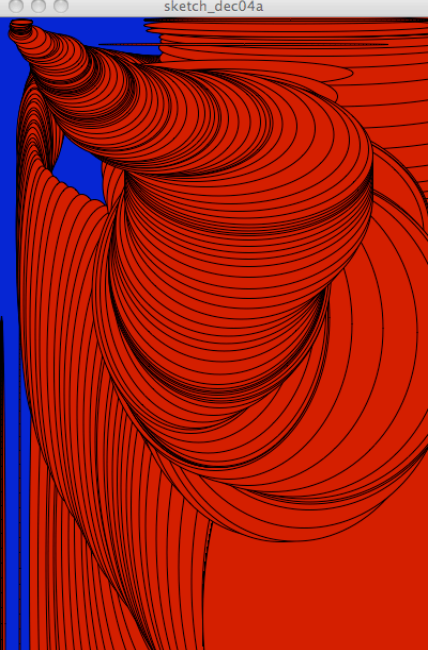



The Junior Woodchuck Manual of Processing Programming for Android Devices

<i>The Image</i>	<i>The Code</i>
	<pre>void setup() { size(400, 600); background(0, 0 , 200); // blue fill(200, 0, 0); //red } void draw() { ellipse(mouseX, mouseY, pmouseX, pmouseY); }</pre>

Chapter 3

Forests and Trees ???

The Image	The Code
	<pre> void setup() { size(400, 600); smooth(); background(200, 200 , 0); fill(0, 200, 0); } void draw() { fill(random(256), random(256), random(256)); ellipse(mouseX, mouseY, pmouseX, pmouseY); } </pre>

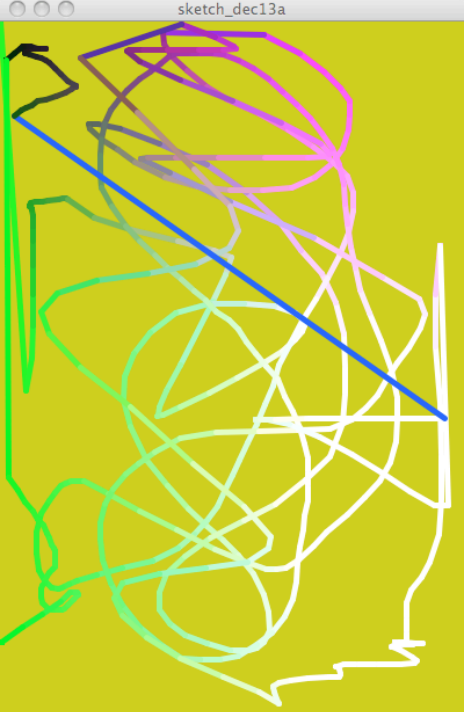
You can, if you wish, spend the next few weeks making drawings like you did in the first class. However, you will miss out on at least two of the big advantages of programming with Processing – animation and games.

If you want to add animation to your program and use that to make a computer game, more work is necessary. The first thing we need to do is to add some “structure” to your code so it is easier for you to write, debug, and modify to make it better.

Let’s get started.

Section 1

Functions Give Us a Map...

The Image	The Code
	<pre>void setup() { size(400, 600); smooth(); strokeWeight(5); background(200, 200 , 0); fill(0, 200, 0); } void draw() { stroke(mouseX, mouseY, pmouseX); line(mouseX, mouseY, pmouseX, pmouseY); }</pre>

We begin with functions. You used Processing's functions in our first class to draw your images. The "deal" with Processing was that Processing wrote the functions and you used them.

For a few special functions we can turn the deal around. We can write functions and Processing will use them.

The first one of these special functions is the `setup()` function. The parentheses are required and they must be empty.

The name must be spelled exactly as shown including the case of the letters. There cannot be any capital or upper case letter.

If we write the function, `setup()` in our code, Processing will look for it, find it, and run it `setup()`!

You may be asking, "what does `setup()` do?"

The answer is, "anything we tell it to do" which is great. Usually we use `setup()` to "set the stage" of our program.

Let's look at the `setup()` function that is at the beginning of this section on page 3:

```
void setup( )
{
  size( 400, 600 );
  smooth( );
  strokeWeight( 5 );
  background( 200, 200 , 0 );
  fill( 0, 200, 0 );
}
```

This `setup()` function tells Processing to do the following:

1. make the window 400 pixels wide and 600 pixels high
2. turn on smoothing so the jagged lines do not look so bad
3. make the lines 5 pixels wide
4. make the background a sorta' ugly yellow
5. make the fill color a shade of green
- 6.

