A Not-So-Quick review of past stuff. . .

System variables:

There are a number of variables declared and initialized by Processing. Two such variables you may have used are:

width - which is the horizontal length of the screen.

height - which is the vertical length of the screen.

mouseX - x mouse location in this frame.

mouseY - y mouse location in this frame.

pmouseX - x mouse location in the previous frame.

pmouseY - y mouse location in the previous frame.

frameCount -number of current frame

There are others.

A general rule for system variables is to use them but never change them. Violate this rule at your own risk…

Global variables:

You have used global variables since Homework #2 when you declared either

```
float x, y, wd, ht;
```

The term "global" means that these variables can be used anywhere in your code at any time.

==========

Now we fold in old and new stuff:

Recently, you had to write function definitions.

A function definition tells Processing exactly what to do when the function is called.

Two functions, that Processing will call if we define them in our code :

- **setup( )**. It is called by Processing and executed AFTER the global variables are declared and initialized.
- **draw( )**. It is called and executed when Processing finishes executing **setup()**.

Here is one possible definition of the setup( ) function

```
void setup( )
{
   size( 400, 400 );
   background( #000000 );
   smooth( );
   fill( 200, 200, 0 );
}
```

The physical forms of all function definitions resemble this one. This definition tells Processing that when this function is executed it has to:
set the size of the window to 400 x 400
make the background black (this is the hex value for black)
turn on smoothing.
set the fill to yellow

The physical components of the function are labeled below:

< ---------- This entire line is the function header ---------- >

```
void                setup          (                    )
```

function return type function name argument list (which is empty)

```
{     opening brace to mark the start of the code in the definition

   list of what to do when the function is called
    size( 400, 400 );
    background( #000000 );
    smooth( );
    fill( 200, 200, 0. );

}     closing brace to mark the end of the function definition
```

When you use the setup( ) function, the first line after the opening brace MUST be a call to size( )! Violate this rule and your web page will not show your program.

The physical form of the draw( ) function is identical: to that of the function setup( ) as shown below:

```
void draw( )
{
     rect( width/2,   height/2,    .2*width,    .3*height);
}
```

If a function is not defined in Processing's API, then we have to define it for Processing. The function drawTarget( ) is not in Processing's API.

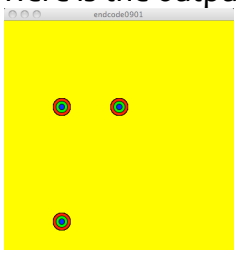
```
float diameter;
void setup(  )
{
   size( 400, 400 );  // This MUST be the first line.
   smooth( );
   background( 255, 255, 0 );

   diameter= 10;
}
```

```
void draw( )
{
   drawTarget( 100, 150 );
   drawTarget( 200, 150 );
   drawTarget( 100, 350 );
}

void drawTarget( float x, float y )  // function definition
{
    fill( 255, 0, 0 );
    ellipse( x, y, diameter*3, diameter*3);
    fill( 0, 255, 0 );
    ellipse( x, y, diameter*2, diameter*2);
    fill( 0, 0, 255 );
    ellipse( x, y, diameter, diameter);
}
```

Here is the output from the execution of this program:



The function **drawTarget( )** is not part of Processing, so Processing has no idea what **drawTarget( )** means unless Jim provides the definition of **drawTarget( )**. This definition is written in the code in the exact same physical form as the definition of **setup( )** and **draw( )**.

There are two differences:
Jim must call **drawTarget( )** if he wants Processing to execute it.
There is "stuff" inside the parentheses of the definition. We will look at that "stuff" in a minute.

When Processing executes the **draw( )** function, it encounters the function call:

```
drawTarget( 100, 150 );
```

Processing looks at its own personal list of function definitions in its API and does not find anything called

```
void drawTarget( float, float )
```

so it looks at Jim's code. If there is a definition telling Processing what to do, Processing is happy and runs the code. If there is no definition, it will not compile Jim's code.

The stuff in the parentheses:

Programming languages such as C, C++, Java and Processing provide a way to send data into a function via argument (or argument) lists. The way this works is straightforward:

The function drawTarget( ) is called here:	**This is the definition of the function drawTarget( ):**

void draw( )
{ arg #1 arg #2
drawTarget(100, 150);
---------- >
}
	arg #1 arg #2
	void drawTarget(float x, float y)
	{
	fill(255, 0, 0);
	ellipse(x, y, diameter*3, diameter*3);
	fill(0, 255, 0);
	ellipse(x, y, diameter*2, diameter*2);
	fill(0, 0, 255);
	ellipse(x, y, diameter, diameter);
	}

The value of the **first argument** shown in blue ➔	is copied to the **first argument** shown in blue. For this call of **drawTarget( )**, **x** will be 100 when the code is executed.
The value of the **second argument** shown in green ➔	is copied to the **second argument** shown in green. For this call of **drawTarget( )**, **y** will be 150 when the code is executed.

This is exactly how Processing executes the functions you have used in the first three home works. Using arguments' and arguments in this manner, we can draw targets anywhere in the window by specifying different x and y values.

The first line of the function definition contains the function's "signature". The signature of the **drawTarget( )** function is shown below in red:

void drawTarget(float x, float y)

We say that the signature of the function **drawTarget( )** is

drawTarget(float, float)

The signature of any function is composed of:

the name of the function

the list of the types of the arguments (not the names, just types)

The idea of a signature is very helpful in understanding how functions work in languages such as Processing. If you can figure out how they work, your efforts in writing code will be much easier. Time spent here is worthwhile to you for the next 13 weeks.

