

## Processing the Danger Shield

Materials by Lindsay Craig, edited by Ben Leduc-Mills



### I. What Processing is:

Processing is a Java based programming environment that draws on PostScript and OpenGL for 2-D and 3-D graphics respectively. Processing is a wonderful entry level program that interfaces easily with Arduino via Serial, making it a simple, yet powerful environment.

### Who created Processing:

Processing was conceived at MIT in 2001 by Casey Reas and Ben Fry. Processing is a FLOSS project (Free, Libre, Open Source Software) with millions of contributors all linked by the Processing website, Processing.org. Processing has a system of software extensions called “libraries”, this allows people to write code and extend the abilities of the original software for various purposes. These “libraries” are available on the website, including the Arduino library which is one way to interface Arduino hardware with your Processing sketches.

### Downloading and installing Processing:

Go to <http://processing.org/download> and select Linux, Mac or Windows depending on what kind of machine you have.

#### For Linux:

Download the .tar.gz file to your home directory, then open a terminal window and type:

```
Tar xvfz processing-xxxx.tgz
```

(replace xxxx with the rest of the file’s name, which is the version number)

This will create a folder named processing-1.5 or something similar. Then change to that directory:

```
cd processing-xxxx
```

and run processing:

```
./processing
```

#### For Mac:

Double-click the .dmg file and drag the Processing icon from inside this file to your applications folder, or any other location on your computer. Double click the Processing icon to start Processing.

#### For Windows:

Double-click the .zip file and drag the folder inside labeled Processing to a location on your hard drive. Double click the Processing icon to start Processing.

## Intro to Processing with the Danger Shield

If you are stuck go to <http://wiki.processing.org/index.php/Troubleshooting> for help.

### PROCESSING CHEAT SHEET

#### DATA TYPES

**Primitive**  
boolean  
byte  
char  
color  
double  
float  
int  
long

**Composite**  
Array  
ArrayList  
HashMap  
Object  
String  
XMLElement

#### Conversion

binary()  
boolean()  
byte()  
char()  
float()  
hex()  
int()  
str()  
unbinary()  
unhex()

#### String Functions

join()  
match()  
matchAll()  
nf()  
nfc()  
nfp()  
nfs()  
split()  
splitTokens()  
trim()

#### Array Functions

append()  
arrayCopy()  
concat()  
expand()  
reverse()  
shorten()  
sort()  
splice()  
subset()

#### Constants

HALF\_PI  
PI  
QUARTER\_PI  
TWO\_PI

#### Assign variables

= assign value to a variable  
; statement terminator  
, separates parameters in function  
separates variables in declarations  
separates variables in array

**Assign variables Example**  
//Format is in variable\_type variable\_name;  
int total;  
//Then you can assign a value to it later  
total = 0;  
//Or, assign a value to it at the same time  
int total = 0;  
//Note: use one of the primitive data types on the left

#### Structure: program structure

setup() defines initial environment properties, screen size, background before the draw()  
draw() called after setup() & executes code continuously inside its block until program is stopped or noLoop() is called.  
size() size() must be first line in setup() defines dimension of display in units of pixels  
noLoop() Stops Processing from executing code within draw() continuously

**Example**  
void setup() {  
size(200, 200);  
background(0);  
fill(102);  
}  
void draw() {  
//Draw code here  
}

#### 2D Primitives

point() draws a point  
point(x, y)  
point(x, y, z) //3D

line() draws a line  
line(x1, y1, x2, y2)  
line(x1, y1, z1, x2, y2, z2) //3D

rect() draws a rectangle  
rect(x, y, width, height)

ellipse() Draws an ellipse  
ellipse(x, y, width, height)

arc() draws an arc  
arc(x, y, width, height, start, stop)

**Arc Example**  
//x & y = coords, width & height = size  
//start + stop = starting and end points (think angle in radians) of circle in  $\pi$  pie  
LINK  
arc(x, y, width, height, start, stop)  
arc(100, 100, 50, 50, PI, 2\*PI) //Sad Face  
arc(100, 100, 50, 50, 0, PI) //Happy Face  
//Note: Play around with start and stop. Use PIE constants or math operators PI/3, .5\*PI

#### Relational

== equality  
> greater than  
>= greater than or equal to  
!= inequality  
<= less than or equal to

**Example**  
if(total == 100){  
//Then do this  
}

#### Iteration

while executes statements while the expression is true  
for loop continues until the test evaluates to false

**while Example**  
while(total < 100){  
total++; //adds 1 to total  
}  
  
**for Example**  
for(int i=0; i<100; i++){  
//Do something here  
}

#### Conditionals

if if statement evaluates to true then execute code  
else extension of if statement executes if equals false  
else if extension of if statement executes if equals true

**if / else / else if Example**  
if(total == 100){  
//total is equal to 100  
}  
else  
if(total < 100){  
//total is smaller than 100  
}  
else{  
//total is bigger than 100  
}

#### Coloring stuff

background() sets background color in RGB or hexadecimal color  
background(value1, value2, value3)  
background(hexadecimal\_value)

fill() sets color for shape  
fill(value1, value2, value3)  
fill(hexadecimal\_value)

stroke() sets color for shape  
stroke(value1, value2, value3)  
stroke(hexadecimal\_value)

**Example**  
//Note call fill or stroke before every shape you are planning on using different colors on each  
stroke(#CCCCFF);  
fill(#FFCCCC);  
rect(100, 100, 50, 50);

#### CONTROL

**Relational Operators**  
== (equality)  
> (greater than)  
>= (greater than or equal to)  
!= (inequality)  
< (less than)  
<= (less than or equal to)

#### Iteration

for  
while

#### Conditionals

break  
case  
?: (conditional)  
continue  
default  
else  
if  
switch()

#### Logical Operators

&& (logical AND)  
! (logical NOT)  
|| (logical OR)

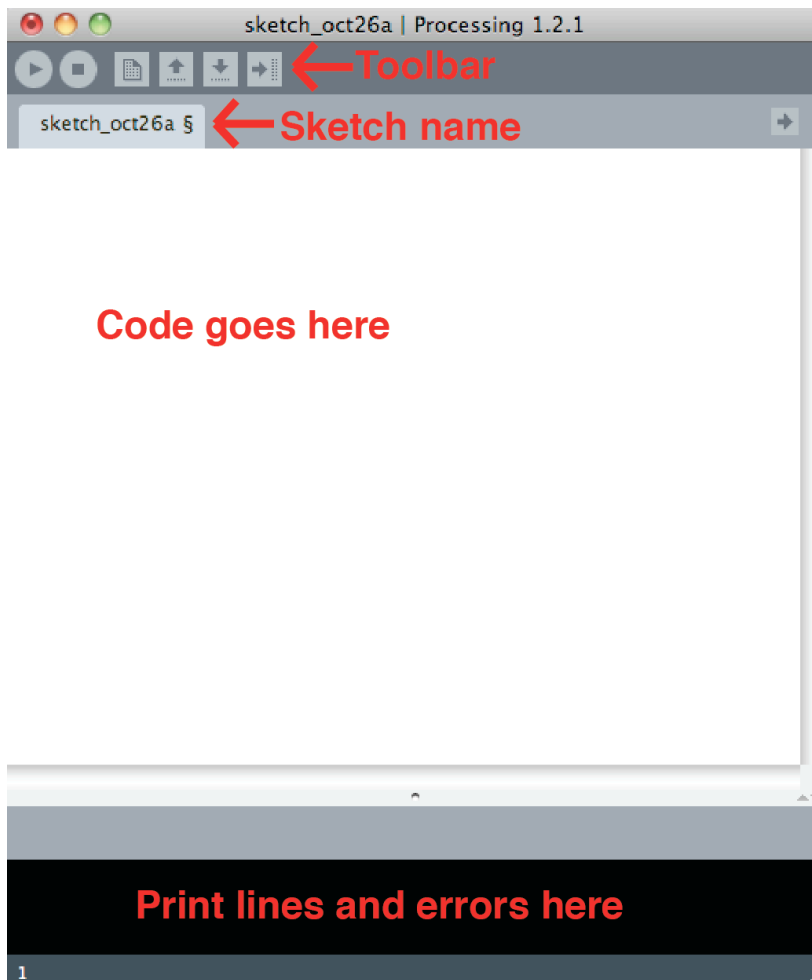
Cheat Sheet courtesy of Chrisdrogaris.com



## II. Parts of a Sketch in Processing:

### The Sketch:

A sketch is a file or project you create in Processing. When you first open up a new sketch it will be completely blank. Below is an example of a blank sketch.



Toolbar buttons:

There are a few key interface buttons that you will need to understand to get started in Processing, the rest you can discover later. These buttons are:

Run:  Stop:  New:  and Save: 



Run executes the code you have written... or it doesn't if you have errors in your code.



Stop ends the execution of your Processing Sketch.



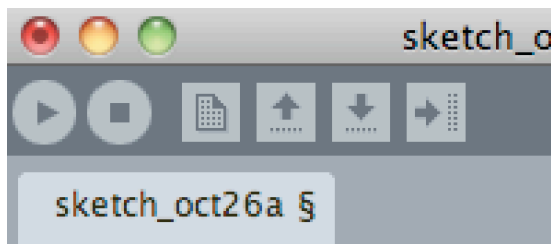
New creates a new blank sketch in Processing.



Save we are all familiar with, it saves your Processing sketch. **Save often and save different versions so you can backtrack if you need to.** This cannot be stressed enough. Create your own saving filename structure to keep track of your progress. You may never use your previous saved files, but if you do you will be very, very thankful that you saved previous versions.

The setup function:

This function runs once, at the very beginning of your sketch. You will use setup to set up certain aspects of your sketch, makes sense right? This is where you will declare things like the size of the window your sketch will appear in, variables you plan to use a lot, and image modes such as smooth (a less pixelated way to draw images) and 3-D image ability. Most importantly for this class you will begin Serial communication in the setup function. The setup function without anything in it looks like this:



```
void setup(){
  Setup code goes here between curly brackets
}
```

The draw loop:

This function is where everything happens in your sketch. The draw loop is the portion of code that keeps repeating while the Processing sketch is open. Any animation, interaction or changes to the images or variables in your sketch will need to be programmed inside of this loop. The draw loop looks like this:



```
void setup () {
}
void draw() {
  Code goes here between the curly
  brackets and repeats forever
}
```

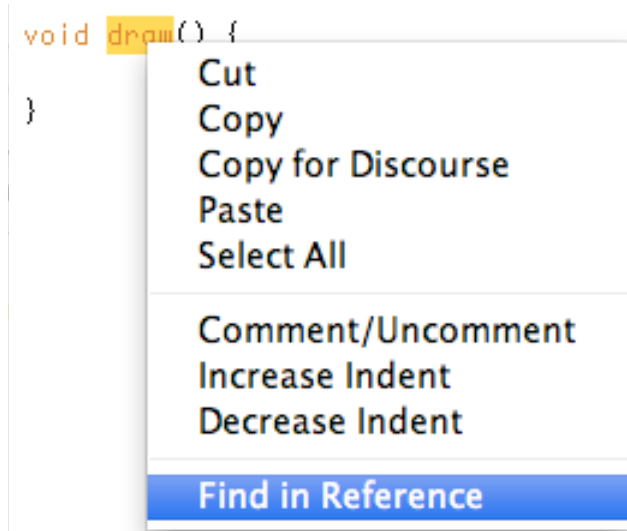
**Draw loop header** (points to `void draw() {`)

**Curly brackets** (points to the opening and closing braces of the `draw()` function)

**Code goes here between the curly brackets and repeats forever** (points to the space between the curly braces)

## Reference in Processing:

One very convenient way to access Processing's Help Reference is to highlight a function or a word used in your code, right click and select Find in Reference. In the example below the word "draw" has been highlighted



This will open Processing's Help File directly to the function or word you have highlighted. This will not work with variable names or anything that is not a "reserved word" already used in the Processing language. Here is what the Help Reference looks like, it is an invaluable tool for both the beginner and the expert:



Let's get started with Processing:



III. Let's draw a dot, a line and some shapes.

To do this first we will need a window to draw the dot in. Type the following code inside your setup function:

```
void setup (){  
  size (700, 500);  
}
```

Go ahead and press Run, you should get a window that is 700 pixels wide by 500 pixels tall. Don't forget the parentheses or semicolon, everything is important in a Processing sketch. The parentheses indicate values inside of them and the semicolon tells the computer to execute the line. If you don't type a line indicating size Processing will create a 100 X 100 window for you.

