

## Functions – More Than You Want to Know... But Not Everything You Need to Know...

*Shiffman does a very good job explaining functions but there were some minor murmuring last term because there were no class notes for this topic. You really need to read that stuff for today.*

*Here is a completely different look at functions – one that exists nowhere else<sup>1</sup>.*

*Let's consider functions not in terms of what they do or what arguments they need but in terms of:*

- who “owns” them
- who “uses” them

*The entity that “owns” a function is the entity that wrote it. In the first week, all of the functions you used were owned by or written by Processing programmers: fill(0), ellipse( 10, 10, 5, 2 )... were all written by Processing programmers some time in the past.*

*When someone “writes” a function (the better term to use is “defines a function”), they put in the code needed to tell Processing how to do whatever task the function is supposed to do. Processing defined all of the functions you have used thus far in the class.*

*The entity that “uses” the function is the entity that types the name of the function and the required arguments somewhere in a program. You were the user of the functions. You used the functions. A better term is call. You called the functions that were owned by Processing. Your programs have been a list of function calls and variable declarations and assignments.*

*Using this nomenclature (remember, this is uniquely a Jim thing...) we can categorize the functions we will use in the next three class meetings.*

*Wait a minute.*

- Who else would write the functions but Processing?
- Who else would call the functions but us?

*This is where some new fun begins.*

---

<sup>1</sup> which should really scare you...

Today we are going to look at the exact reverse. We are going to **write** or **define** the functions and Processing **will** use or **call** them...

There are two functions that Processing will call we define them in our program:

- **setup( )**
- **draw( )**

If we do not define them, Processing does not care – it is just one thing less for Processing to do today.

### BIG CHANGE ALERT

Starting with homework #3 all of our code must be either

- variable declarations
- function definitions

We will explain this in class.

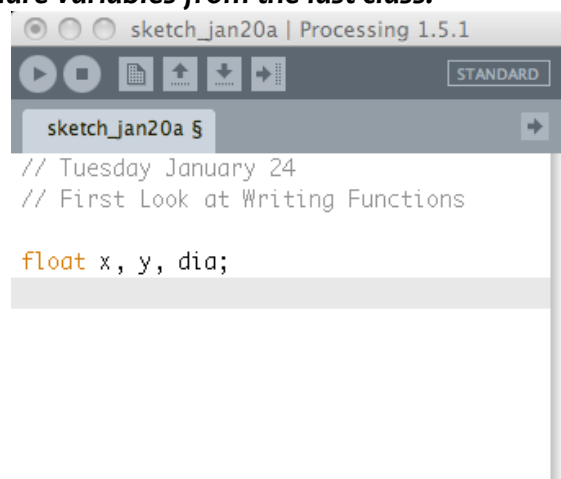
The agreement we have with Processing is the following:

1. Processing will take care of any variables that we declare first.
2. Processing will call (execute the code inside) the **setup( )** function next – if we have one defined (written) in our program.
3. Processing will then call (and execute the code inside) the **draw( )** function if we have one defined (written) in our program. **B/T/W Processing executes the draw( ) function over and over again (about 60 times per second) until we either stop it or close the graphics window.**

Here is the structure of all of our programs from now on:

- variable declarations (and possible initializations)
- definition of the **setup( )** function
- definition of the **draw( )** function

We know how to declare variables from the last class:



```
sketch_jan20a | Processing 1.5.1
STANDARD
sketch_jan20a $
// Tuesday January 24
// First Look at Writing Functions

float x, y, dia;
```

This code tells Processing to reserve three pieces of memory configured to hold decimal numbers and name them **x**, **y**, and **dia**.

Now how do we define or write a function? The physical form of all function definitions is the same. Here are the skeletons for the definitions of a **setup( )** and a **draw( )** function:

```
sketch_jan20a | Processing 1.5.1
sketch_jan20a §
// Tuesday January 24
// First Look at Writing Functions

float x, y, dia;

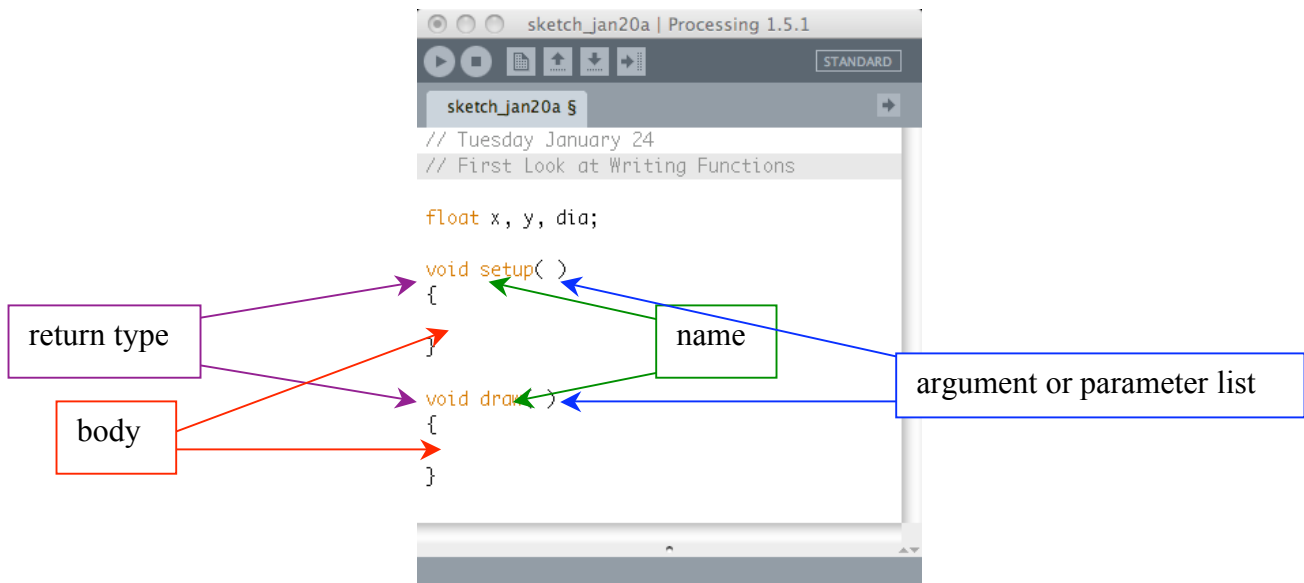
void setup( )
{

}

void draw( )
{

}
```

