?

```
if (age > 18)
    discount = 0;
```

```
while (age > 18)
    discount = 0;
```

Assume that the person's age is **20**. The leftmost code will set the discount to **0** and move on. The rightmost code will loop forever, continually setting the discount to **0**.

As it turns out, every **for** loop can be expressed in terms of a **while** loop. For example, recall the algorithm we used to sum the numbers between 1 and 10:

```
int total = 0;
for (int i=1; i<=10; i++)
    total = total + i;
print(total);
```

Here it is re-written using a **while** loop:

```
int total = 0;
int i=1;
while (i<=10){
    total = total + i;
    i++;
}
print(total);
```

However, as a "rule of thumb", a **while** loop should only be used when there is **uncertainty** in how many times the loop will occur. That is, you should only use a **while** loop when the condition to stop the loop is generated from an unexpected event, not when a fixed counter is to be used.

2.7 Practice examples!

When developing your own algorithms to solve a problem, it is important to understand the difference between what has already been given to you (i.e., parameters) and what you need to figure out on your own. In the following examples, see if you can identify the algorithm parameters and develop a computational model by figuring out what additional computations and variables that you will need to solve the problem.

Example:

Bob and Steve went on a vacation together. During the trip Bob paid for all the food and for the hotel. Steve paid for the gas and for the entertainment. Write code that computes the amount of money that Bob owes Steve (or Steve owes Bob) after the trip, assuming that they decided to split the expenses evenly.

First, we need to determine what data (i.e., algorithm parameters) that we are given. From the problem, we see that they are the expenses incurred by Bob and Steve. If we were to develop an application for this problem these expenses would likely be entered into the user interface (perhaps through a series of text fields). To keep things simple, however, we will represent these expenses as simple **float** variables.

```
float   f = 187.23;    // food expenses (paid by Bob)
float   h = 458.91;    // hotel expenses (paid by Bob)
float   g = 133.67;    // gas expenses (paid by Steve)
float   e = 67.54;     // entertainment expenses (paid by Steve)
```

Next, we need to figure out the computations required to obtain the solution. Since we are splitting the costs between two people, we should all expenses and cut the total in half to determine how much each one pays. We can store this result in a variable as well:

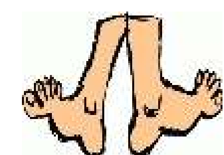
```
float each;     // cost that each person should pay
each = (f + h + g + e) / 2;
```

Finally, we need to determine who owes who. If Bob paid more than Steve, then **(f + h)** will be greater than **(g + e)**. In that case Steve would owe Bob **(f + h) - each**. Otherwise, Bob will owe Steve **(g + e) - each**:

```
// Determine who owes the money
if ((f+h) > (g+e))
   println("Steve owes Bob $" + (f + h - each));
else
   println("Bob owes Steve $" + (g + e - each));
```

Example:

There are **n** kids in a room. As it turns out, some kids have socks on (with or without shoes), some kids are wearing shoes (with or without socks), and some kids are wearing both socks & shoes. Develop a computational model & algorithm to determine how many kids are barefoot?



What are the algorithm parameters?

We know there are **n** kids, which can vary, so **n** is a parameter.

```
int n = 34;  // #kids in room (34 arbitrarily chosen for example)
```

We will also need to know which of these kids are wearing socks and which ones are wearing shoes. There are 3 combinations (again numbers chosen arbitrarily for this example):

```
int socks = 23;              // #kids who have socks on
int shoes = 16;              // #kids who have shoes on
int socksAndShoes = 7;       // #kids with socks AND shoes on
```

So, how many kids are barefoot? Well, anyone who is not wearing socks or shoes. At first, you may think that the answer is **(n - socks - shoes)**. However, some of the kids are wearing both socks AND shoes together. Therefore these kids would be deducted twice from the **n** kids, resulting in the wrong answer. We must therefore need to distinguish the kids who are ONLY wearing socks and ONLY wearing shoes:

```
int socksOnly = socks - socksAndShoes;
int shoesOnly = shoes - socksAndShoes;
```

Now we can compute the final answer by subtracting all **socksOnly**, **shoesOnly** and **socksAndShoes** kids from the total ... leaving the barefoot kids as an answer:

```
println("Barefoot = " + (n - socksOnly - shoesOnly - socksAndShoes));
```

Interestingly, we do not need the intermediate variables socksOnly and shoesOnly. That is because we are only storing this value for use exactly one time later in the program. So we could have simply written the following:

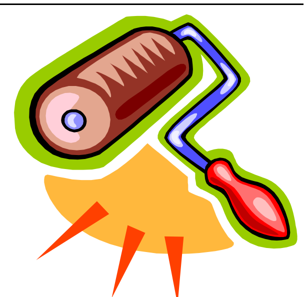
```
println("Barefoot = " + (n - (socks - socksAndShoes) -
                             (shoes - socksAndShoes) - socksAndShoes));
```

which is just this simple result:

```
println("Barefoot = " + (n - socks - shoes + socksAndShoes));
```

Example:

A team of **n** people work *together* painting houses for the summer. For each house they paint they get **$256.00**. If the people work for **4** months of summer and their expenses are **$152.00** per month, how many houses must they paint for each of them to have one thousand dollars at the end of the summer?



Once again, start by determining what information you have available as incoming parameters:

```
int     n = 6;                        // people painting the houses
float   incomePerHouse = 256.00;      // money for painting 1 house
int     monthsToWork = 4;             // months for painting
float   monthlyExpenses = 152.00;     // monthlyExpenses
```

OK. Now we just need to determine the computations. We need to think of what we are trying to do. If each person wants to make $1000 by the end of the 4 months, then as a group, they need to earn **$1000 x n** plus enough to cover their expenses. The expenses are $152 per month for 4 months. This is the "goal" amount of money to be earned:

```
float  goal = (n * 1000) + (monthsToWork * monthlyExpenses);
```

Now that we know how much money the group needs to make, we need to determine how many houses need to be painted in order to make that amount of money:

```
println("Houses = " + (goal / incomePerHouse));
```

That's it ! Notice that the **1000** is a fixed/constant value, as opposed to a parameter.

A constant is a single piece of data that does not change throughout the algorithm

In a more general version of this problem, we can make any (or all) of these values to be adjustable parameters.

In each of the above examples, the parameters and variables are all numbers. When programming, however, sometimes the variables and parameters may be of a different nature. For example, sometimes the input to an algorithm may be in the form of text, such as a person’s name or an address. Or perhaps the variables come in the form of yes/no answers (i.e., true/false). Consider these next examples ...

Example:

Assume that you want to take a vote among **5** friends to find out whether or not they agree to some issue (e.g., like if wearing speedos at a pool party is acceptable). Each person votes yes or no. Develop an algorithm that determines the majority response (either yes or no).

