Yes, this stuff is on the exam.
Know it well.
Read this before class and bring your questions to class.

Starting today, we can no longer write our code as a list of function calls and variable declarations as we have done in the first three programs:

```
size( 400, 400, P3D );

int depth = 0;
int edge = (int)(.07*width);

println("Printing the initial J");

background( #04075A );
lights( );
fill( #F79007 );

// Cap of J
// back row of blocks
pushMatrix( );
  translate( width*.5, height*.2, depth - edge - 5);
  fill( #F79007 );
  box( edge);
popMatrix( );
```

We are moving from the "basic" style of coding to the "structured" style where "structured" means that everything is done inside of "function definitions".

Before we get into what "function definitions" are, let's review variables.

Variables are used in our programs to store data for use in our code. Homework #2 (hopefully) demonstrated the value of using variables with arithmetic so we can easily move and/or resize the initials. Processing has about 8 different kinds or "types" of variables. We have used **float**. We will look at these in greater depth a bit later.

*Variables can be described by the type of data they store (**float**) and, by their owner - who declares them and how they are initialized. The discussion that follows looks at variables based on their owner.*

System variables:

There are a number of variables declared and initialized by Processing. We have seen two thus far:

- **width** *- which is the horizontal length of the screen.*
- **height** *- which is the vertical length of the screen.*

*Processing declares these variables as **int** (meaning that they cannot have a decimal or fractional part) and initializes them to 100.*

*If we call the function **size(some-int-value, some-int-value);** Processing will take the first argument in the parentheses and assign that value to **width**. Processing then takes the second argument and assigns that value to **height**.*

We will encounter other system variables as we continue to work with Processing.

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

Global variables:

The term "global" is new for us but we have used global variables since Homework #2. When you declared either

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

you were declaring what we call "global variables." The term "global" means that these variables can be used anywhere in your code at any time. For right now, we will leave the explanation at this level. Some of you might be thinking, "is there any other kind of variable?" and the answer is "yes." And we will soon encounter another kind of variable.

======================
Starting with homework #4, our code will have to consist of only variable declarations and function definitions.

"OK - what is a function definition?"

The word definition tells us what the word means.

A function definition explains in often painful detail what Processing must do when our code calls the function.

When we code this function call that tells Processing to execute the **translate** function:

```
  translate( width*.5, height*.2, depth - edge - 5);
```

some piece of code must define for Processing exactly what to do with the values in the argument list and exactly how to do a translation. Processing does not "know" how to **translate**. It has to have very precise and exact directions.

It is time to add function definitions to our code. Right now, function definitions will be divided into three categories:

- The first category has the functions defined in Processing and executed if our code calls them such as the following

```
        pushMatrix( );
            translate( width*.5, height*.2, depth - edge - 5);
            fill( #F79007 );
            box( edge);
        popMatrix( );
```

The definitions of these functions (like all of the functions in the API) are buried somewhere in Processing out of sight to us.

- The second category has functions that we can define in our code but they are called by Processing. This is new for us. These functions that we define will be called automatically for us at the right time by Processing -- MUST never be called in our code.

The first one of these is named **setup( )**. It MUST NEVER have arguments. If we define the function **setup()** in our code, this will be called by Processing and executed AFTER the global variables are declared and initialized.
The second one of these is named **draw( )**. It MUST NEVER have arguments. Processing calls and executes this when it finishes executing **setup()**.

The **setup()** function should be used to "set up" the initial values of variables, **color**, **strokeWeight**, and draw any background stuff that is needed. If we relate this to a stage play - the **setup( )** function sets the stage so **draw( )** can perform the play. (There is more to **draw( )** than we are showing here... heh... heh...heh...).

Here is one possible definition of the **setup( )** function

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

The physical form of all function definitions resemble this one. This definition tells Processing that when this function is executed it has to:

1. set the size of the window to 400 x 400
2. make the background black
3. turn on smoothing
4. set the fill to a somewhat dim shade of yellow

The physical components of the function are labeled below:

< ================ this 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

This is the function **body**

<
|
|
|
|
|
|
<

Each function has what we call a **signature**. *The* **signature** *of this function is:*

setup( )

Details on signatures are on page 8.

The stuff above described with the blue text will be on exam 1.

The function return type (**void** *in this example*) *tells Processing what this function returns. Some functions like the squre root function return values. In programming, we can define functions that are marked* **void** *which return nothing. Unlike functions in math, programming functions do not always have to return values. Typically,* **void** *functions "do stuff" instead of "returning values." such as* **fill(0)** *or* **smooth( )**.

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 as shown below is identical to that of the function* **setup( )**

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

Compare the two definitions. You will see:

- *a return type,*
- *function name,*
- *parentheses,*
- *an opening brace,*
- *a list of what must be done*
- *and a closing brace.*

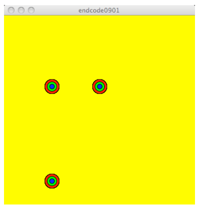
Processing will execute **setup( )** *first and then execute* **draw( )**.

A third category of functions has the functions that we define and we call.

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

   radius = 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 idea for the function **drawTarget( )** *comes from deep somewhere inside Jim's brain. The function 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 associated with this function:

- *First, Jim must call* **drawTarget( )** *if he wants Processing to execute it.*
- *Second, There is "stuff" inside the parentheses in the header line of the definition. We will look at this "stuff" in a minute.*

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

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

Processing looks at its built-in list of function definitions (the API) and does not find anything called **drawTarget(float, float)**. *So it looks at Jim's code. If there is a definition of* **drawTarget( )** *telling Processing what to do, it is happy and runs the code. If there is no definition, it will not compile Jim's code.*

The stuff in the parentheses:

Shiffman calls the stuff in the parentheses of the definition the arguments. He refers to the stuff in the parentheses of the function call as parameters. This terminology is not uniform and is confusing to everyone but especially confusing to novices. Just remember that arguments and parameters are the same exact thing - consider them to be the same word. You say "toe-may-toe: and I say "tah-mah-toe"...[1]

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

The function drawTarget( ) is called here:
 	
`void draw( )`
`{                arg #1      arg #2`
`  drawTarget(  100,         150 );`
 	
`}`
	This tells Processing what to do when drawTarget ( ) is called
	
	`                   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 in the call shown in blue ➔	is copied to the first argument in the definition - also shown in blue. So for this call of **drawTarget( )**, **x** will be **100** when the code is executed.
The value of the second argument sin the call shown in green ➔	is copied to the second argument in the definition - also shown in green. So 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. By using arguments' in this manner, we can draw targets anywhere in the window by specifying different **x** and **y** values. Think about this and bring your questions to class next time.

Back to the "signature of the function" first mentioned on page 5 ➔ The first line of the function definition contains the "function signature". The signature of the **drawTarget( )** function is shown below in red:

void drawTarget(float x,   float y)

[1] (very old joke...)

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

drawTarget(float , float)

The signature is the name of the function and the list of the types of the arguments.

We begin here on next time… Bring your questions to class.