Variables - One More (but not the last) Time with feeling…

All variables have the following in common:

- a name
- a type (**int**, **float**, …)
- a value
- an "owner"

We can describe variables in terms of:

- who owns them (Processing or us)
- who declares them
- who initializes them
- who changes them
- who can use them
- how long the actually "live" during the execution of our program.

Here is what we have talked about thus far:

Item	System Variables	Global Variables
Who owns them?	Processing	We do
Who declares them?	Processing	We do
Who initializes them?	Processing	Processing to a default initial value: int to zero float to zero.zero char to a null character boolean to false If these values are not useful to us, we have to initialize the variable to values that are useful and we should do this in the setup( ) function.
Who changes them?	Processing	We do
Who can use them?	Both Processing and us	We do

How long do they "live"?	**As long as the program is running**	**As long as the program is running**

We need to expand this list with two additional categories: **function definition arguments** *and* **local variables**. *We have to do this because you can get into trouble (your code will not run) through no fault of your own (until you read this...). Some of you may have already hit this problem. From the past:*
System variables are just that - variables maintained by Processing. We can use them but the code that declares and initializes them is hidden from us.
Global variables are declared in our code and must be outside any visible set of braces. The following code has four global variables marked in red:

```
int i;
void setup( )
{
   size( 400, 400 );
   fill( 0 );
}
float f;
void draw( )
{
   rect( 0, 0, width, height) ;
}
char c;
void keyPressed( )
{
   println( "Value of variable i is " + i );
   println( "Value of variable f is " + f );
   println( "Value of variable c is " + c );
   println( "Value of variable b is " + b );
}
boolean b;
```

Global variables do not have to be declared at the top of the code but for 15-102, we prefer that you do declare them at the top. It makes helping you easier for us.

The code above produces this output:

```
Value of variable i is 0
Value of variable f is 0.0
Value of variable c is □
Value of variable b is false
```

The four global variables are named **i**, **f**, **c**, *and* **b**. *They are global because they are declared outside the braces of the three function definitions. Note the initial values provided by Processing. The* **char** *(short for character) variable is initialized to a "special" null character that does not print anything so we do not "see" it. Word put the red box in this document to indicate that "something" is there.*

Function Definition Arguments are similar to system and global variables in that they have a name, type, value, and an owner. The differences between the function argument and the system and global variables can be seen in the other items in our list:

Item	System Variables	Global Variables	Function Definition Arguments
Who owns them?	Processing	We do	The function does
Who declares them?	Processing	We do	The function does in the function's header
Who initializes them?	Processing	Processing to a defined initial value: int to zero float to zero.zero char to space boolean to false If these values are not useful to us, we have to initialize the variable	Processing initializes them by copying the value of the argument in the function call into the function definition argument.
Who changes them?	Processing	We do	Only this function can change them.
Who can use them?	Both Processing and us	We do	Only this function can use them.
How long do they "live"?	As long as the program is running	As long as the program is running	Only as long as the function is being executed by Processing. When the execution of the function is complete, the memory used by the argument is returned to the operating system for reuse.

It is very important to remember two things:

1. *The initial value comes from the argument in the function call*
2. *The argument exists only while the function is being executed.*

Local variables are very similar to function definition arguments. Once again, we return to our list to find the differences:

Item	System Variables	Global Variables	Function Definition Arguments	Local Variables
Who owns them?	Processing	We do	The function does	The function does
Who declares them?	Processing	We do	The function does in the function's header	We do between the braces of the function
Who initializes them?	Processing	Processing to a defined initial value: int to zero float to zero.zero char to space boolean to false If these values are not useful to us, we have to initialize the variable	Processing initializes them by copying the value of the parameter in the call into the function argument.	We have to. If we do not initialize them before we attempt to use them, Processing will not compile our code
Who changes them?	Processing	We do	Only this function can change this argument	Only this function can change this local variable
Who can use them?	Both Processing and us	We do	Only this function can use this argument	Only this function can use this local variable
How long do they "live"?	As long as the program is running	As long as the program is running	Only as long as the function is being executed by Processing. When the execution of the function is complete, the memory used by the argument is returned to the operating system for reuse.	Only as long as the function is being executed by Processing. When the execution of the function is complete, the memory used by the argument is returned to the operating system for reuse.

