How To Create A Simple Version Of ‘Flappy Bird’ In “Processing” Using The p5.js Library

By: Dylan Melton

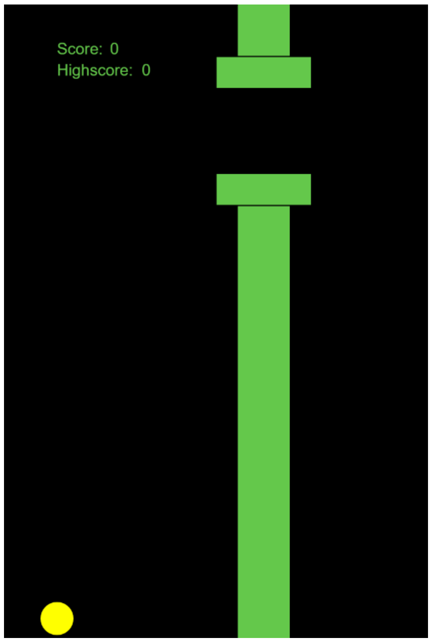
Table Of Contents

Section 1: Introduction

Programming can be used to create many different concepts like games, simulations, applications, and many more. Each type of program can be used by different people for their specific needs. For those programs, a different language of coding and/or a different library can be used. Having the knowledge of being able to code to these specific needs is a great skill to have.

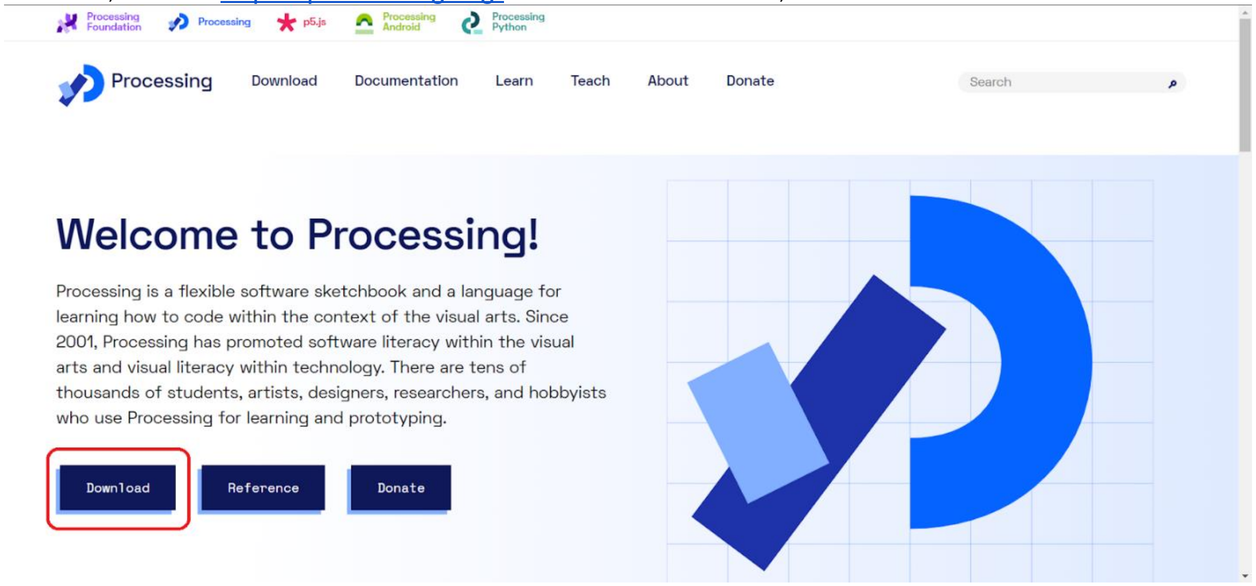
In this tutorial, I will be going to show you how to create a simple game known as 'Flappy Bird' in the application "Processing" using a coding library called p5.js. 'Flappy Bird' was a well-known mobile game where you would have to tap your screen to make the bird "fly" and try to avoid the incoming pipes, trying to get through as many pipes as you could without hitting one. The game seems like a very simple concept, which it is.

Here is what the finished program will look like, and although it doesn't look like much, there's quite a bit that goes into making it work like the game 'Flappy Bird'. This is to get the basic mechanics of the game to work, you can still go a bit further to make it look like the real game, but in this tutorial, I will **not** be going over that.



Section 2: Downloading Processing

To get started we are going to need to download the program called "Processing". In your browser, search https://processing.org/. From the home site, click **Download**.

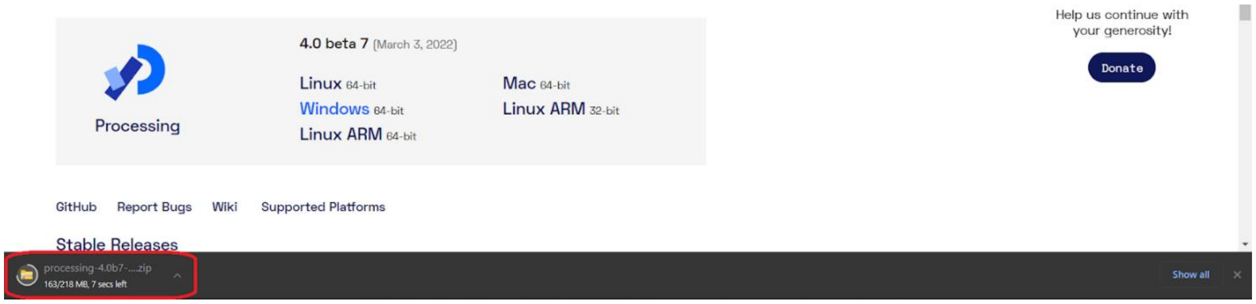


Welcome to Processing!

Processing is a flexible software sketchbook and a language for learning how to code within the context of the visual arts. Since 2001, Processing has promoted software literacy within the visual arts and visual literacy within technology. There are tens of thousands of students, artists, designers, researchers, and hobbyists who use Processing for learning and prototyping.

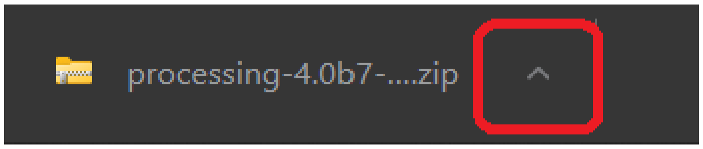
There it should've taken you to a page where you are able to choose the kind of system you are using, Mac, Windows, or Linux. Click on the option that you need depending on your system.

****Disclaimer** This will be for a Windows user install.**

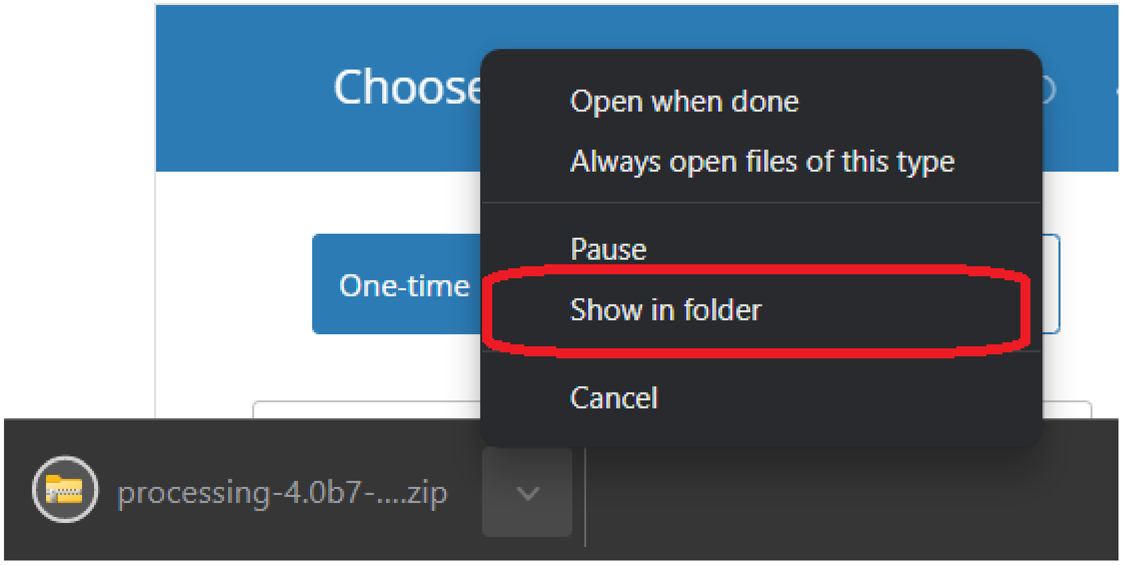


4.0 beta 7 (March 3, 2022)

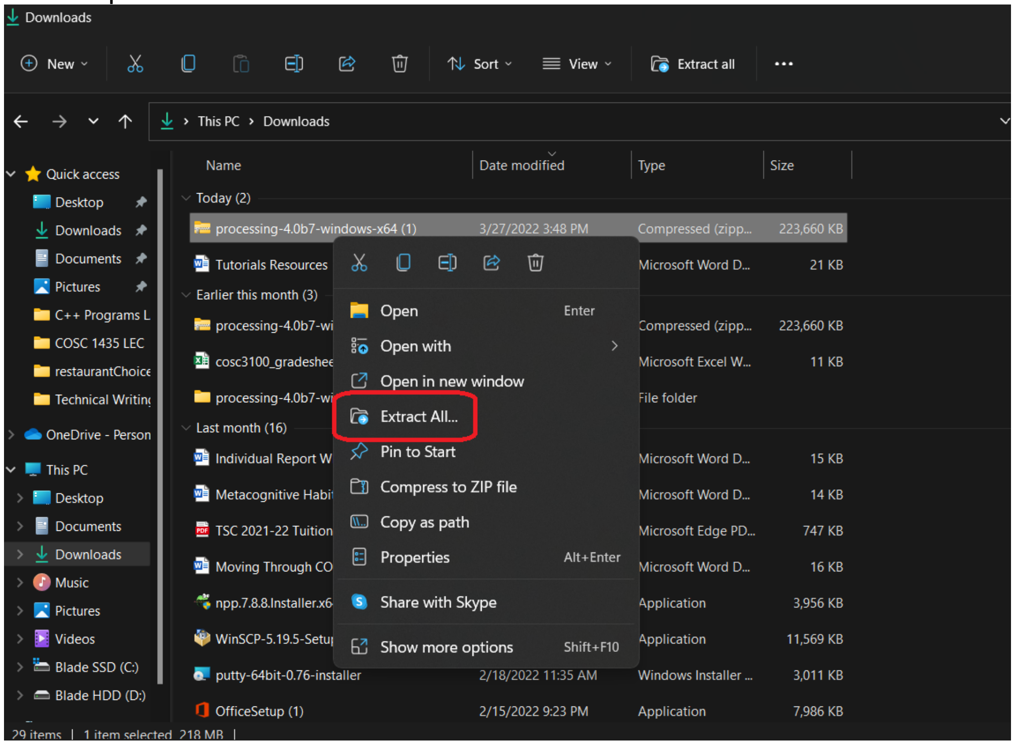
After you choose your option, a folder should start to download. After it is finished downloading, click the up arrow.