Since this program has a `setup()` function, Processing will look for it, find it, and do these for tasks first. Processing does them only one time.

You can use the `setup()` function to do whatever you want to do to get your program ready to run.

Look very carefully at the way the `setup()` function is written.

The rules for having a `setup()` function that Processing will use are:

- it must be named `setup()`
- it must have an opening `{` after the parentheses
- it must have a closing `}` at the end of the function

If you follow these rules, you are set.

There is another function that Processing will use if we write it. This is the `draw()` function.

Again, The parentheses are required and they must be empty.

The rules are the same for writing this function:

- it must be named `draw()`
- it must have an opening `{` after the parentheses
- it must have a closing `}` at the end of the function

What we write inside the `{ }` is what we want Processing to draw.

You may be thinking, "Wait a minute – we drew stuff last time without all of this function stuff – what's the deal?"

The "deal" is that Processing is using or running, or as most programmer say, executing our `draw()` function 60 times a second.

That's right, it is doing whatever we tell it to do 60 times every second.

This is our "animation engine" – we use this to move things in the window. The `draw()` function drives the animation.

But before we can explain how this works, we need to add one more tool to our toolbox – variables.

Here is the code that `draw()` function that drew the image on page one of this chapter:


```
void draw( )  
{  
  stroke( mouseX, mouseY, pmouseX );  
  line( mouseX, mouseY,  
        pmouseX, pmouseY );  
}
```

We need to look at the stuff in the parentheses.

Read on...

Section 2

Variables Give us Directions for Using the Map...

<i>The Image</i>	<i>The Code</i>
	<pre>void setup() { size(400, 600); smooth(); strokeWeight(5); background(200, 200 , 0); fill(0, 200, 0); } void draw() { stroke(mouseX, mouseY, pmouseX); line(mouseX, mouseY, pmouseX, pmouseY); }</pre>

Variables are used by programming languages to store data that changes from time to time as our program is executed. Processing programs often use four different kinds or types of variables. Right now, we will use only one – the type **float**. A float variable can have a fractional or decimal part such as 3.14159 or similar value.

Just like functions where Processing can write them and we can use them, Processing has variables that it “owns” and changes and we can use them.

In the code in the section header on the previous page we can see four of these in use:

```
void draw( )  
{  
    stroke( mouseX, mouseY, pmouseX );  
    line( mouseX, mouseY,  
          pmouseX, pmouseY );  
}
```

We need to understand one very important fact about how Processing executes or runs our draw() function. Each execution of the draw() function generates what we call a new frame. The frame is the drawing window where our figures are displayed. Each new frame is a transparent window that is displayed on top of or over the previous frames. This is the way cartoons were made before computers. Unless we color the background of the new frame, we can see the old frames that are under the new frame.

Each execution of draw() assigns new, current values to these variables:

The variables mouseX and mouseY store the pixel location of the mouse for the current frame.

The variables pmouseX and pmouseY store the pixel location of the mouse in the previous frame.

This means that the color of the line which is set by the stroke() function will have these values in the new frame:

- red will be the current x coordinate location of the mouse
- green will be the current y coordinate location of the mouse
- blue will be the x coordinate location of the mouse in the previous frame.

The line will be drawn at this location in this frame:

- the line will begin at the current x coordinate of the mouse and
- the current y coordinate of the mouse
- and will be drawn to the previous x coordinate of the mouse and
- the previous y coordinate of the mouse.

Processing has other variables that we will use very soon.

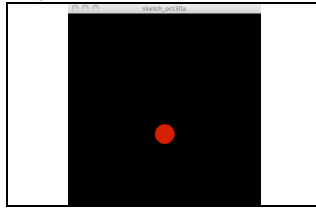
There is another side to this variable story. We can build and use our own variables. Here is a program that has three variables:

```
// these are variables
float x, y, diam;

void setup( )
{
  size( 400, 400 );
  smooth( );
  x = 200.0;
  y = 250.0;
  diam = 42.0;
}

void draw( )
{
  background( 0 ); // black
  fill( 200, 0, 0 ); // dark red
  ellipse( x, y, diam, diam );
}
```

This program makes this output in the window:



The ellipse is located *x* pixels away from the left edge and *y* pixels down from the top edge. The ellipse is *diam* pixels wide and high.