Here are the labels for each part:



Lets take these one at a time. First the **return type**. In math class we learned that functions return values. The square root function takes an argument (4), computes the square root of the argument and returns it (2). This is how your calculator works. Processing has functions like this. Next class we will work with a function named **random( )** that returns a random number. The number it returns is a **float** number (has a decimal) so the **return type** of **random( )** is **float**.

In programming languages like Processing we also have functions that do things but do not return any values that we can use in our program. The functions you have been using are such functions. The **rect( )** function draws a rectangle but does not return any value to our program. When a function does not return any value, Processing considers its **return type** to be **void**.

Our **setup( )** and **draw( )** functions must have a **return type** of **void** or Processing will not call them.

The **name** of the function must begin with an alpha character. The remaining characters must be alpha or digit. By **convention** the first letter is lower case – please follow this convention so we can read and understand your code. The name should describe what the function does. We will use our **setup( )** function to tell Processing how to “set things up” for our program. This is where we put code that we need to do before the program begins to draw.

The two parentheses make up the **argument list**. The **argument list** for **setup( )** and **draw( )** must be empty or Processing will not call them. Later we will define functions with arguments in the **argument list**.

The **body** of the function is within a pair of braces **{ }** is were we tell Processing exactly what we want the function to do when it is called.

The **body** of our **setup( )** function is where we put the code in that we want processing to execute before any drawing begins.

The first line of code in **setup( )** must be

**size( magic-number, magic-number);**

Use numbers for the arguments. Do not use variables or expressions. They might work but not always.

The **body** of our **draw( )** function is where we put the function calls to draw what we want to display in the window.