IF we return to the Processing API for the function **fill( )**, it shows us that there are eight different ways to call **fill( )**:

Syntax

```
fill(gray)
fill(gray, alpha)
fill(value1, value2, value3)
fill(value1, value2, value3, alpha)
fill(color)
fill(color, alpha)
fill(hex)
fill(hex, alpha)
```

This list is actually a list of eight different definitions of the function named fill. That's right, eight. Programming languages like Processing need a way to figure out which of the eight definitions of fill to use.

The way Processing does is involves the signature of the function call. As stated above, the signature is the name of the function and the list of the type of the arguments in the parentheses.

For the 8 definitions shown above, the signatures are:
fill(float)
fill(float, float)
fill(float, float, float)
fill(float, float, float, float)
fill(color)
fill(color, float)
fill(hex value)
fill(hex value, float)

*I*f you are asking , "can we use **int** *?*", you can. You can substitute an **int** for a **float** value. Processing just adds a .0 to the **int** making it a **float**. *Y*ou cannot substitute a float for an int value.

When Processing executes your code and encounters a call of the function **fill( )**, it determines the signature of the call.

So if we have this code:
fill(255, 17 , 43);

Processing determines that the signature is
fill(int, int, int)

None of the 8 definitions of the fill function match this signature but Processing knows it can substitute **ints** for **floats** so it chooses the definition with the signature:

fill(float, float ,float)

and follow the code in that definition.

Let's see how this works with functions we define and call.
In the class code shown above all of the calls originally had the same signature. Later Jim changed the calls to look like this:

```
void draw( )
{
  drawTarget( 100, 150 );
  drawTarget( 200, 150, 50 );
  drawTarget( 300, 150, 25, 50 );
}
```

Each call had a different signature.

```
drawTarget( float, float);
drawTarget( float, float, float);
drawTarget( float, float, float, float);
```

If we look at the function definitions in the code, we see:

Signature is drawTarget (float, float)	Signature is drawTarget (float, float, float)	Signature is drawTarget (float, float, float, float)

void drawTarget
(float x, float y)
{
fill(255, 0, 0);
ellipse(x, y, dia*3, dia*3);
fill(0, 255, 0);
ellipse(x, y, **2, **2);
fill(0, 0, 255);
ellipse(x, y, *, *);
}
	void drawTarget
	(float x, float y, float d)
	{
	fill(255, 0, 0);
	ellipse(x, y, d*3, d*3);
	fill(0, 255, 0);
	ellipse(x, y, d*2, d*2);
	fill(0, 0, 255);
	ellipse(x, y, d, d);
	}
		void drawTarget
		(float x, float y,
		float wd, float ht)
		{
		fill(255, 0, 0);
		ellipse(x, y, wd*3, ht*3);
		fill(0, 255, 0);
		ellipse(x, y, wd*2, ht*2);
		fill(0, 0, 255);
		ellipse(x, y, wd, ht);
		}

Processing matches the signature of the call to the signature of the definition and follows the code inside that version of the function.

If Processing cannot match the signature of the call with the signature of any of the definitions, it will not compile and run your program. Instead, it will display an error message.

Major important question and a new topic for discussion – "How does the data get from the call?"

```
drawTarget( 100, 150 );
```

to the definition?

```
void drawTarget   ( float x, float y )
{
   fill( 255, 0, 0 );
   ellipse( x, y, dia*3, dia*3);
   fill( 0, 255, 0 );
   ellipse( x, y, dia*2, dia*2);
   fill( 0, 0, 255 );
   ellipse( x, y, dia, dia);
}
```

The cheap and easy answer is, "automatically."

But that is not fair to you and your understanding. The detailed answer is a straightforward process.

First Processing finds the right function definition to execute by comparing the signature of the call to the signature of the definition.

Next it goes to the call and determines the value of the first argument in the function call

```
drawTarget( 100, 150 );
```

-If the argument is a constant like the one above, it uses the constant's value.
- If the argument is a variable, it looks up the value.
- If the argument is an expression, it evaluates the expression.

Processing then copies the values of the arguments in the call into the arguments of the definition:

```
void drawTarget   ( float x, float y )
```

The arguments in the definition are like variables. They have a type (float) and a name (x). The difference is that they get their initial values when Processing copies the values of the arguments in the call into the arguments in the definition.

Processing copies the first argument's value in the call into the first argument of the definition:

The first argument in the call is the constant 100 so we see this:

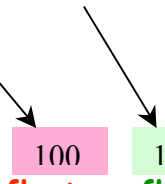
The call: `drawTarget( 100, 150 );`

The definition `void drawTarget ( float x, float y )`

Processing copies the value of second argument in the call into the second argument of the definition and we see this:

The call:

`drawTarget( 100, 150 );`



The definition

`void drawTarget ( float x, float y )`

This technique of copying the values in the argument list of the call into the argument list of the definition is referred to as the argument binding.

Two important thing to know is:
-- how Processing "knows" which function to execute
and
-- how the values get from the function call to the function definition.

Final Thoughts:

Understanding how functions work in Processing is a major foundation block to the remaining work You have to work with this and until you are comfortable.

You will have to write functions in every homework starting now.

You will have to write functions on the exams and these functions will require arguments.

You will be shown function calls and you will have to write the signatures of the calls.

You will be shown function definitions and will have to write the signatures of the definitions.

You will have to draw the figures shown on the previous page and label them.

Read the previous set of notes and these notes again and bring your questions to class next time.