When Processing executes the `draw()` function, it actually looks up the values stored in the variables *x*, *y*, and *diam* and uses those values to draw the ellipse 200 pixels away from the left edge, 250 pixels down from the top and 42 pixels wide and high.

What we cannot see is that Processing is drawing a new frame with the red circle 60 times a second.

We can change that by altering or varying the value on one or both variables. Let's change the value of *x*. We will start with the program with the variable *x* having a value of zero.

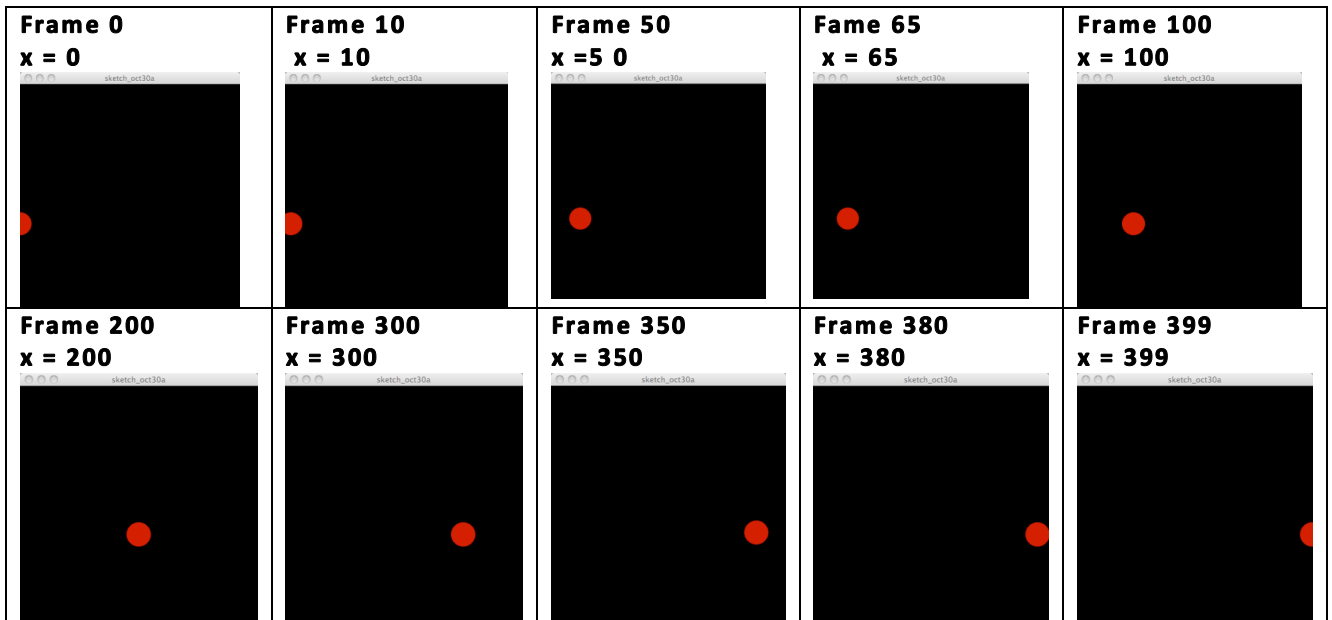
Then in each frame after we have drawn the circle, we will alter the value of *x* by adding one to its value. Let's look at the code and the result and then explain how this works. Here is the code;

```
// these are variables
float x, y, diam;

void setup( )
{
  size( 400, 400 );
  smooth( );
  x = 0.0;
  y = 250.0;
  diam = 42.0;
}

void draw( )
{
  background( 0 ); // black
  fill( 200, 0, 0 ); // dark red
  ellipse( x, y, diam, diam );
  x = x + 1;
}
```

And here is the result of several frames:



As you can see in the figures above, as the value of x changes, the circle move further to the right. If we continue to run the program, the circle will disappear off the right side of the window.

The reason we cannot see the previous frames behind the new frame is because each new frame has a black background that hides the older frame.