Once again, we begin by identifying the incoming parameters ... which is the 5 votes. Each vote is either "yes" or "no" ... which can be stored as a String in Processing/Java:

```
// The 5 votes (values arbitrarily chosen for this example)
String vote1 = "yes";
String vote2 = "no";
String vote3 = "no";
String vote4 = "yes";
String vote5 = "no";
```

To determine the majority, we will need to count the "yes" (i.e., true) votes and the "no" (i.e., false) votes and compare the counts. We can actually use just count the "yes" votes since the "no" votes is 5 minus the "yes" votes. Here is the counter (set to zero because we have not started counting yet):

```
int     yesVotes = 0;
```

Now we just need to add them all up:

```
if (vote1 == "yes")
  yesVotes = yesVotes + 1;
if (vote2 == "yes")
  yesVotes = yesVotes + 1;
if (vote3 == "yes")
  yesVotes = yesVotes + 1;
if (vote4 == "yes")
  yesVotes = yesVotes + 1;
if (vote5 == "yes")
  yesVotes = yesVotes + 1;
```

Finally, to determine the majority, we compare the two counters:

```
if (yesVotes >= 3)
  println("Speedos allowed");
else
  println("No speedos allowed");
```

An alternative solution can make use of the char data type, so that the votes are just single characters **'y'** or **'n'** as follows:

```
char    vote1 = 'y';    // The 5 votes (values arbitrarily
char    vote2 = 'n';    //              chosen for this example)
char    vote3 = 'n';
char    vote4 = 'y';
char    vote5 = 'n';
...
if (vote1 == 'y')
   yesVotes = yesVotes + 1;
...
```

We could have also done this with boolean values.

> *A **boolean** is a value that is either **true** or **false**.*

In other words, **yes** is the same as **true** and **no** is the same as **false** like this:

```
// The 5 votes (values arbitrarily chosen for this example)
boolean     vote1 = true;
boolean     vote2 = false;
boolean     vote3 = false;
boolean     vote4 = true;
boolean     vote5 = false;
...
if (vote1 == true)
   yesVotes = yesVotes + 1;
...
```

Choosing the appropriate data type can be beneficial. For example, we could chose **int** as our data type, setting the "yes" votes to 1 and "no" votes to 0:

```
// The 5 votes (values arbitrarily chosen for this example)
int     vote1 = 1;
int     vote2 = 0;
int     vote3 = 0;
int     vote4 = 1;
int     vote5 = 0;
...
if (vote1 == 1)
    yesVotes = yesVotes + 1;
...
```

By doing this, we can really simplify the code by eliminating all of the **IF** statements to add up the votes and we don't really need the **yesVotes** variable at all:

```
if ((vote1 + vote2 + vote3 + vote4 + vote5) >= 3)
   println("Speedos allowed");
else
   println("No speedos allowed");
```

Example:

Consider developing a simple computational model that computes a price for patrons who want to go to the theatre. Assume that there is a discount of 50% for women that are senior (i.e., 65 or older) or to girls who are 12 and under. For all other people, the discount should otherwise be 0%. Develop an algorithm that displays the appropriate discount for a particular person buying the ticket.

We can define the incoming parameters to be the **age** and **gender** of the person:

```
// The age and gender is chosen arbitrarily for this example
int     age = 72;
char    gender = 'f';
```

Now we need to determine the discount by considering the age and gender as given in the problem definition:

```
int     discount = 0;
if (gender == 'f') {
    if (age >= 65)
        discount = 50;
    else if (age <= 12)
        discount = 50;
}
println("The discount is " + discount + "%");
```

Notice that there is no need for an **else** statement for the first **if** because the discount was set to zero and a decision was only necessary to set it to 50 in the two particular cases.

We can actually simplify this code by combining the **if** statements. To do this, we need to make use of some other logical operators which are common in computer programming. They are known as **AND**, **OR** and **NOT** operators. They allow you to work with Boolean values (i.e., true/false values) to combine them in logical ways in order to achieve an overall Boolean result.

Notice that the person gets a discount if the gender is female and the person is either 65 or older **OR** the person is 12 or under. The 2nd and 3rd **if** statements can be combined logically as follows:

> **if age** >= 65 **OR age** <= 12 **then** the discount = 50%.

Being of female gender is also a requirement for the discount. So, we can even combine the 1st **IF** statement as well using an **AND**:

> **if gender** is female **AND** (**age** >= 65 **OR age** <= 12) **then** the discount = 50%.

Notice that we put brackets around the age-checking code. This is important. Notice how the decision to place parentheses can alter the logic of the expression and reduce ambiguity:

> **if** (**gender** is female and **age** >= 65 or **gender** is female and **age** <= 12) **then …**
> **if** (**gender** is female and **age** >= 65) or (**gender** is female and **age** <= 12) **then …**
> **if** (**gender** is female and (**age** >= 65 or **gender** is female) and **age** <= 12) **then …**
> **if** (**gender** is female and (**age** >= 65 or **gender** is female and **age** <= 12)) **then …**

Which is the correct understanding of the problem? The 2nd one. When programming, it is important to be as clear as possible in your code. Therefore, try to be aware of the need for parentheses when the code seems complex.

Below is a "truth table" explaining the results of using any two boolean values, say b_1 and b_2, in an **if** statement:

b_1	b_2	if (b_1 **and** b_2)	if (b_1 **or** b_2)	if (**not** b_1)	if (b_1)
false	false	false	false	**true**	false
false	**true**	false	**true**	**true**	false
true	false	false	**true**	false	**true**
true	**true**	**true**	**true**	false	**true**

Notice that the **and** results in **true** only when both Booleans are **true**, and **false** otherwise. Conversely, the **or** results in **false** only when both Booleans are **false**, and **true** otherwise. Also note that the **not** results in the opposite value of the Boolean. Of course, we can combine multiple **and/or/not** operators within the same **if** statement as in our example.

In Processing/JAVA, we cannot use AND, OR and NOT. Instead, the following three **boolean** operators are to be used:

- **&&** … the same as saying **AND**
- **||** … the same as saying **OR**
- **!** … the same as saying **NOT**

So here would be the resulting Processing/JAVA code:

```
int     discount = 0;
if (gender == 'f') {
    if ((age >= 65) || (age <= 12))
        discount = 50;
}
println("The discount is " + discount + "%");
```

or ...

```
int     discount = 0;
if ((gender == 'f') && ((age >= 65) || (age <= 12)))
        discount = 50;
println("The discount is " + discount + "%");
```

Example:

Consider writing an algorithm that takes the number grade of a student (i.e., from **0%** to **100%**) and outputs a letter grade (from **F** to **A+**). To do this, we need to first understand the computational model … that is, which letter grade corresponds to which number grades:

A = 80% - 100%
B = 70% - 79%
C = 60% - 69%
D = 50% - 59%
F = 0% - 49%

The input parameter is the number grade:

```
// Value chosen arbitrarily for this example
int   grade = 68;
```

Now to solve the problem, we need to check each of the grade ranges in turn:

```
if (grade >= 80)
     print("A");
if ((grade >= 70) && (grade <= 79))
     print("B");
if ((grade >= 60) && (grade <=69))
     print("C");
if ((grade >= 50) && (grade <= 59))
```

```
        print("D");
if (grade < 50)
        print("F");
```

Notice how each **if** statement checks to see whether the **grade** lies within a specific range. This code will ensure that the correct grade letter is printed. However, there is a small issue with the above code in regards to efficiency. The algorithm is correct, but it is not efficient.

Assume that the grade entered was **92%.** The first **if** statement will be evaluated and the condition will be satisfied resulting in "A" being printed. However, the algorithm will then continue to check the conditions of all the other if statements as well … which will all evaluate to **false** anyway. Precious computer CPU (i.e., processing) time is wasted by evaluating the remaining **if** statements since the result was already obtained from the first **if** statement.