Now let's put a dot (one single pixel) in your window by typing the following inside of the draw loop, but replace **x** with a number smaller than 700 and **y** with a number smaller than 500:

```
void draw(){  
  point (x, y);  
}
```

Again, don't forget the comma or semicolon, they're important and you will get an error message if you forget them. Try different values for x and y to gain an understanding of how Processing uses coordinates and the x and y. Values of 0 are at the absolute left side of the window for x and at the very top for y. What happens if you choose a number that is larger than your window is wide or tall? Does Processing wrap around to the other side of the window or does the point disappear? Try it out for yourself.



IV. Next we are going to draw a single line in the window.

Delete (or comment out using // before the text) the line that created your dot. Replace it with the following line inside of the draw loop. (We left the draw loop syntax out from this point on, you're a big kid now.)

```
line(x1, y1, x2, y2);
```

Replace **x1** and **y1** with the coordinates where your line will start and **x2** and **y2** with the coordinates where your line will end. Pretty simply, huh? Delete this line or comment it out before moving on to the next step.



V. Now let's draw three shapes, a rectangle, a circle and a triangle. Type the lines below and substitute numbers for the red variables, each of which are explained below.

```
triangle(x1, y1, x2, y2, x3, y3);
rect(x, y, width, height);
ellipse(x, y, width, height);
```

Triangle is pretty straightforward and you should understand what **x1**, **y2** and the rest are at this point. (The three different points of the triangle.)

Rectangle's **x** and **y** coordinates specify the upper left corner of the rectangle, with **width** and **height** being the height and width of the rectangle. There are a lot of functions that use width and height. These values are measured in pixels, just like lines and windows, so to find out where the shape ends just add the starting position (x for width and y for height) to the width or height values.

Ellipse is almost the same as rectangle except the **x** and **y** coordinates indicate the center of the ellipse you are creating.

The order in which you write the code for the shapes will effect which shape is on top of the others with the first shape, in this case the triangle, being on the bottom, or behind all the other shapes.

Play around with different values for these shapes until you feel comfortable with them. If you're already comfortable with coding these shapes either use a bunch of these shapes to create a more complicated image, or try some of the more advanced shapes on the handout provided.



VI. Leave the code for your three shapes in the Sketch and play with the outline of the shapes.

Changing the outline:

Every shape in processing has a stroke (outline) and a fill (internal color). To change the width of the outline of your shape use the line below, replace **p** with the number of pixels you want for the width of your outline. You will need to write this code before the line that creates the shape you want to affect.

```
strokeWeight(p);
```



This line will change the outline for all the shapes you draw after it, so make sure to set it back to 1 after you are done drawing shapes with thick lines, otherwise everything will have thick outlines.

To create shapes with no outline use the line below. You will need to write this code before the line that creates the shape you want to affect. Don't forget to turn the outline back on once you are done by writing `strokeWeight(p)`; after the line that creates your shape with no outline.

```
noStroke();
```

To get your outline back use the line of code below.

```
stroke();
```

To create a shape with just an outline and no “fill” color use the line below. You will need to write this code before the line that creates the shape you want to affect.

```
noFill();
```

To create shapes with a color after this line has been used you must use the `fill(color)`; line which is outlined in the next section. You will need to write this code before the line that creates the shape you want to affect.



## VII. Adding Color to Your Sketch.

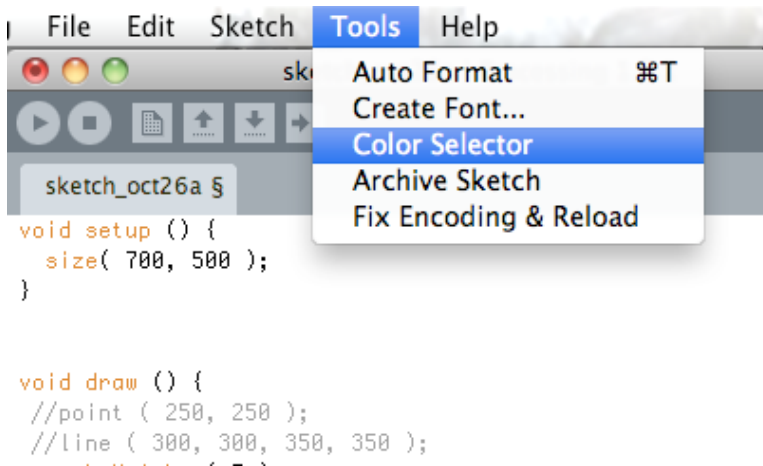
Next let's give each of the shapes a color. To do this type the following line inserted just before a line of code that creates a shape.

```
fill(red, blue, green);
```

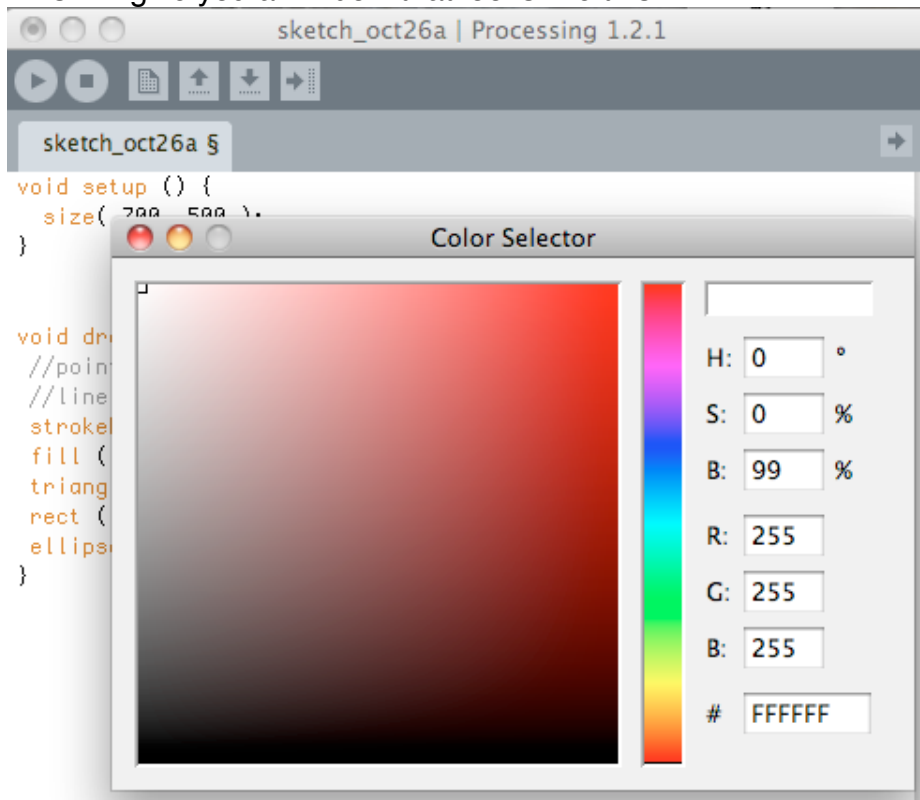
Or to control the color of your outline:

```
stroke(red, blue, green);
```

Each of the variables, red, blue and green, are replaced with a number ranging from 0 to 255. The lower number the number the less of that color is present in the one color this line represents. An example of three RGB values that you would use to create purple are red: 195, green: 3 and blue: 255. For help figuring out these three numbers in Processing select Color Selector in the Tools menu.



This will give you a window that looks like this:



You can see each of the three RGB (R, G, B) colors that when added together create the color you have selected. In this example each of the colors are maxed out at 255 giving us the color white.

Another way to use the fill function is with a hexadecimal value. In the color selector the hexadecimal value is indicated by the # character. If you don't know what a hexadecimal value is check the image above to see how to get a

hexadecimal value using the color selector. In this example the hexadecimal value is #FFFFFF which signifies white. We aren't going to go into Hexadecimal now, but it is an important programming concept that you will bump into soon or later if you are programming. The hexadecimal value for purple is displayed as #C303FF. To use a hexadecimal value in the fill function type the following and replace `hexadecimalValue` with your hexadecimal value.

```
fill(#hexadecimalValue);
```

Or to control the color of your outline:

```
stroke(#hexadecimalValue);
```

To give your shapes or stroke transparency, simply add the variable `transparency` to the RGB values already in the fill function like below. The transparency, or "alpha" value is represented by a number between 0 and 255 where 0 indicates the object is completely see through and 255 represents an object that is completely solid.

```
fill(red, blue, green, transparency);  
or  
fill(#hexadecimalValue, transparency);
```

Or to control the color of your outline:

```
stroke(red, blue, green, transparency);  
or  
stroke(#hexadecimalValue, transparency);
```

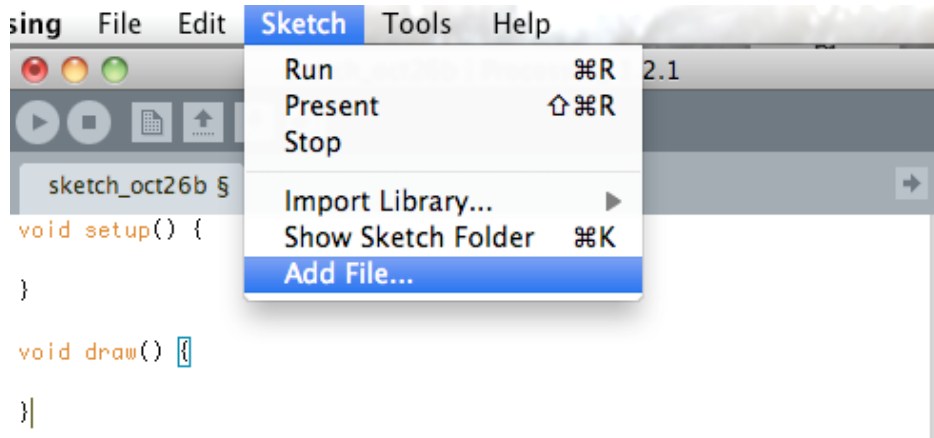
Colors that have transparency and are laid over each other in the draw function will mix to create a new color. Give each of your three shapes a different color before continuing with the activity.



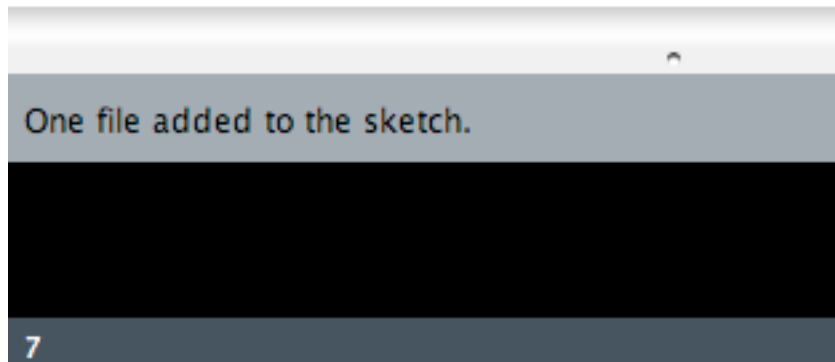
## IIx. Importing Images.