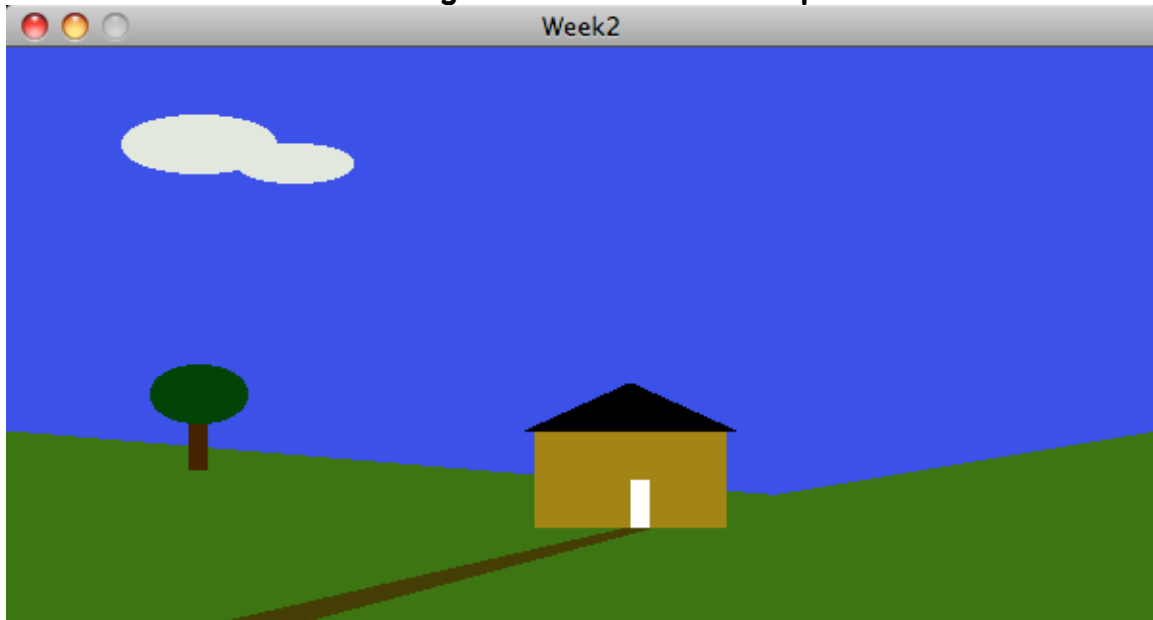
So this is one way we can use variables but this animation stuff is more than we want to use right now. You can, if you want to experiment with this during the coming week. We will come back to it later in the course sequence. We looked at it here to show you one way we can use variables.

Before we go back to animation, we want to look at two other ways to use variables to make our programs more “useful.”

So for now, we put animation away and move ...

onward...

Let's return to Jim's drawing of at the end of the previous set of notes.



Here is some of the code that draws this figure:

```
fill( 147, 114, 3 );
rect( 275, 200, 100, 50 );
fill( 0 );
triangle( 270, 200, 380, 200, 325, 175 );
fill( 255 );
rect( 325, 225, 10, 25 );
fill( 54, 48, 1 );
quad( 325, 250, 335, 250, 150, 300, 110, 300 );
```

It is difficult to figure out what function draws what part except for the roof which is probably drawn by the triangle() function.

When Jim wrote this, he added what we call comments to the code so he would know what the functions were drawing. The comments begin with two slashes and ignored by Processing when the program is executed.

Here is the commented form of the program:

```
// house
// front
fill( 147, 114, 3 );
rect( 275, 200, 100, 50 );
// roof
fill( 0 );
triangle( 270, 200, 380, 200, 325, 175 );
```

```
// door
fill( 255 );
rect( 325, 225, 10, 25 );
// walk
fill( 54, 48, 1 );
quad( 325, 250, 335, 250, 150, 300, 110, 300 );
```

So what does this have to do with variables? As this code is written, absolutely nothing.

But what if Jim's boss asked him to move the house to the right a bit and make it a bit bigger but make the height of the roof smaller. Oh, make the door larger and move it left a bit.

Jim would have to rewrite almost the entire house drawing code. All of the arguments in the function calls would have replaced with new values that would do what the boss asked Jim to do.

This is because all of the arguments that Jim used in the functions are the actual numbers of the locations. These numbers are called literal constants. The value of 42 is always 42. You cannot change it.