We can avoid this inefficiency by created ***nested if statements***. This means that we insert successive **if** statements and make use of the **else** portion of the **if** statements as follows:

```
if (grade >= 80)
        print("A");
else {
        if ((grade >= 70) && (grade <= 79))
                print("B");
        else {
                if ((grade >= 60) && (grade <=69))
                        print("C");
                else {
                        if ((grade >= 50) && (grade <= 59))
                                print("D");
                        else {
                                if (grade < 50)
                                        print("F");
                        }
                }
        }
}
```

Notice how the successive **if** statements are shown indented, as they lie within the **else** part of the previous **if** statement. What happens if **92%** is entered ? The first **if** statement is evaluated and the **else** part of it is ignored. So, none of the other **if** statement conditions are evaluated. This is more efficient.

Notice now the 2nd **if** statement that checks the "B" range. Since it is in the **else** part of the first **if** statement, the **grade** must therefore be less than or equal to **79**, since if not, the algorithm would have stopped on the first **if** statement. So, we do not need to check whether or not the **grade** is less than **79** in the 2nd **if** statement. Likewise, the upper bound of **69** and **59** need not be checked in the other **if** statements. Therefore, here is even better code:

```
if (grade >= 80)
    print("A");
else {
    if (grade >= 70)
        print("B");
    else {
        if (grade >= 60)
            print("C");
        else {
            if (grade >= 50)
                print("D");
            else
                print("F");
        }
    }
}
```

Often, when we have a sequence of such nested **if/else** statements, we can simplify the writing of the code further. First, the braces are not needed since there is only one JAVA statement for each **else** body ... which is actually the next **if/else** statement. The entire **if/else** along with the code inside of it actually acts as **one** Processing/JAVA statement, although it is complex. Since the braces are not needed then, we often align the **if** statements on the left side as follows:

```
if (grade >= 80)
    print("A");
else if (grade >= 70)
    print("B");
else if (grade >= 60)
    print("C");
else if (grade >= 50)
    print("D");
else
    print("F");
```

This code is much more readable, but it requires you to understand that only one of the **if** statement bodies will be evaluated.

Example:

Consider another example in which we are given an integer representing a **month** and we would like to determine the number of days in that month (we will assume that it is not a leap year). Here is the table of information that we need to know to begin:

Month	**Jan**	**Feb**	**Mar**	**Apr**	**May**	**Jun**	**Jul**	**Aug**	**Sep**	**Oct**	**Nov**	**Dec**
Days	31	28	31	30	31	30	31	31	30	31	30	31

Here is a solution based on the incoming month parameter, which should be an integer in the range from 1 to 12:

```
// Value of 7 has been arbitrarily chosen for this example
int  month = 7;

if (month == 1)
     print(31);
else if (month == 2)
     print(28);
... etc ...
else if (month == 12)
     print(31);
```

However, you can see that the combined **if** statements will take up 24 lines of code! Since there are only 3 values for the days in the months (i.e., 31, 30 and 28), there should be a way to arrange it in a format like this …

```
if (...) then          // Jan, Mar, May, Jul, Aug, Oct, Dec
   print 31
else if (...) then     // Apr, Jun, Sep, Nov
   print 30
else print 28          // Feb only
```

We would just need to group the months into the appropriate category and use **||** operators to decide which ones match the corresponding **if** statement:

```
if ((month == 1) || (month == 3) || (month == 5) || (month == 7) ||
    (month == 8) || (month == 10) || (month == 12))
     print(31);
else if ((month == 4) || (month == 6) || (month == 9) || (month == 11))
     print(30);
else
     print(28);
```

This seems much shorter. How can we shorten the algorithm description even further?

We can re-arrange the code to check for the 28 day month first, and the 31 day months last:

```
if (month == 2)
     print(28);
else if ((month == 4) || (month == 6) || (month == 9) || (month == 11))
     print(30);
else
     print(31);
```

Wow. The code works the same way, but is much shorter, cleaner and nicer. It is often the case that we can re-arrange our algorithm steps in this manner like this in order to make it more readable and simpler to understand.

In special cases where there is a list of fixed values that we want to make a decision on (e.g., month is a number from 1 to 12), we can use what is known as a ***switch statement***.

The **switch** statement has the following format:

```
switch (aPrimitiveExpression) {
    case val1:
        /*one or more lines of JAVA code*/;
        break;
    case val2:
        /*one or more lines of JAVA code*/;
        break;
    ...
    case valN:
        /*one or more lines of JAVA code*/;
        break;
    default:
        /*one or more lines of JAVA code*/;
        break;
}
```

In the above code, **aPrimitiveExpression** is either a primitive variable (e.g., a variable of type **int**, **char**, **float**, etc…) or any code that results in a primitive value. The values of $\mathbf{val_1}$, $\mathbf{val_2}$, …, $\mathbf{val_N}$ must all be primitive constant values of the same type as **aPrimitiveExpression**.

The **switch** statement works as follows:

1. It evaluates **aPrimitiveExpression** to obtain a value (the expression MUST result in a primitive data type (e.g., int, char, etc.), it **cannot** be an object (e.g., string, more on this later)).

2. It then checks the values **val1**, **val2**, …, **valN** in order from top to bottom until a value is found equal to the value of **aPrimitiveExpression**. If none match, then the **default** case is evaluated.

3. It then evaluates the statements corresponding to the **case** whose value matched.

4. If there is a **break** at the end of the lines of code for that **case**, then the **switch** statement quits. Otherwise it continues to evaluate *all* of the successive **case** statements that follow(!) ... until a **break** is found or until no more cases remain.

Here is how we can use a **switch** statement for our **DaysInMonth** code …

```
switch(month) {
    case 2: print(28); break;
    case 4:
    case 6:
    case 9:
    case 11: print(30); break;
    default: print(31);
}
```

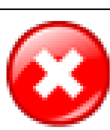
Note that when the month is **4**, **6**, **9**, or **11**, then the **print(30);** is evaluated. The code is not necessarily much shorter, but it is simpler to read. This is the main advantage of a **switch** statement. However, the value of the cases must be **primitive literals**. That is, they cannot

be expressions, ranges (nor Strings). Nor can we make use of the logical operators such as **and** and **or**. So these three examples will not work:

```
switch (age) {
    case 1 to 12:  price = 5.00; break;   // Won't compile
    case 13 to 17: price = 8.00; break;   // Won't compile
    case 18 to 54: price = 10.00; break;  // Won't compile
    default:    price = 6.00;
}
```

```
switch (name) {
    case "Mark":   bonus = 3; break; // Won't compile
    case "Betty":  bonus = 2; break; // Won't compile
    case "Jane":   bonus = 1; break; // Won't compile
    default:       bonus = 0;
}
```

```
switch (month) {
    case 2:                     print(28); break;
    case 4 || 6 || 9 || 11: print(30); break;   // Won't compile
    default:                    print(31);
}
```



Example:

Consider writing a program that will be placed at a kiosk in front of a bank to allow customers to determine whether or not they qualify for the bank's new "Entrepreneur Startup Loan". Assume that this kind of loan is only given out to someone who is currently employed and who is a recent University graduate, or someone who is employed, over 30 years of age and has at least 10 years of full-time work experience.

The program should display information to the screen as well as ask the user various questions … and then determine if the person qualifies.

What questions should be asked?

- Are you currently employed?
- Did you graduate with a university degree in the past 6 months?
- How old are you?
- How many years have you been working at full time status?

What are the parameters for this problem?