Next let's draw a background image we have on our computer in our Processing window. To do this first you need to follow three steps to actually get the image imported into your Processing Sketch.

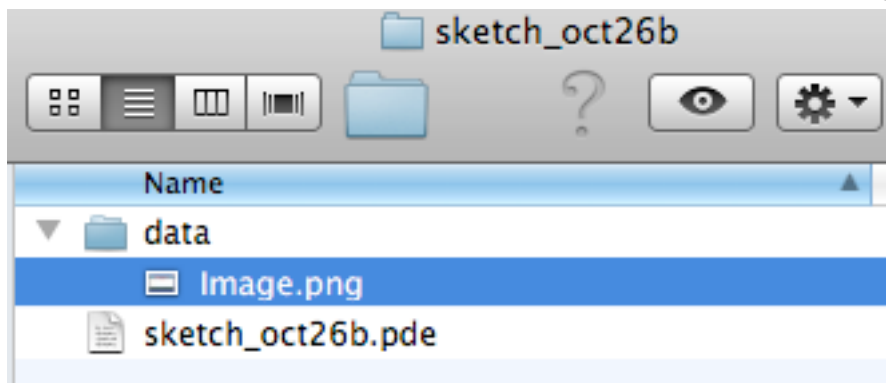
First select Add File from the Sketch menu and find the file you wish to add to your Sketch.



Processing plays well with the following image file types: .jpg, .png and .gif, it also likes raster and vector images (these last two are math based image types instead of pixel image types, meaning they will never create distortion when scaled). If everything goes well you should see a message that reads “One file added to the sketch” below the area where you write code.



This means Processing has automatically created a folder called Data inside your Sketch folder with the image file inside of the Data folder. You can also drag and drop images and fonts (you will also need to import fonts to use them) into this folder as an alternative way to complete this first step of adding files. After you have completed this step check your Sketch folder and make sure Processing has created a data folder.



Next you will create an object to store your image in. You will create this object to store your image inside your setup function. An object is like a variable, but it's for a Class, don't worry about Classes right now, we'll go into that later. The object name can be anything you like (I named mine **img**), as long as it makes sense to you. But the Class, **PImage**, needs to be the same whenever you are using an imported image. Make sure you type this line above the setup function because you want to be able to use this variable anywhere in your sketch. If you wrote it inside the setup function or the draw function you would only be able to use it inside those functions. We'll name the object **img01** because you will be creating more image objects later, which we will call **img02**, **img03**, etc... To create your image object, type the line above your setup function.

```
PImage img01;
```

Now you have an instance of an object from the PImage Class(this stands for Processing Image) to store the information that is your actual image in. Next you have to assign the data file to the object you created. To do this you will assign your **image data file** to the **img01** object (or whatever you named it if you didn't use **img01**) using the **loadImage** function as typed below. Make sure to include quotation marks around the **image data file** name as well as the file extension (.jpg, .png, and .gif). The file name is also case sensitive so make sure you type it exactly as it appears in the folder. Place this code in the setup function as well.

```
img01 = loadImage ("Image.png");
```

Okay, you're almost ready to actually draw the image in your Processing Window. To do this you just need to type one more line using the **image function**. The **image function** has three parameters or variables; the image object you just created, x position and y position. Type the following line and substitute the x and y positions where you wish your image to display for the variables **x** and **y**. Remember that these variables indicate where the upper left corner of your image will start. Place this code inside your loop function.

```
image (img01, x, y);
```

To control the size of the image simply add two more variables for width and height. If you leave width and height out of the function, or just give them a value of 0 the image will draw itself at whatever size the original picture is in the data file you imported.

Type the following line inside the draw loop and substitute the size you wish your image to display in pixels for the variables `width` and `height` (or you can enter 0 or leave them out altogether to display at the original image's size). The variables `width` and `height` are global variables that are always equal to the width and height of your canvas window. Don't make Processing draw the image larger than the original file unless you are okay with image distortion (or you are using a vector file format like .svg).

```
image(img01, x, y, width, height);
```

To see this background image press the Run button.

Make sure your background image covers the whole window that Processing is drawing without covering any of your existing shapes. To do this, make sure that you type the image function command (the line above) before any of your shape lines. Once you have done that add at least three images to the Data file folder in your Processing Sketch.

Now load these other images and assign them to PImage objects. The code to do this is shown below in case you forgot.

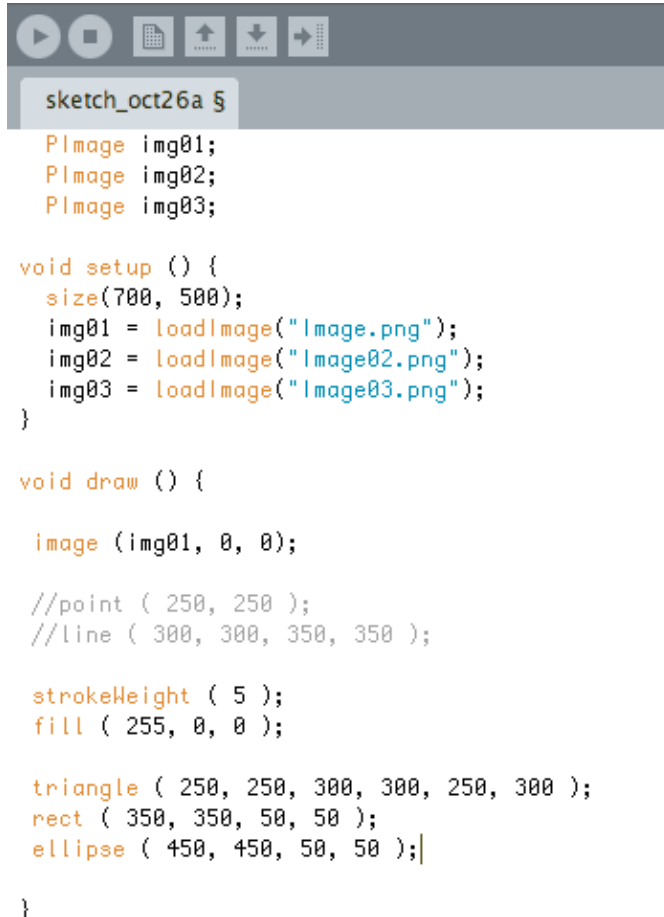
Make sure you write this code above your setup function.

```
PImage img02;  
PImage img03;
```

And make sure you write this code inside your setup function.

```
img02 = loadImage ("Image02.png");  
img03 = loadImage ("Image03.png");
```

Here is what your code might look like at this point:



```

PImage img01;
PImage img02;
PImage img03;

void setup () {
  size(700, 500);
  img01 = loadImage("Image.png");
  img02 = loadImage("Image02.png");
  img03 = loadImage("Image03.png");
}

void draw () {

  image (img01, 0, 0);

  //point ( 250, 250 );
  //line ( 300, 300, 350, 350 );

  strokeHeight ( 5 );
  fill ( 255, 0, 0 );

  triangle ( 250, 250, 300, 300, 250, 300 );
  rect ( 350, 350, 50, 50 );
  ellipse ( 450, 450, 50, 50 );
}

```



## IX. Creating a Function.

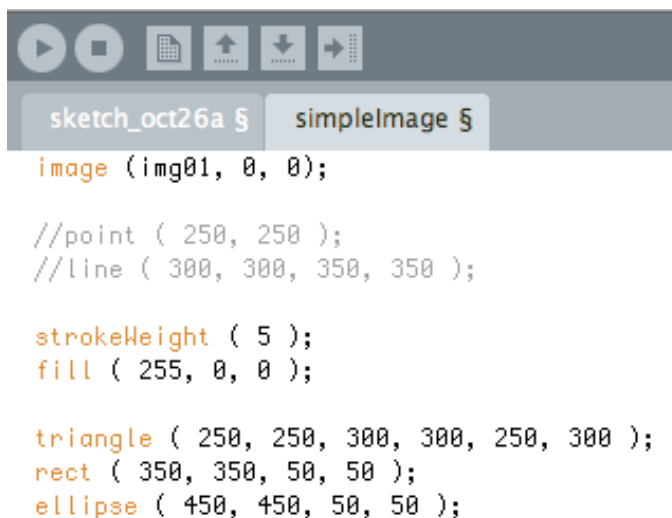
Now we will take the sketch as it stands so far and turn it into what is called a Function. A Function is a way to package a bunch of code that pertains to one action(s) or image(s) separate from your draw loop, it can be as complicated or as simple as you like. This way the programmer can use one line of code to call the function instead of typing all the code over and over again every time he or she wants to use the code in the Function.

The first thing you will do is to create a new tab in which you can place the Function. To do this simply click on the tab button and select New Tab.



Then you will have to create a name for this new tab, or File. Choose something that makes sense, but don't worry, you can always rename it later. I named mine simpleImage. Finally click OK, this will create a new tab with a name that you can write code in.

Now cut and paste all the code **inside the brackets of your draw loop** into the new tab. It should look something like this:



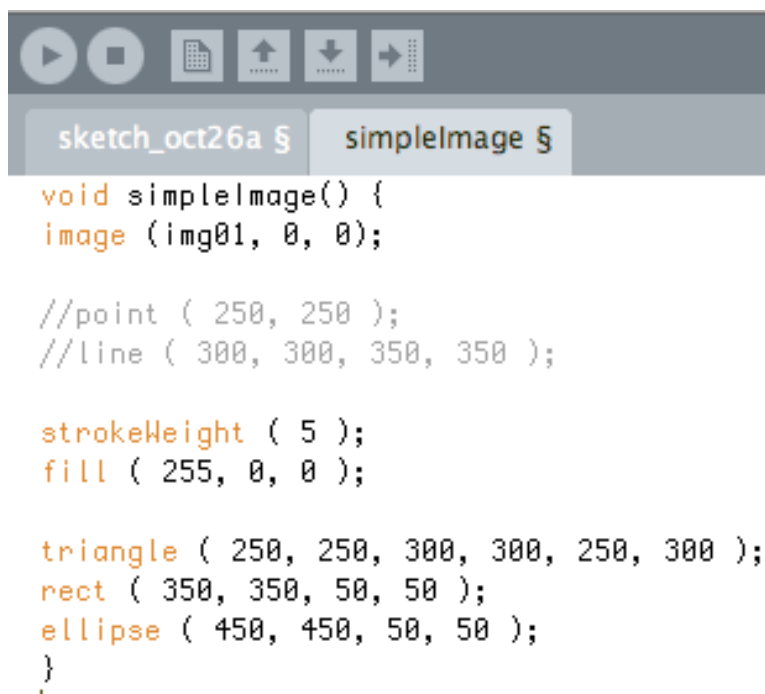
Next we will need to create a Function Header that holds the code you just copy and pasted. A Function Header is the code at the beginning of the Function as well as the parenthesis that go around the code. To create your Function Header create a blank line at the beginning of the code by pressing enter and then type the following:

```
void functionName ( ) {
```



**functionName** is the **name of the Function** and can be whatever you choose, but it should be relatively short and make sense to you. Programmers don't capitalize Function names to make it easy to remember that it is a Function and not a Class. (We will go into Classes later.) **Void** simply means that the function will not send any information back to the Draw Loop when it is "called". As you gain more experience with programming you may use this aspect of functions to create some more complicated code. Finish off the Function by placing a "closed" curly bracket at the end of the code like this: }.