But what if Jim used variables instead of the constants. Suppose he drew the house with code like this:

```
// variables:
float x = 275;    // location of the upper left corner
float y = 200;    // of the brown rectangle of the house
float wd = 100;   // width of the house
float ht = 50;    // height of the house
// house
// front
fill( 147, 114, 3 );
rect( x, y, wd, ht );
```

This code will draw the brown rectangle that forms the structure of the house using variables instead of constants. If Jim's boss wants the house moved to the right, all Jim as to do is change the value of x:

```
float x = 375;
```

In a similar way he can make the house smaller by changing the variables wd and ht:

```
float wd = 75;
float ht = 30;
```

This works fine for the brown rectangle but what about the roof. That does not use the values (200, 100) for any of the triangle vertices or the values wd or ht for the width or height of the roof.

In order to do this we need some arithmetic. Processing does arithmetic just like you do. The rules for adding, subtracting, multiplying, and dividing are the same. The rules for using parentheses are the same.

The only difference you need to know is the multiplication operator. In your arithmetic classes you learned that the multiplication operator is the X. Unfortunately programming languages like Processing cannot tell the difference between the variable x and the operator X so Processing uses a different multiplication operator - the asterisk that is on the key with the number 8.

Instead of typing:

```
0.9 x x
```

we type

```
0.9 * x
```

Processing reads this a "zero point nine times the variable x."

if we have these variables:

```
float x = 275;    // location of the upper left corner
float y = 200;    // of the brown rectangle of the house
float wd = 100;   // width of the house
float ht = 50;    // height of the house
```

and we draw the rectangle like this:

```
fill( 147, 114, 3 );
rect( x, y, wd, ht );
```

then we can draw the roof like this:

```
// roof
fill( 0 );
//      left vertex      right vertex      roof peak
triangle( x-5, y,      x+wd+5, y,      x+(wd/2), y-(ht/2);
```

Processing looks up the values of the variables and substitutes them:

```
//      left vertex      right vertex      roof peak
triangle( 275-5, 200,      275+100+5, 200,      275+(100/2), 200-(50/2);
```

which becomes:

```
//      left vertex      right vertex      roof peak
triangle( 270, 200,      380, 200,      325, 175 );
```

If we use similar arithmetic to draw the entire house, then Jim can move it and resize it by just changing the values of the variables used to draw the house.

Now, this can get complicated if you have a complex drawing that uses angles and shapes that are not easy to compute with simple arithmetic. Jim used graph paper to lay out the original drawing when he planned the picture the first time.

If you have followed this, you may be able to see how variables make our figures easier to move and resize.

The first program that drew the house with constants as arguments works fine but is very difficult to change. Some programmer call the use of constants like Jim did for his drawing, the "use of magic" numbers. The constants are referred to as "magic numbers" because they sorta appear like magic.

The use of magic numbers will work but it makes things difficult to alter. One other problem with magic numbers is that it is not easy for other programmers to figure out what your code is doing.

Compare these two lines of code

```
rect( 275, 200, 100, 50 );  
triangle( 270, 200, 380, 200, 325, 175 );
```

to these two lines of code which do the exact same thing:

```
rect( x, y, wd, ht );  
triangle( x-5, y, x+wd+5, y, x+(wd/2), y-(ht/2);
```

and you can get an idea of how the triangle is related to the rectangle more easily in the second set of code.

If you have some time, experiment with variables and arithmetic in the coming week. It will make some of the work we do later much easier.

Another way to move things...

What if we have this house?

```
// house
// front
fill( 147, 114, 3 );
rect( 275, 200, 100, 50 );
// roof
fill( 0 );
triangle( 270, 200, 380, 200, 325, 175 );
// door
fill( 255 );
rect( 325, 225, 10, 25 );
// walk
fill( 54, 48, 1 );
quad( 325, 250, 335, 250, 150, 300, 110, 300 );
```

and we want to move it.

and we do not want to replace the constants with variables and arithmetic.

Can we do this?

Think about it.

How can we?

Think again before going to the next page...

The answer is a trick. A rather neat trick that Processing makes possible with a function.

The constants in the house drawing code are based on the known location of the (0, 0) point or the origin. By default, every execution of the draw() function begins with the (0, 0) point in the upper left corner. X increases to the right and y increases down.