Well ... it is the user responses to the above questions. We will therefore need to ask the user for his/her responses. Unfortunately, there is no simple command in Processing to grab an integer from the user. So, we will be using some JAVA code in this example (although the code also works in Processing). However, we do not want to explain the code in this course. So, you will not need to remember or memorize how to get input from the user in this manner. For now, just know that the following code gets the answers (i.e., incoming parameters) to the above 4 questions:

```
String  employed = javax.swing.JOptionPane.showInputDialog(
     "Are you currently employed ?");
String  degree = javax.swing.JOptionPane.showInputDialog(
     "Did you graduate with a university degree in the past 6 months ?");
int  age = Integer.parseInt(javax.swing.JOptionPane.showInputDialog(
     "How old are you ?"));
int  years = Integer.parseInt(javax.swing.JOptionPane.showInputDialog(
     "How many years have you been working at fulltime status ?"));
```

Note that the variables **employed** and **degree** will hold a String which should be either "yes" or "no". We are not doing any error-checking in our code to look for a variety of input such as "YES", "Yes", "y", "Y", "NO", "No", "n", "N" ... which are also valid logically.

Now, how do we solve the problem? Well, we need to consider the cases that allow a person to qualify and those that don't. Then, we need to write appropriate code using correct logic:

```
if (employed == "YES") {
   if (degree == "YES")
      println("Congratulations, you qualify for the loan.");
   else {
      if (age >= 30) {
         if (years >= 10)
            println("Congratulations, you qualify for the loan.");
         else
            println("Sorry. You do not qualify.  You must have " +
                    "worked at least 10 years at full time status.");
      }
      else
         println("Sorry. You do not qualify.  You must be a " +
                 "recent graduate or at least 30 years of age.");
   }
}
else
   println("Sorry. You must be currently employed to qualify.");
```

You may have noticed that some **if** statements are nested within others. Of course, the order that the **if** statements are evaluated in can vary. That is, the check for employment can be done after the check for the degree, age and years worked. However, since employment is necessary in all special cases, it is good to check for that first so that the user code completes quicker when the user is unemployed. In fact, we could intermix the user input with the **IF** statements as follows:

```
String  employed = ...;
if (employed == "YES") {
    String  degree = ...;
    if (degree == "YES")
        println("Congratulations, you qualify for the loan.");
    else {
        int  age = ...;
        if (age >= 30) {
            int  years = ...;
            if (years >= 10)
                println("Congratulations, you qualify for the loan.");
            else
                println("Sorry. You do not qualify.  You must have " +
                        "worked at least 10 years at full time status.");
        }
        else
            println("Sorry. You do not qualify.  You must be a " +
                    "recent graduate or at least 30 years of age.");
    }
}
else
    println("Sorry. You must be currently employed to qualify.");
```

This code may seem a little more cluttered, but it has the advantage that the program ends quickly and abruptly as soon as any information is entered from the user that disqualifies him/her. After all, there is nothing more annoying that having to fill out a form with a lot of information in it only to find out that the first piece of information disqualified you!

In time, you will get used to adjusting your code accordingly to make it more efficient and user-friendly.

Although **if** statements are quite easy to use, it is often the case that students do not **fully** understand how to use **boolean** logic. As a result, sometimes students end up writing overly complex and inefficient code ... sometimes even using an **if** statement when it is not even required!

To illustrate this, consider the following examples of "BAD" coding style. Try to determine why the code is inefficient and how to improve it. If it is your desire to be a good programmer, pay careful attention to these examples.

Example 1:

```
boolean   male = ...;

if (male == true)
    println("male");
else
    println("female");
```

Here, the **boolean** value of **male** is *already* **true** or **false**, we can make use of this fact:

```
boolean   male = ...;

if (male)
   println("male");
else
   println("female");
```

Example 2:

```
boolean   adult = ...;

if (adult == false)
     discount = 3.00;
```

Here is a similar situation as above, but with a negated **boolean**. Below is better code.

```
boolean   adult = ...;

if (!adult)
     discount = 3.00;
```

Example 3:

```
boolean   tired = ...;

if (tired)
     result = true;
else
     result = false;
```

Above, we are actually returning the identical **boolean** as **tired**. No **if** statement is needed:

```
boolean   tired = ...;

result = tired;
```

Example 4:

```
boolean   discount;

if ((age < 6) || (age > 65))
     discount = true;
else
     discount = false;
```

The discount is solely determined by the **age**. No **if** statement is needed:

```
boolean discount;

discount = (age < 6) || (age > 65);
```

Example5:

```
boolean   fullPrice;

if ((age < 6) || (age > 65))
     fullPrice = false;
else
     fullPrice = true;
```

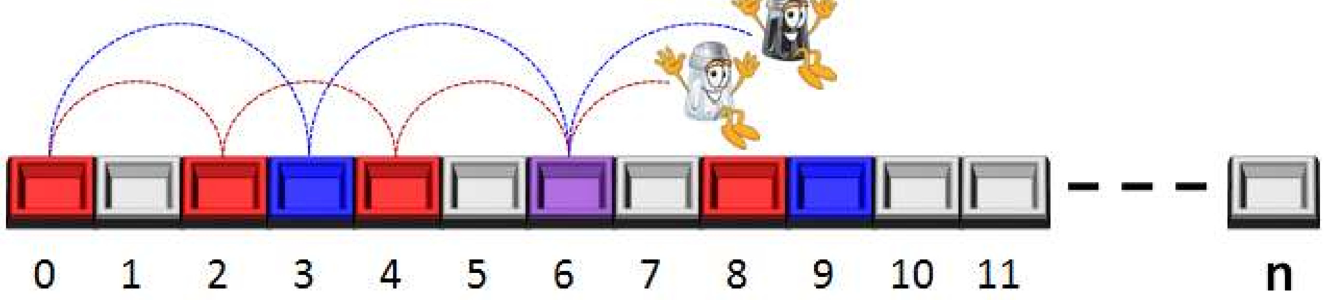
Just like above, we do not need the **if** statement:

```
boolean   fullPrice;

fullPrice = !((age < 6) || (age > 65));
or …
fullPrice = (age >= 6) && (age <= 65);
```

Example:

Imagine that you are creating a game where two players are moving along a one-dimensional grid (i.e., path). One player always jumps forward **2** steps at a time, while the other always jumps forward **3** steps at a time. Develop an algorithm that figures out how many grid locations have not been landed on if the two players start at the same location (i.e., 0) and they each jumped up to grid location **n**.



How do we approach this problem? First, examine the grid locations covered by player 1:

0, 2, 4, 6, 8, 10, 12, 14, … (seems to be the even numbered locations)

and those covered by player 2:

0, 3, 6, 9, 12, 15, 18, 21, … (seems to be the locations that are multiples of 3)

We could try to figure out a formula… but can we do this with some kind of loop counter? What if we examine each location at a time … can we determine by the location number whether or not it would be landed on?

```
int spotsNotLandedOn = 0;

for (int i=0; i<=n; i++) {
   if ((i%2 != 0) && (i%3 != 0))
      spotsNotLandedOn++;
}
println(spotsNotLandedOn);
```

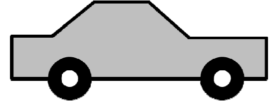
How could we change the algorithm so that we displayed a list of all the locations that the two players met at along the way?

