
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 Problem-Solving With 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:

Two-handed Algorithm	One-handed Algorithm
<p>AlgorithmX:</p> <ol style="list-style-type: none"> 1. get off couch 2. walk to kitchen 3. go to refrigerator 4. chooseDrink() 5. drink it <p>chooseDrink():</p> <ol style="list-style-type: none"> 1. open refrigerator 2. if there is a carton of juice then { 3. pourDrink() 4. } 5. otherwise if there is a soda then { 6. take soda 7. open soda 8. } 9. close refrigerator <p>pourDrink():</p> <ol style="list-style-type: none"> 1. take the carton 2. close refrigerator 3. getGlass() 4. pour lemonade or juice into glass 5. go to refrigerator 6. open refrigerator 7. put carton in refrigerator <p>getGlass():</p> <ol style="list-style-type: none"> 1. go to the cupboard 2. open cupboard 3. take a glass 4. close cupboard 	<p>chooseDrink():</p> <ol style="list-style-type: none"> 1. open refrigerator 2. if there is a carton of juice then { 3. pourDrink() 4. } 5. otherwise if there is a soda then { 6. take soda 7. open soda 8. go to counter 9. put soda down on counter 10. go to refrigerator 11. } 12. close refrigerator 13. go to counter 14. pick up drink <p>pourDrink():</p> <ol style="list-style-type: none"> 1. take the carton 2. go to counter 3. put carton down on counter 4. go to refrigerator 5. close refrigerator 6. getGlass() 7. pick up carton 8. pour lemonade or juice into glass 9. put down carton 10. go to refrigerator 11. open refrigerator 12. go to counter 13. pick up carton 14. go to refrigerator 15. put carton in refrigerator <p>getGlass():</p> <ol style="list-style-type: none"> 1. go to the cupboard 2. open cupboard 3. take a glass 4. put glass down on counter 5. close cupboard 

Notice all the changes that are necessary now because you 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).

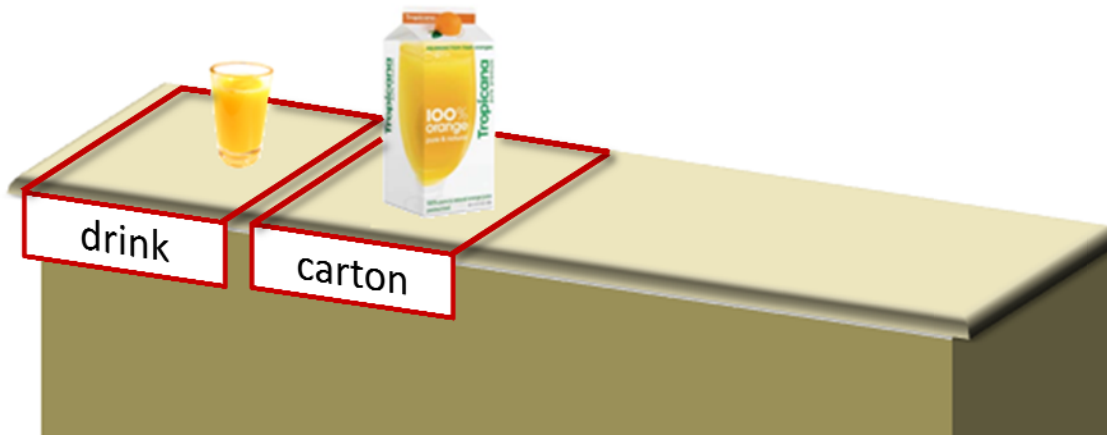
In reality, such details are usually an obvious part of the solution and need not be mentioned. However, it is important to indicate **what** information is being stored and **where** it is being stored so that we can use it later in the program. For example, what if we placed the carton of juice down 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, during steps **9** through **12**, both the **glass** and the **carton** are sitting on the counter. 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, when there is no juice, the algorithm gives back a soda can as the result. So, perhaps instead of “glass”, we could use the label “drink” instead:

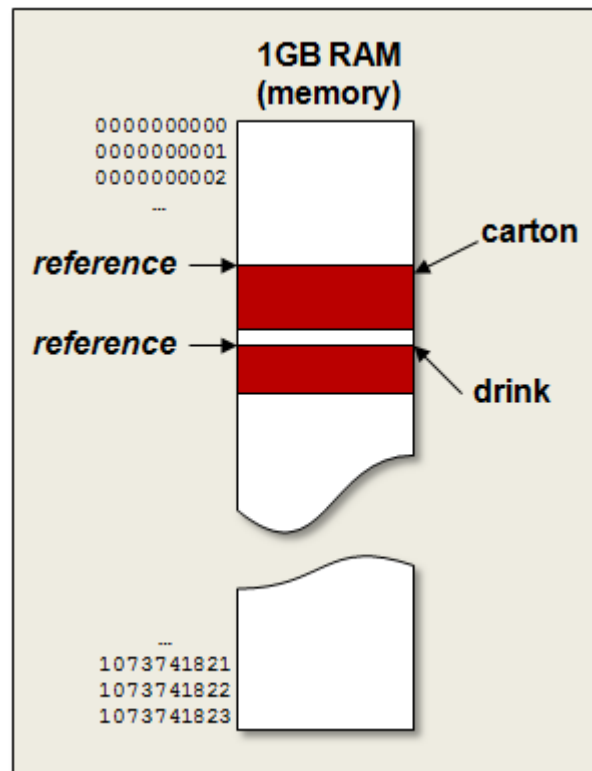


So, each variable that we make will **have its own counter space on the counter top**. When we create (or **declare**) a variable, we are really **reserving space** for an object on the counter top. And as with any reservation, we need to have a name for the reservation ... which corresponds to the label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

AlgorithmX:

1. get off couch
2. walk to kitchen
3. go to refrigerator
4. **drink** ← result of **chooseDrink()**
5. drink the **drink**

chooseDrink():

1. open refrigerator
2. **if** there is a carton of lemonade or orange juice **then** {
3. **drink** ← result of **pourDrink()**
4. }
5. **otherwise if** there is a soda **then** {
6. **drink** ← the soda
7. open **drink**
8. }
9. close refrigerator
10. **return drink**

pourDrink():

1. **carton** ← the carton
2. **drink** ← result of **getGlass()**
3. pour **carton** contents into **drink**
4. go to refrigerator
5. put **carton** in refrigerator
6. **return drink**

getGlass():

1. go to the cupboard
2. open cupboard
3. **drink** ← the glass
4. close cupboard
5. **return drink**

Notice that the **drink** variable represents either a can of soda, an empty glass or a glass with lemonade or orange juice in it, depending on the line of the program. The ← 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 **go to the counter** and **put soda down on counter** and **go to refrigerator** are all combined into one storage step as **drink** ← **soda**. That's because leaving the refrigerator and heading over to the counter was done as a means to store the soda can 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 instruction.

Similarly, the **go to counter** and **pick up drink** combination as well as the **go to counter** and **pick up carton** combination is analogous to simply getting the value of the variable in that it gets the object stored at that location on the countertop. The functions **chooseDrink()**, **pourDrink()** and **getGlass()** all bring back (i.e., **return**) a drink, whether it is a full glass, an empty glass or a soda can. The **drink** is returned (from each function) back to the function that called it.

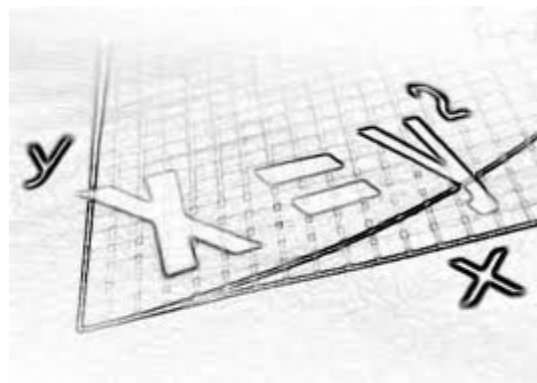
*The **return value** is the value returned as a result of the function.*

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**.

Recall from chapter 1 that a *parameter* is a piece of data that is provided to a function. 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.

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 develop a computational model and identify the incoming parameters and variables that you will need.

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 an algorithm to compute the amount of money that Bob owes Steve (or Steve owes Bob) after the trip, assuming that they decided to split the expenses evenly ?

How many algorithm parameters are there ? What are they ?

Algorithm: TripExpenses

f: food cost
h: hotel cost
g: gas cost
e: entertainment cost

1. **each** $\leftarrow (f + h + g + e) / 2$
2. **difference** \leftarrow **each** $- (g + e)$
3. **if difference** < 0 **then** Bob owes Steve the **difference**
4. **otherwise** steve owes bob the **difference**



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?

Algorithm: Barefoot

n: # of kids in total
x: # kids with socks on
y: # kids with shoes on
z: # kids with socks & shoes on



1. **socksOnly** $\leftarrow x - z$
2. **shoesOnly** $\leftarrow y - z$
3. **print** ($n - z - \text{socksOnly} - \text{shoesOnly}$)

Can you re-write the algorithm without using the **socksOnly** and **shoesOnly** variables ?

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?

Algorithm: PaintHouses

n: # people on team

1. **goal** $\leftarrow ((n * \$1000) + (4 \text{ months} * \$152 \text{ per month}))$
2. **houses** $\leftarrow (\text{goal} / \$256 \text{ per house})$
3. **print houses**



Notice that the **1000**, **4**, **152** and **256** here are all fixed/constant values, as opposed to parameters.

*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 examples:

Example:

Assume that you are given a name of a person in one of the following formats:

“Firstname Lastname”

“Lastname, Firstname”

“Firstname MiddleInitial. LastName”

“Lastname, Firstname MiddleInitial.”

You want to develop an algorithm that will determine whether or not the name is properly formatted to one of these formats. The output should be “YES” if properly formatted, otherwise “NO”. How would you write the algorithm? Assume that the **name** is the only incoming parameter in the form of a bunch of consecutive characters.

Algorithm: NameFormat

n: the name

1. **if** (n does not start with capital letter) **then** print NO
2. **if** (n has a lower case letter after a space character) **then** print NO
3. **if** (n has *weird* characters in it) **then** print NO
4. **if** (n has a comma and there is no space after it or no letter before it) **then** print NO
5. **if** (n has a middle initial that has more than one character) **then** print NO
6. **otherwise** print YES

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 not wearing speedos at a pool party). Each person votes yes or no. Develop an algorithm that determines the majority response (either yes or no).