Here is what my Function looks like at this point:



```
void simpleImage() {
  image (img01, 0, 0);

  //point ( 250, 250 );
  //line ( 300, 300, 350, 350 );

  strokeWeight ( 5 );
  fill ( 255, 0, 0 );

  triangle ( 250, 250, 300, 300, 250, 300 );
  rect ( 350, 350, 50, 50 );
  ellipse ( 450, 450, 50, 50 );
}
```

I choose to call it simpleImage. Feel free to change the name of your function from functionName to simpleImage or whatever makes sense to you.

At this point if you press "play" nothing will happen because you have not called the simpleImage function yet. The code is there, inside the function, but it doesn't actually get executed yet.

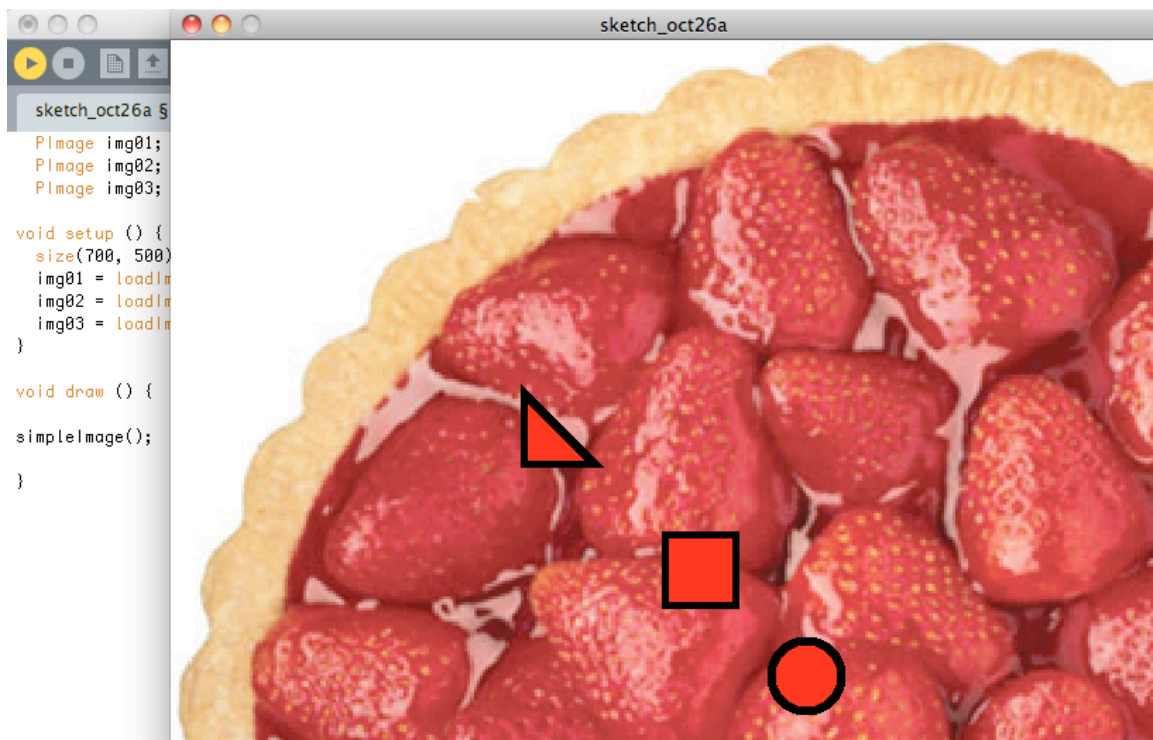
If all went well you should be able to execute your code by "calling" your Function from inside your draw loop. To do this simply create a blank line inside the draw loop curly brackets and type the name of your Function followed by two parenthesis and a semicolon.

## Intro to Processing with the Danger Shield

Here's what it looks like when I call my simpleImage Function from the draw loop:

```
void draw () {  
  
    simpleImage();  
  
}
```

Here's what happens when I press the play button in Processing:

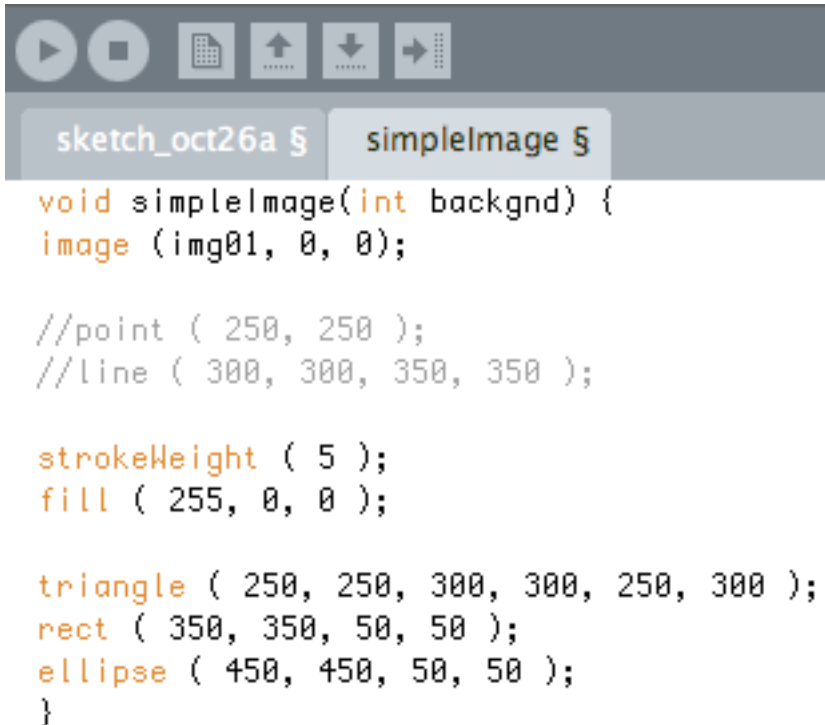




## X. Passing the Function Arguments.

By now you may be asking what the point of creating a Function was if the code acts exactly the same as it did before when it was inside the draw loop. One of the great things about creating Functions is that it makes your draw loop **a lot** less cluttered. Where you used to have lines and lines and lines of code, you now have one nice tidy line that acts the same as all the previous lines of code. Another great thing is that you can pass the Function “arguments” (this is a fancy way to say variable values and other pieces of code) from the draw loop that effect aspects of the Function. It sounds complicated, but we will start with a couple of simple examples and soon you will understand that there are literally an infinite number of ways this can be useful.

First we need to “declare” arguments in your Function Header. To do this, simply type the argument type (int, char, boolean, Class name, etc...) followed by the argument name (which can be anything you like as long as it makes sense to you) inside the empty parentheses after the Function name. If you wish to pass more than one argument to the Function you will need to declare each argument inside the parentheses, separating them with a comma. For now we will only worry about arguments that are variables. I have added an integer variable called `backgnd` to my `simpleImage` Function Header, now it looks like this:



```

void simpleImage(int backgnd) {
  image (img01, 0, 0);

  //point ( 250, 250 );
  //line ( 300, 300, 350, 350 );

  strokeWeight ( 5 );
  fill ( 255, 0, 0 );

  triangle ( 250, 250, 300, 300, 250, 300 );
  rect ( 350, 350, 50, 50 );
  ellipse ( 450, 450, 50, 50 );
}

```

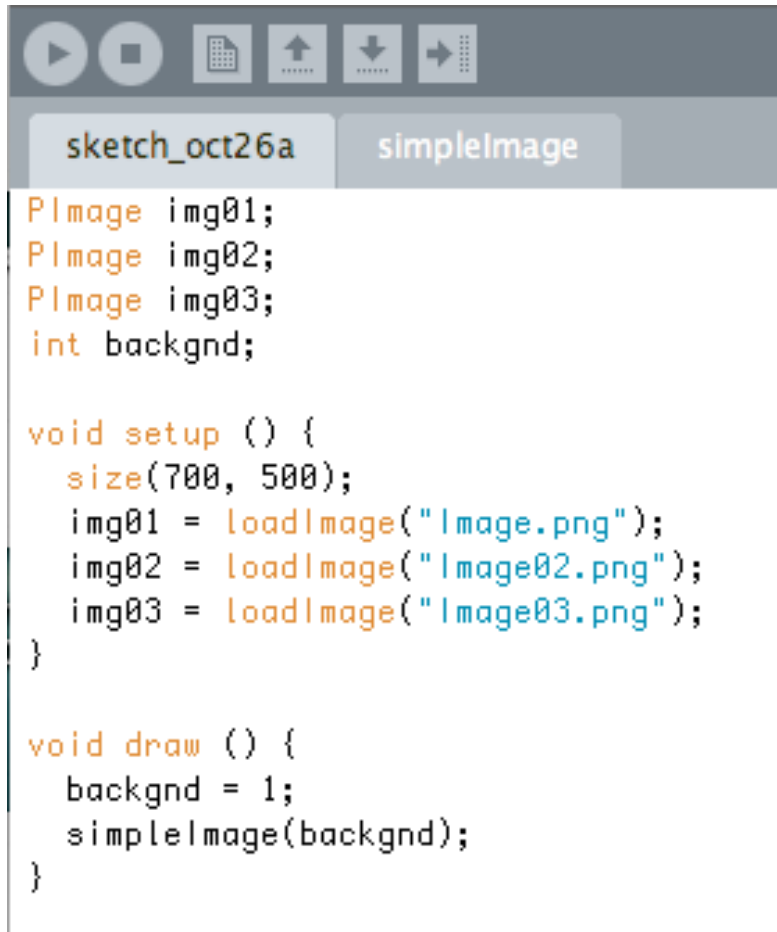
Add an integer variable to your Function Header that you will use to control the background. You can call it background or you can call it something else, just make sure you enter the variable type integer before the name of the variable.

In order to use this **variable** in the Processing sketch we need to declare it. To do this type the following line of code above your draw loop. This way the variable is a global variable, meaning you can use it anywhere.

```
int backgnd;
```

Now we need to make sure that every time we call the Function from the draw loop we supply the Function with variable values. This is called “passing” the Function variables. To do this simply call the Function the same as before by typing the Function name, followed by a set of parentheses and, of course, don’t forget the semicolon. Only this time type your variable value (or values) inside of the parentheses. Make sure that the variable value corresponds to the variable type. This means that if you declared an integer variable in the Function Header you must pass the Function a number between 2,147,483,647 and - 2,147,483,648 with no decimals. If you have questions about variable types see the section on variables included with this handout. One last thing to note about passing variables to a Function is that if you pass multiple variables you will need to make sure that you pass them in the same order you declared them. Also make sure to separate your variables with commas. Here is what it looks like

when I declare the variable, set it equal to the number one and then pass that variable to the `simpleImage` function:



```

PImage img01;
PImage img02;
PImage img03;
int backgnd;

void setup () {
  size(700, 500);
  img01 = loadImage("Image.png");
  img02 = loadImage("Image02.png");
  img03 = loadImage("Image03.png");
}

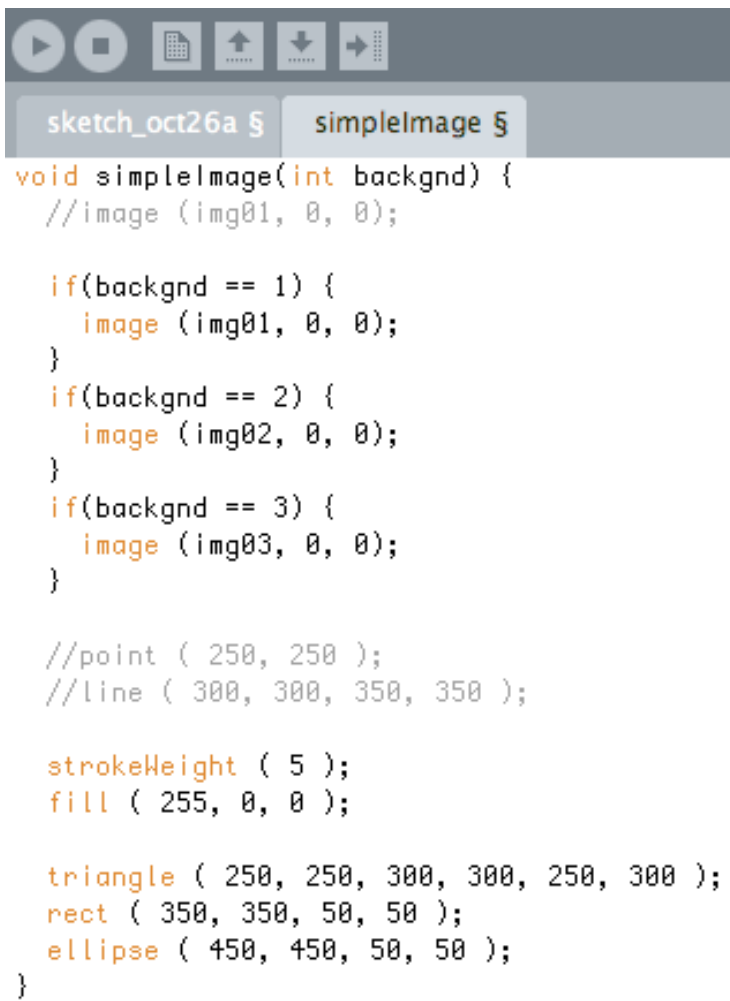
void draw () {
  backgnd = 1;
  simpleImage(backgnd);
}

```

Pass your Function an integer with a value between (and including) one and three. If you choose to place more background images in your Data Folder you can pass your Function an integer that goes as high as the number of images you placed in the Data Folder. If you feel comfortable passing your Function more than one variable go ahead and plug more variables into the Function Instance.