We would just need to find the locations that were multiples of 2 or 3:

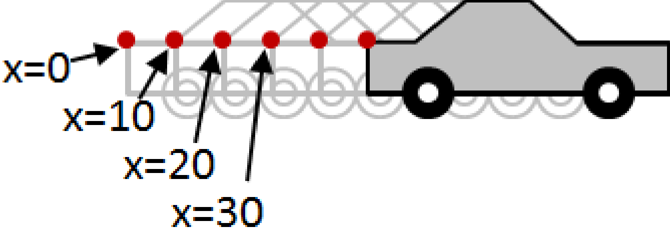
```
for (int i=0; i<=n; i++) {
   if ((i%2 == 0) && (i%3 == 0))
      println(i);
}
```

Example:

Consider writing a program that will cause a car to be displayed on the window, moving across the window to the right. The car can be drawn easily, kind of like drawing the house as we did before.



Now we need to understand what is happening as the car moves. What changes as the car moves along to the right? The horizontal position changes … which is the **x** coordinate of the car's points. So, we would need to introduce a variable, say **x**, to represent the horizontal location of the car and use it in our program. Likely the **x** refers to some fixed part of the car's image … perhaps the top/leftmost point:



In Processing the code would look like this:

```
for (int x=0; x<=width; x=x+10)
    drawCarAt(x);  // details left out
```

A few things to note. First, **width** is a pre-defined parameter in Processing that is set to the width of the window (in pixels). Similarly, there is a **height** parameter set to the height of the window (in pixels). Notice as well now that the **x** value is increased by **10** each time. The value of **10** represents the car's speed. If we use a smaller number, such as **5**, the car will appear to move slower across the screen, as it will move only half as far each time that we redraw it. A larger value, such as **20**, will double the speed.

Now what if we wanted the car to speed up? The value of **10** would have to start smaller, perhaps at **1** and then increase. So the loop increment (i.e., the *speed* in this case) would need to increase each time as well as the **x** value.

```
int speed = 0;
for (int x=0; x<=width; x=x+speed) {
   drawCarAt(x);          // details left out
   speed = speed + 2;
}
```

Notice the need for a new **speed** variable and how this variable is used to move the car. The **2** here is the rate of acceleration. For smaller numbers, the car accelerates slower, but for larger numbers it accelerates faster.

How can you adjust the code above so that the car speeds up until it gets to the center of the window and then slows down (i.e., decelerates) so that it stops at the right of the window?

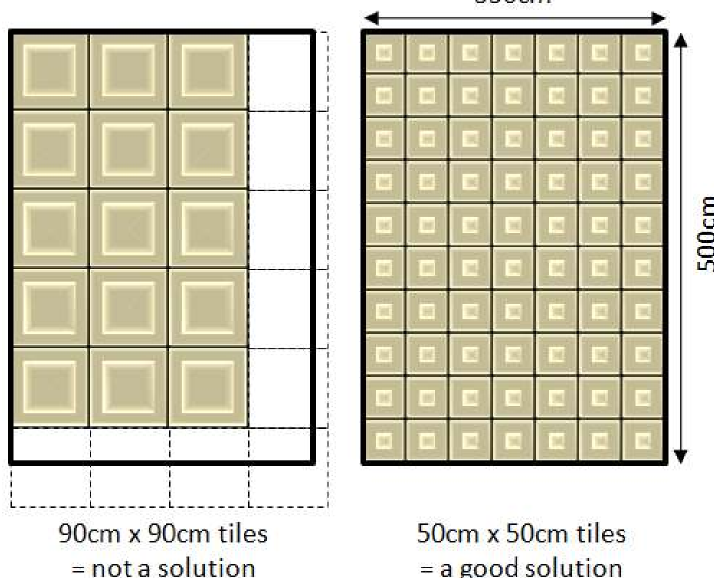
```
int speed = 0;
for (int x=0; x<=width; x=x+speed) {
   drawCarAt(x);          // details left out
   if (x < width/2)
      speed = speed + 2;
   else
      speed = speed - 2;
}
```

Example:

Assume that you have a nice rectangular room that measures $\mathbf{W}_{cm}$ **x** $\mathbf{L}_{cm}$. You want to place tiles down on the floor arranged in a grid pattern so that the entire floor is covered. However, you do not want to cut any tiles! Assuming that you can buy pre-cut square tiles of any size, what size of tiles should you buy?

How do we approach the problem? Once again, make sure that we understand the problem.

Consider the picture to the right which has a **350_{cm} x 500_{cm}** room. In order for the tiles to fit properly, we can only have whole tiles across any row and any column. That means, if we have **R** tiles across a row, then for tiles that are **T_{cm} x T_{cm}**, then **RxT** must equal exactly **350**. In other words, **350 / T** must be a whole number, not a fraction. So the **T** must divide evenly into **350**. Similarly, **T** must divide evenly into **500** if we are to fit them properly in each column as well.



Certainly we could use 1_{cm} x 1_{cm} tiles in our example above, but that would require **175000** tiles (surely you would not want to lay those down yourself) ! In fact, here are all the possible solutions for our example:

Tile Size	Tiles Required
1$_{cm}$ x **1**$_{cm}$	350 x 500 = **175000**
2$_{cm}$ x **2**$_{cm}$	175 x 250 = **43750**
5$_{cm}$ x **5**$_{cm}$	70 x 100 = **7000**
10$_{cm}$ x **10**$_{cm}$	35 x 50 = **1750**
25$_{cm}$ x **25**$_{cm}$	14x 20 = **280**
50$_{cm}$ x **50**$_{cm}$	7 x 10 = **70**

Likely, the favored solution is the one that requires the least amount of tiles … which is the **50**$_{cm}$ x **50**$_{cm}$ tile solution. The number **50** happens to be the ***greatest common divisor*** (i.e., **GCD**) … or ***greatest common factor*** (i.e., **GCF**) of the numbers **350** and **500**. In fact, the problem that we are trying to solve requires us to find the GCD of our two numbers. Can you think of a simple solution to find that number?

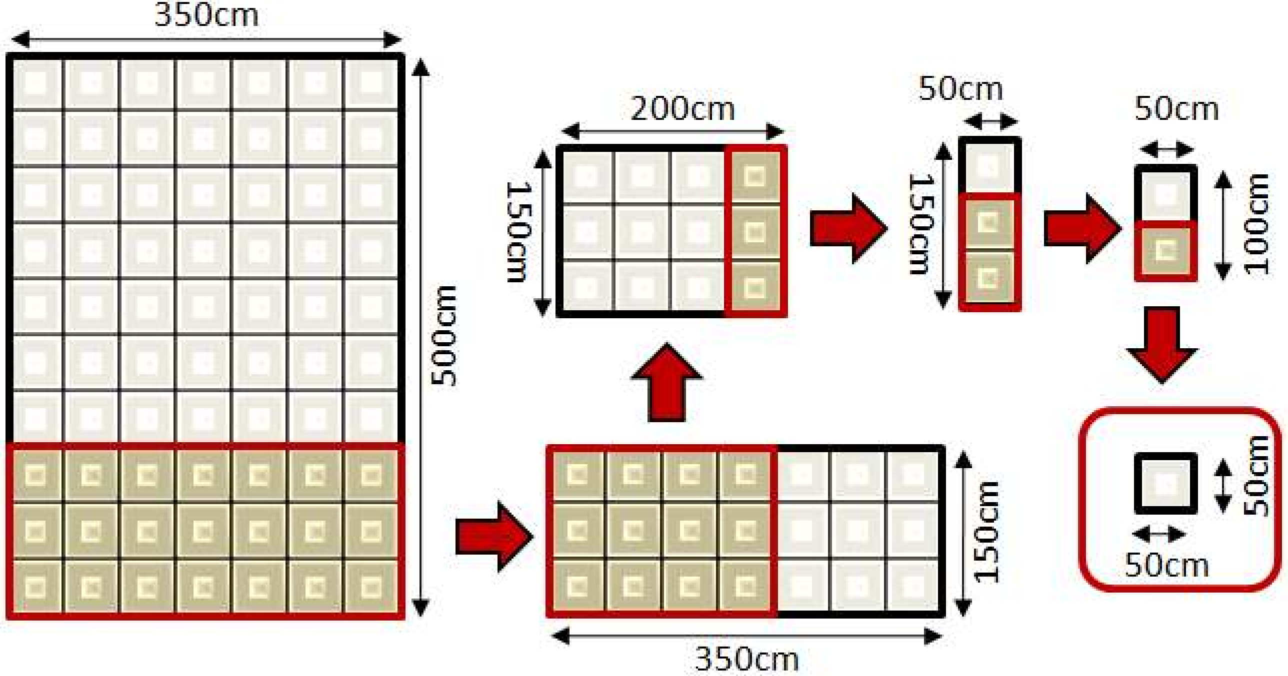
The parameters for the problem are the #'s that represent the floor dimensions:

```
int W = 350;
int L = 500;
```