Here is Jim's original code. The only change is a partial sun in the upper left corner drawn with its center at (0, 0);

```
// sun
fill( 200, 200, 0 );
ellipse( 0, 0, 100, 100 );

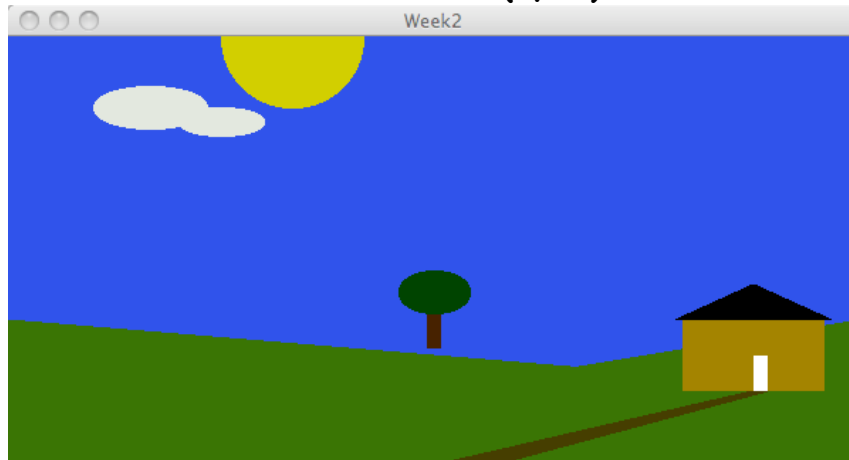
// house
// front
fill( 147, 114, 3 );
rect( 275, 200, 100, 50 );
// roof
fill( 0 );
triangle( 270, 200, 380, 200, 325, 175 );
// door
fill( 255 );
rect( 325, 225, 10, 25 );
// walk
fill( 54, 48, 1 );
quad( 325, 250, 335, 250, 150, 300, 110, 300 );

// tree
// trunk
fill( 54, 25, 1 );
rect( 95, 180, 10, 40 );
// leaves
fill(1, 54, 1 );
ellipse( 100, 180, 50, 30 );
```

And here is the figure the code draws



Now here is Jim's house drawn with the same code plus one new line of code. The sun is still at the coordinates (0, 0):



Remember, the sun is drawn at the (0, 0) point.

How is this possible?

The answer is that Processing has a function that lets us shift the coordinate plane (the (0, 0) point) to the left or right and up or down. Here is the code that does this:

```
// shift the (0, 0) location
translate( 200, 0 );
// sun
fill( 200, 200, 0 );
ellipse( 0, 0, 100, 100 );
```

The `translate()` function takes two arguments.

- The first argument is the amount of shift (in pixels) to the left (a positive value) or to the right (a negative value).
- The second argument is the amount of shift (in pixels down(a positive value) or up (a negative value).

The translation remains in effect for everything drawn from this point to the end of draw.

The translation is canceled and returns to (0, 0) at the start of the next frame. You can translate as many times as you want to as long as you keep track of where you are. Notice that Jim's tree is floating in space in the last drawing... This is not good!!!

But, beware !! You can translate off screen and not see what you draw.

You should play with this before the next class. You can also use this in animation but we will talk about that next time...

Remember, this is a exploratory program – go explore and get lost in the code. If it does not work, bring it to class next time.

Your Assignment for next time:

Put a `setup()` function in your program and set the size of the window to something reasonable – try `size(400, 400);` Put a `draw()` function in your program and do the following:

Design a symbol or emblem to represent you. This is sorta' like a coat of arms or family crest. It can have your initials or what ever you want but it should represent you. We will use this symbol in a game a bit later.

Since it will be used in a game, there are a few limits you should keep in mind if you want your game to look good.

Design your emblem to fit inside a circle. We will be moving this around the screen and it looks better if it round or very nearly round. You do not have to show the circle but you can.

Here is the tough part. Try to fit your emblem or coat of arms in a circle that is 200 pixels in diameter. This could be tough but give it a try.

Remember to take a picture of your emblem with the `saveFrame()` function.

If you are really getting into this stuff, try to design your emblem using the variables and arithmetic that Jim did starting on page 9.

If you get this working, and still have some time, interest, and energy, try to move your emblem around the screen by setting your `x` and `y` variables to the values of `mouseX` and `mouseY`. When you move the mouse in the window, your emblem should follow the mouse.

If you get this working, look up the `noCursor()` function and see what it does. Call this in your `setup()` function after you set the size of the window.