Now that you have done all this work to declare and pass variables it's time to use one of the variables you have passed to your `simpleImage` Function in the code. We're going to create a bunch of `If` Statements that change the background image of your sketch depending on what variable you pass your `simpleImage` Function. You will type the following code inside of the `simpleImage` Function. To do this first comment out the line that uses the `image` function to draw the background image by typing two `//` marks on the same line as the `image` function, just before it. Next type at least three `If` Statements

that use the **image function** to draw different background images depending on what the value is of the variable you are passing your Function. If you're not sure how an If Statement works check the handout on If Statements. There are a couple different ways to do this if you are familiar with coding, but here is what the code should look like in the simplest form:



```
void simpleImage(int backgnd) {
  //image (img01, 0, 0);

  if(backgnd == 1) {
    image (img01, 0, 0);
  }
  if(backgnd == 2) {
    image (img02, 0, 0);
  }
  if(backgnd == 3) {
    image (img03, 0, 0);
  }

  //point ( 250, 250 );
  //line ( 300, 300, 350, 350 );

  strokeWeight ( 5 );
  fill ( 255, 0, 0 );

  triangle ( 250, 250, 300, 300, 250, 300 );
  rect ( 350, 350, 50, 50 );
  ellipse ( 450, 450, 50, 50 );
}
```

If you're familiar with coding go ahead and write a case/switch statement to replace these if statements, otherwise skip to the next page. To do this declare a variable to check in your case/switch statement. In this case you will most likely use the variable that already exists, "backgnd". Here is an example of the code you would use; variables that may differ have been highlighted in red. Simply write more case statements for variable values other than 1. Case/switch statements will help you save on memory and are easier to keep track of than a bunch of if statements.

```
switch (backgnd) {
case: 1
```

```
//place code here
break;
}
```

Try passing your Function different integer variables from your Draw Loop to make sure your If Statements are working. For example if I want to see my second image as the background I would set my backgnd variable equal to two. Pretty sweet, huh? If your backgrounds are similar to each other, but slightly different, you could create an animation for your background with a little more code. For example, use pictures of a setting sun in sequential order to make it look like nighttime is arriving. (Hint: Look up For Loops and use one to change the integer variable you pass to your Function in the Draw Loop to make this happen)



## XI. Sending Information from The Danger Shield.

Ok. Now you are ready to load code onto your Danger Shield hardware and use it to send values to your Processing Sketch. To do this you will need to have a functioning Arduino set up and understand Serial Communication basics. If you haven't played around with this before take a minute to familiarize yourself with Arduino and the concept of Serial Communication. Don't worry about learning a new language; Arduino works almost the same as Processing, so everything we have talked about so far applies to the Arduino Environment as well, although there are a few differences. These differences are because Processing is based on the programming language Java while Arduino is based on the programming language C.

For the code you will load onto your Danger Shield (or any other Atmega device) there are four basic parts you will need to send variable values from inside your main Loop function (this function is basically like your Draw Loop, but it is on the Arduino instead of in the Processing Sketch).

The first thing we need in order to send information from the Danger Shield to the Processing Sketch takes place in the setup function and simply begins Serial Communication between the Arduino and your desktop computer. That line of code looks like this (remember to put it in the setup function!):

```
Serial.begin(9600);
```

The number 9600 is the "Baud Rate", or speed at which the computers talk to each other. Without this number it is possible for one computer to talk faster than the other computer is listening. This will result in gibberish being communicated, because, although the gibberish would contain the correct information, the

receiving computer would not be able to decipher it at the correct speed. Imagine a sound being slowed down to the point where you can't understand it, although the sound may be someone speaking a sentence you can understand, if the sound is slowed down too much you won't be able to hear the actual sentence in order to understand it.

Second, you need to create a variable that is equal to a reading of the sensors on your hardware. To do this you will use the `digitalRead( )`; and `analogRead( )`; functions. Here is an example of creating a variable and assigning it a value you get from reading one of your Danger Shield sensors (the first slider), although like in the example you will need to create and assign the `SLIDER1` variable before hand:

```
val = analogRead(SLIDER1); //read slider1
```

Or you might write the code like this, because the slider is on pin 2 of the Arduino:

```
val = analogRead(2); //read Arduino pin # 2 which slider1 is attached to
```

This code does two things; it creates a variable named `val` and assigns it the value that the `analogRead( )`; function gets from `slider1` (which is attached to Arduino pin # 2).

Next you need to send the information via Serial Communication to your Processing Sketch on your computer. To do this you will be using the `Serial.print( )`; function. Here is an example of a line of code that sends a variable value to your computer from the Arduino Hardware:

```
Serial.print(val, DEC);
```

This is a similar concept to "passing" variables. In this case you are using `Serial.print( )`; to pass the variable `val`, and `DEC` indicates that your variable is a base ten number (a normal, everyday number). You need the `DEC` portion of this line because the computer thinks in hexadecimal (base 16) and otherwise you'll see some letters when your Processing Sketch receives the values instead of plain old numbers.

Next you will need a way to separate the many different values you will be sending from the Danger Shield to your Processing Sketch. You didn't think you'd just be sending one value did you? To separate your values you need to send a character that is not a number (because otherwise the computer not be able to



tell this difference between the numbers). In the Processing the Danger Shield example code we have created a variable to store this character called `DELIMITER`, in this code `DELIMITER` is a comma, but it could be any non-number character. If you decide not to use a variable like `DELIMITER` make sure that you place single quotes around your non-number character like the second example of code below. Here are two examples of a line of Arduino code that send the `DELIMITER` value to your Processing Sketch:

```
Serial.print(DELIMITER);
```



Or you might write this line of code like this:

```
Serial.print(',', '');
```


Notice how similar this line of code is to the line of code you used to send your variable `val` to the Processing Sketch.

So, that's almost everything you'll need for the code on your Danger Shield (or any other Arduino or Atmega hardware) in order to send information. Just repeat the last three steps of reading a sensor into a variable, using `Serial.print()`; to send the value and then sending a delimiter value. The only other thing to remember is that the very last variable you send should use `Serial.println()`: to send the variable value instead of `Serial.print()`;. It's just a two letter difference, but it is important because the `Serial.println()`; sends a carriage return (this is the same as pressing the enter or return button) after the value. Without the carriage return the Processing Sketch would not be able to tell when you are done sending the set of data.

Here is what all of the code I am loading onto my Arduino looks like:  
(I had to cut it up into sections because it's too big to see all at once.)

Pin definitions, first part of code	Setup function, second part of code
 <pre>ProcessingDanger06 § // Pin definitions #define SLIDER3 0 #define SLIDER2 1 #define SLIDER1 2 #define KNOCK 5 #define CAP1 2 #define CAP1 9 #define BUTTON1 10 #define BUTTON2 11 #define BUTTON3 12 #define LED1 5 #define LED2 6 #define BUZZER 3 #define TEMP 4 #define LIGHT 3 #define LATCH 7 #define CLOCK 8 #define DATA 4  char START_BYTE = '*'; char DELIMITER = ','; char END_BYTE = '#'; int val;</pre>	 <pre>ProcessingDanger06 § void setup() {    pinMode(BUZZER, OUTPUT);   pinMode(LED1, OUTPUT);   pinMode(LED2, OUTPUT);   digitalWrite(LED1, HIGH);   digitalWrite(LED2, HIGH);   pinMode(LATCH, OUTPUT);   pinMode(CLOCK, OUTPUT);   pinMode(DATA, OUTPUT);   digitalWrite(10, HIGH);   digitalWrite(11, HIGH);   digitalWrite(12, HIGH);   Serial.begin(9600);   establishContact(); }</pre>

Loop function, third portion of code



```
ProcessingDanger06 §
void loop(){

  Serial.print(START_BYTE, BYTE);

  val = analogRead(SLIDER1); //read slider1
  Serial.print(val, DEC);
  Serial.print(DELIMITER);

  val = digitalRead(BUTTON1); //read button1
  Serial.print(val, DEC);
  Serial.print(DELIMITER);

  val = analogRead(SLIDER2); //read slider2
  Serial.print(val, DEC);
  Serial.print(DELIMITER);

  val = digitalRead(BUTTON2); //read button2
  Serial.print(val, DEC);
  Serial.print(DELIMITER);

  val = analogRead(SLIDER3); //read slider3

  Serial.print(val, DEC);
  Serial.print(DELIMITER);
}
```

Second portion of the loop function, fourth portion of code.

```
val = digitalRead(BUTTON3); //read button3
Serial.print(val, DEC);
Serial.print(DELIMITER);

val = analogRead(TEMP);
Serial.print(val, DEC);
Serial.print(DELIMITER);

val = analogRead(LIGHT);

Serial.print(val, DEC);
Serial.print(DELIMITER);
Serial.println(END_BYTE, BYTE);
}
```

The establishContact function, called from setup function, fifth portion of code.

```
void establishContact() {
  while (Serial.available() <= 0) {
    Serial.println("Hello"); // send a starting message
    delay(300);
  }
}
```

You can just copy this if you like, but make sure you go back through this handout and understand what the four main portions are. The most important is the establishContact () function, which is described below.

The Serial.available () function causes the Arduino to wait until it has received a byte worth of information and then Serial.available () is set equal to whatever the byte is. The Arduino checks to see if there is a byte in Serial.available (), which indicates that it has made contact with the computer and Processing sketch. If there is not any Serial information available the Arduino sends the message "Hello" to the Processing sketch and continues to wait for a byte of information in Serial.available (). Once the Arduino has received a byte of Serial information it is ready to start spitting information over the Serial line to the Processing sketch.



## XII. Receiving Information from the Danger Shield.

Let's go back to the Processing Sketch now and type in the code we will need for the Processing Sketch to receive the information your Arduino is sending.

The very first thing you will need to do is import the Library that Processing uses to work with Serial Communication. Libraries are portions of code that other programmers have written. This means that you can use functions that exist in the Library in your own code by first importing the Library, creating an Instance of a Class found inside the library and then typing the Functions found inside that Class using something called "Dot notation". Dot notation just means that first you will type the Library name, followed by a period, and then the name of the function in the Library you are using. Don't worry if this seems complicated, we'll go through it step by step and tell you when you are using Dot notation.