To begin the algorithm, consider first getting all 5 votes and storing them. Then use the votes to determine the majority.



Algorithm: Majority

v1, v2, v3, v4, v5: the votes of the 5 people

```

1.  yesCount ← 0
2.  noCount ← 0
3.  if v1 = yes then yesCount ← yesCount + 1
4.  otherwise noCount ← noCount + 1
5.  if v2 = yes then yesCount ← yesCount + 1
6.  otherwise noCount ← noCount + 1
7.  if v3 = yes then yesCount ← yesCount + 1
8.  otherwise noCount ← noCount + 1
9.  if v4 = yes then yesCount ← yesCount + 1
10. otherwise noCount ← noCount + 1
11. if v5 = yes then yesCount ← yesCount + 1
12. otherwise noCount ← noCount + 1
13. if yesCount > noCount then display YES
14. otherwise display NO

```

In this example, the yes and no parameters are considered to be **boolean** values.

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

So we can say that **yes** is the same as **true** and **no** is the same as **false**.

Now that you understand how to identify algorithmic parameters and variables, as well as how to develop and use simple computational models, it is important to discuss how these variables are represented in a real computer.

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.

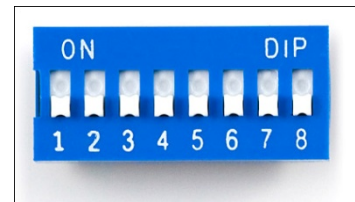
```

011010010010011101101101
001000000111001101101101
011000010111001001110100
011001010111001000100000
011101000110100001100001
011011100010000001101101
011110010010000001100010
01101111011100110110011

```

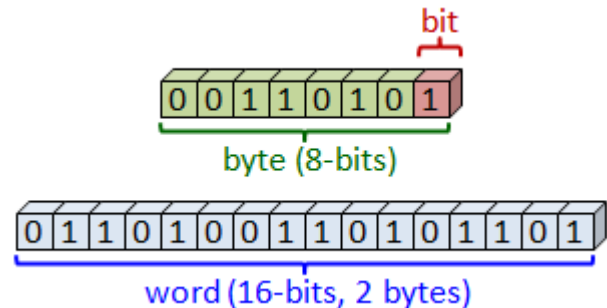


A group of 8 bits is known as a **byte**. A byte can represent 256 combinations of 1's and 0's. That is, if we think of each bit as being a switch which is either on or off, we can flip the 8 switches in 256 unique combinations. This allows a single byte to store a number from **0 to 255**.



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** (American Standard Code for Information Interchange). 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 is the ASCII table (provided as a reference only ... DO NOT try to memorize it):

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	{ } ~ □

So, for example, the letter “**A**” corresponds to number **65** in the ASCII table which is number **01000001** in binary bits (we will not discuss bit representation any further in this course). There are also versions of extended ASCII 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.

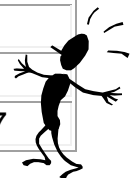
In addition, we can actually use bytes to represent real numbers (also called floating-point numbers) such as **3.14159265**. Also, by making assumptions on one particular bit in a byte (i.e., the **most significant bit**, also called the **sign bit**), we can allow the numbers to be either negative 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;
```

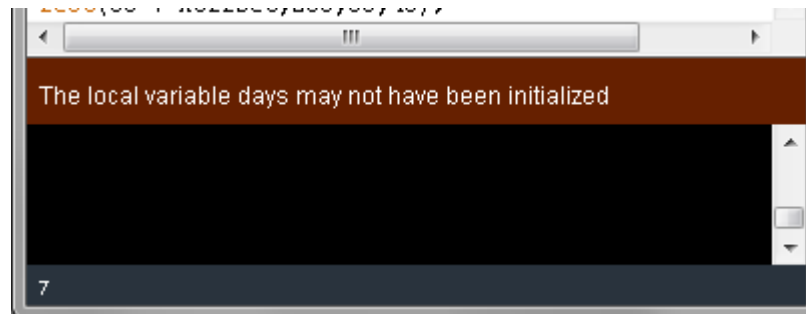
The above examples show all 8 primitive possible types that you may use in Processing/JAVA programs. Note that these will differ from language to language. Notice as well that there is a ; character after each line, as with any step of an algorithm.

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) that can hold a value of the given type.

Declaring a variable, DOES NOT assign it any value, it only reserves space for the variable. In processing, after you declare a variable, you MUST give it a value **before** you use it. 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.

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 (i.e., no spaces in the name). As standard convention, multiple word names should have every word capitalized (except the first).

Here are some good examples of variable names:

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





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.

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:

 <code>int days;</code> <code>days = 10.2789;</code>	 <code>int days;</code> <code>days = 10;</code>
 <code>boolean hungry;</code> <code>hungry = 'y';</code>	 <code>boolean hungry;</code> <code>hungry = false;</code>
 <code>char sex;</code> <code>sex = "F";</code>	 <code>char sex;</code> <code>sex = 'F';</code>

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
```

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;
```

Normally, constants use uppercase letters with underscores (i.e., **_**) separating words.

2.3 Conditional Statements

We have already seen the need to make decisions in our program based on various input and certain calculations. Recall, for example, the **TripExpenses** algorithm:

Algorithm: TripExpenses

1. **each** $\leftarrow ((f + h + g + e) / 2)$
2. **difference** $\leftarrow (each - (g + e))$
3. **if difference** < 0 **then** Bob owes Steve the **difference**
4. **otherwise** steve owes bob the **difference**

Notice here that a decision had to be made as to whether Bob owed Steve the difference or vice-versa. The **if/then/otherwise** here is known as a **conditional statement** (often simply called an **if statement**).

Example:

Often, the **otherwise** part can be left off of an **if** statement. For 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:

Algorithm: TheatreDiscount

p: person buying theatre ticket

1. **discount** $\leftarrow 0$
2. **gender** \leftarrow gender of **p**
3. **age** \leftarrow age of **p**
4. **if gender** is female and **age** > 64 or **gender** is female and **age** < 13 **then** {
 discount $\leftarrow 50$
}
5. **print discount**



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

You may notice that step 4 is a little ambiguous because of the “and”s and “or”s being used. That is, notice how the decision differs based on the placement of parentheses:

if (gender is female and age > 64 or gender is female and age < 13) then ...
if (gender is female and age > 64) or (gender is female and age < 13) then ...
if (gender is female and (age > 64 or gender is female) and age < 13) then ...
if (gender is female and (age > 64 or gender is female and age < 13)) 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.

As it turns out, “**and**”, “**or**” and “**not**” are common operators in computer science, called **logical 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.

For example, **age > 64** results in either **true** or **false** as does **age < 13**. Also, **gender is female** will also produce a **true** or **false** result. When we **and/or** these **true/false** values together, we end up with an overall **true/false** result that is used by the **if** statement to decide whether or not to evaluate the code within the body of the **if** statement.

Below is a “truth table” explaining the results of using any two boolean values, say **b₁** and **b₂**, in an **if** statement:

b₁	b₂	if (b₁ and b₂)	if (b₁ or b₂)	if (not b₁)	if (b₁)
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.

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%

Algorithm: GradeToLetter

```

grade:    the number grade of a student

1.  if (grade is between 80 and 100) then
2.      print "A"
3.  if (grade is between 70 and 79) then
4.      print "B"
5.  if (grade is between 60 and 69) then
6.      print "C"
7.  if (grade is between 50 and 59) then
8.      print "D"
9.  otherwise
10.     print "F"

```



Notice how each **if** statement checks to see whether the **grade** lies within a specific range. We can re-write this using **and** operators as follows:

Algorithm: GradeToLetter2

```

grade:    the number grade of a student

1.  if (grade >= 80 and grade <= 100) then
2.      print "A"
3.  if (grade >= 70 and grade <=79) then
4.      print "B"
5.  if (grade >= 60 and grade <= 69) then
6.      print "C"
7.  if (grade >= 50 and grade <= 59) then
8.      print "D"
9.  otherwise
10.     print "F"

```

This is more realistic when programming because very few (if any at all) programming languages have a “**between**” kind of command that allows you to check values within a certain range.

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%**. Step **1** and **2** will be evaluated. However, the algorithm will then continue with steps **3**, **5**, and **7** by checking those **if** statements ... which will all evaluate to **false** anyway.

We can avoid this inefficiency by created **nested if statements**. This means that we insert successive **if** statements into the **otherwise** part of the earlier **if** statements as follows:

Algorithm: GradeToLetter3

```

grade:    the number grade of a student

1.  if (grade >= 80 and grade <= 100) then
2.      print "A"
3.  otherwise {
4.      if (grade >= 70 and grade <=79) then
5.          print "B"
6.      otherwise {
7.          if (grade >= 60 and grade <= 69) then
8.              print "C"
9.          otherwise {
10.             if (grade >= 50 and grade <= 59) then
11.                 print "D"
12.             otherwise
13.                 print "F"
                }
            }
        }
    }

```

Notice how the successive **if** statements are shown indented, as they lie within the **otherwise** part of the previous **if** statement. What happens if **92%** is entered? Steps **1** and **2** are evaluated, but then since steps **4** through **13** are inside the **otherwise** part of step **1**'s **if** statement, they are not evaluated. This is more efficient.

Notice as well the **if** statement of step **4**. Since it is in the **otherwise** part of step **1**'s **if** statement, the **grade** must be less than or equal to **79**, since if not, the algorithm would have stopped on step **2**. So, we don't need to check whether or not **grade <= 79** in step **4**. Likewise, the upper bound of **69** and **59** need not be checked in steps **7** and **10**. Here is the better code:

Algorithm: GradeToLetter3

```

grade:    the number grade of a student

1.  if (grade >= 80 and grade <= 100) then
2.      print "A"
3.  otherwise if (grade >= 70) then
4.      print "B"
5.  otherwise if (grade >= 60) then
6.      print "C"
7.  otherwise if (grade >= 50) then
8.      print "D"
9.  otherwise
10.     print "F"

```


Notice how the **otherwise** statements are also aligned nicely one after another. Since there is only one line to evaluate in each **if** statement, we do not need the braces **{ }**. In such a scenario it is often the case that we line up the statements as shown, since it is more intuitive to read.

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 the algorithm:

Algorithm: DaysInMonth

```

month:  a month from 1 to 12

1.  if (month is 1) then
2.      print 31
3.  otherwise if (month is 2) then
4.      print 28
5.  otherwise if (month is 3) then
6.      print 31
7.  otherwise if (month is 4) then
8.      print 30
9.  etc..
...

```

However, you can see that the combined **if** statements will be 24 lines long! Since there are only 3 values for 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
otherwise if (...) then     // Apr, Jun, Sep, Nov
    print 30
otherwise                    // Feb only
    print 28

```

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

Algorithm: DaysInMonth2

month: a month from 1 to 12

1. **if** ((**month** is 1) or (**month** is 3) or (**month** is 5) or (**month** is 7) or (**month** is 8) or (**month** is 10) or (**month** is 12)) **then**
2. print 31
3. **otherwise if** (**month** is 4) or (**month** is 6) or (**month** is 9) or (**month** is 11) **then**
4. print 30
5. **otherwise** 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:

Algorithm: DaysInMonth2

month: a month from 1 to 12

1. **if** (**month** is 2) **then**
2. print 28
3. **otherwise if** (**month** is 4) or (**month** is 6) or (**month** is 9) or (**month** is 11) **then**
4. print 30
5. **otherwise** 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.