To find the gcd, we can just check all possibilities. This program will start off with an attempt to see whether or not the smaller number divides evenly into the larger one. If that is true, then we have our answer and the loop quits. Otherwise, the program keeps subtracting **1** from the potential **gcd** until one is found. Ultimately, this number will keep decreasing to **1**, and that will be a common divisor to any number (although it's the ***least*** common divisor). The program assumes that neither number is zero or negative to begin with.

```
int        gcd = min(W,L);
boolean    found = false;
while (!found) {
    if ((W%gcd == 0) && (L%gcd == 0))
        found = true;
    else
        gcd = gcd - 1;
}
println(gcd + "x" + gcd + " tiles should be used");
```

The above solution will require **300** iterations of the **WHILE** loop (i.e., **gcd** decreases from 350, 349, 348, 347, … down to **50**). There are more efficient solutions. For example, since the **gcd** divides both **350** and **500**, we can see that it is still possible to find the **gcd** by ignoring a large 350_{cm} x 350_{cm} portion of the floor area and concentrating on the remaining area:

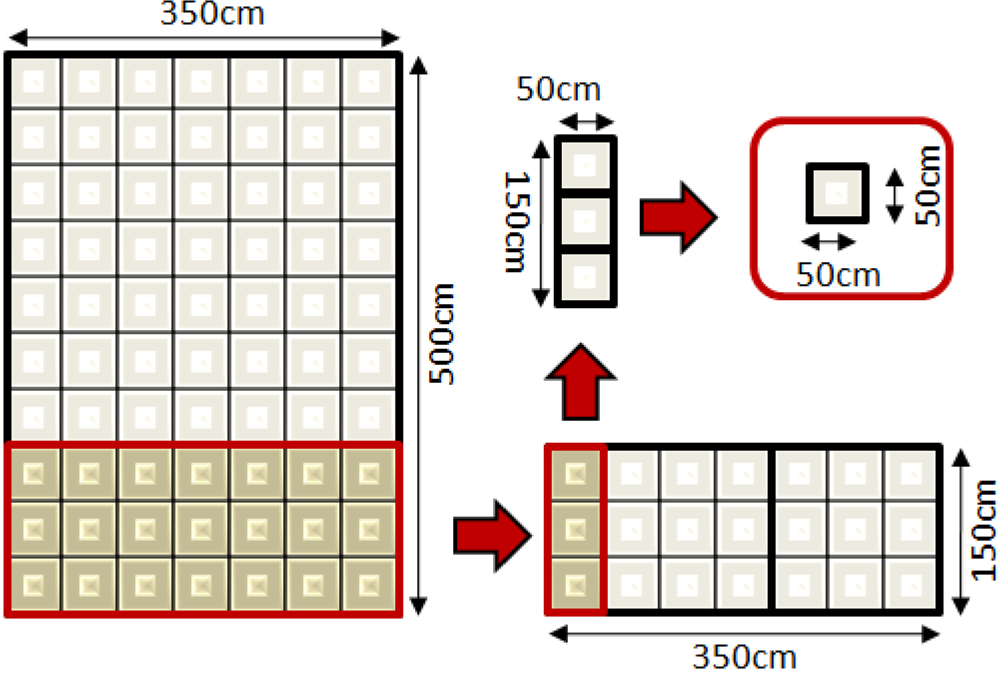


As seen in the diagram, given that we have an **n** x **m** floor area remaining, we can continually extract an **n** x **n** floor area (if **n** < **m**) or an **m** x **m** floor area (if **m** < **n**) until we end up with a remaining floor area in which **n** = **m**. In this case, **n** (or **m**, since they are equal) is the **gcd**. We can adjust our algorithm to compute the answer in this manner by repeatedly extracting the minimum of the dimensions:

```
int        a = width;
int        b = length;
while (a != b) {
    if (a > b)
        a = a - b;
    else
        b = b - a;
}
println(a + "x" + a + " tiles should be used");
```

This algorithm produces a better solution … which requires only **5** iterations of the **while** loop!

In fact, it can be improved even further (i.e., only **3** iterations of the **while** loop) by using the modulus operator which takes multiples of the lower dimension away in one step:



Can you figure out how to write this code?

2.8 Math & Trigonometry

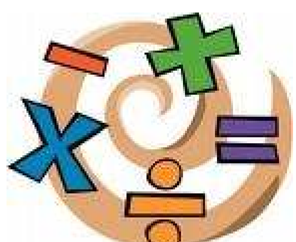
Obviously, a computer can compute solutions to mathematical expressions. We can actually perform simple math expressions such as:

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 Processing/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 Processing/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 Processing/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. Perhaps a more often usage of the modulus operator is to provide a kind of wrap-around effect when increasing or decreasing an integer.

Example:

Recall our algorithm to move a car across the window:

```
for (int x=0; x<=width; x=x+10)
     drawCarAt(x);   // details left out
```

What if we wanted the car to drive off the right edge of the window and then re-appear on the left side again? We could adjust the algorithm as follows:

```
int x = 0;
while (true) {    // repeat forever
   drawCarAt(x);  // details left out
   x = x + 10;
   if (x > width)
       x = 0;
}
```

We can eliminate the **IF** statement and reduce this code simply by using the modulus operator:

```
int x = 0;
while (true) {    // repeat forever
   drawCarAt(x);  // details left out
   x = (x + 10) % width;
}
```

Of course, the above code examples cause the car to re-appear suddenly on the left side of the window. How could we adjust it so that the car **"drives in"** from the left side instead of appearing suddenly ? See if you can figure this out.