When the arrow is clicked, it will bring up multiple options. The option you need to choose is the “Show in folder” option.



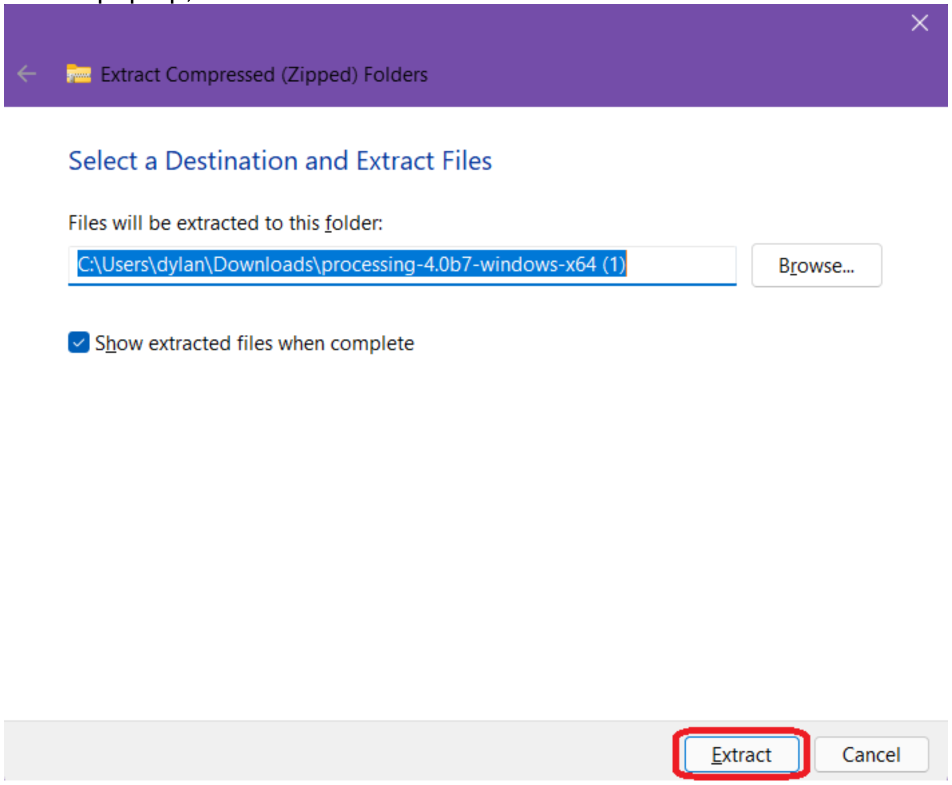
That should bring you to the location of the folder. When it opens, right click on the .zip folder and choose the option “Extract All”.



Name	Date modified	Type	Size
Today (2)			
processing-4.0b7-windows-x64 (1)	3/27/2022 3:48 PM	Compressed (zipp...	223,660 KB
Tutorials Resources		Microsoft Word D...	21 KB
Earlier this month (3)			
processing-4.0b7-wi		Compressed (zipp...	223,660 KB
cosc3100_gradeshee		Microsoft Excel W...	11 KB
processing-4.0b7-wi		File folder	
Last month (16)			
Individual Report W		Microsoft Word D...	15 KB
Metacognitive Habi		Microsoft Word D...	14 KB
TSC 2021-22 Tuition		Microsoft Edge PD...	747 KB
Moving Through CO		Microsoft Word D...	16 KB
npp.7.8.8.Installer.x6		Application	3,956 KB
WinSCP-5.19.5-Setu		Application	11,569 KB
putty-64bit-0.76-installer	2/18/2022 11:35 AM	Windows Installer ...	3,011 KB
OfficeSetup (1)	2/15/2022 9:23 PM	Application	7,986 KB

29 items | 1 item selected 218 MB |

A new window will pop up, and click "Extract"

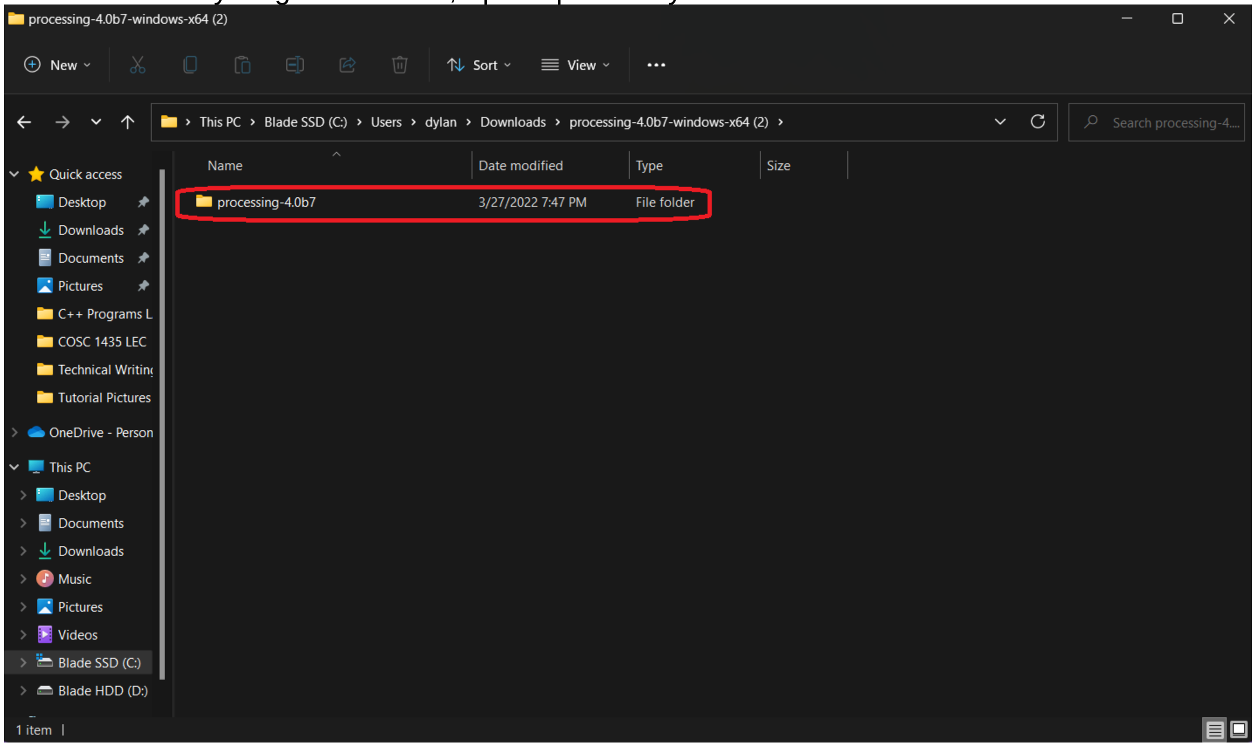

Extract Compressed (Zipped) Folders

Select a Destination and Extract Files

Files will be extracted to this folder:

C:\Users\dylan\Downloads\processing-4.0b7-windows-x64 (1)

As soon as everything is extracted, open up the only folder.


processing-4.0b7-windows-x64 (2)

This PC › Blade SSD (C:) › Users › dylan › Downloads › processing-4.0b7-windows-x64 (2) ›

Name	Date modified	Type	Size
processing-4.0b7	3/27/2022 7:47 PM	File folder	

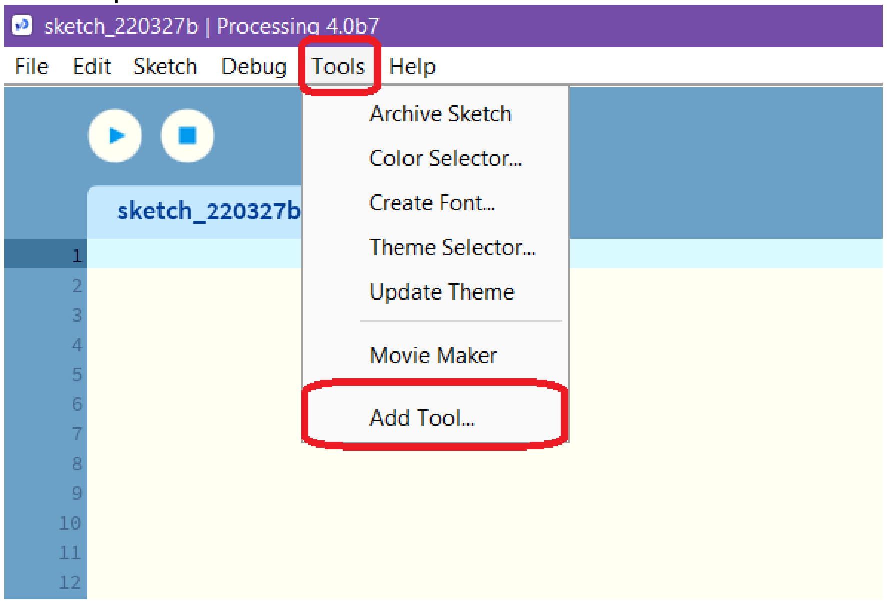
1 item

Once the folder is open, you can finally open the "processing" application by double clicking it.