Example:

Here is a larger, more complex example. See if you can understand where you should use nested **if** statements.

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 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 ?

Here is an algorithm:

Algorithm: LoanQualificationKiosk

```
1.  print welcome message
2.  employed ← ask user if he/she is currently employed
3.  hasDegree ← ask user if he received a university degree within past 6 months
4.  age ← ask user for his/her age
5.  yearsWorked ← ask user for # years worked at full time status

6.  if (employed is true) then {
7.      if (hasDegree is true) then
8.          print "Congratulations, you qualify!"
9.      otherwise {
10.         if (age >= 30) then {
11.             if (yearsWorked >= 10) then
12.                 print "Congratulations, you qualify!"
13.             otherwise
14.                 print "Sorry, you do not qualify.  You must have
                    worked at least 10 years at full time status."
15.         }
16.         otherwise
17.             print "Sorry, you do not qualify.  You must be a
                    recent graduate or at last 30 years of age."
18.     }
19. }
```

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 in step 6 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 (from lines 2 through 5) with the **if** statements as follows:

Algorithm: LoanQualificationKiosk

```
1.  print welcome message
2.  employed ← ask user if he/she is currently employed
3.  if (employed is false) then
4.      print "Sorry, you must be currently employed to qualify."
5.  otherwise {
6.      hasDegree ← ask user if he received a univ. degree within past 6 months
7.      if (hasDegree is true) then
8.          print "Congratulations, you qualify!"
9.      otherwise {
10.         age ← ask user for his/her age
11.         if (age < 30) then
12.             print "Sorry, you do not qualify.  You must be a
                  recent graduate or at last 30 years of age."
13.         otherwise {
14.             yearsWorked ← ask user for # years worked at full time status
15.             if (yearsWorked >= 10) then
16.                 print "Congratulations, you qualify!"
17.             otherwise
18.                 print "Sorry, you do not qualify.  You must have
                           worked at least 10 years at full time status."
19.         }
20.     }
21. }
```

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 than 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.

2.4 Counting

Consider a common situation in which you want to count the total of a set of values. For example, assume that you are having a pizza lunch and you want your employees to tell you how many slices they each want. Your goal is to determine how many pizzas to buy (assume 8 slices per pizza ... and all just plain pepperoni). Write an algorithm for doing this, assuming that there are n people:

Algorithm: PizzaCount

1. start with a total of 0 slices
2. **repeat** n times {
3. ask a person for the number of slices that they want
4. add those slices to the total
- }
5. divide the total slices by 8 and print the answer



It seems straight forward. Here is a more formal version:

Algorithm: PizzaCount

1. **total** \leftarrow 0
2. **repeat** n times {
3. **number** \leftarrow ask a person for the number of slices that they want
4. **total** \leftarrow **total** + **number**
- }
5. print **total** / 8

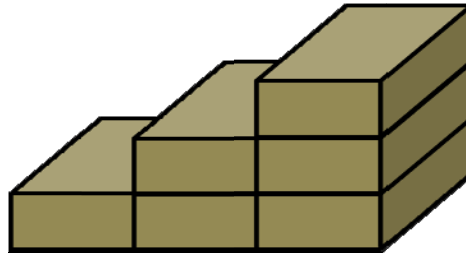
The code is logical and straight forward. In real life though, the code will usually produce a non-integer number (e.g., **4.3** pizzas). In this example we would want to increase **4.3** to the whole number **5** so that we have an integer number of pizzas. In math, the **ceiling** function is used to get the nearest integer *above* the given real value (likewise, the **floor** function gives us the nearest integer *below* the given value).

Example:

What would differ in the above code if we wanted to find the average number of pizza slices that each person will eat? Only one thing will change... can you figure it out?

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:

$$1 + 2 + 3 + 4 + \dots + n$$

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:

Algorithm: Stairs

```
1.   total ← 0
2.   currentHeight ← 1
3.   repeat n times {
4.       total ← total + currentHeight
5.       currentHeight ← currentHeight + 1
6.   }
7.   print total
```

The above algorithm demonstrates a very popular form of repetition in computer programs ... that of counting from **1** to some fixed value (i.e., n in this case). You may notice that the counter is called **currentHeight** and that it starts at 1 but each time through the loop it increases by 1. Hence, **currentHeight** 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.

The **currentHeight** variable above is an example of a **loop counter** or **loop index** because it adds 1 each time through the loop. Most loops that you will use when you are programming will involve these simple forms of counters.

Example:

Another kind of counting involves searching through some items to enumerate (i.e., count) them. Perhaps we need to count the number of items that match some kind of search criteria. For example, how would we write an algorithm that went through a list of n people and count how many people were adults ?

Algorithm: CountAdults

```
1.  set the total of the adults to 0.
2.  repeat n times {
3.      get a person and look at his/her age
4.      if the person is 18 years of age or older then {
5.          add one adult to the total
        }
    }
6.  display the total number of adults
```



Can you identify the variables being used here ? You should have identified:

- **total** - since it changes during the program.
- **age** - since it is found in step 3 and used later in step 4.

Here is the more formal version:

Algorithm: CountAdults

```
1.  total ← 0
2.  repeat n times {
3.      age ← the age of a person
4.      if age >= 18 then {
5.          total ← total + 1
        }
    }
6.  print total
```

Example:

Suppose that we wanted to print out the even numbers from **1** to **100**. How could we do this? Do you know how to check whether or not a number is even? We can check if the remainder after dividing by two is zero. The modulus operator (**%** in Processing and JAVA) gives the remainder after dividing. We just need to put this into a loop:

Algorithm: OddNumbers

```

1.   for each number n from 1 to 100 {
2.       if n modulus 2 = 0 then {
3.           print n
           }
       }

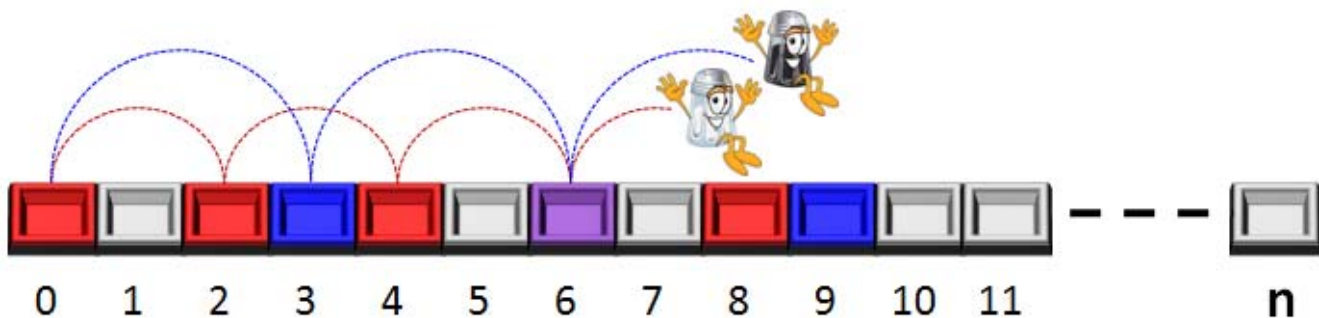
```



This kind of loop is called a **for** loop (or **for each** loop) because it repeats the loop for a set of particular values. Notice that it iterates through each location number from **1** through to **100** and that the numbers are used within the loop (i.e., in step 2). The number **n** itself is called the **loop variable** because it varies (i.e., changes) each time through the loop. In this case, the *loop variable* is the same as the *loop index* or *loop counter*, as it simply counts from **1** to **n**.

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 ?

Algorithm: HopCoverage

```
1.   spotsNotLandedOn ← 0
2.   for each locationNumber from 0 to n {
3.       if the locationNumber is not divisible by 2 and not divisible by 3 then {
4.           spotsNotLandedOn ← (spotsNotLandedOn + 1)
5.       }
6.   }
7.   print spotsNotLandedOn
```

Again, we see the **for** loop used as a way of counting now from **0** to **n**.

Example:

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:

Algorithm: HopMeetings

```
2.   for each locationNumber from 0 to n {
3.       if locationNumber is divisible by both 2 and 3 then {
4.           print locationNumber
5.       }
6.   }
```

Example:

Often, a situation arises where we have a 2-dimensional grid of locations. For example, imagine selling seats for an event taking place at a stadium. The seats are often arranged in rows and columns for each section of the stadium. Imagine that some seats are sold (red), but others are available (gray):

		columns											
		0	1	2	3	4	5	6	7	8	9	10	11
ROWS	0												
	1												
	2												
	3												
	4												
	5												
	6												
	7												

How could we write an algorithm that counted the available seats ?

Algorithm: CountSeats

```

1.  seatsAvailable ← 0
2.  for each seat {
3.      if the seat is available then {
4.          seatsAvailable ← seatsAvailable + 1
5.      }
6.  }
7.  print seatsAvailable

```

Its not too difficult to come up with this algorithm and it is logical. Now although the algorithm above is correct, step 2 is often not precise enough when it comes to programming. There would need to be a systematic way of getting each seat. Somehow, we need to indicate what order to get the seats in. Just think of how you would count the seats. You would likely count the available seats in each row and then go to the next row...and the next one... and so on until you completed all rows.