Here's the code you use to import the Processing Serial Library:

```
import processing.serial.*;
```

You can also import libraries through the Sketch tab in Processing and selecting "import library" and then selecting the library you wish to import. Importing a library through the menu will create a line of code similar to the line above at the very top of your code. When you import libraries they should always be imported at the very top of your sketch.

The code used to import a library is an example of Dot notation; it simply means that you are importing something from the Serial Library of your Processing Library. (It is possible to have Libraries inside of Libraries; this is kind of like the Kiddie section Library inside your local physical Library.) The \* in this line of code means that the computer should import all the Classes in the Serial Library. Don't forget both of the periods in this line of code; they're both important!

Another way to import a Library is to go to the Sketch Menu and select Import Library and select the Library you wish to import from the list.

After you have imported your Library create a Boolean variable called "firstContact". We will use this variable to check to see if Serial communication has started yet. This way the Arduino knows to keep checking for beginning communication if it doesn't hear anything from the Processing sketch. Here is the code you will use to create that variable:

```
boolean firstContact = false;
```

Next we need to create an “object” which is like an Instance of a Class. One Serial Class Object to store the USB port that your computer and Danger Shield are talking over and an array of integers to store the data you get sent from the Danger Shield. Arrays are just a list of variables arranged inside of square parentheses; see the handout on arrays if you’re unsure about them. The important thing to know is that you can store and access a bunch of variables of the same data type in one array.

Here is the code you will use to create this object and array, type these above the setup function:

```
Serial usbPort;  
int [ ] sensorData;
```

`usbPort` and `sensorData` are just the **variable and Serial Object names** I chose, they really can be anything you like, they just need to make sense. Right now this array and object are empty, we will give the object some values inside the setup function next. This is called “Initializing” the object. Before we do that let’s talk about Classes for a second.

Classes are a way to organize code similar to Functions. The difference is that Classes can have functions inside of them. Right now this might seem like a pain in the butt, however it is very useful once you start doing more complicated things. Some key things to remember is that while you pass Functions variables, you will pass Classes arguments and that Classes are a little bigger (or maybe just more important) than Functions because they can have Functions inside of them. Arguments can be more complicated than variables, but are a similar concept.

These next two lines of code are very important because they establish where the Serial Communication is coming from that your Processing Sketch receives. Without these lines your Processing Sketch is kind of like a person waiting for a letter without a mailbox to check, even if the letter has been sent and delivered the person has no hope of actually getting it! The USB port is the digital mailbox where all the information is delivered. Below is the code you will use to define your USB port Class by creating a new Instance of the Class with some variables. Type it inside your setup function.

```
usbPort = new Serial (this, Serial.list( ) [0], 9600);  
usbPort.bufferUntil ( '\n' );
```

The first line **creates a new Serial object** it's very similar to creating an Instance of a Function. (Remember Instances? If not go back to the Creating a Function section.) The values inside the parentheses do three different things; **this** makes sure that the code in this line references this particular object of the Serial Class, the command `Serial.list( ) [0]` lists the available Serial Port on the computer in an array, the number **0** is the first value in the array, which is the Serial Port Processing uses, and **9600** is the **baud rate** that your computers at which your computers are communicating. Make sure that this **baud rate** matches the **baud rate** in your Arduino code you are using with the Danger Shield.

The second line of code is much simpler. It uses Dot Notation to access the `bufferUntil` Function in the Serial Class. This Function buffers data coming in through the Serial Port until it receives the **value in the parentheses**. In this case the Processing Sketch is waiting for `\n` which is what the Arduino sends for a carriage return (Which is what comes at the end of `Serial.println( ) ;`), the reason it is inside single quotes is because the variable type is char. Once the `bufferUntil` Function receives this value it will call a `serialEvent` Function that you will type elsewhere. These qualities of the `bufferUntil` Function are already defined inside of the Serial Class. (See how useful Classes are already?)

All right, now we're ready to define our `serialEvent` Function. To define the Function let's start with just the Function header. This is the portion of the code that the Processing Sketch looks for whenever there is data incoming on the Serial Port. It looks like this:

```
void serialEvent ( Serial usbPort ) {
//code goes here
}
```

By now you may recognize that you are passing this `serialEvent` Function the **object usbPort** and that it's **type** is a **Class called Serial**. This Function is important because it is called every time there is data available in the Serial buffer. This means that this Function happens every single time that the computer running your Processing Sketch receives some Serial Communication from the **usbPort object** (Which is your **Serial Communication hardware Port**). So this is where you will define what the computer does with the information coming over the **Serial Communication Port**.

The code you will write inside this particular `serialEvent` Function will listen to the information coming over the USB port and save it into a String variable (a String variable is a fancy way to say a bunch of characters) and then divide the String into sections and save those sections in your array that you created at the

beginning of this section. If it sounds complicated, don't worry, we'll walk through each of the pieces of code. The code you will type inside of this `serialEvent` Function (this means inside of the `serialEvent` curly brackets) looks like this:

```
String usbString = usbPort.readStringUntil ('\n');
if (usbString != null) {
    usbString = trim(usbString);
    int sensors[ ] = int (split (usbString, ','));
    for (int sensorNum = 0; sensorNum < sensors.length;
sensorNum++) {
        println( "Sensor " + sensorNum + ": " + sensors
[sensorNum] );
    }
}
```

The **first line** creates a String variable called `usbString` and fills it with the data coming over the `usbPort` object (which was created using the Serial Class) using the Serial Class's Function `readStringUntil`. See the Dot Notation? Remember Dot Notation? The `readStringUntil` Function enters character variables into the `usbString` until it reads the characters inside the parentheses. You may remember that `\n` is what the Arduino will send after any `Serial.println( );` function and this indicates that the last sensor value has been sent to the Processing Sketch.

The **second line (and matching closed curly bracket)** is an `If` Statement that simply tells the computer to continue with the code inside the curly brackets if the `usbString` is not empty. In this code the characters `!=` mean not equal to and the word "null" means empty.

The **third line** uses the `trim` function to remove the carriage return (created by the `Serial.println( );` Function) and any "whitespace" characters from the beginning and end of the String in `usbString`. Whitespace characters include space, carriage return, tab and the Unicode "nbsp" character. Unicode "nbsp" stands for "non-break space" and is a character that acts like a space, but doesn't take up as much physical space.

The **fourth line** uses the `split` Function to divide up the `usbString` into integer variables and enters these variable values into the `sensors` array. The `split` Function takes two arguments; the name of the String to be split and the character that indicates the place where the split should occur. Remember back when you were sending `Serial.print( );` Functions from your Danger Shield Arduino? The second argument in the `split` Function should match the character you sent as a delimiter.

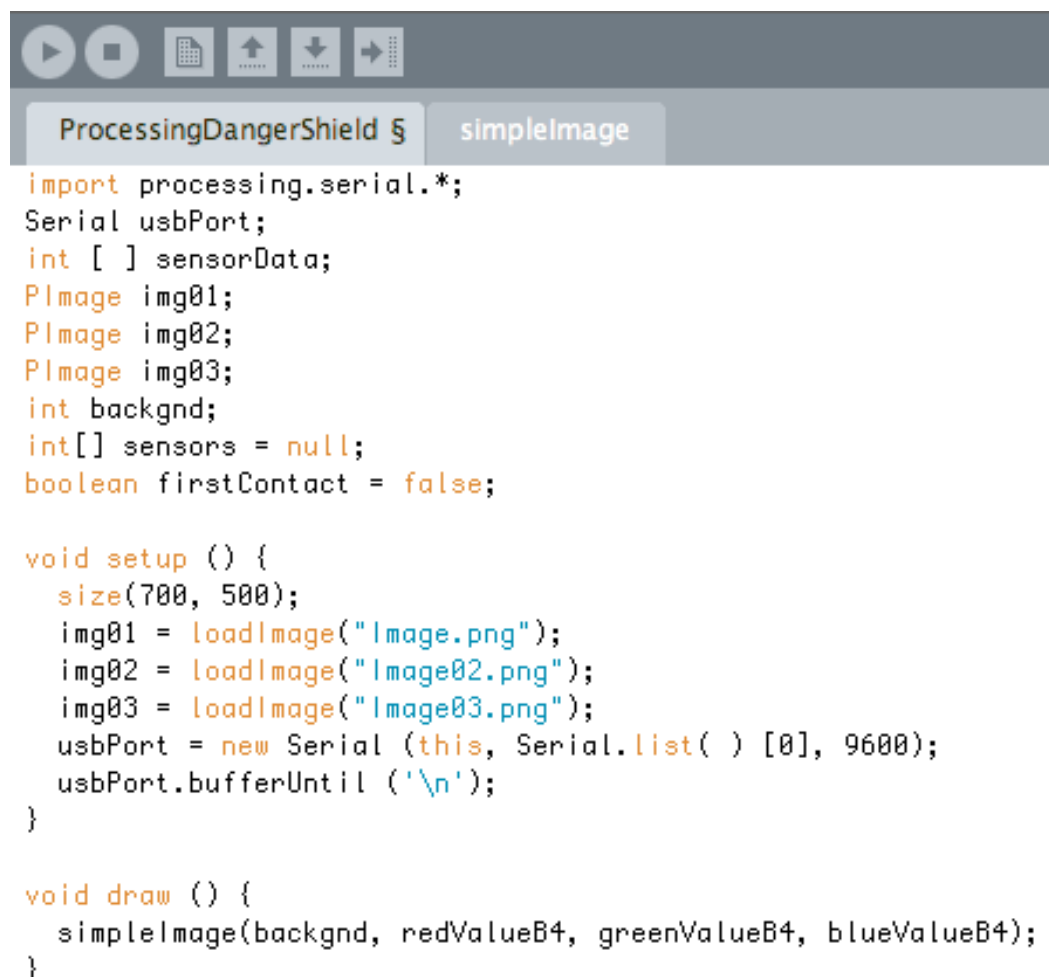


The **fifth line (and matching closed curly bracket)** is a simple for loop that cycles through the various values in the sensors array. If you need a refresher on how for loops work see the attached handout.

The **sixth and final line** is a `println()` function which displays the values of the sensors array that the for loop is cycling through.

The **fifth** and **sixth lines** are not strictly necessary in order to get information from the Danger Shield hardware, but it is nice to be able to see the values coming over the USB port and very useful for troubleshooting.

Here is what my code looks like so far in my Processing sketch now that I have added the Serial Communication portions:



```
import processing.serial.*;
Serial usbPort;
int [ ] sensorData;
PImage img01;
PImage img02;
PImage img03;
int backgnd;
int[] sensors = null;
boolean firstContact = false;

void setup () {
  size(700, 500);
  img01 = loadImage("Image.png");
  img02 = loadImage("Image02.png");
  img03 = loadImage("Image03.png");
  usbPort = new Serial (this, Serial.list() [0], 9600);
  usbPort.bufferUntil ('\n');
}

void draw () {
  simpleImage(backgnd, redValueB4, greenValueB4, blueValueB4);
}
```

```
void serialEvent ( Serial usbPort ) {
  String usbString = usbPort.readStringUntil ('\n');
  if (usbString != null) {
    usbString = trim(usbString);
    if (firstContact == false) {
      if (usbString.equals("Hello")) {
        usbPort.clear();
        firstContact = true;
        usbPort.write('A');
        println("contact");
      }
    }
    else {
      int sensors[ ] = int(split(usbString, ','));
      for (int sensorNum = 0; sensorNum < sensors.length; sensorNum++) {
      }
      usbPort.write("A");
    }
  }
}
```



### XIII. Integrating the Information from the Danger Shield into your Function.

Now that there is data coming into your Processing Sketch from your Arduino you will need to plug that data into the Classes and Functions of your Processing Sketch. Currently the variables you got from the Danger Shield are listed in an array named sensors. To make things simpler, first you can assign these array values to integer variables. If you are comfortable with arrays feel free to replace these variables with sensors array values. If you do this remember that arrays start with the number 0, so sensors[1] is actually the second value in the sensors array.