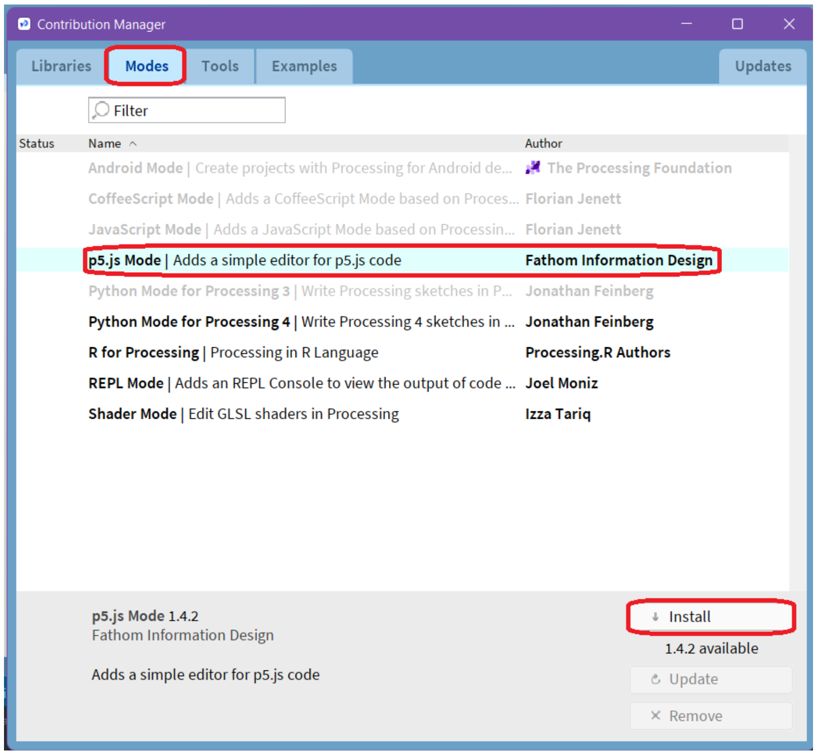
Now you have successfully installed "processing"!

Section 3: Setting Up Processing

Now that the application is open, the setup to get the p5.js library is actually very easy. At the top left of the window, there's a tab called "Tools", click that and there will be a drop down and in there click the option for "Add Tool".



That should take you to a menu with four tabs. Click on the tab "Modes", and then click "p5.js" choice and "install".



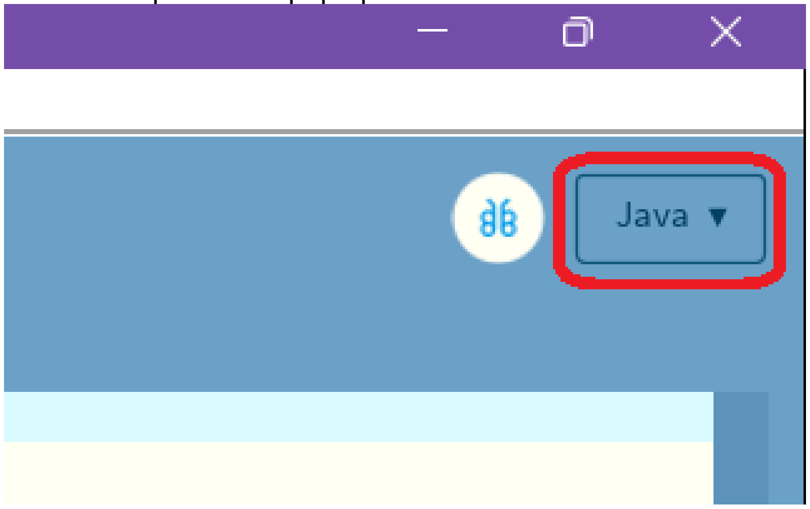
Status	Name	Author
	Android Mode | Create projects with Processing for Android de...	The Processing Foundation
	CoffeeScript Mode | Adds a CoffeeScript Mode based on Proces...	Florian Jenett
	JavaScript Mode | Adds a JavaScript Mode based on Processin...	Florian Jenett
	p5.js Mode | Adds a simple editor for p5.js code	Fathom Information Design
	Python Mode for Processing 3 | Write Processing sketches in P...	Jonathan Feinberg
	Python Mode for Processing 4 | Write Processing 4 sketches in ...	Jonathan Feinberg
	R for Processing | Processing in R Language	Processing.R Authors
	REPL Mode | Adds an REPL Console to view the output of code ...	Joel Moniz
	Shader Mode | Edit GLSL shaders in Processing	Izza Tariq

p5.js Mode 1.4.2
Fathom Information Design

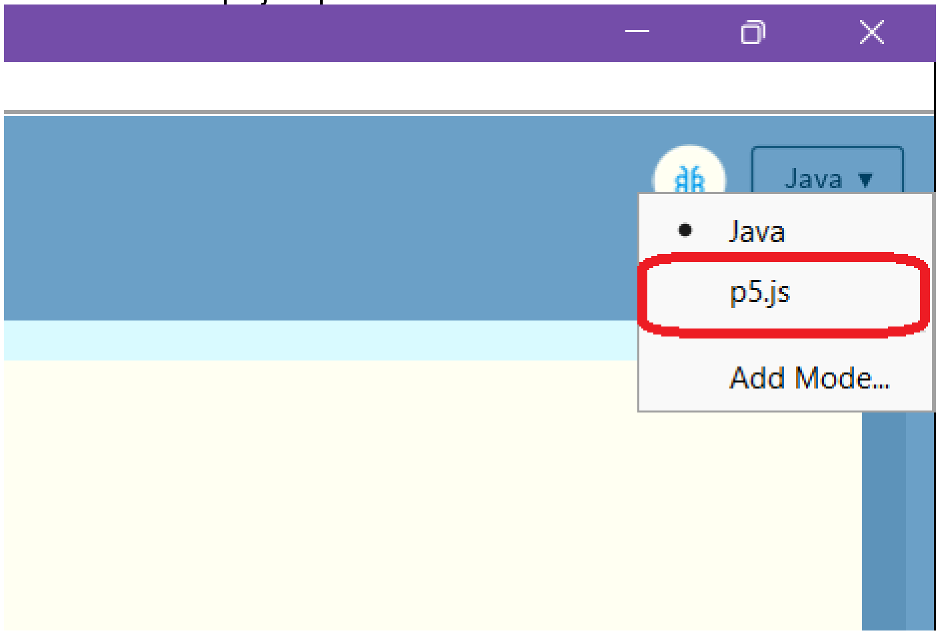
Adds a simple editor for p5.js code

1.4.2 available

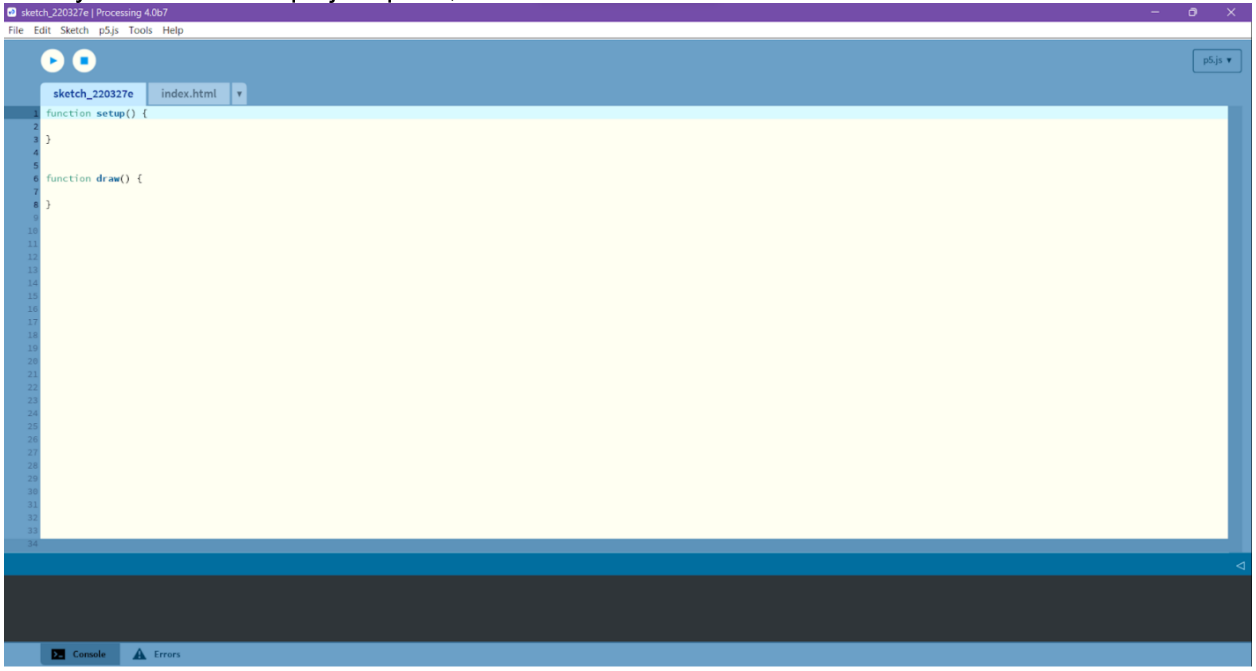
After it is installed, in the main page at the top right you'll see the box that has "Java" with an arrow. Click that and some options will pop up.



You will need to choose the "p5.js" option.



After you choose the “p5.js” option, this is what the window should look like.


sketch_220327e | Processing 4.0b7
File Edit Sketch p5.js Tools Help
p5.js ▾
sketch_220327e
index.html
Console
Errors

```
function setup() {

}


function draw() {

}
```

When you’re at this point, you are ready to begin coding!