So, to do this, we would choose one row (e.g., row 0) and then loop through the seats in that row... which is the same as going through the columns of the grid. This loop would then need to be inside a bigger loop that made sure that we systematically went through each row. Here is what we would do:

Algorithm: CountSeatsSystematically

```

1.   seatsAvailable ← 0
2.   for each row {
3.       for each column {
4.           if the seat at this row and column is available then {
5.               seatsAvailable ← seatsAvailable + 1
6.           }
7.       }
8.   }
9.   print seatsAvailable

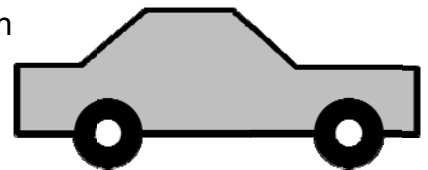
```

Notice that the inner loop goes through the seats in a particular row and that this is repeated for each row. Whenever we include one loop inside another like this, it is known as **nested looping**, or simply **nested loops**.

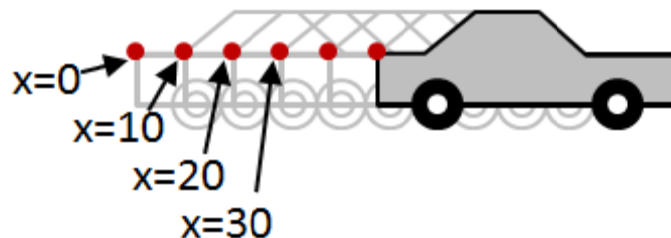
Nested loops are quite common in programming. They often appear in situations involving data arranged in a grid pattern such as applications that manipulate data tables, pictures or images, and graphics involving x/y coordinate systems.

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, kinda 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:



Algorithm: DrawCar

```
1.   for successive x locations from 0 to windowHeight {
2.       draw the car at position x
3.       x ← x + 10
    }
```

windowWidth here is a fixed constant representing the width of the window. Notice how **x** starts at **0** and then continues to increase by some constant value (in this case 10). The value of **10** will represent the 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.

A more “proper” way to write the **for** loop in our algorithm is to specify this fixed increase amount each time (called the **loop increment**). We can re-write this as follows:

Algorithm: DrawCar2

```
1.   for x locations from 0 to windowHeight by 10 {
2.       draw the car at position x
    }
```

The “**by 10**” here is a way of saying that **x** will be increased by **10** each time the loop restarts.

Example:

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.

Algorithm: AccelerateCar

```
1.   speed ← 0
2.   for x locations from 0 to windowHeight by speed {
3.       draw the car at position x
4.       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.

Example:

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 ?

Algorithm: DecelerateCar

```
1.   speed ← 0
2.   for x locations from 0 to windowWidth by speed {
3.       draw the car at position x
4.       if x < (windowWidth /2) then
5.           speed ← speed + 2
6.       otherwise
7.           speed ← speed - 2
      }
```

2.5 Conditional Iteration

In the above examples, we used variables and loops to show how we can count within an algorithm. In each case there was a fixed number of items to iterate through. However, the situation often arises in 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. 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. For example, scanning continues until there are no more items. We will now consider a few more examples that show the need for **conditional looping** (i.e., looping that requires some kind of stopping condition).

Example:

Consider the slight variation of the problem in which we want to find the average of a set of grades... but we don't know in advance how many there will be (i.e., **n** is no longer a parameter). How would we write the algorithm ?

Algorithm: Average

```

1.   sum ← 0
2.   repeat until no more grades available {
3.       x ← the next grade
4.       sum ← sum + x
5.   }
6.   compute the average to be sum / (total number of grades).
7.   print the average.

```



The condition for stopping the loop is when no more grades are available. Depending on where the grades came from, this condition would need to be specified more clearly (e.g., if the user has stopped entering them manually, or if we reached the end of a file, or if some time limit has been reached, etc..).

Both **sum** and **x** are variables here, but there is another “hidden” variable. We need to have the total number of grades in order to compute the average. Where *is* this total? We need to keep count of the number of grades that have been entered. So, we’ll need this variable which starts at 0 and then increases each time we get a new grade.

Here is the more formal version:

Algorithm: Average1

```

1.   sum ← 0
2.   count ← 0
3.   repeat until no more grades are available {
4.       x ← the next grade
5.       sum ← sum + x
6.       count ← count + 1
7.   }
8.   average ← sum / count
9.   print average

```

This kind of repeat loop is called a **repeat-until** loop because it repeats a loop until a certain condition occurs.

Often, it is re-written with the “*until*” part at the bottom as follows:

Algorithm: Average2

```

1.   sum ← 0
2.   count ← 0
3.   repeat {
4.       x ← the next grade
5.       sum ← sum + x
6.       count ← count + 1
7.   } until no more grades are available
8.   average ← sum / count
9.   print average

```

The “until part” is called the **loop condition**, since it is the part of the algorithm that decides whether or not to continue looping or to stop. Sometimes the word **while** is used to describe the condition for looping as follows:

Algorithm: Average3

```

1.   sum ← 0
2.   count ← 0
3.   while (more grades are available) {
4.       x ← the next grade
5.       sum ← sum + x
6.       count ← count + 1
7.   }
8.   average ← sum / count
9.   print average

```

This kind of repeat loop is called a **while** loop because it repeats the loop as long as (or while) a certain condition still occurs. The *while* loop is more popular in programming than the *repeat-until* loop, although they do the same thing. Notice that the condition is the opposite from the repeat-until. That is, the condition of the “while loop” indicates when to *continue* the loop, whereas the condition of the “repeat-until” loop indicates when to *stop* the loop.

We will make use of the **while** loop construct more often now in our examples, since this is used in JAVA and Processing, whereas the **repeat-until** looping construct is not.

As it turns out, every **repeat** or **for** loop can be expressed in terms of a **while** loop.

For example, recall the **AccelerateCar** algorithm:

Algorithm: AccelerateCar

```

1.   speed ← 0
2.   for x locations from 0 to windowHeight by speed {
3.       draw the car at position x
4.       speed ← speed + 2
   }

```

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

Algorithm: AccelerateCarWhileLoop

```

1.   speed ← 0
2.   x ← 0
3.   while (x <= windowHeight) {
4.       draw the car at position x
5.       speed ← speed + 2
6.       x ← x + speed
   }

```

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.

The **while** loop of the **Average3** algorithm, for example, could get its grades one-by-one from the user, who may type-in a special number to stop the process (e.g., -1). That would be an unexpected event to trigger the stopping of the loop. Therefore the while loop would be a good choice. If however, we knew that there were exactly 75 grades **available**, a **for** loop may be used more naturally, asking for grades from 1 to 75 and then stopping automatically.

Here is another example requiring a **while** loop ...

Example:

Consider this example that simulates a cashier scanning items at a checkout line in a store. A **while** loop here would fit naturally since there is no way the cashier could know in advance how many items will be scanned ... it could be 1, 10, 50, etc... In real life, the cashier would likely press a particular button (e.g., **total**, or **done**) once all items are scanned.

Try to identify the variables that would be needed by looking for values that will “vary” during the program or values that are needed in multiple parts of the program

Algorithm: CashierSales

```

1.   while (there are more items) {
2.       scan the next item
      }
3.   add the tax to the total
4.   get the payment from the user
5.   compute the change as payment – total - tax
6.   give the change to the customer

```



You should have identified:

- **total cost** - since it changes during the program.
- **tax** - since it is computed in step 3 and used later in step 5.
- **payment** - since it is received in step 4 and used later in step 5.
- **change** - since it is computed in step 5 and used later in step 6.

Likely, however, you may have missed a “hidden” variable. Notice that step 2 is kind of vague. Within that step, we need to add the **price** of the item to the **total cost**. Likely it would make sense to first get the **price** and afterwards add it to the **total** in a second sub step. Here is the more formal code with the variables:

Algorithm: CashierSales

```

1.   total ← 0
2.   while (there are more items) {
3.       price ← result of next scanned item
4.       total ← total + price
      }
5.   tax ← total * 0.13
6.   payment ← payment from customer
7.   change ← payment – total – tax
8.   give back change to customer

```

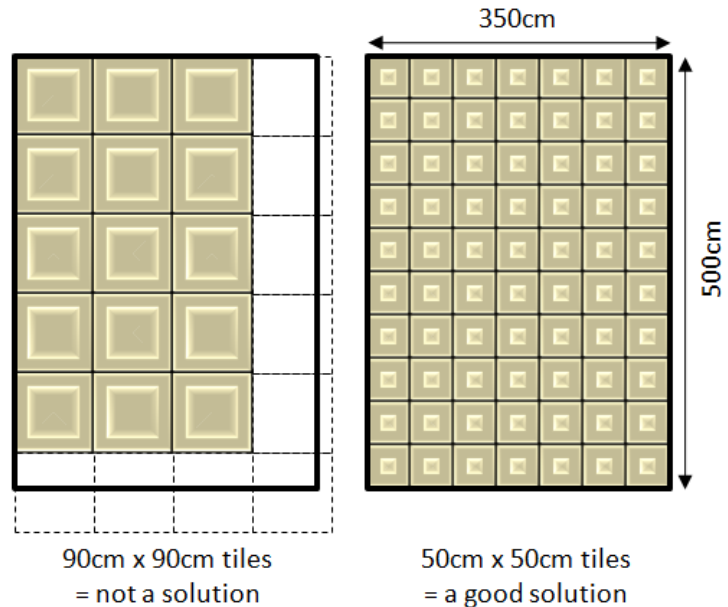
The **0.13** here is a constant. While it is true that the tax value may change after a few years, (and so it is not a permanent value forever) the value does not change while the program is running and is therefore considered constant.

Example:

Assume that you have a nice rectangular room that measures $W_{cm} \times H_{cm}$. 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 $350_{cm} \times 500_{cm}$ 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 $T_{cm} \times T_{cm}$, then $R \times T$ 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 $1_{cm} \times 1_{cm}$ 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
$1_{cm} \times 1_{cm}$	$350 \times 500 = 175000$
$2_{cm} \times 2_{cm}$	$175 \times 250 = 43750$
$5_{cm} \times 5_{cm}$	$70 \times 100 = 7000$
$10_{cm} \times 10_{cm}$	$35 \times 50 = 1750$
$25_{cm} \times 25_{cm}$	$14 \times 20 = 280$
$50_{cm} \times 50_{cm}$	$7 \times 10 = 70$

Likely, the favored solution is the one that requires the least amount of tiles ... which is the $50_{cm} \times 50_{cm}$ 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?

Algorithm: SimpleGCD