Processing actually has many other pre-defined functions that you can use within your programs. Here are just a few of the standard mathematical ones:

- **min(a, b)** – returns the smallest of **a**, **b**, and **c**(optional)
- **max(a, b)** – returns the largest of **a**, **b**, and **c**(optional)
- **round(a)** – rounds **a** up or down to the closest integer

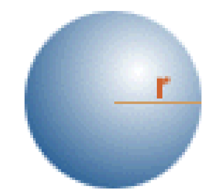
- **pow(a, b)** – returns **a** to the power of **b**
- **sqrt(a)** – returns the square root of **a**
- **abs(a)** – returns the absolute value of **a** (i.e., it discards the negative sign)

Similar functions are usually available in all programming languages, although their syntax and parameters may vary a little. For example, in JAVA, here is what these functions would be called:

- **Math.min(a, b)** – returns the smallest of **a** and **b**
- **Math.max(a, b)** – returns the largest of **a** and **b**
- **Math.round(a)** – rounds **a** up or down to the closest integer
- **Math.pow(a, b)** – returns **a** to the power of **b**
- **Math.sqrt(a)** – returns the square root of **a**
- **Math.abs(a)** – returns the absolute value of **a** (i.e., it discards the negative sign)

Example:

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 Processing 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).

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

In Processing, **PI** is defined as a constant with the value 3.14159265358979323846 (in Java, we use **Math.PI**). Here is the simplest, most straight forward solution:

```
int   r = 25;
println(4 * PI * pow(r,3) / 3.0);
```

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

```
println(4 * PI * (r*r*r) / 3.0);
```

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

```
println(4 * 3.14159265358979323846 * (r*r*r) / 3.0);
```

Or we could even pre-compute 4**π**/3 first (which is roughly 4.1887903) :

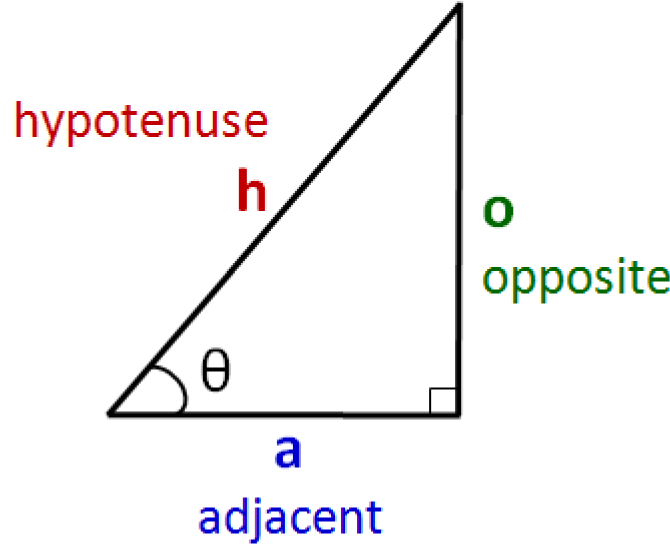
```
println(4.1887903 * pow(r,3));
```

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).

Besides these basic math functions, there are other VERY useful functions that are often needed in computer science. For example , trigonometric functions are central to computer graphics and for modeling and simulating objects that move around on the screen.

Trigonometry is all based on the angles of a right-angled triangle. Recall that a right-angled triangle has a hypotenuse ... which is the edge opposite to the right angle:

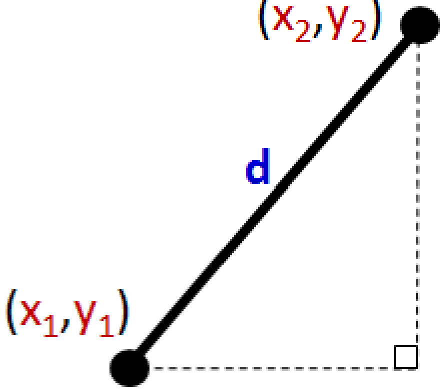


Given one of the other angles, **θ**, of the triangle (either of the ones that is not 90°), we can relate the lengths of the triangle's sides with one other as follows:

sine(θ) = **o**/**h** → "soh"
cosine(θ) = **a**/**h** → "cah"
tangent(θ) = **o**/**a** → "toa"

You may also remember the following formula for calculating the length of **h**:

$$h = \sqrt{a^2 + o^2}$$



What does all of this have to do with computer science ? Well, for one thing, geometry problems are often encountered in computer science and they often require us to determine the distance between two points as follows:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

In Processing, however, there is a function for doing this:

- **dist(x_1, y_1, x_2, y_2)** – returns the distance between points **(x_1, y_1)** and **(x_2, y_2)**

and in JAVA, here is the function:

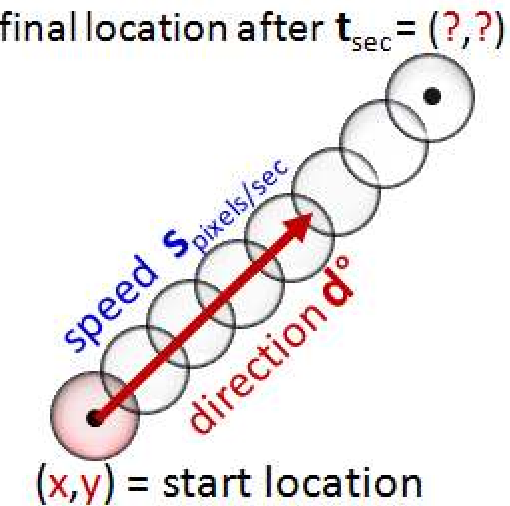
- **Point.distance(x_1, y_1, x_2, y_2)** – returns the distance between points **(x_1, y_1)** and **(x_2, y_2)**

There are countless situations in computational geometry and in computer graphics where knowing the distance between two points is important. We will see examples of this later in the course.

Getting back to the trigonometric functions, they are required in many computational problems as well as simulation and computer graphics problems.

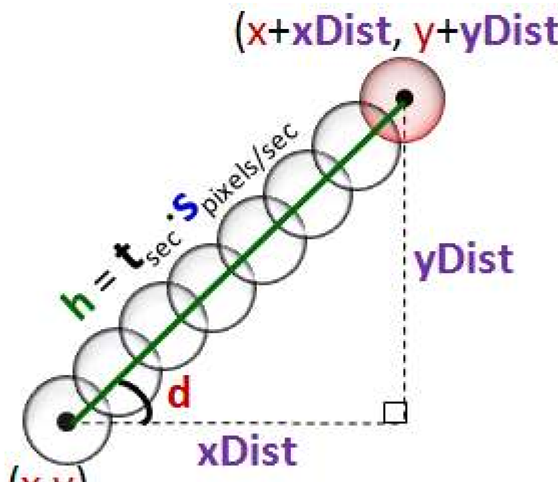
Example:

Consider programming a game in which a ball is moving along at some direction **d** (in degrees with respect to the horizontal axis) and speed **s** (in pixels per second). Given that the ball starts at position (**x**,**y**), where do we redraw the ball after **t** seconds ?



To solve this problem, we simply apply trigonometry. To begin, we need to understand the triangle formed between the start and end location.

The distance between the start and end location cannot be directly computed using the **dist()** function since we do not know the ending location. However, we do know that the distance travelled is (**speed** x **time**). The speed is in pixels per second and the time is in seconds, so the distance travelled, **h**, will have units of pixels.



To determine the final location of the ball at time **t**, we just need to determine the amount of movement in the horizontal and vertical directions (i.e., **xDist** and **yDist**). Then we can add those distances to the original (**x**, **y**) location to get the final location.

We can plug in our known trigonometric formulas:

sin(d) = **yDist**/**h** → **yDist** = **h**·**sin**(d)
cos(d) = **xDist**/**h** → **xDist** = **h**·**cos**(d)

And **voila**! We have the final location!!