****Disclaimer**** Through out the program, ( ) [ ] { }, will be used. All of these have a different functionality in programming.

( ) are used to hold in a condition of a function.

[ ] are used with arrays.

{ } are used to keep in code that will be executed if the function they are a part of is called.

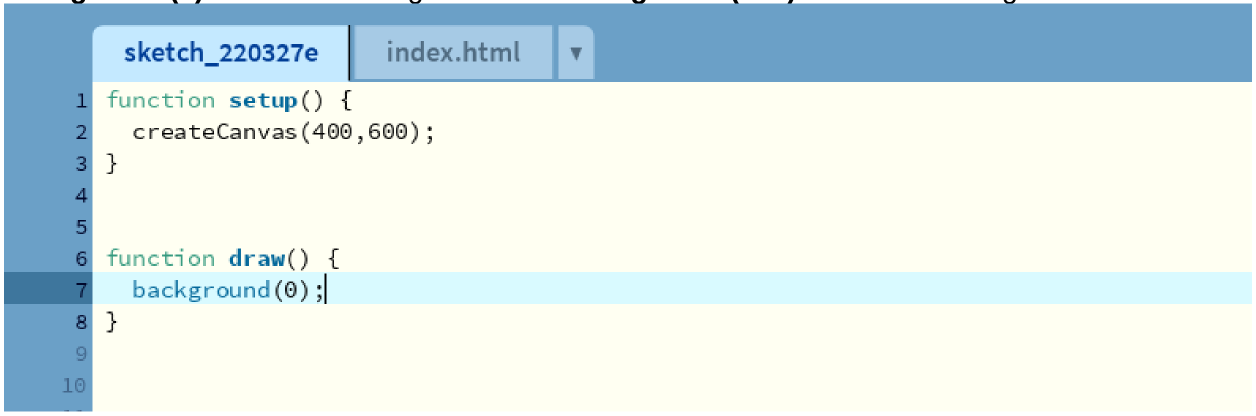
Also, don’t forget a semicolon after every line of code, but do **NOT** use them at the end of conditions or functions. The code inside the block will not execute correctly or at all.

Section 4: Creating The Code

To start off the coding, we are going to need a window or a “canvas” to put everything onto. So, to do that, in the setup function block of code, you need to input **createCanvas(x,y)**, a built-in function in the p5.js library. The ‘x’ value is how wide the window will be, and the ‘y’ value is how tall the window is. Just to get things started, we’re going to input an ‘x’ value of 400 and a ‘y’ value of 600, so the window should be 400x600 pixels.
****Disclaimer**** The canvas is made from (0,0), the top left corner, going to the right is a positive ‘x’ value and going down is a positive ‘y’ value.

The window will just be blank if you run it at this moment, so we need to add some color to it. To do that we need to color the background. So in the **draw()** function block we need to input **background(r,g,b)** where r,g,b corresponds to the value of red, green, and blue values respectfully. The values of these range from 0 to 255. You can mess around with the values of these to get a color you want, but for this tutorial I’m going to make the background black. Something special about the background function is that if you just input one number from 0 to 255, it’ll make the color a grayscale, so it’ll be a different shade of gray, no other color.

background(0) is a black background and **background(255)** is a white background.



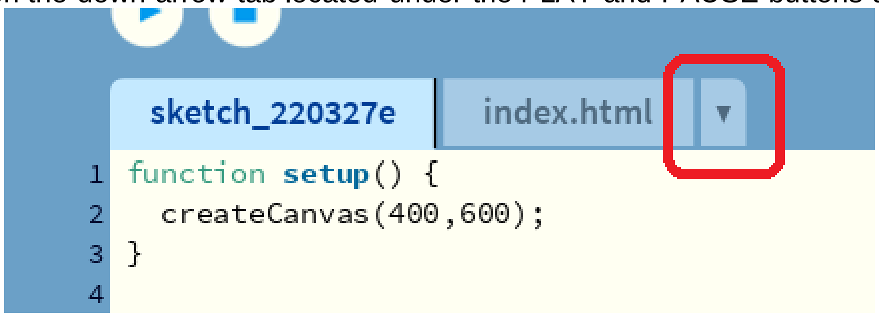
```
function setup() {
  createCanvas(400,600);
}


function draw() {
  background(0);
}
```

At this point, you will have a black canvas 400x600 pixels.

Section 4.1: The Bird

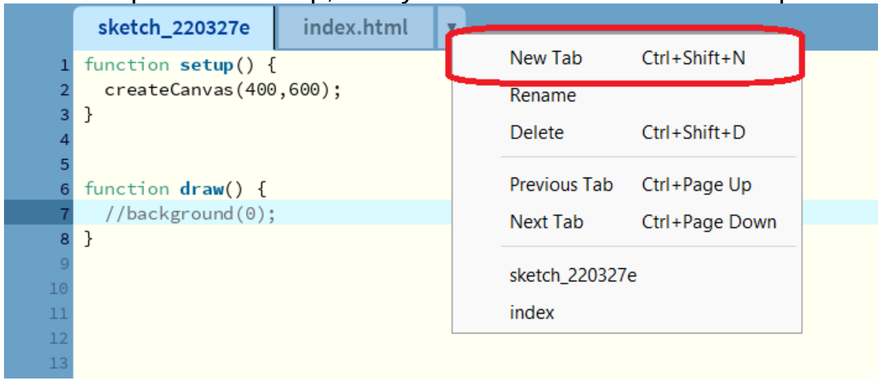
Now onto making the bird object. Before we start making the bird, we need to understand what the bird needs to be doing. In the mobile game, the bird started off in the center of the screen and once it started, the bird would fall without any input, and after an input it would fly up.

So, to start making it, we first need to make a separate java file called **bird.js**. To do that, you need to click on the down arrow tab located under the PLAY and PAUSE buttons at the top left.



```
function setup() {
  createCanvas(400,600);
}
```

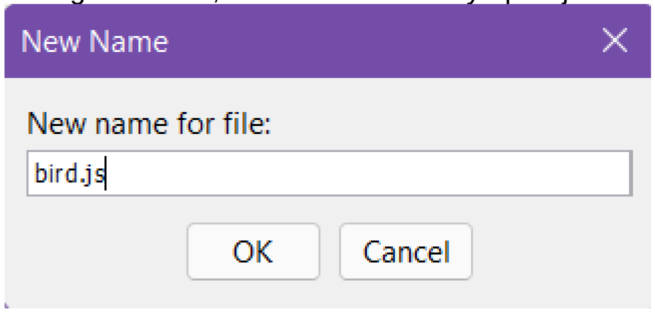
That will make several options show up, and you will need to choose the option of “New Tab”.



```
function setup() {
  createCanvas(400,600);
}


function draw() {
  //background(0);
}
```

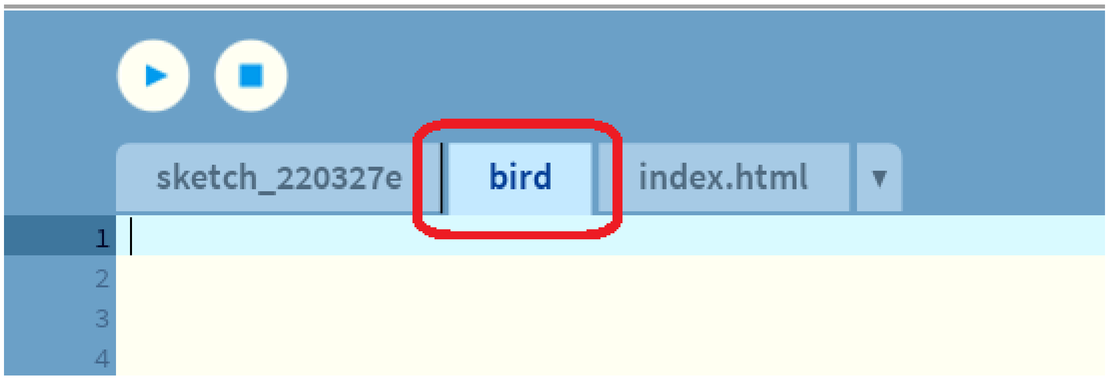
A window will pop up, asking you to name the tab. Name it **bird.js** to make things simplistic and easy to understand.

****Disclaimer**** When naming new tabs, make sure to always put .js at the end.



New name for file: bird.js

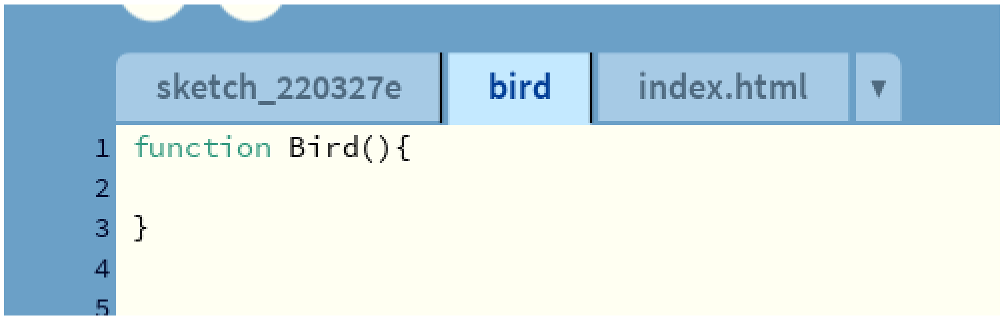
Now you’ll see the new tab “bird” appear in the list of tabs.



Adding the new tab automatically places the file location in the "index.html" tab so you don't have to worry about adding the **bird.js** file to it.