```

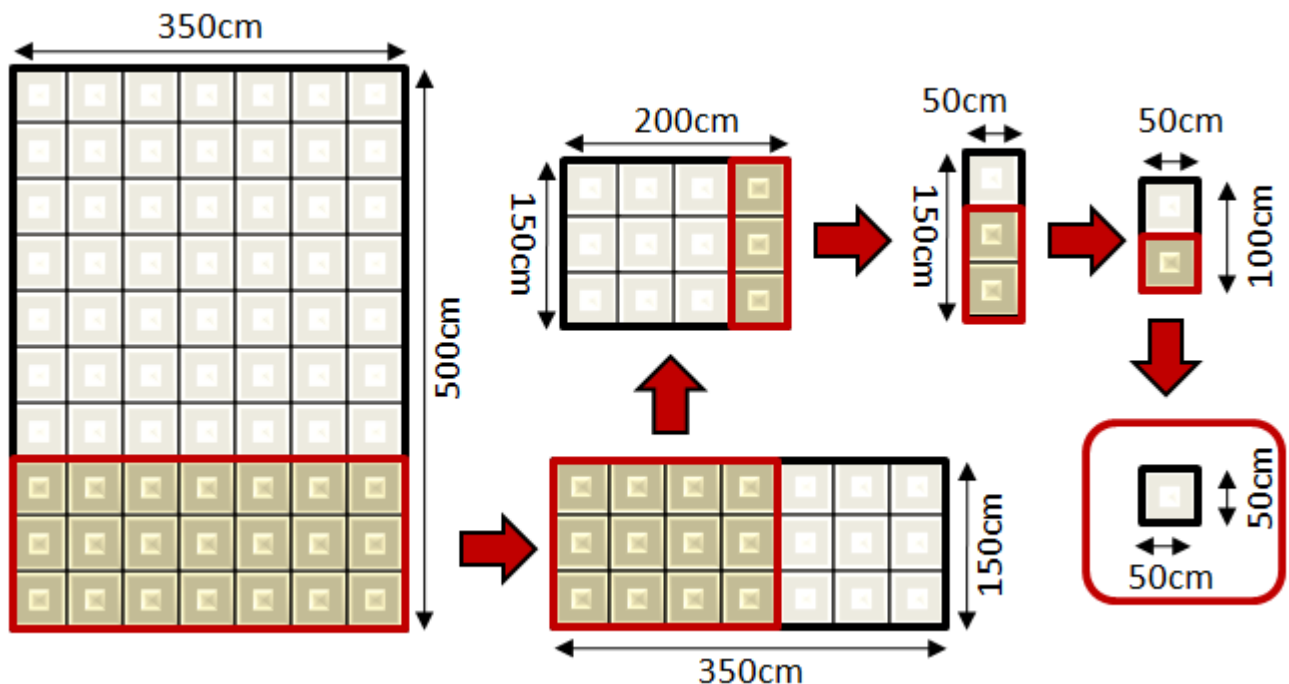
n1, n2:           numbers to which we need to find the GCD

1.  gcd ← minimum of n1 and n2
2.  found ← false
3.  while (not found) {
4.      if (gcd divides evenly into n1) AND (gcd divides evenly into n2) then
5.          found ← true
6.      otherwise
7.          gcd ← gcd - 1
8.  }
print(gcd)

```

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 while 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.

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 $n \times m$ floor area remaining, we can continually extract an $n \times n$ floor area (if $n < m$) or an $m \times m$ 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:

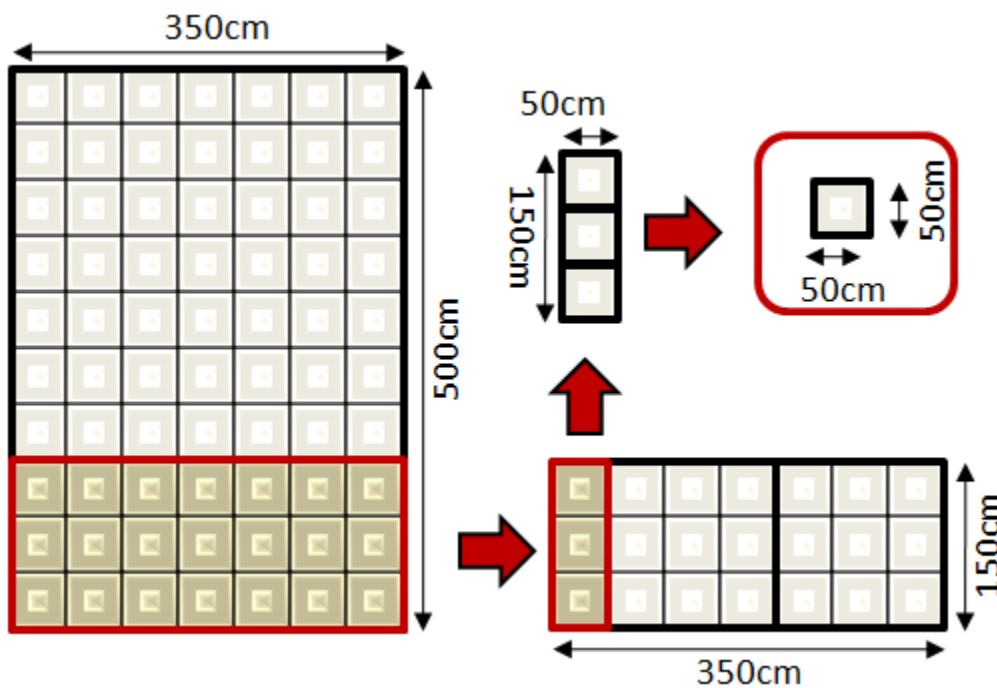
Algorithm: BetterGCD

n1, n2: numbers to which we need to find the GCD

1. **a** ← n1
2. **b** ← n2
3. **while** (a is not equal to b) {
4. **if** a > b **then**
5. **a** ← a - b
6. **otherwise**
7. **b** ← b - a
8. **}**
9. **print(a)**

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:



We'll leave the details for you to figure out as a practice exercise.

2.6 Control Structures in Processing

As we have seen already through various examples, the need to repeat steps of an algorithm often arises as well as the need to make a decision using **if/otherwise** statements. These are called **control structures** in that they specify the flow of control through the program.

The control structures that we have been using in our pseudocode is quite similar to that which is used in Processing (and JAVA). Here, for example is some code that will run in Processing:


```
int grade = 0;
if (grade >= 50) {
    print("Congratulations! ");
    print(grade);
    println(" is a passing grade.");
}
else {
    print(grade);
    println(" is quite low. Oh well, there's always next term.");
}
```

Notice how the **if** statements are not capitalized and that we do not use the word **then**. Also, notice that in place of **otherwise**, we use the word **else**.

The braces **{ }** specify the code that is inside the **if** or **else** part of the conditional statement. If there is only one line within the **if** or **else** 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:

```
int grade = 0;
if (grade >= 50)
    print("Congratulations! ");
    print(grade);
    println(" is a passing grade.");
else
    print(grade);
    println(" is quite low. Oh well, there's always next term.");
```



The code above will not compile. Since the brackets are missing, the code is assumed to have only one line in the **if** body as follows:

```
int grade = 0;
if (grade >= 50)
    print("Congratulations! ");
print(grade);
println(" is a passing grade.");
else
    print(grade);
println(" is quite low. Oh well, there's always next term.");
```




It then sees the **else** as being out of place. In processing, the **else**, and anything after it will then be ignored completely. So the result would be:

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

In JAVA, the situation is different. You would get a compile error saying: **'else' without 'if'** and the code would not run at all.

An even worse scenario is when Processing/JAVA does not notice the error at all. Consider the following:


```
int grade = 0;
if (grade >= 50)
    print("Congratulations! ");
    print(grade);
    println(" is a passing grade.");
println("All Done.");
```



In the above code, here is the output:

```
0 is a passing grade.
All Done.
```

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

```
int grade = 0;
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 tells Processing/JAVA that there is no body for the `if` statement. Thus, the `println(...)` line is outside the `if` statement altogether and is therefore always evaluated.

Recall the GradeToLetter3 algorithm:

Algorithm: GradeToLetter3

grade: the number grade of a student

1. **if** (`grade` \geq 80 and `grade` \leq 100) **then**
2. print "A"
3. **otherwise if** (`grade` \geq 70) **then**
4. print "B"
5. **otherwise if** (`grade` \geq 60) **then**
6. print "C"
7. **otherwise if** (`grade` \geq 50) **then**
8. print "D"
9. **otherwise**
10. print "F"

Here is the Processing/JAVA equivalent:

```
int grade = ___; // ignore for now where we got this grade from

if ((grade >= 80) && (grade <=100)) print("A");
else if (grade >= 70) print("B");
else if (grade >= 60) print("C");
else if (grade >= 50) print("D");
else print("F");
```

Of course, we could space this out on a few more lines, but the result would be the same.

Notice the use of `&&`. This is called a **Boolean operator** in Processing/JAVA. Here are three useful **boolean** operators:

- `&&` ... the same as saying **and**
- `||` ... the same as saying **or**
- `!` ... the same as saying **not**

You will notice as well that the code used the `>=` operator. This is called a **logical operator** that determines whether one number is greater than or equal to another. It takes the two 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 (i.e., give) a value to a variable.

Recall as well the **DaysInMonth2** algorithm:

Algorithm: DaysInMonth2

month: a month from 1 to 12

1. **if (month is 2) then**
2. **print 28**
3. **otherwise if (month is 4) or (month is 6) or (month is 9) or (month is 11) then**
4. **print 30**
5. **otherwise print 31**

This could be implemented in Processing/JAVA as follows:

```
int month = ___; // ignore for now where we got this month from

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

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 **val₁**, **val₂**, ..., **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, it **cannot** be an object (more on this later)).
2. It then checks the values **val₁**, **val₂**, ..., **val_N** 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 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 ...

```
int month = ___; // ignore for now where we got this month from

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.

One thing that needs mentioning is that 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);  
}
```



Getting back to the **IF** statement, although they 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);
```

Now what about the **repeat**, **for** and **while** loops ? They too are control structures. Recall the stair-counting algorithm:

Algorithm: Stairs

```

1.   total ← 0
2.   currentHeight ← 1
3.   repeat n times {
4.       total ← total + currentHeight
5.       currentHeight ← currentHeight + 1
6.   }
7.   print total

```

What would this look like in Processing ? Here is the solution:

```

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

```

Let us discuss this code a bit. Notice that the **total** starts off at 0 and then the **currentHeight** is added to it, just as in our pseudocode, and finally the **total** is printed.