The next page shows a sample of how we use these two functions in a program:

```

// Homework #2
// Copyright Jim Roberts 2012

float x, y, dia;

void setup( )
{
  size( 400, 400);
  x = 150;
  y = 150;
  dia = 200;

  smooth( );
  background( 0 );
}

void draw( )
{
  strokeWeight( .05*dia );

  // Surrounding Background
  fill( 0 );
  stroke( 200, 200, 0 );
  beginShape( );
  vertex( x, y-dia/2);
  vertex( x + dia/3, y-dia/3);
  vertex( x + dia/2, y );
  vertex( x + dia/3, y+dia/3);
  vertex( x, y+dia/2 );
  vertex( x - dia/3, y+dia/3);
  vertex( x - dia/2, y );
  vertex( x - dia/3, y-dia/3);
  endShape( CLOSE );

  noFill( );
  beginShape( );
  vertex( x, y-dia/1.5);
  vertex( x + dia/2.5, y-dia/2.5);
  vertex( x + dia/1.5, y );
  vertex( x + dia/2.5, y+dia/2.5);
  vertex( x, y+dia/1.5 );
  vertex( x - dia/2.5, y+dia/2.5);
  vertex( x - dia/1.5, y );
  vertex( x - dia/2.5, y-dia/2.5);
  endShape( CLOSE );

  // First part of R
  stroke( 255, 0, 0 );
  beginShape( );
  curveVertex( x - .2*dia, y + .5*dia );
  curveVertex( x - .2*dia, y + .4*dia );
  curveVertex( x - .2*dia, y );
  curveVertex( x - .2*dia, y - .3*dia );
  curveVertex( x - .1*dia, y - .4*dia );
  curveVertex( x + .1*dia, y - .4*dia );
  curveVertex( x + .2*dia, y - .3*dia );
  curveVertex( x + .2*dia, y - .05*dia );
  curveVertex( x + .1*dia, y );
  curveVertex( x , y );
  curveVertex( x - .2*dia, y );
  curveVertex( x - .3*dia, y - .2*dia );
  endShape( );

  // second part of R
  beginShape( );
  curveVertex( x - .3*dia, y - .2*dia );
  curveVertex( x - .2*dia, y );
  curveVertex( x , y );
  curveVertex( x + .2*dia, y + .1*dia );
  curveVertex( x + .2*dia, y + .4*dia );
  curveVertex( x + .2*dia, y + .5*dia );
  endShape( );

  // J
  stroke( 0, 255, 0 );
  beginShape( );
  curveVertex( x - .4*dia, y - .1*dia );
  curveVertex( x - .35*dia, y - .2*dia );
  curveVertex( x - .3*dia, y - .27*dia );
  curveVertex( x - .24*dia, y - .3*dia );
  curveVertex( x - .17*dia, y - .37*dia );
  endShape( );
  line( x - .3*dia, y - .27*dia,
        x - .3*dia, y + .27*dia );

  beginShape( );
  curveVertex( x - .25*dia, y + .30*dia );
  curveVertex( x - .3*dia, y + .27*dia );
  curveVertex( x - .35*dia, y + .17*dia );
  curveVertex( x - .40*dia, y );
  curveVertex( x - .5*dia, y + .1*dia );
  endShape( );

  // A right side
  stroke( 0, 0, 255 );
  beginShape( );
  curveVertex( x + .25*dia, y - .4*dia);
  curveVertex( x + .33*dia, y - .3*dia);
  curveVertex( x + .39*dia, y - .2*dia);
  curveVertex( x + .43*dia, y - .1*dia);
  curveVertex( x + .45*dia, y );
  curveVertex( x + .42*dia, y + .1*dia);
  curveVertex( x + .38*dia, y + .2*dia);
  curveVertex( x + .35*dia, y + .3*dia);
  curveVertex( x + .28*dia, y + .3*dia);
  endShape( );

  // a left side
  beginShape( );
  curveVertex( x + .41*dia, y - .4*dia);
  curveVertex( x + .33*dia, y - .3*dia);
  curveVertex( x + .27*dia, y - .2*dia);
  curveVertex( x + .22*dia, y - .1*dia);
  curveVertex( x + .21*dia, y );
  curveVertex( x + .24*dia, y + .1*dia);
  curveVertex( x + .28*dia, y + .2*dia);
  curveVertex( x + .29*dia, y + .3*dia);
  curveVertex( x + .25*dia, y + .3*dia);
  // control point

  // a horizontal
  line( x + .21*dia, y, x + .45*dia, y );
}

```

Using this idea of who defines the function and who calls them we can add a third dimension – when are they called:

- We call Processing's functions when we need them.
- Processing calls `setup( )` if we define it
- Processing calls `draw( )` if we define it.

There are two more additions to this family of function descriptions:

- Next time we will look at functions that Processing calls **if** we define them **when** certain actions or events occur.
- In the following class we will look at functions that we define and we call.

Beginning with Homework #3, we will always use a `draw( )` and a `setup( )` function in our code.

## And now... Variables...

Now, let's apply this idea of ownership and use to variables. Last time we introduced you to the variables `x`, `y`, and `wd` and `ht`:

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

The line of code declares the variables `x`, `y`, and `dia` as variables that can store a decimal number or a `float`. This line of code means that we **own** them.

And we **use** them when we initialize them or assign them meaningful values:

```
x = 100;  
y = 100;  
wd = 50;  
ht = 50;
```

Processing does not own or assign them values. It will change their values if we tell it to:

```
x = x * 2;
```

and Processing will look up their values and use them to do something

```
ellipse( x, y, wd, ht);
```

but Processing will not alter them or look them up unless we tell it to do so.

We will call these variables **global variables** because they are globally available for use anywhere in our program.

Processing has its own variables. Jim calls the **system variables**. In general when you read or hear the word **system**, we are referring to **Processing** or the **operating system** on your computer or an part of them that is against us ... be very careful... sorry...

System variables are variables that Processing declares and assigns values. Processing changes their values as needed. We can change them **BUT WE MUST NEVER DO THIS** because the results can be disastrous. We can use them in our code **but we must never change their values**.

Here are a few of the **system variables** you might be able to use in some way in homework #3:

<b>width</b>	This is the horizontal dimension of the window. Its default value is 100. If we call the size function, the first argument is used as the value of width.
<b>height</b>	This is the vertical dimension of the window. Its default value is 100. If we call the size function, the second argument is used as the value of width.
<b>mouseX</b>	This is the x or horizontal location of the mouse in the window <b>this frame</b> <sup>2</sup> .
<b>mouseY</b>	This is the y or vertical location of the mouse in the window <b>this frame</b> .
<b>pmouseX</b>	This is the x or horizontal location of the mouse in the window <b>last frame</b> .
<b>pmouseY</b>	This is the y or vertical location of the mouse in the window <b>last frame</b> .
<b>frameCount</b>	This is the <b>number of this frame</b> <sup>3</sup> .

One more thing – you may want to look up the function **random()** in the API. It might be useful in homework #3 also.

<sup>2</sup> We will explain “this frame” in class.

<sup>3</sup> Ditto...