****Disclaimer**** To start off any new tab, we need to make a constructor function in the tab.

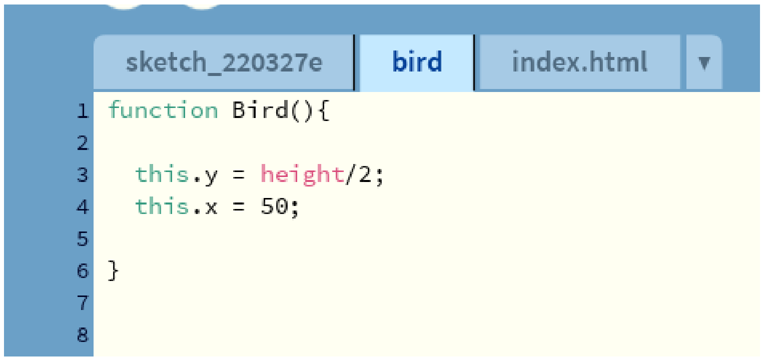
That will look like this.



```
function Bird(){

}
```

Now that we have the base of the **Bird()** function, we need to add the characteristics that make the bird act like the bird from the mobile game. To start off, we can make variables for the bird's location, the **x** coordinate, and the **y** coordinate. In the mobile game the bird starts off in the center of the screen, but mostly to the left. Knowing this we make a **y** variable called **this.y** setting it equal to ***height***/2, making the value half of the canvas **y** value. The **x** variable will be called **this.x** and can be set to any number you want, but for this tutorial I will set it equal to 50.

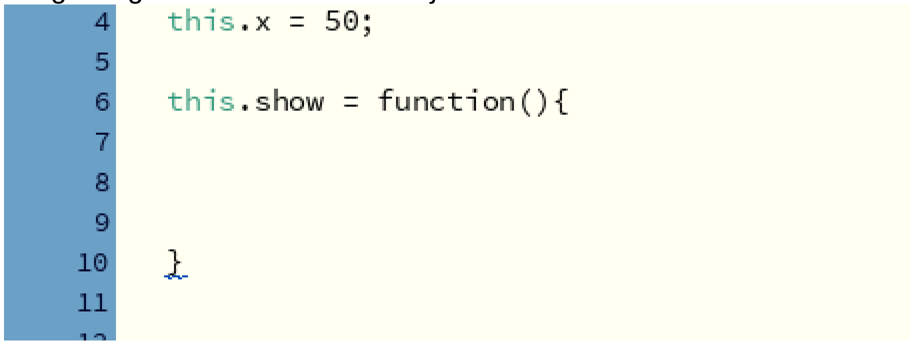
****Disclaimer**** ***height*** is a built-in variable in the p5.js library as well as many others. Also, in JavaScript when in a constructor function most things named, like variables and new functions made, will need to start with '**this.**'.



```
function Bird(){

  this.y = height/2;
  this.x = 50;

}
```

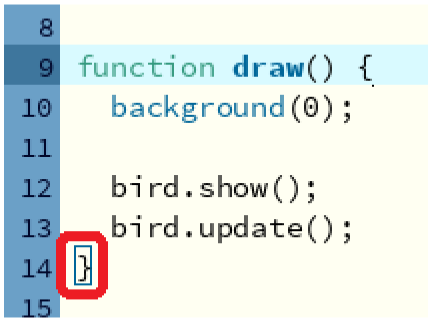
Now we need to make a new function inside the **Bird()** function. This function will be what makes the bird appear on screen.

The function can be called whatever you want, but for this tutorial I will call it '**show**' since it makes most sense. Just like the variables, it needs to start with '**this.**' and then set it equal to **function()** making it look like '**this.show = function()**'. And since it is a function, it needs a { bracket at the beginning of the function and a } bracket at the end.



```
this.x = 50;

this.show = function(){



}

```

****Disclaimer**** You can click on the { bracket after the **draw()** function and see which } bracket it's attached to. You will see a little blue box around the } bracket it's attached to. You can do this with any of the { brackets.



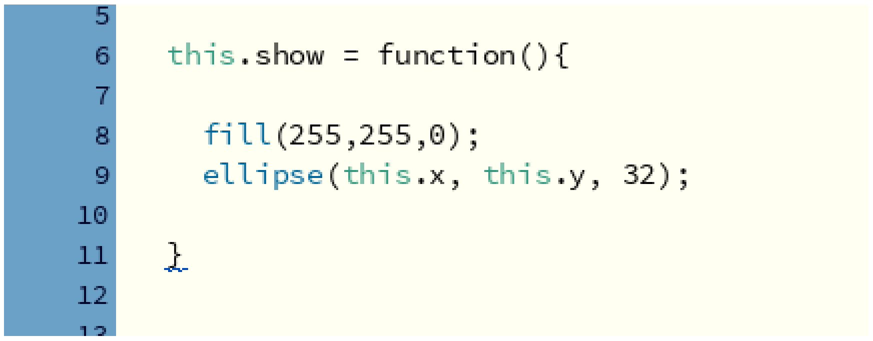
```

function draw() {
  background(0);

  bird.show();
  bird.update();
}

```

To make the bird appear, we need to add a shape within the **this.show** function. To do that we can write **ellipse(x,y,w,h)**, a built-in function from the p5.js library. Where the **(x)** value is the x coordinate on the canvas, the **(y)** value is the y coordinate on the canvas, the **(w)** and **(h)** value are the width and height of the ellipse.

If you want a circle, you can just input a **(w)** value and the program will automatically make that **(w)** value an **(h)** value. The **this.x** and **this.y** variables we made earlier will replace the **(x)** and **(y)** in the **ellipse()** function. The **(w)** value will be set as 32. Making it look like **ellipse(this.x,this.y,32)**.

To add color to the shape, we can add the **fill(r,g,b,a)** function before the **ellipse()** function where the **(r)(g)(b)** values are the red, green, and blue values respectfully ranging from 0 to 255 and the **(a)** value is the alpha value ranging from 0 to 100. The alpha value is an optional value and is basically the transparency of the color. So for this tutorial I'll make the ellipse yellow, making the **fill()** function look like **fill(255,255,0)**.

The code should look like this.

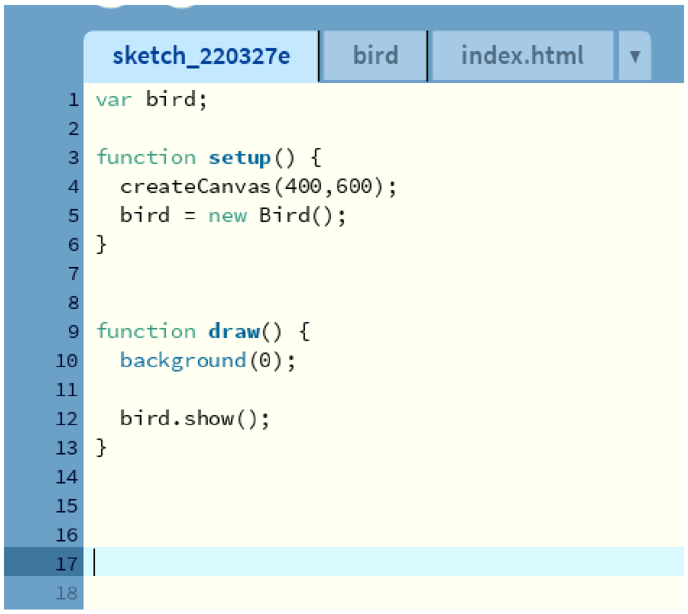


```
this.show = function(){

  fill(255,255,0);
  ellipse(this.x, this.y, 32);

}
```

If you run the program right now, you'll notice that you don't see the "bird" you just made and only the background. That's because we need to add the **show()** function we made in the "**bird**" tab to your main sketch tab.

To do that we need to add **bird.show()** in the **draw()** function block after the **background()** function, create a **bird** variable before the **setup()** function, and a *new* **bird** variable in the **setup()** function. To create a variable, you need to write "var (name of variable)". Note, the parentheses are not apart of the actual naming, they are there to show that's what the variable will be called. So for the **bird** variable it should look like "var bird", and the *new* **bird** variable should look like "bird = new **Bird()**". Writing it like this gets the information from the **Bird()** function in the "**bird**" tab.

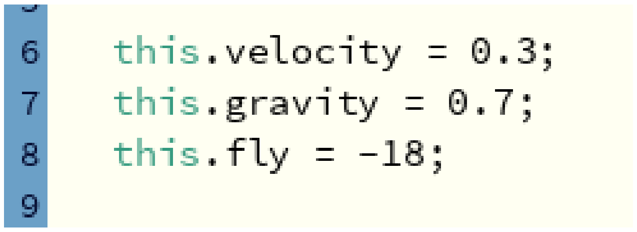
At this point the main sketch tab code should look like this.



```
var bird;

function setup() {
  createCanvas(400,600);
  bird = new Bird();
}


function draw() {
  background(0);

  bird.show();
}
```

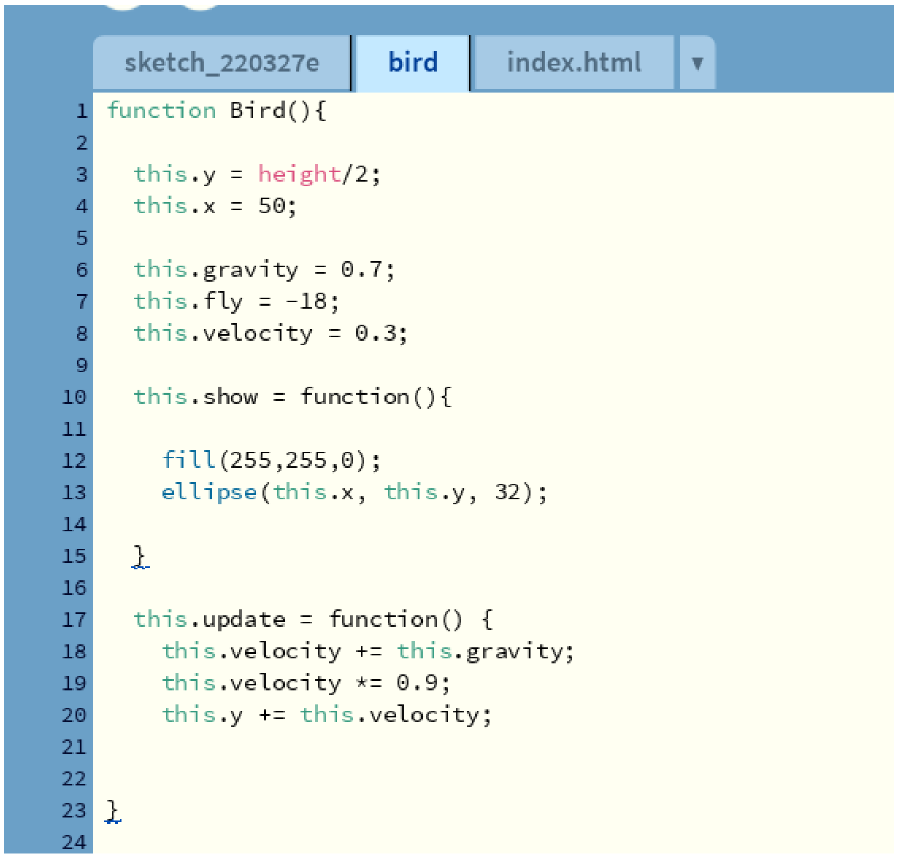
Now we need to make the bird fall when there is no input and to do that we need to make a new function in the "**bird**" tab. This function will be made just like the **show()** function. We need to call it **update()**, and this function should look like **this.update()** with the { } brackets making the start and end of the block of code for the function.