Notice that the **for loop** has what looks like parameters and code within the parentheses () and that it has braces { } just as a function or procedure does. 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.

Lets break down the code within the parentheses. You may notice that there are two semi-colons that break things into two portions as follows:

(*initializer* ; *loopTest* ; *countExpression*)



Each of these portions 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.

So in our example, the **loopTest** checks to see if the **currentHeight <= n**. If it evaluates to **true**, then it keeps looping, otherwise it no longer repeats the loop code. The **loopTest** is checked before the loop starts. If it is false right away, then the entire **for** loop is ignored and never evaluated. Otherwise, the loop is evaluated at least once. At the end each loop iteration (i.e., after the loop body has been evaluated), the **countExpression** is evaluated and then the **loopTest** is performed again in order to decide whether or not to continue another round at the top of the loop.

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

```
currentHeight = currentHeight + 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.

Often, in order to keep code simple to read, a programmer will use the letter **i** to represent the loop variable (**i** being short for “*index*”). Here is how our code will look with **i** instead:

```
for (int i=1; i<=n; i++) {  
    total = total + i;  
}
```

The code looks much simpler. In addition, if the body of a **for** loop has only one line of code in it, then the brace **{ }** characters are not needed:

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

Recall as well that we sometimes want to count by more than 1 each time.

For example, the code to accelerate a car increased the **x** value by a speed component as follows:

Algorithm: AccelerateCar

```

1.   speed ← 0
2.   for x locations from 0 to windowWidth by speed {
3.       draw the car at position x
4.       speed ← speed + 2
    }

```

Here is what the Processing code would look like:

```

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

```

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 **speed** instead of by 1 each time.

What would happen if the stopping-expression of a **for** loop evaluated to **false** right away as follows:

```

for (int x=0; x>=width; x++) {
    ...
}

```

In this case, **x** starts at **0** and the stopping condition determines that it is not greater than or equal to **width**. Thus, the loop body never gets evaluated. That is, the **for** loop does nothing ... your program ignores it.

A similar unintentional situation may occur if you accidentally place a semi-colon **;** after the round brackets by mistake:

```

for (int x=0; x>=width; x++) ; {
    ...
}

```



In this situation, JAVA assumes that the **for** loop ends at that semi-colon **;** and that it has no body to be evaluated. In this case, the body of the loop is considered to be regular code outside of the **for** loop and it is evaluated once.

Hence the above code is understood as:

```
for (int x=0; x>=width; x++) {
}
...
```

Alternatively, in Processing, we could have used **while** loops in our solutions as follows:

```
int total = 0;
int currentHeight = 1;

while (currentHeight <= n) {
    total = total + currentHeight;
    currentHeight++;
}
println(total);
```

```
int speed = 0;
int x=0;
while (x <= width) {
    drawCarAt(x);
    speed = speed + 2;
    x = x + speed;
}
```

You should notice that the **currentHeight** and **x** variables are still needed, but is now defined outside the loop. However, it is often the case that a **while** loop waits for some event, as opposed to some counter reaching a known limit. We will do more examples later.

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 (n < 100) ; {
    println(n++);
}
```



```
// 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 (n >= 0)
    println(n--);
```

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**.

2.7 Procedures & Functions in Processing

You may recall, from our discussion of abstraction, that it is often a good idea to simplify our overall algorithm by making it higher-level. That means, we could hide details that are unnecessary. For example, if we wanted to create a home scenery, it may make sense to develop a high-level algorithm like this:

Algorithm: DrawScenery

1. draw the house
2. draw the laneway
3. draw the car
4. draw the lawn
5. draw the trees

This is an easily understood algorithm which hides all the unnecessary details of “how” to draw the various things. Recall our program for drawing a simple house:

```
size(300,300);

// Draw the house
rect(100,200,100,100);
triangle(100,200,150,150,200,200);
rect(135,260,30,40);
point(155,280);
```

How could we abstract out and make this a higher-level algorithm ? We could create a **drawHouse()** procedure that will draw the house. Then our program would look simpler like this:

```
size(300,300);
drawHouse();
```

What would the **drawHouse()** procedure look like ? Well in Processing and JAVA, this is the format for declaring a simple procedure:

```
void procedureName () {  
    // Write your procedure's code here  
}
```

Notice that procedure has **void** at the front. This indicates that there is no value to be returned from the procedure.

The brace characters (i.e., { }) indicate the code's body:

*The **body** of a function or procedure is the code that is evaluated each time that the function or procedure is called.*

So then, our **drawHouse()** procedure would look as follows:

```
size(300,300);  
drawHouse();  
  
void drawHouse () {  
    rect(100,200,100,100);  
    triangle(100,200,150,150,200,200);  
    rect(135,260,30,40);  
    point(155,280);  
}
```

The code itself looks more complicated because nothing looks hidden at all! Actually, the above code, by itself, will not compile in Processing. In general, when writing a program in Processing, all of our program code must lie within a function or procedure of some kind. There are exceptions to this:

- we may declare “global variables” outside a function at the top of our program
- when we are not creating any functions or procedures of our own, Processing will allow us to write a simple sequence of code that does not need to be inside a function or procedure.

So, Processing has provided a useful procedure called **setup()** into which we can place our code. The **setup()** procedure is called one time automatically by Processing whenever we start or restart the program. Here is code that will now compile and run in Processing:

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

```

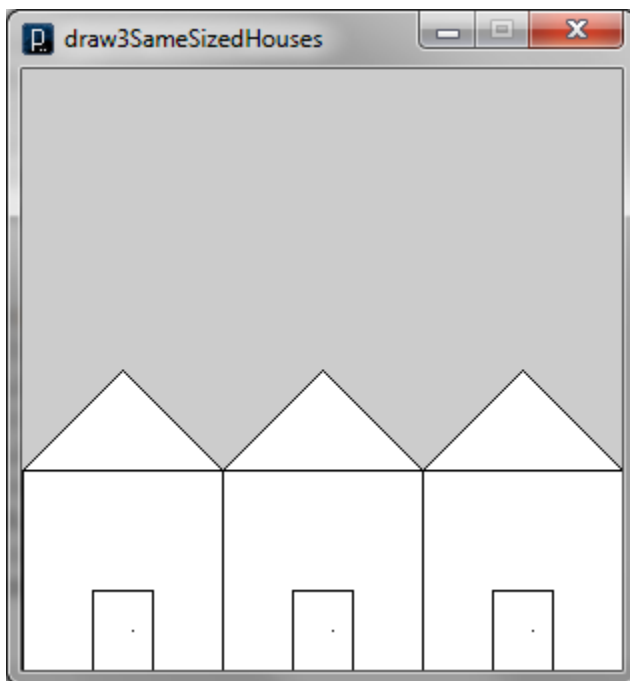
void drawHouse () {
    rect(100,200,100,100);
    triangle(100,200,150,150,200,200);
    rect(135,260,30,40);
    point(155,280);
}

```

Notice that the code is simply made up of two procedures. Again, there seems to be more code, but notice how the main algorithm (i.e., the code in the **setup()** procedure) is much simpler now.

As a side point, in JAVA, C, C#, C++ and similar languages, there is a procedure similar to **setup()** called **main()** ... which is called automatically upon program startup. You will see more of this in the follow-up course.

Abstraction, is just one reason for creating a procedure or function. However, efficiency (in the amount of code that is to be written) is another. 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 is with respect to the values shown underlined in red. 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 the value is always the **x** coordinate of the shape being drawn ... the 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).*

Notice how we can re-write the code with an x-offset:

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

    size(300,300);

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

    // 2nd house
    xOffset = 100;
    rect((0+xOffset),200,100,100);
    triangle((0+xOffset),200,(50+xOffset),150,(100+xOffset),200);
    rect((35+xOffset),260,30,40);
    point((55+xOffset),280);

    // 3rd house
    xOffset = 200;
    rect((0+xOffset),200,100,100);
    triangle((0+xOffset),200,(50+xOffset),150,(100+xOffset),200);
    rect((35+xOffset),260,30,40);
    point((55+xOffset),280);
}
```

It seems that we have written **more** code now, so how could this be more efficient? As written, it is NOT more efficient. However, you should notice that the portion of the code that actually draws each house is exactly the same!

That means, we could create a procedure for drawing the house and simply call it three times 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);
}

```

What would the **drawHouse()** procedure look like now? Well, you may notice that we need to pass in a parameter to the procedure in order to indicate the x offset. Here is the format for passing in parameters to a function:

```

void procedureName (t1 n1, t2 n2, ..., tk nk) {
  // 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, **t₁, t₂, ..., t_k** are the *types* of the parameters to the function while **n₁, n₂, ..., n_k** are the *names* of the parameters. While inside the function, each parameter is available for us to use as ... just as we would use any other variable.

So then, our **drawHouse()** procedure would now look like this:

```

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

```

As you can see, the **xOffset** parameter looks just like a variable declaration, but inside the brackets now. It tells the Processing/JAVA compiler to reserve space for this incoming integer value which can be used within the procedure.

Where does **xOffset** get its value? When we call the procedure. Each time we call it with a different offset.

Here is the simplified code:

```

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

  // draw 3 houses with different x offsets
  drawHouse(0);
  drawHouse(100);
  drawHouse(200);
}