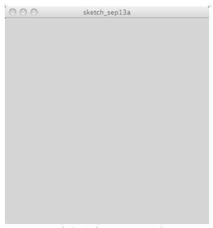
If we declare a variable within the braces of a function, that variable is a local variable. This point is very important. Forgetting it can cost you a great deal of time.

Here is how forgetting this can get you into trouble:

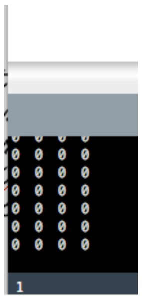
```
int x, y, wd, ht;
void setup( )
{
   size( 400, 400 );
   fill( 0 );
   int x = 100;
   int y = 50;
   int wd = 15;
   int ht = 10;
}

void draw( )
{
   rect( x, y, wd, ht) ;
   println( x + "  " + y + "  " + wd + "  " + ht );
}
```

This code compiles and runs and produces this output in the window:



We are expecting to see a black rectangle located 100 pixels to the right, 50 pixels down with a width of 15 pixels and an height of 10 pixels. But we see nothing. A clue comes from the **println( )** *function call. Here is what we see in the console window:*



```
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
```

The values of the global variables are all zero. So what happened???

```
int x, y, wd, ht;
void setup( )
{
   size( 400, 400 );
   fill( 0 );
   int x = 100;
   int y = 50;
   int wd = 15;
   int ht = 10;
}

void draw( )
{
   rect( x, y, wd, ht) ;
   println( x + "  " + y + "  " + wd + "  " + ht );
}
```

Let's return to the code. Look at the **setup( )** *function. The four lines with red text are variable declarations. Any time you see:*

```
data-type  variable-name;
```

or

```
data-type  variable-name =  some-value;
```

You are looking at a variable declaration. Since the four lines with red text are within the braces of the **setup( )** *function, they are declaring local variables owned by the setup function.*

At this point in the execution of the program, there are eight variables:

- *four global variables named* **x, y, wd, ht**
- *four local variables named* **x, y, wd,** *and* **ht**

Processing is very comfortable with this situation. It works with the local variables if they exist. This code alters the values of the four local variables and does not do anything with the global variables. Their values remain unchanged so they are zero.

One very important thing to remember is this row from our comparison grid:

Item	System Variables	Global Variables	Function Definition Arguments	Local Variables
How long do they "live"?	As long as the program is running	As long as the program is running	Only as long as the function is being executed by Processing. When the execution of the function is complete, the memory used by the argument is returned to the operating system for reuse.	Only as long as the function is being executed by Processing. When the execution of the function is complete, the memory used by the argument is returned to the operating system for reuse.

These four local variables exist only as long as it takes Processing to execute all of the "stuff" our code requires it to execute in the **setup( )** function. Once the **setup( )** function is finished, the four local variables, **x**, **y**, **wd**, and **ht** are destroyed leaving only the four unchanged global variables.

The code shown above can be fixed by removing the type name float from the four lines of code that are attempting to initialize the variables:

```
int x, y, wd, ht;
void setup( )
{
   size( 400, 400 );
   fill( 0 );
   x = 100;
   y = 50;
   wd = 15;
   ht = 10;
}
. . .
```

This is very subtle and you need to be aware of it. We will revisit these ideas again in the course.

When do I use a local variable and when do I use a global variable?

Part of the answer is rather easy to explain:
If the value of the variable is needed in more than one function, it probably should be a global variable.

This part is not that simple: For some cases, if the value of the variable is used only within one function, it can be a local variable.

The fuzzy part gets into efficiency. In an intro course, we usually do not worry about efficiency. However, we are not a typical intro course. We will do some animation and if we are not careful, we can write code that does not allow for smooth, fast, pleasant animation. Jim will show you an example of this in class.

The processor in the computer can only do so much in a finite amount of time. Processing has a default frame rate of 60 frames per second. This rate is possible only if the processor can do all we tell it to in a slice of time that is 1/60th of a second or less. If we tell it to do more than it can accomplish in 1/60th of a second, the animation rate slows down. Each line of code takes time. If we can eliminate a line of code, we reduce the amount of work the processor has to do in the 1/60th of a second for each frame and it can contribute to a smoother, more pleasant animation.