Where we made the variables **this.x** and **this.y,** we need to make three more variables. One is going to be for the **velocity** or how fast the "bird" will drop. We can't just have this variable alone cause the bird will fall with a constant speed, so we need to add a **gravity** variable making the **velocity** variable change every time the **update()** function is called. Then we're going to need a variable that reverses the direction the "bird" is moving, so we're going to add a **fly** variable. Now, all these variables can be tuned to your liking, but for this tutorial I've gotten the values close enough to make the game playable. The value for **this.velocity** should be equal to 0.3, the value for **this.gravity** should be equal to 0.7, and the value for **this.fly** should be equal to -18.



```
this.velocity = 0.3;
this.gravity = 0.7;
this.fly = -18;
```

Inside the **update()** function we need to assign the changing values of the **this.velocity** variable and the **this.y** variable. So first, the **this.velocity** variable should have the **this.gravity** variable added to it, looking like "this.velocity += this.gravity". In the next line, the new result of the **this.velocity** variable should be multiplied by 0.9, looking like "this.velocity *= 0.9". Finally the **this.y** variable should have the **this.velocity** value added to it, looking like "this.y += this.velocity". Doing all of this should make the "bird" fall with acceleration, so it shouldn't be falling at a constant speed. This is what the code should be looking like at this point.



```
function Bird(){

  this.y = height/2;
  this.x = 50;

  this.gravity = 0.7;
  this.fly = -18;
  this.velocity = 0.3;

  this.show = function(){

    fill(255,255,0);
    ellipse(this.x, this.y, 32);

  }

  this.update = function() {
    this.velocity += this.gravity;
    this.velocity *= 0.9;
    this.y += this.velocity;


}

```

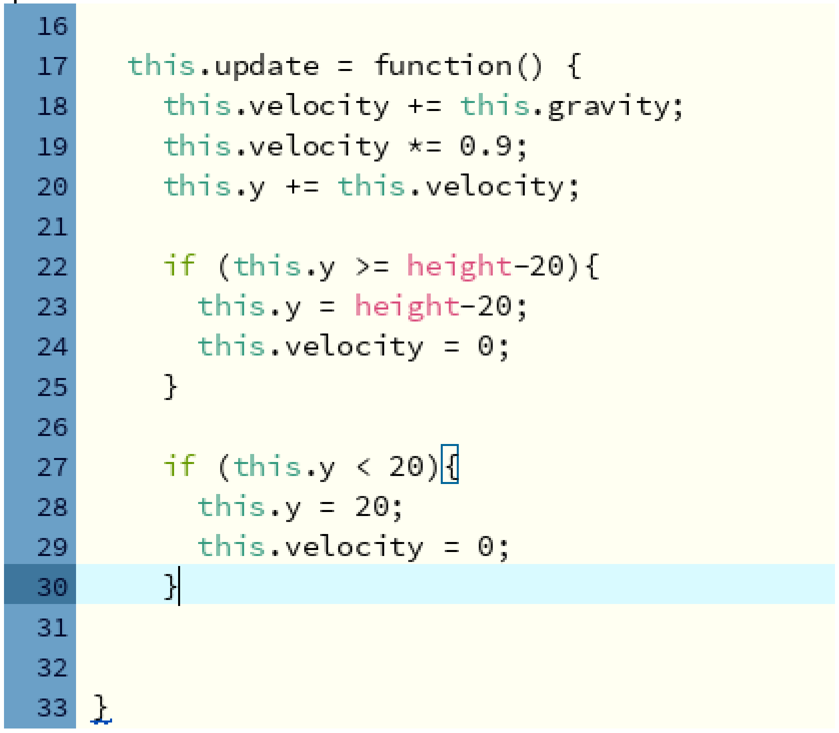
If you want to check to see if everything is working correctly, add the **update()** function to the main sketch tab right after **bird.show()** and then hit the PLAY button. Here you should see the "bird" fall, but it keeps going right off the screen. Now we need to add something that limits where the "bird" can go. To do this, we need to add two conditional statements in the **update()** function in the "**bird**" tab.

In the **update()** function right after "this.y += this.velocity" we need to put in two **if(condition)** statements, where the **(condition)** is what we want to test to see if it's *true* or *false*. The **if()** statement should look like "if (condition){ }" where the { } brackets hold the code that will be executed if *true*. The first test is to see if the "bird's" **this.y** value is at the bottom of the screen. For this tutorial, I have tuned the numbers as much as possible to make sure there are no glitchy animations.

So like setting the **this.y** value in the beginning, we can see if the **this.y** value is greater than or equal to the window's **height** minus 20. If this statement is true, we need to set the **this.y** value to be equal to the **height** minus 20, and the **this.velocity** value to zero to make it look like it stopped at the bottom of the screen.

The second test is to see if the "bird's" **this.y** value is at the top of the screen. For this one we need to see if the "bird's" **this.y** value is less than to 20. If this statement is true, we need to set the **this.y** value to be equal to 20, and the **this.velocity** value to zero to make sure the "bird" doesn't fly off the top of the screen.

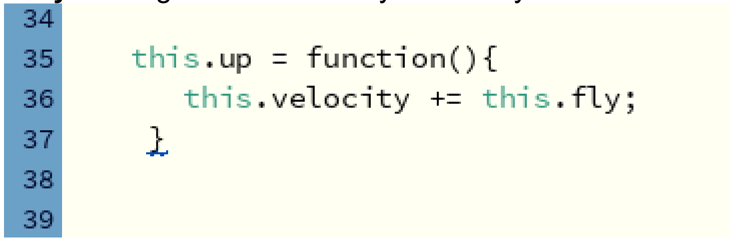
This is what this part of code should look like.



```

  this.update = function() {
    this.velocity += this.gravity;
    this.velocity *= 0.9;
    this.y += this.velocity;

    if (this.y >= height-20){
      this.y = height-20;
      this.velocity = 0;
    }

    if (this.y < 20){
      this.y = 20;
      this.velocity = 0;
    }


}
```

Now the "bird" won't fall off the screen or fly off the screen. But now we need to make the "bird" fly up, and we want that to happen every time we hit a certain key, in this case it will be the *SPACE* bar of your computer.

First, we need to make another function in the “**bird”** tab. For this tutorial I named the function ‘**up**’ because it makes the most sense. By this point you should know how to write a function, but just in case it should look like “this.up = function(){ }” where the { } brackets hold the code that will run if the function is called. In this function we just need to add the value of **this.fly** to the value of **this.velocity** looking like “this.velocity += this.fly”



```
  this.up = function(){
     this.velocity += this.fly;
   }
```

Now we need to add that whenever the *SPACE* bar is hit, the **up()** function is called and executed. To do that we need to add an **if()** statement with a specific function as it’s condition in the main sketch tab after the **draw()** function block.

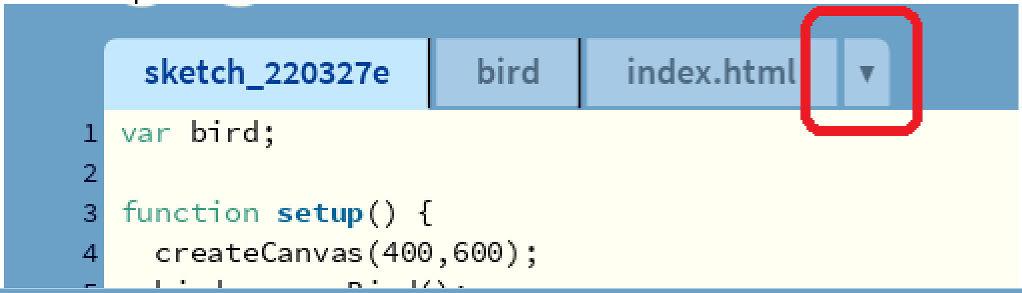
The function we need to add is called **keyPressed()** with an **if()** statement inside. We need to test if the *SPACE* bar was hit, and there is a specific keyword called **key** and we need to see if it’s equal to “ “. That is a whitespace, or an actual space, between the quotes, so the condition should look like ‘if(key == “ “){ }’, and the code that should run is the **up()** function from the “**bird**” tab. Like the **show()** and **update()** function, it should look like ”bird.up()”. If everything is correct, when run you should see the “bird” fall, and when the *SPACE* bar is hit the “bird” flies up.

That is the completion of making your “bird”! Now onto the pipes!