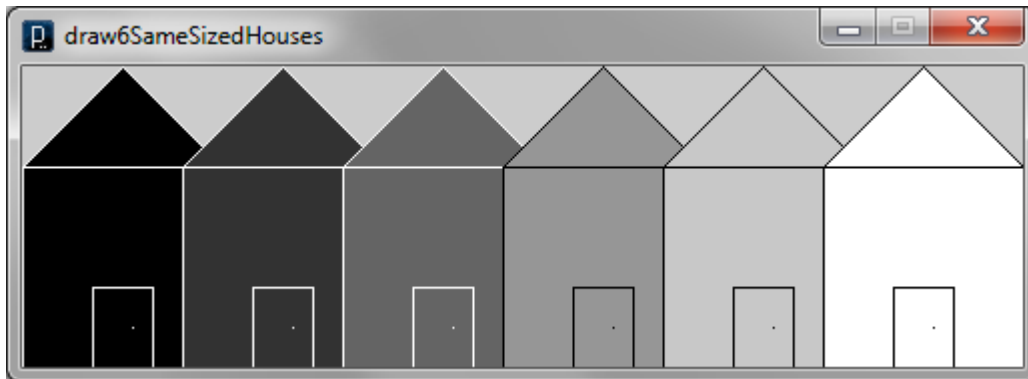
void drawHouse(int xOffset) {
  rect((0+xOffset),200,100,100);
  triangle((0+xOffset),200,(50+xOffset),150,(100+xOffset),200);
  rect((35+xOffset),260,30,40);
  point((55+xOffset),280);
}

```

Notice that we no longer need the **xOffset** variable that was declared outside the procedure because we can simply pass in the value for the offset each time we call the procedure.

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**
- 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);
}

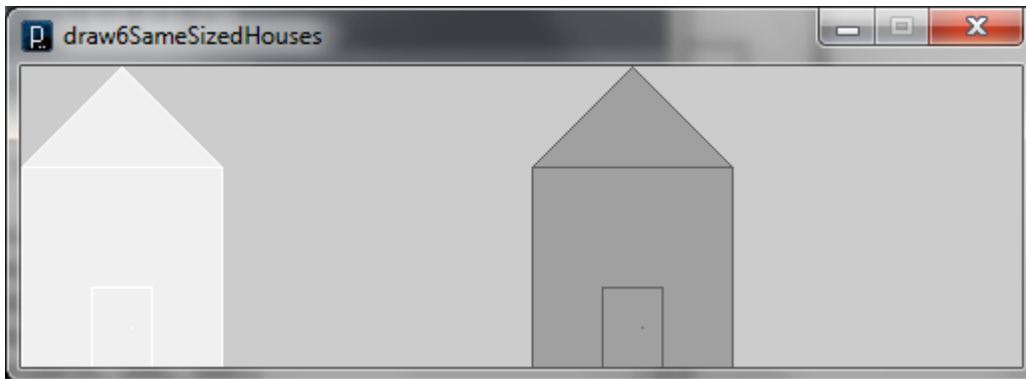
```

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 if 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:

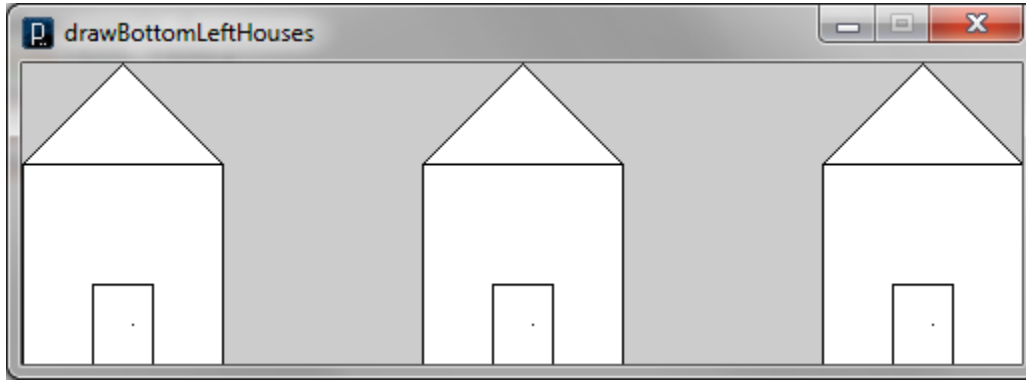
The method drawHouse(int, int, int) ... is not applicable for the arguments (int, int)

A similar error would also occur if you passed in too many parameters to the procedure.

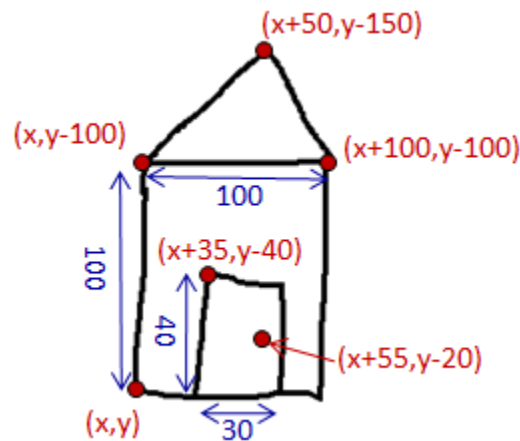
Example:

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);  
}
```



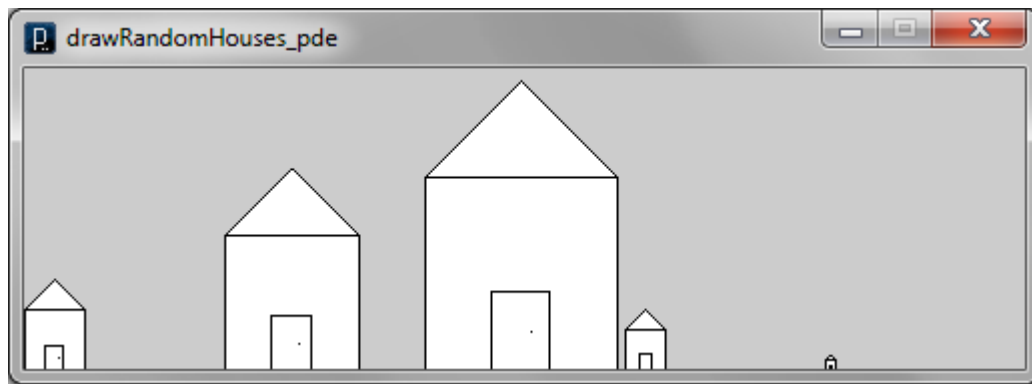
The code is not too difficult, but we must understand how the scale works. The **setup()** method has already adjusted for the position of the bottom-left corner of the houses. All that remains is to ensure that the dimensions are all somehow adjusted by the scale value:

```
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 moving the scale parameter into the procedure and 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**.

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);
}
```

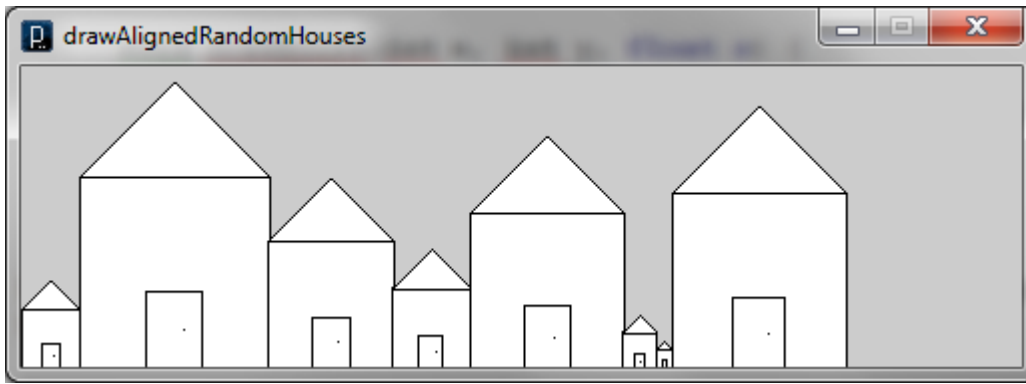
Notice how the variable **s** is declared within the procedure as a *local variable*. That means that **s** cannot be used outside of the procedure.

Of course, if we don't want our houses to overlap, we should ensure that we adjust our **x** offsets to be at least **100** pixels from one another:

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

    for (int x=0; x<=400; x=x+100)
        drawHouse(x, 150);
}
```

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:

Algorithm: DrawPackedHouses

1. **xOffset** \leftarrow 0
2. **width** \leftarrow random value
3. draw the 1st house at **xOffset**
4. **xOffset** \leftarrow **xOffset** + **width**
5. **width** \leftarrow random value
6. draw the 2nd house at **xOffset**
7. **xOffset** \leftarrow **xOffset** + **width**
8. **width** \leftarrow random value
9. etc...

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);
    ...
}
```

Comparing our algorithm with the code that we already have tells us that step 2 of the algorithm is done as the first couple of lines while we are drawing the house (i.e., within the

drawHouse() procedure). Lines 4 and 7 of the algorithm, however, require the width of the previously drawn house in order to compute the offset for the next house to be drawn.

You should realize that we need to get back the width of the house after we draw it, so that it can be used to compute the offset of the next house. That means, our **drawHouse()** procedure must actually become a **function** ... in that we now need a value returned from it.

Functions are created the same way that procedures are, but with one exception. Instead of **void**, a function must declare the *type* of the value returned as follows:

```
tr functionName (t1 n1, t2 n2, ..., tk nk) {
    // Write your function's code here
}
```

t_r here is the *return type* of the function:

*A **return type** is the type of the value that is returned from a function.*

So a function is exactly the same as a procedure, except that it must return a value of the type specified as its return type.

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 100*s;
}
```

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

The above code, however will not compile. It will return an error saying:

cannot convert from float to int .

It gives this error because the function requires an integer to be returned (i.e., return type is **int**) but we are trying to return **100*s** which is a **float**. We need to convert the return value to an integer.

Processing offers some pre-defined conversion functions for converting between the various data types:

- `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**

So in our function, we need to use `return int(100*s)` in order to get the integer that we need as a result. As a side note, however, 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**

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

Algorithm: DrawPackedHouses2

1. `xOffset ← 0`
2. `xOffset ← xOffset + drawHouse(xOffset)`
3. `xOffset ← xOffset + drawHouse(xOffset)`
4. `xOffset ← xOffset + drawHouse(xOffset)`
5. `etc...`

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;
  for (int i=0; i<8; i++)
    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. There isn't much more to say about them at this point.