Suppose we have a function that computes the same value repeatedly for each animation:

```
void draw( )
{
   ellipse( width/2, height/2, frameCount%width,
                               frameCount%height);
}
```

In this example, the code draws an ellipse that grows from a very small circle to a large circle. The maximum diameter is the **width** or **height** of the window (the window is square).

The circle is always located at the center of the circle so the two arguments in green evaluate to the same value for every execution of the **draw( )** function.

However, since the value of **frameCount** changes for each execution of the **draw()** function, the values of the blue arguments are constantly changing. We could save some execution time for each execution of the **draw()** function if we declared global variables that store the values of half of the **width** and half of the **height**. Note that we could not measure the time we are saving in this simple example but we would safe a very small fraction of a second.

```
int halfWidth, halfHeight;
void setup( )
{
  size( 400, 400 );
  fill( 255 );
  halfWidth = width/2;
  halfHeight = height/2;
}
void draw( )
{
   ellipse( halfWidth, halfHeight,
            frameCount%width, frameCount%height);
}
```

As you write your code, you should begin to develop an "eye" for redundant or unneeded code.
Here is a common novice example:

```
void setup( )
{
   size( 400, 400 );
}
void draw( )
{
   background( 0 );
   smooth( );
   fill( 255, 0, 0 );
   rect( frameCount%width, height/2, width*.1, height*.2 );
}
```

There are a number of redundant or unneeded lines of code in the **draw()** *function. See if you can find them before reading further…*

Every execution of **draw( )** *does this:*

```
   background( 0 );
   smooth( );
   fill( 255, 0, 0 );
   rect( frameCount%width, height/2, width*.1, height*.2 );
```

This is a better way to code **draw( )***:*

```
   background( 0 );
   rect( frameCount%width, halfHeight, rectWidth, rectHeight );
```

The **smooth()** *function needs to be called only one time - do this in* **setup()***;*

Since there is only one rect and it is always filled with red, **fill( )** *needs to be called only one time.*
- do this in **setup()***;*

If there are multiple rects or ellipses of different color, this strategy will not work.

Declare global variables and initialze them in **setup()** *to be half of the* **height***, 10% of the* **width** *and 20% of the* **height***.*

```
int halfHeight;
float  rectWidth, rectHeight;
void setup( )
{
   size( 400, 400 );
   fill( 255 );

   halfHeight = height/2;
   rectWidth = .1*width;
   rectHeight = .2*height;
}
void draw( )
{
   background( 0 );
   smooth( );
   fill( 255, 0, 0 );
   rect( frameCount%width, halfHeight, rectWidth, rectHeight);
}
```

This type of coding may never affect your code but it makes your code much easier to read and edit.

Declaring functions that are not void in our code:

Thus far we have written only **void** functions. Many of you asked what **void** meant. We told you that it meant that the function "returns nothing". Not a very satisfying answer. Maybe we can add more depth to that answer.

Here is the structure of a function definition:

```
return type   name  open-paren  argument-list  close-paren
open-brace
  stuff-to-do
close-brace
```

and an example

```
void drawInitials( int x, int y, int wd, int ht)
{
  fill( 0 );
   . . .
}
```

The function **drawInitials( )** returns no values that the program can use. It draws our initials. So we say that since it returns no values, it is a **void** function.

Here is another function definition:

```
int doubleArgumentValue( int number)
{
  return number*2;
}
```

We would probably never write this as a function since doubling the value of a variable is very easy - just multiply it by 2.

But it does show how we can define a function to compute and return the value computed.

The call could look like any of these:

```
  text( doubleArgumentValue ( 42 , 100, 100 ) );
```

or

```
   int x = 42;
   int doubleX = doubleArgumentValue( x );
```

or

```
   int x = 42;
   int y = 24;
   rect( doubleArgumentVaule( x ),
         doubleArgumentVaule( y ), x, y );
```

Using functions like this is ideal for complex arithmetic expressions that are used multiple times in your code. Write the expression one time in a function and call the function where you need the value. Another advantage is that if you find that you coded the expression incorrectly, you only have to make a single correction.