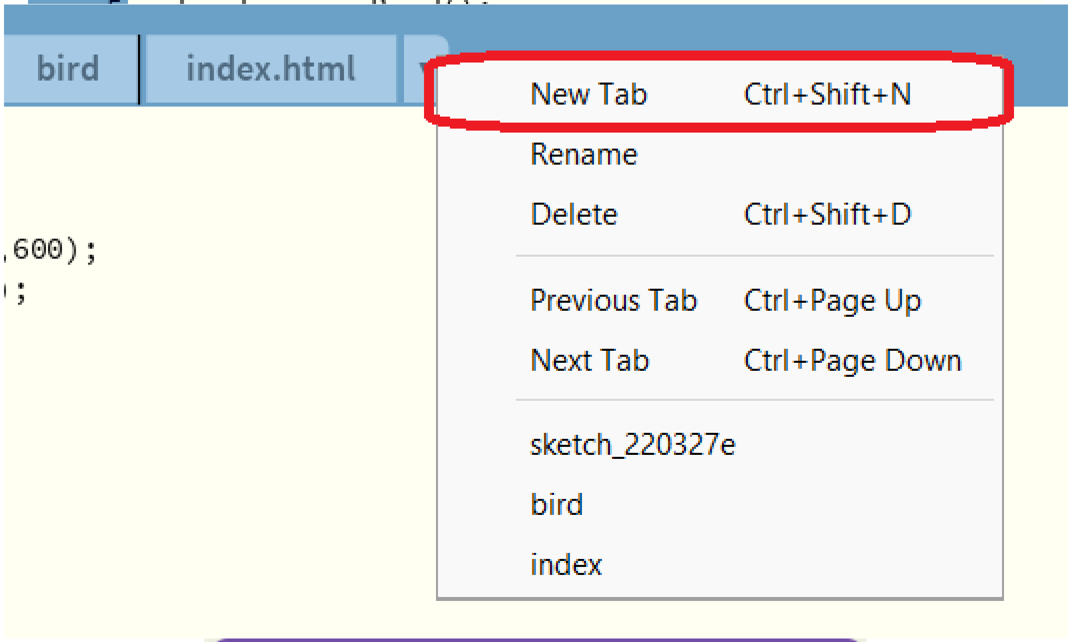
Section 4.2: The Pipes

In the mobile game, the “pipes” seem to appear at random heights and are moving constantly from the right with the same distance in between them. Creating the “pipes” is a little bit more difficult than making the “bird”, but similar.

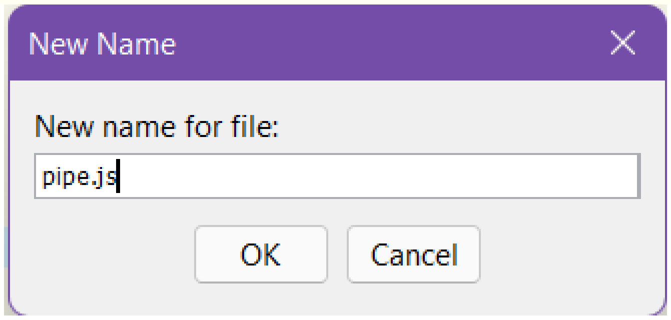
To start off, we’re going to need to make a separate tab and constructor function for the “pipes”. To make it is exactly like how we made the “**bird**” tab, but just name it “pipe.js”. Here’s a quick refresher of the steps to make the new tab.



```
var bird;

function setup() {
  createCanvas(400,600);
```

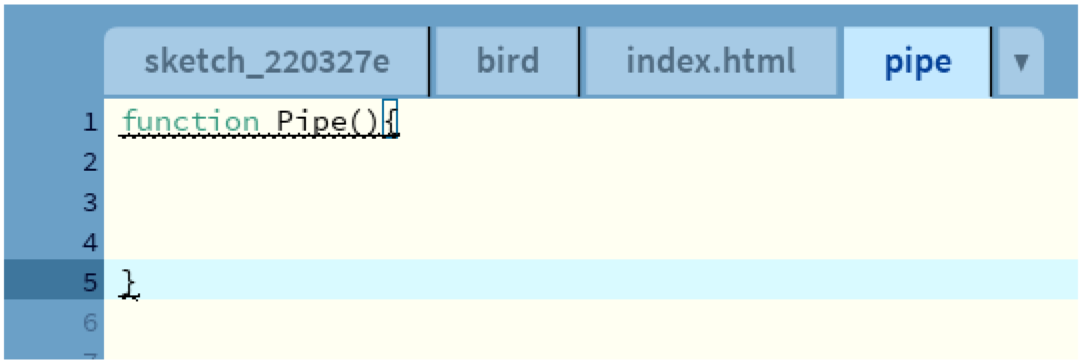


```
,600);
;
```



New name for file: pipe.js

And to make the constructor function for the “**pipe**” tab.



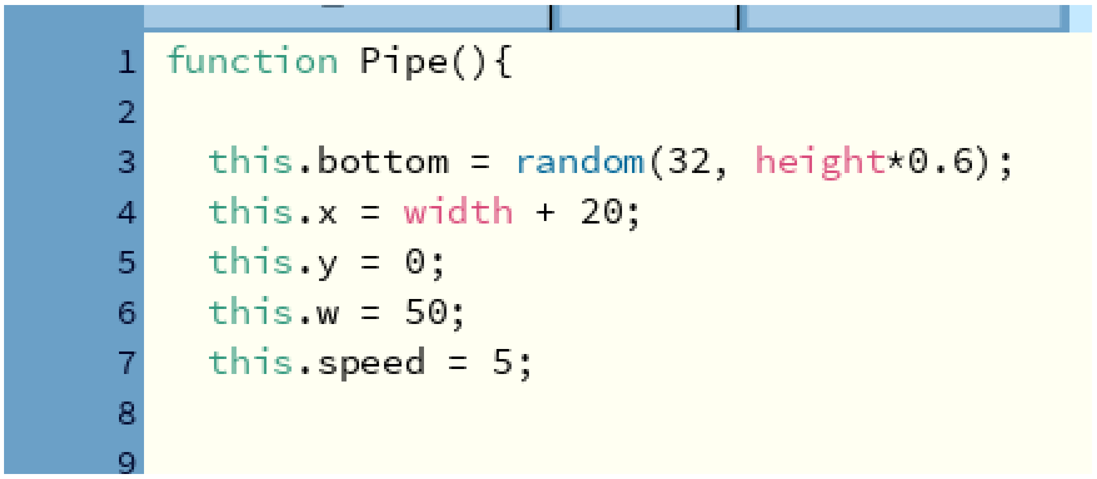
```
function Pipe(){



}
```

To start off we need to make some certain variables to make the “pipes”. The “pipes” are going to be made up of rectangles, so just like the “bird” we’re going to need an **x** coordinate, a **y** coordinate, a width, height. We are also going to need a speed at which they will be moving. Also just like the “bird” variables, all of these will start with a **this.**.

We’re going to want the pipes to start off the right of the screen so the **this.x** variable will be equal to **width** plus 20, and the “pipe” starts at the top of the screen making the **this.y** value equal to zero. The **this.w** value of the “pipe”, for this tutorial, will be equal to 50.

Now the height variable is going to be a bit different. To be easy to remember things, for this tutorial, the height variable will be named **this.bottom**. We want the height of the “pipes” to differ from each other, so to do that we need to set the **this.bottom** variable equal to **random(min, max)**, where the **(min)** value is the minimum number the value can be and **(max)** is the maximum number the value can be. For the best experience, we’re going to want the “pipes” to range from a bit off the top and a bit off the bottom, but remember we still need space for the bottom “pipe”. So, for this tutorial, the **(min)** is set to 32 and the **(max)** is **height** multiplied by 0.6. Then the **this.speed** variable will be set to 5.

This is what the variables should resemble.



```
function Pipe(){

  this.bottom = random(32, height*0.6);
  this.x = width + 20;
  this.y = 0;
  this.w = 50;
  this.speed = 5;


```

Now that we have the needed variables to create the “pipes”, we need a function where we show the “pipes”. Like in the “**bird**” tab, we need to make a **this.show()** function looking like “this.show = function() { }” where the { } brackets hold the code that will run when the **this.show()** is executed. In the **this.show()** function, we want to fill the “pipes” with color, and make the different rectangles that make up the “pipes”. To do that we need to put the **fill(r,g,b,a)** function first. To make the “pipes” green as shown in the beginning, the values should be **(100,200,75)**, but these values can be whatever you’d like.

Getting the “pipes” to look symmetrical will take a bit of thinking, but it’s doable. The “pipes” are made up of four different rectangles, so we’re going to need to call four different **rect(x,y,w,h)** functions. Where the **(x)** value is the rectangle’s **x** coordinate, the **(y)** value is the rectangle’s **y** coordinate, and the **(w)** and **(h)** values are the rectangle’s width and height. Put these different functions all on different lines. The first two, or the long bases of the “pipes”, are easy to make. The last two, or the heads of the “pipes” can be a bit tricky to get right.

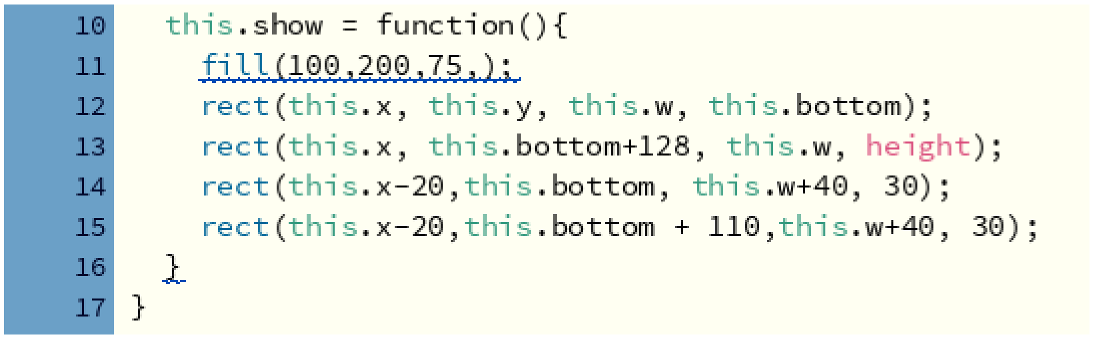
****Disclaimer**** Rectangles are made from the top left corner going down to the bottom right corner.