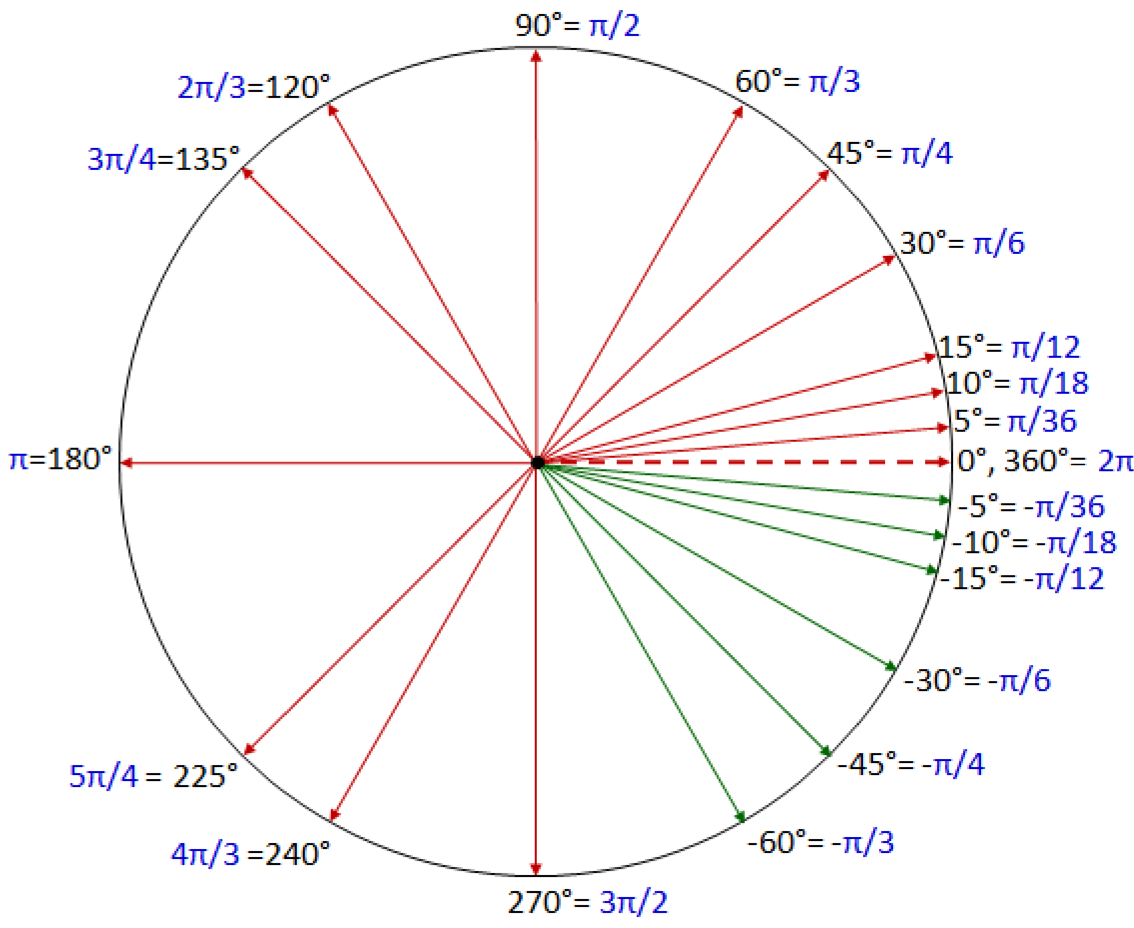
Whenever doing trigonometry, we must always understand the difference between degrees and radians. Recall that all angles are represented as either **degrees** or **radians**. However, in most programming languages, all trigonometric functions require angles to be specified in radians. Here are the functions in Processing:

- **sin(a)** – returns the sine of angle **a** (which must be in radians)
- **cos(a)** – returns the cosine of angle **a** (which must be in radians)
- **tan(a)** – returns the tangent of angle **a** (which must be in radians)

Here are the JAVA equivalent functions:

- **Math.sin(a)** – returns the sine of angle **a** (which must be in radians)
- **Math.cos(a)** – returns the cosine of angle **a** (which must be in radians)
- **Math.tan(a)** – returns the tangent of angle **a** (which must be in radians)

Do you remember what radians are ? All degrees and radian values are with respect to a horizontal line that points right … which is the 0° (and 0 radians) angle. A positive increase in angle represents a counter-clockwise spin around a circle, while negative angles represent a clockwise spin. Here is how angles and radians relate to each other:



Just a few common values are shown above. Note also that the negative values all have equivalent positive values as well. So, for example, an angle of 300° (or 5π/3 radians) is the same as the angle -60° (or -π/3 radians).

It is a good idea to understand the above diagram to get used to working with angles. Although the functions all require angles in radians, it is sometimes conceptually easier to store angles as degrees (since it is more intuitive to think in degrees). Because of this, there are conversion functions for converting back and forth between radians and degrees. Here are the conversion functions in Processing:

- **degrees(r)** – returns the degree value for radian angle **r** (computed as 360***r**/2π)
- **radians(d)** – returns the radians value for degree angle **d** (computed as 2π***d**/360)

Here are the JAVA equivalent functions:

- **Math.toDegrees(r)** – returns the degree of angle **r** (which is in radians)
- **Math.toRadians(d)** – returns the radians of angle **d** (which is in degrees)

One more important pre-defined function in most programming languages is the **random** function. In computer science it is often necessary to use random numbers in order to provide variety in our program. For example,

- If we are simulating a colony of ants roaming around on the screen, if we want the ants to seem realistic in their movements, there is a certain degree of unpredictability that must be allowed in the way they move. If, for example, the ants always moved in the same patterns, the simulation would not look realistic.

- If we are testing our program to see if it behaves with unpredictable user input, we may need to generate random data or supply data at random intervals to test the timing of our program. Some algorithms in computer science are based on data that is assumed to be in truly random order.

Obtaining a truly random number is difficult with a computer which is based on 1's and 0's stored in memory. The problem of generating truly random numbers is an open area in computer science that is still being studied. However, many languages supply functions that produce what is called ***pseudo-random*** numbers. That is, the numbers "seem" random, but are actually a fixed sequence of numbers based on some starting point (called the ***seed***) and some function that is applied to that seed successively. Some random number generators simply use the computer's current clock value (i.e., time of day in milliseconds) as a means of obtaining the seed or computing the next random number.

Anyway, there is no need for an in-depth discussion on random number generators. All that is important is that you know that there is a function in Processing that computes a pseudo-random floating point number in the range of 0 to **n**:

- **random(n)** – returns a random floating point number **x** such that $0.0 \le x < n$.

So, to get a random number from 0.0 to 99.9999999999, we would call `random(100)`. We can "adjust" the result of this function to obtain a number in any range that we want.

For example, if we wanted an integer in the range from 15 to 67, inclusively, we would do this:

```
(int)random(53)) + 15 // where 53 = 67 - 15 + 1
```

In JAVA, the random number generator is different:

- **Math.random()** – returns a random floating point number **x** such that 0.0 ≤ **x** < 1.0.

It always generates a random number from 0.0 to 0.999999999 and if we want to get it within a certain range we need to do additional multiplications:

```
100 * Math.random()             // from 0.0 to 99.99999999
(int)(53 * Math.random()) + 15  // from 15 to 67
```