2.8 Math & Trigonometry

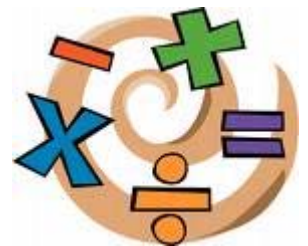
Obviously, a computer can compute solutions to mathematical expressions. We can actually perform simple math expressions such as:

$$30 + 5 * 2 - 18 / 2 - 2$$

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:

$$\begin{aligned} &30 + 5 * 2 - 18 / 2 - 2 \\ &30 + 10 - 18 / 2 - 2 \\ &30 + 10 - 9 - 2 \\ &40 - 9 - 2 \\ &31 - 2 \\ &29 \end{aligned}$$



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):

$$\begin{aligned} &(30 + 5) * (2 - (18 / 2 - 2)) \\ &35 * (2 - (18 / 2 - 2)) \\ &35 * (2 - (9 - 2)) \\ &35 * (2 - 7) \\ &35 * -5 \\ &-175 \end{aligned}$$

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:

```

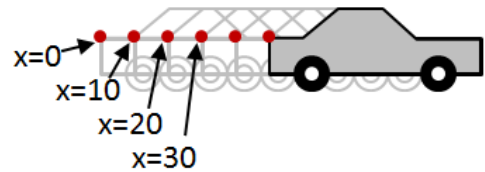
10 % 2    // results in the remainder after dividing 10 by 2 which is 0
10 % 3    // results in the remainder after dividing 10 by 3 which is 1
10 % 4    // results in the remainder after dividing 10 by 4 which is 2
39 % 20   // results in the remainder after dividing 39 by 20 which is 19

```

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:



Algorithm: DrawCar

```

1.   for successive x locations from 0 to windowHeight {
2.       draw the car at position x
3.       x ← x + 10
   }

```

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:

Algorithm: DrawCarWrapAround1

```

1.   x ← 0
2.   repeat {
3.       draw the car at position x
4.       x ← x + 10
5.       if (x > windowHeight) then
6.           x ← 0
   }

```

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

Algorithm: DrawCarWrapAround2

```

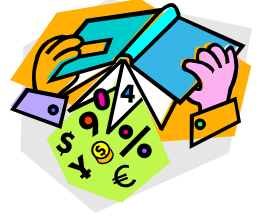
1.   x ← 0
2.   repeat {
3.       draw the car at position x
4.       x ← (x + 10) % windowHeight
   }

```

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)

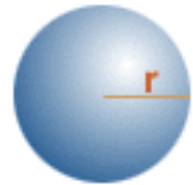


Example:

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);
```

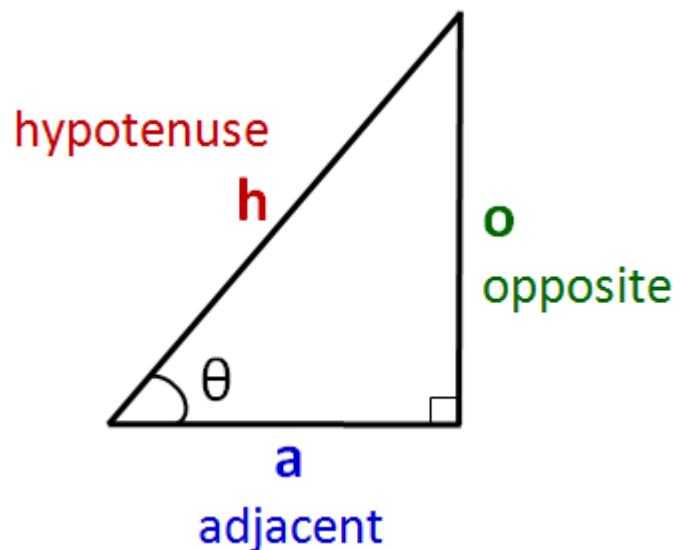
Or we could even pre-compute $4\pi/3$ first (which is roughly 4.1887903) :

```
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:

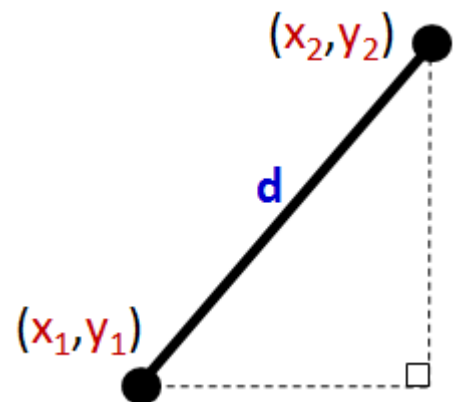
$$\begin{aligned} \text{sine}(\theta) &= o/h && \rightarrow \text{"soh"} \\ \text{cosine}(\theta) &= a/h && \rightarrow \text{"cah"} \\ \text{tangent}(\theta) &= o/a && \rightarrow \text{"toa"} \end{aligned}$$

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.

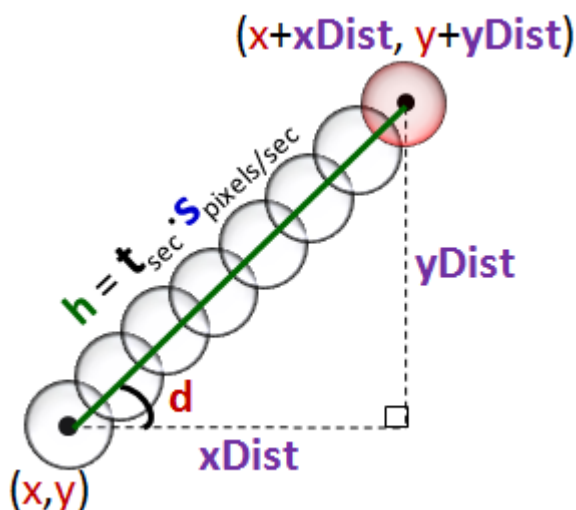
Getting back to the trigonometric functions, they are required in many computational problems as well as simulation and computer graphics problems.

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.



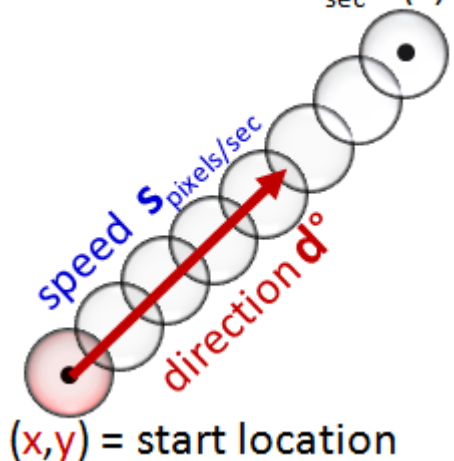
To determine the final location of the ball at time t , we just need to determine the amount of movement in the horizontal and vertical directions (i.e., $xDist$ and $yDist$). Then we can add those distances to the original (x, y) location to get the final location.

We can plug in our known trigonometric formulas:

$$\begin{aligned} \sin(d) &= yDist/h && \rightarrow yDist = h \cdot \sin(d) \\ \cos(d) &= xDist/h && \rightarrow xDist = h \cdot \cos(d) \end{aligned}$$

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

final location after $t_{sec} = (?, ?)$



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:

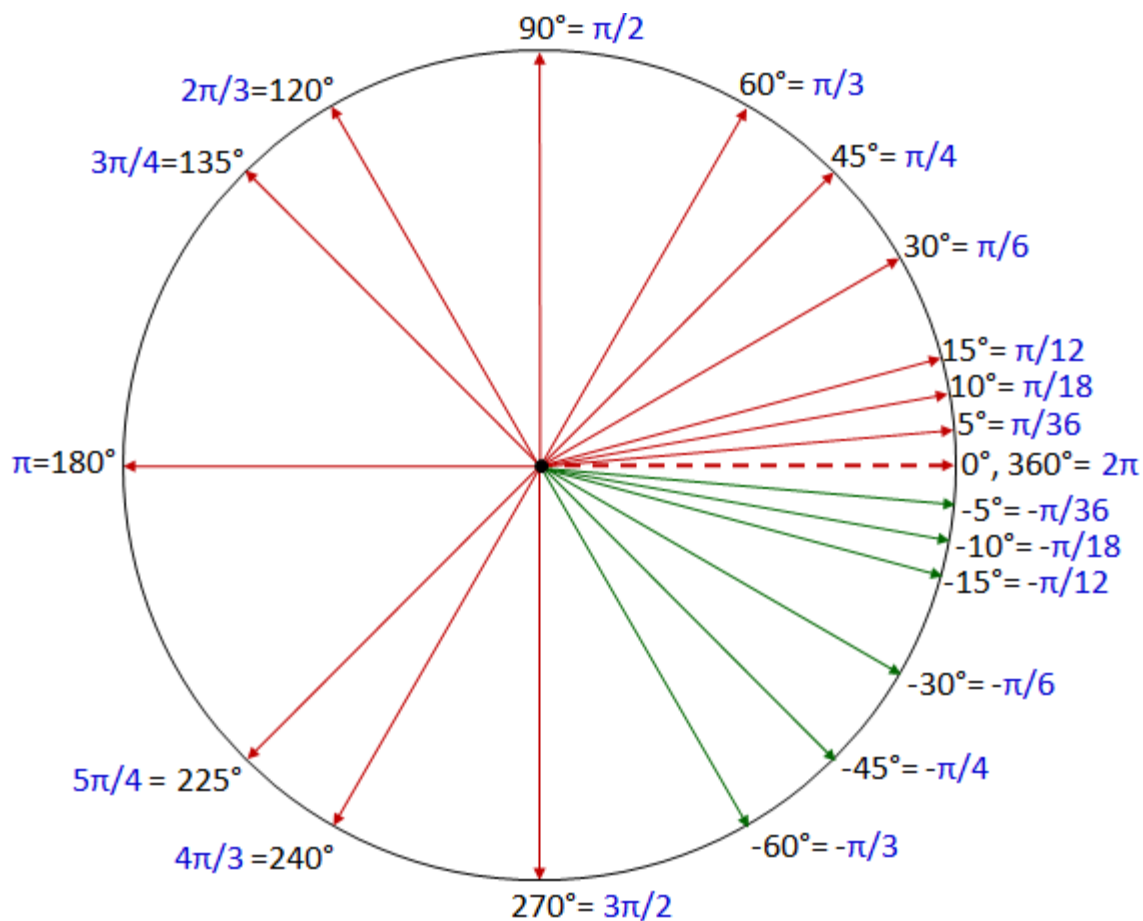
- **sin(a)** – returns the sine of angle **a** (which must be in radians)
- **cos(a)** – returns the cosine of angle **a** (which must be in radians)
- **tan(a)** – returns the tangent of angle **a** (which must be in radians)



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 $5\pi/3$ radians) is the same as the angle -60° (or $-\pi/3$ 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 $360*r/2\pi$)
- **radians(d)** – returns the radians value for degree angle **d** (computed as $2\pi*d/360$)

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 $0.0 \leq x < n$.

So, to get a random number from 0.0 to 99.999999999, 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 $0.0 \leq x < 1.0$.

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
```