To assign the first value in the sensors array (which is coming from Slider # 1 on the Arduino and the Danger Shield) to a variable simply type the following line of code:

```
int sensorValue01 = sensors[0];
```

By now you probably understand exactly what this line does, but in case you don't we'll go over it again.

The first portion of this line, **int**, is necessary to create an **integer** variable. In this case the **integer** variable is named **sensorValue01**, but you can name it

anything that makes sense to you. The `single equals sign` is used to actually assign a value to the `sensorValue01` integer variable. `sensors[ ]` is the array that you are getting information from and the number `0` is the place inside the array that the computer is looking for the variable value.

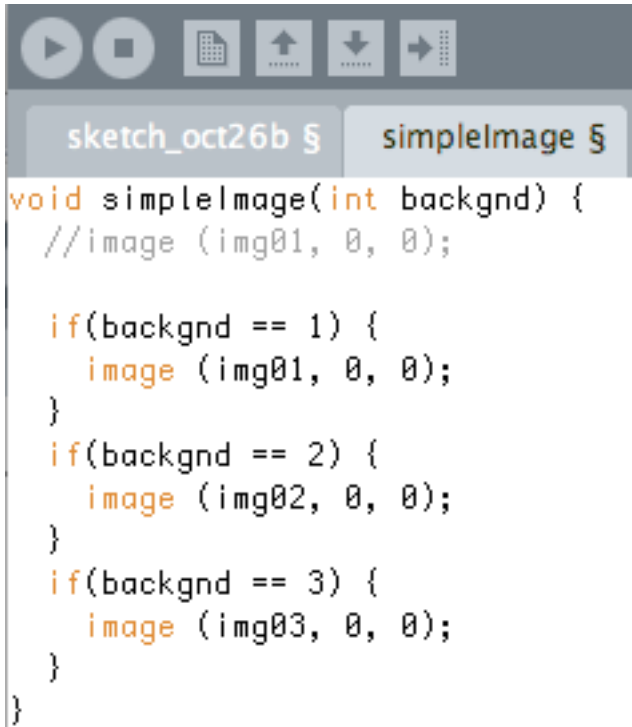
Now that you have one of the variables from the slider on the Arduino and Danger Shield in your Processing sketch you can use that variable to change something in your Processing `draw()` loop.

Do this for all the sensor values.

Let's use the value you will get from the first slider on your Arduino DangerShield sketch to change your Processing sketch's background. To do this you will need to write the line written below. We will explain this code in the next section. Insert this code in place of the line of code you are currently using to set the variable `backgnd` to either one, two or three inside of the draw loop.

```
void draw () {  
  backgnd = int (map ( sensorValue01, 1, 1023, 1, 3 ));  
  simpleImage(backgnd);  
}
```

You will also need to change your `simpleImage` function so that it knows how to use the various values that you are sending it in `backgnd`. One simpler way to do that is to use a bunch of if statements. Here is a simple version of that code:



```

void simpleImage(int backgnd) {
  //image (img01, 0, 0);

  if(backgnd == 1) {
    image (img01, 0, 0);
  }
  if(backgnd == 2) {
    image (img02, 0, 0);
  }
  if(backgnd == 3) {
    image (img03, 0, 0);
  }
}

```

Now that you've seen how easy and fun that was next you will map the value in `sensorValue01` to the range that Processing uses to represent RGB colors. The value in `sensorValue01` goes up to 1023, while the value you need to represent the color red only goes up to 255. Luckily for you there is a function designed specifically to do this. Below is an example and explanation.

Write this code above your setup function so you will be able to use this variable anywhere in your sketch. Set it equal to zero so in case you don't get any values from your DangerShield it won't break your sketch.

```
float redValue = 0;
```

Write this code inside your draw loop.

```
redValue = map ( sensorValue01, 0, 1023, 0, 255 );
```

`redValue` is the **name of the variable** we are creating and its **type** is **float**. `redValue` needs to be a float variable because when you are changing the original value in `sensorValue01` you may wind up with a decimal, which will not play nicely with an integer variable. Now that the computer has created a **variable** use the single equals sign to assign the value that comes out of map to that variable.

The two sets of numbers that follow the **variable** inside the parenthesis are the two sets of numbers that define the how small the variable can get and how large the variable can get. The first two numbers, `0` and `1023`, define the range that the variable starts in and the second set of numbers, `0` and `255`, define the range we are squishing the variable down to. For example, if the **original variable's** value is 512 (halfway between 0 and 1023), then the **map function** will make `redValue` equal to approximately halfway between 0 and 255, or around 128. If you're still not sure about the **map function** feel free to look it up in the Processing reference.

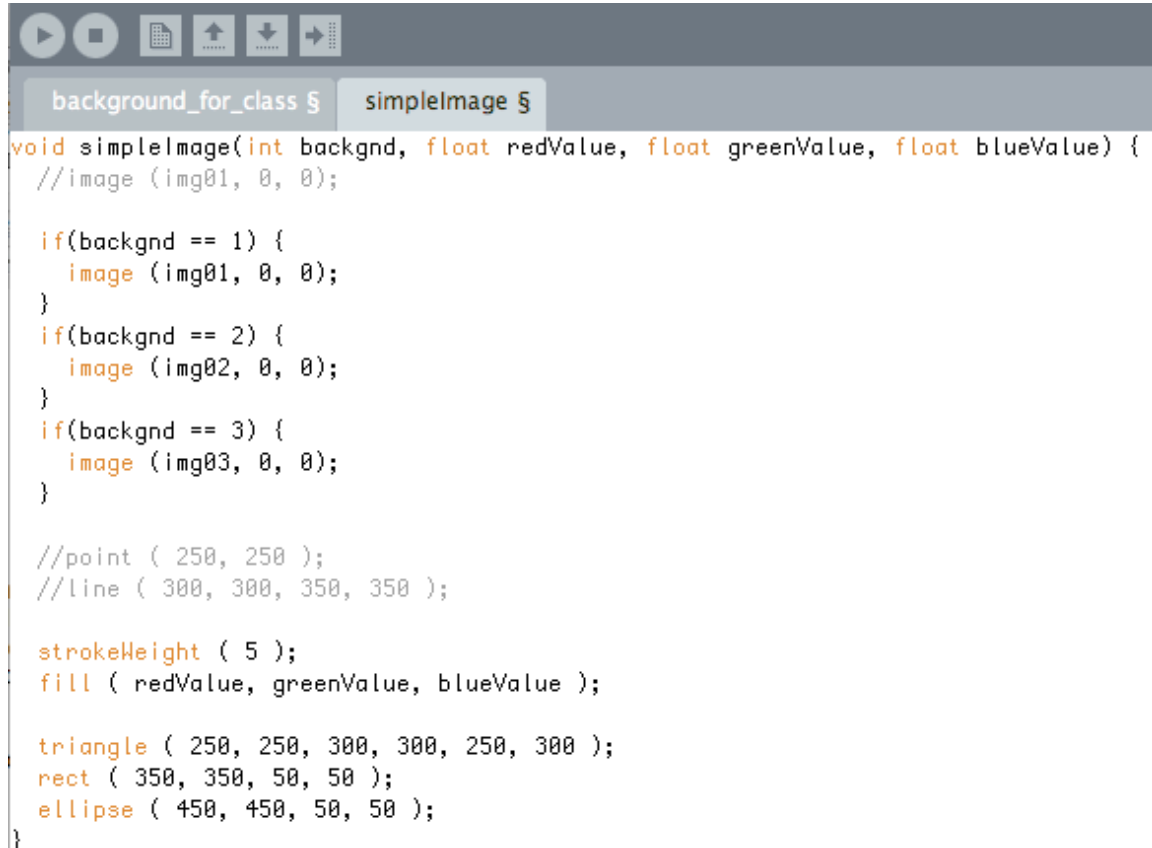
Go back and assign the rest of the values inside your sensors array ( `sensors [ ]` ) to variables with names that make sense to you. If you are continuing to use the examples provided in this text assign `sensors[ 1]` to `sensorsValue02`. Remember that even though the place in the array is #1, it is the second value because place #0 also holds a value.

Then write the map function code from above three times, except instead of creating more variables called `redValue`, create variables called `greenValue` and `blueValue`, and assign slider # 2 (or `sensorValue03`) to the `greenValue` variable and slider # 3 (or `sensorValue05`) to the `blueValue` variable.

Next you need to make sure you pass these variables to your `simpleImage` function. You've already done this once by passing the `backgnd` variable to the `simpleImage` function so you should be familiar with this concept. Make sure you pass all three color variables. If you're running into trouble make sure that you have the variable declared when you call the `simpleImage` function from your draw loop as well as in the `simpleImage` function header.

Plug the `redValue`, `greenValue` and `blueValue` variables into your existing `fill( )` function that occurs before your shape functions in the `simpleImage`

function. Now you can control the color of these shapes with your sliders. So now my `simpleImage` function looks like this:



```

background_for_class §  simpleImage §
void simpleImage(int backgnd, float redValue, float greenValue, float blueValue) {
  //image (img01, 0, 0);

  if(backgnd == 1) {
    image (img01, 0, 0);
  }
  if(backgnd == 2) {
    image (img02, 0, 0);
  }
  if(backgnd == 3) {
    image (img03, 0, 0);
  }

  //point ( 250, 250 );
  //line ( 300, 300, 350, 350 );

  strokeWeight ( 5 );
  fill ( redValue, greenValue, blueValue );

  triangle ( 250, 250, 300, 300, 250, 300 );
  rect ( 350, 350, 50, 50 );
  ellipse ( 450, 450, 50, 50 );
}

```

As you move the sliders up and down you should see a change in the color of your shapes.



## XIV. Using the Buttons on The DangerShield to Set a Variable

But what if you don't want the color of your shape to change until you push the button on your DangerShield below the slider? That means you need to create a second set of variables (I'm going to call them `redValueB4`, `greenValueB4`, etc...) to keep track of the colors before you press the button and a way to set your first set of variables (`redValue`) once the button has been pushed.

First things first we have to declare our new variables on the global scale. This means we declare them at the very top of the sketch outside of any other functions. I'll bet you didn't even know you had been declaring global variables. To declare these variables type the following lines of code:

```
int redValueB4;
int blueValueB4;
int greenValueB4;
```

Next you will need to change the line of code that takes the readings out of the array that you get out of your `SerialEvent` function so that the readings get stored in your first variable. You may not remember this line of code so I have rewritten one of these lines below:

So change this line:

```
redValue = map ( sensorValue01, 0, 1023, 0, 255 );
```

To this line:

```
redValueB4 = map ( sensorValue01, 0, 1023, 0, 255 );
```

Do this for the other two colors as well.

Now all you need to do is create a couple `if` statements that set the `redValue` variable equal to `redValueB4` if the button below the slider that represents `redValue` has been pushed.

Remember that the buttons read HIGH (or 1) when they have not been pushed and **LOW (or 0)** when they have been **pushed** because they use pull up resistors. Also remember that the button values come right after the slider values in your `SensorValue` array.

Here is an example of the code you will use to make this **button** pushing action happen, write it in your draw loop before you call your `simpleImage` function:

```
if ( sensorValue02 == 0 ) {
    redValue = redValueB4;
}
```

Do these two steps for the other two colors as well.

Pretty easy, huh?

If you would like to be able to tell what color the sensor represents before you push it, simply create a square for each slider that is filled with the color of the variable `redValueB4` (or `greenValueB4` or `blueValueB4`). Put these shapes in a corner of your window and make them fairly small so they don't take up space. By now you should be able to do this on your own, but if you need help go back and look at the shape and fill sections at the beginning of this handout.



## XV. Changing the Background Using the Photoresistor

As a last step before you start playing around with Processing and the DangerShield yourself let's use the photoresistor to change your background. If you chose backgrounds that are the same place, but different times of day, this would be a great way to use the light in your surroundings to change the background of your sketch to match.

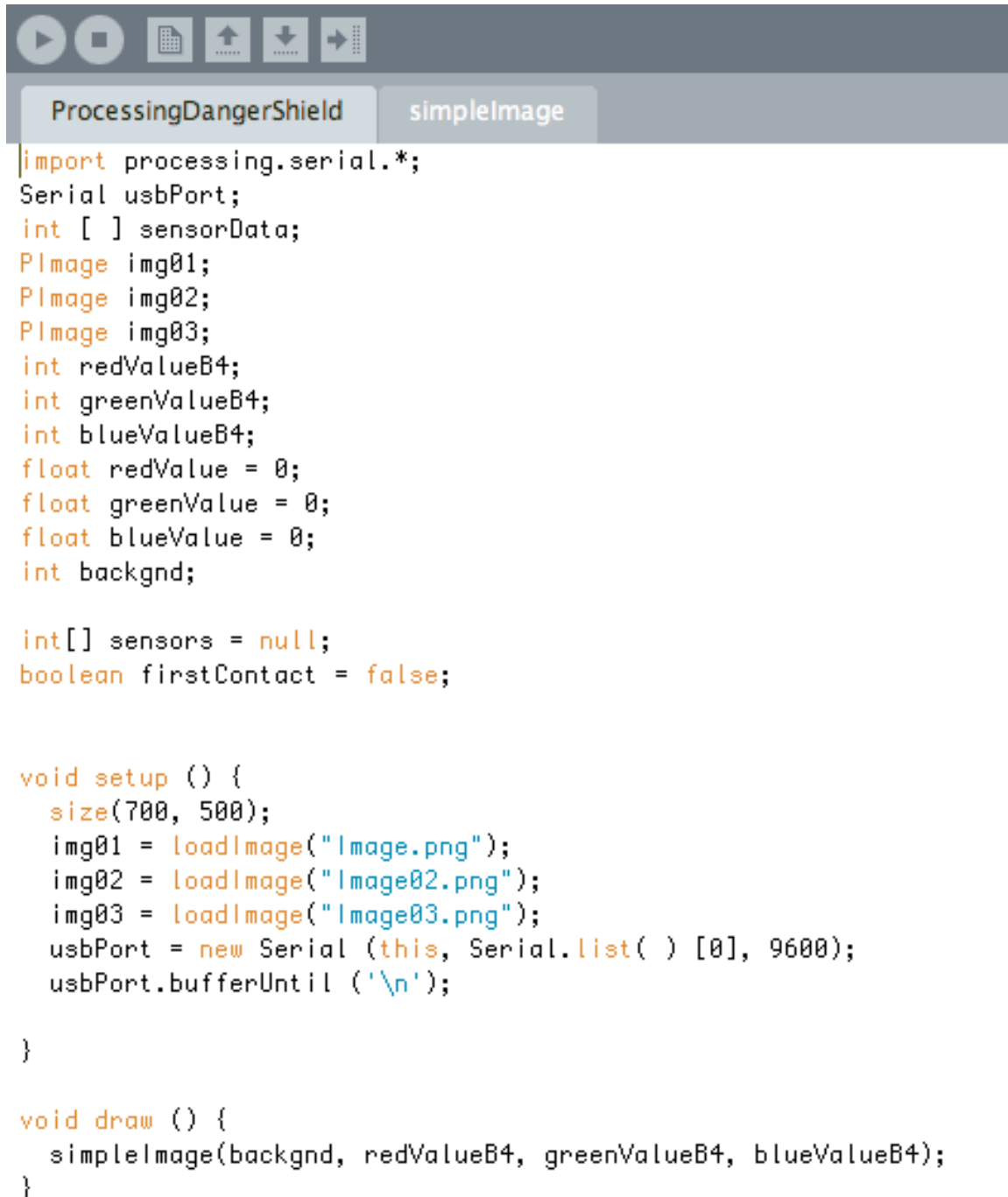
The photoresistor is the eighth value in your `sensorValue` array, so you will access it using the variable `sensorValue07`. Depending on how many different background images you have you will need to map the value of `sensorValue07` to the number of background images you have and assign that variable to the `backgnd` variable. Put this code in your draw loop before you call your `simpleImage` function and it will change the `backgnd` variable that interacts with your `simpleImage` function. Below is an example of the code you might use to do that. I have mapped the `sensorValue07` to the numbers one through three simply because that is how many background images I have. You may have many more, so feel free to change the code.

```
backgnd = int ( map(sensorValue07, 0, 1023, 1, 3) );
```

The `int ( )` function, which we used before when mapping other variables, simply changes the value that the `map` function creates into an integer. This is necessary because otherwise your `simpleImage` function, which receives an integer variable, would not know what to do if the `map` function returned a decimal.



Now my code looks like this (I have broken it up into multiple parts so it will fit):



```

import processing.serial.*;
Serial usbPort;
int [ ] sensorData;
PImage img01;
PImage img02;
PImage img03;
int redValueB4;
int greenValueB4;
int blueValueB4;
float redValue = 0;
float greenValue = 0;
float blueValue = 0;
int backgnd;

int[] sensors = null;
boolean firstContact = false;

void setup () {
  size(700, 500);
  img01 = loadImage("Image.png");
  img02 = loadImage("Image02.png");
  img03 = loadImage("Image03.png");
  usbPort = new Serial (this, Serial.list( ) [0], 9600);
  usbPort.bufferUntil ('\n');
}

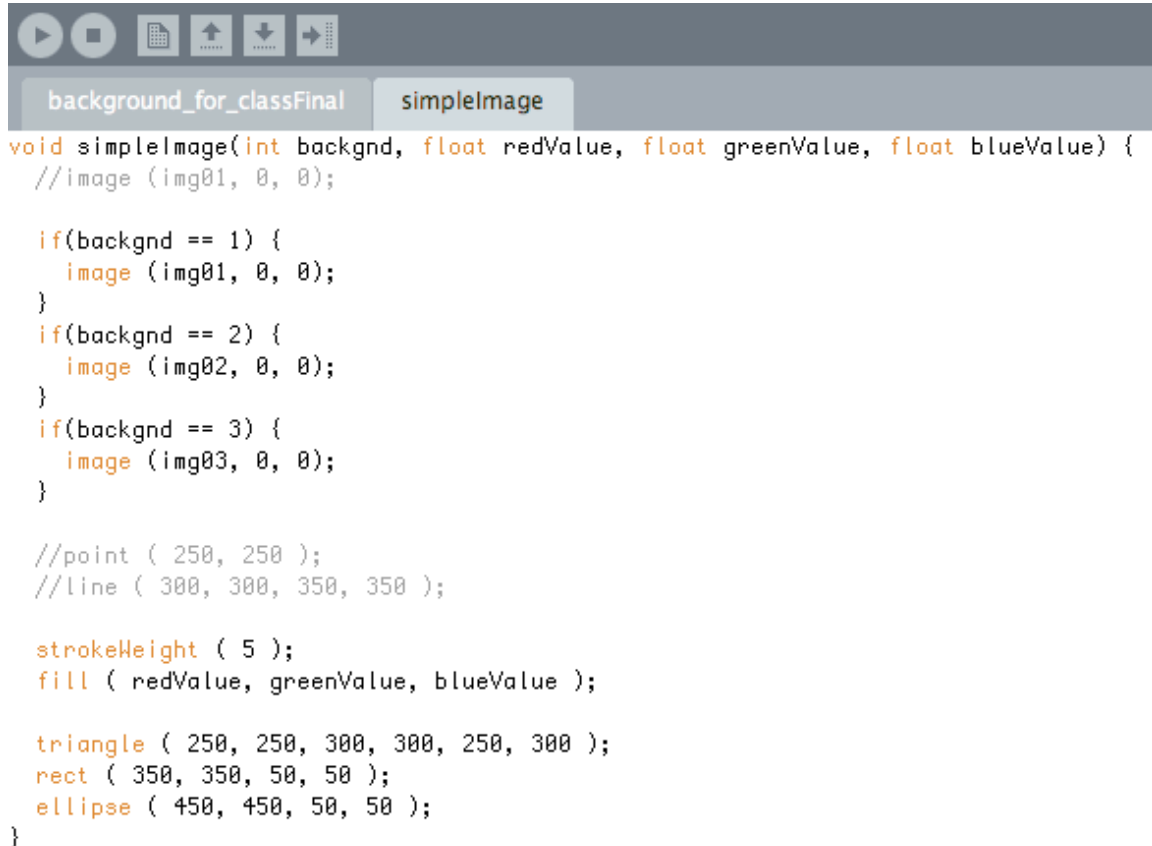
void draw () {
  simpleImage(backgnd, redValueB4, greenValueB4, blueValueB4);
}
  
```

```

void serialEvent ( Serial usbPort ) {
  String usbString = usbPort.readStringUntil ('\n');
  if (usbString != null) {
    usbString = trim(usbString);
    if (firstContact == false) {
      if (usbString.equals("Hello")) {
        usbPort.clear();
        firstContact = true;
        usbPort.write('A');
        println("contact");
      }
    }
    else {
      int sensors[ ] = int(split(usbString, ','));
      for (int sensorNum = 0; sensorNum < sensors.length; sensorNum++) {
      }
      int slider1 = sensors[0];
      int button1 = sensors[1];
      int slider2 = sensors[2];
      int button2 = sensors[3];
      int slider3 = sensors[4];
      int button3 = sensors[5];
      int photoCell = sensors[7];
      backgnd = int (map ( photoCell, 1, 950, 1, 3 ));
      redValueB4 = int (map (slider1, 0, 1023, 0, 255));
      greenValueB4 = int (map (slider2, 0, 1023, 0, 255));
      blueValueB4 = int (map (slider3, 0, 1023, 0, 255));
      if (button1 == 0) {
        redValue = redValueB4;
      }
      if (button2 == 0) {
        greenValue = greenValueB4;
      }
      if (button3 == 0) {
        blueValue = blueValueB4;
      }
      usbPort.write("A");
    }
  }
}

```

And my simpleImage function looks like this:



```
void simpleImage(int backgnd, float redValue, float greenValue, float blueValue) {
  //image (img01, 0, 0);

  if(backgnd == 1) {
    image (img01, 0, 0);
  }
  if(backgnd == 2) {
    image (img02, 0, 0);
  }
  if(backgnd == 3) {
    image (img03, 0, 0);
  }

  //point ( 250, 250 );
  //line ( 300, 300, 350, 350 );

  strokeWeight ( 5 );
  fill ( redValue, greenValue, blueValue );

  triangle ( 250, 250, 300, 300, 250, 300 );
  rect ( 350, 350, 50, 50 );
  ellipse ( 450, 450, 50, 50 );
}
```



## XVI. The Next Step

Um... wow. That's a lot to process. But by now you should have the basic skills necessary to get sensor information off of Arduino hardware into your Processing sketches on your computer. You can do anything with this. On the DangerShield we still haven't used the temperature sensor, the capacitive touch sensor or the seven segment display. Feel free to play around with those, or learn more about Processing on your own and create a game, or an animation, or a data logging sensor system, or an amazing invention!