The first **rect(x)** value will be the **this.x** variable we made earlier, the **rect(y)** value will be the **this.y** variable we made earlier, the **rect(w)** value will be the **this.w** variable we made earlier, and the **rect(h)** value will be the **this.bottom** variable we made earlier.

The second **rect(x)** will be the **this.x** variable we made earlier. Now the **y** coordinate needs to be at a certain distance from the bottom of the first rectangle, so the **(y)** value will be the **this.bottom** variable plus 128. The **(w)** variable will be the **this.w** variable we made earlier, and the height needs to go to the very bottom of the screen making the **(h)** variable the **height** variable built-in to the program.

The third **rect(x)** needs to start a bit to the left of the first rectangle’s **x**, so the **(x)** value needs to be the **this.x** variable minus 20. The **y** coordinate needs to be at the same level as the bottom of the first rectangle, so the **(y)** value needs to be the **this.bottom** variable we made earlier. The width needs to be wider than the first rectangle, but since it starts 20 to the left of the first rectangle it needs to finish at 20 to the right of the first rectangle, making the **(w)** value the **this.w** variable plus 40. The **(h)** value needs to be 30, for best experience.

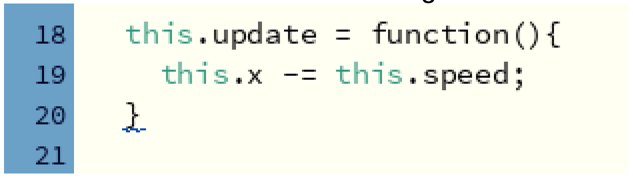
The final **rect(x)** needs to start at the same spot as the third’s, making the **(x)** value the **this.x** variable minus 20. The **y** coordinate needs to be a certain distance from the top of the second rectangle, making the **(y)** value the **this.bottom** variable we made earlier plus 110. The **(w)** and **(h)** values are the same as the third rectangle’s **(w)** and **(h)** values.



```
  this.show = function(){
    fill(100,200,75,);
    rect(this.x, this.y, this.w, this.bottom);
    rect(this.x, this.bottom+128, this.w, height);
    rect(this.x-20,this.bottom, this.w+40, 30);
    rect(this.x-20,this.bottom + 110,this.w+40, 30);
  }
}
```

If the program is run right now, nothing will happen since the “pipes” or rectangles are being made off the screen, so we need to make them appear on screen and move to the left.

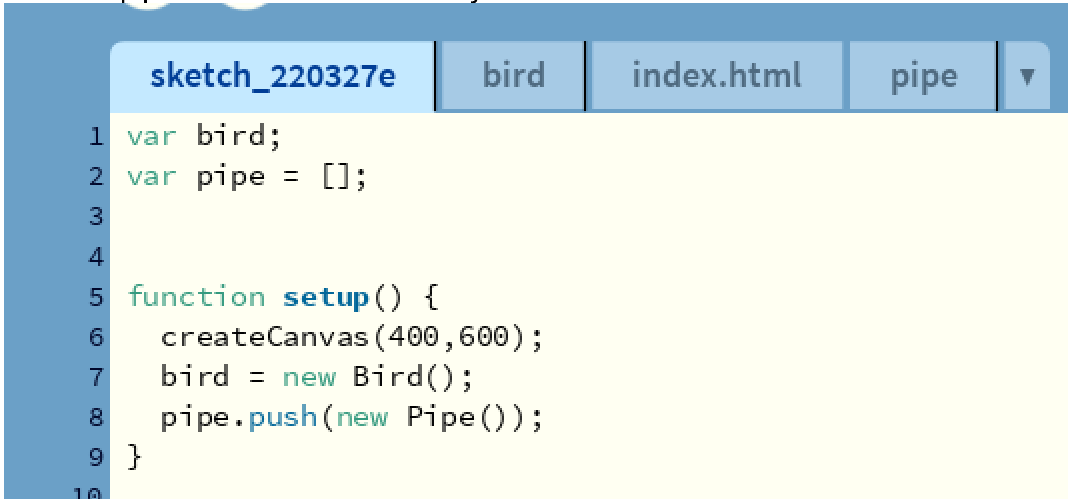
To start off we need to make an **this.update()** function in the “**pipe**” tab, after the **this.show()** function, just like we did in the “**bird**” tab, and in that function we just need to subtract the **this.speed** variable from the **this.x** variable since moving to the left is a negative value.



```
  this.update = function(){
    this.x -= this.speed;
  }

```

If we were to make the “pipe” move at this moment, only one “pipe” will be made, but we need multiple “pipes” to be made forever. To do that, we need to make an *array* to hold the number of “pipes” being made.

To start, we need to make an *array* variable called **pipe** in the main sketch tab, after the **bird** variable, just like we did for the **bird** variable. Instead of just declaring the variable, we need to set it equal to an unspecified *array*, looking like “[ ]”. That will look like “var pipe = [ ]” making the number inside the *array* undefined, able to make as many pipes as we want. To create the “pipes” is a bit different than how we did the “bird”. We need to call the **push(condition)** function where the condition is what we want to force to be made. The condition will look like the *new* **Bird()** variable, but we will use *new* **Pipe()**. We want to make sure that the **push()** function is attached to the **pipe** variable, so this variable will look like “pipe.push(new Pipe())” and this will force a new pipe to be made in the array.

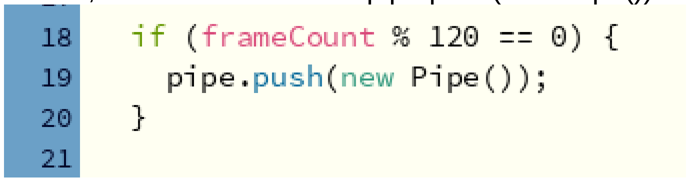


```
var bird;
var pipe = [];


function setup() {
  createCanvas(400,600);
  bird = new Bird();
  pipe.push(new Pipe());
}
```

Still, no pipes are being made. Now we need to make an **if()** statement, that if true, it will call the "pipe.push(new Pipe())" variable to make a new "pipe" in the *array*. Also, we need to make a **for()** loop to make a new "pipe" every time the **for()** loop is executed.

****Disclaimer**** Make sure that these are in the **draw()** function right after the **bird.show()** and **bird.update()** functions.

An easy way to make sure that the "pipes" are created at equal lengths, we can check if the certain frame you're on is divisible by a certain number equaling zero. There is a specific function apart of the p5.js library called **frameCount**. So, in an **if()** statement we need to see if "frameCount % 120 == 0", making sure that the frame you are on is divisible by 120 equaling a remainder of 0. If that is true, we need to call the "pipe.push(new Pipe())" variable.



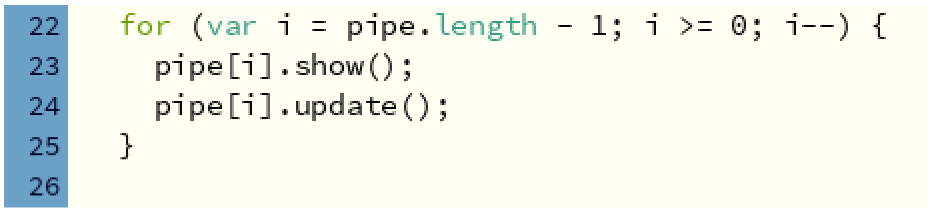
```
if (frameCount % 120 == 0) {
  pipe.push(new Pipe());
}
```

Now that we have our condition for making a new "pipe" in the *array*, we need to make a loop that will show and update the "pipe" made. This will be a **for(s1 ; s2 ; s3)** loop, looking like "for(){ }", where **(s1)** is usually the initialized variable used in the **for()** loop, **(s2)** is the condition to check if *true* or *false*, **(s3)** is usually the increments or decrements the **(s1)** variable by 1. In the block for the **for()** loop, add the **this.show** and **this.update** function we made in the "**pipe**" tab, and these will look like the ones we used for **bird.show()** and **bird.update()** but make sure the **pipe** has index 'i', [i], at the end of it to make sure it's a part of the *array*.

****Disclaimer**** The variable for a **for()** loop that is usually used is **i** and between each statement use a semicolon.

(s1) needs to be set equal to "pipe.length – 1"
(s2) needs to be greater than or equal to zero, "i >= 0"
(s3) needs to decrement by 1, in shorthand that looks like "i - - "

****Disclaimer**** Make sure that this **for()** loop is right after the **if()** statement we just made in the main sketch tab.



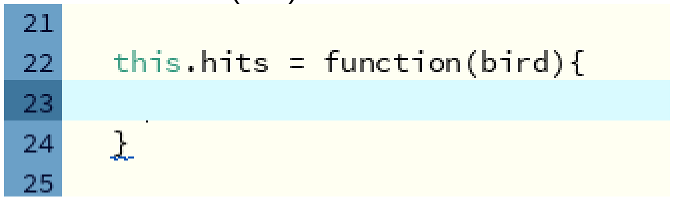
```
for (var i = pipe.length - 1; i >= 0; i--) {
  pipe[i].show();
  pipe[i].update();
}
```

If the program is run, you will finally see the "pipes" being made continuously while moving to the left. Now we need to see if the "bird" is hitting one of the "pipes".

Section 4.3: Collision Detection

At this point, we have the “bird” able to fly and the “pipes” being generated at random heights and moving at a constant speed towards the “bird”. But the “bird” just goes through the “pipes” without any indication that it did, so we have to check if the “bird” is in the same place as any part of the “pipes”.

For this, we have to create a **function()** in the “**pipe”** tab after the **this.update()** function. The function needs to return a **bird** value, so the keyword **bird** needs to be inside of the parentheses of **this.update**. For this tutorial, I have named the function **this.hits**, so the function will look like “this.hits = function(bird)”.

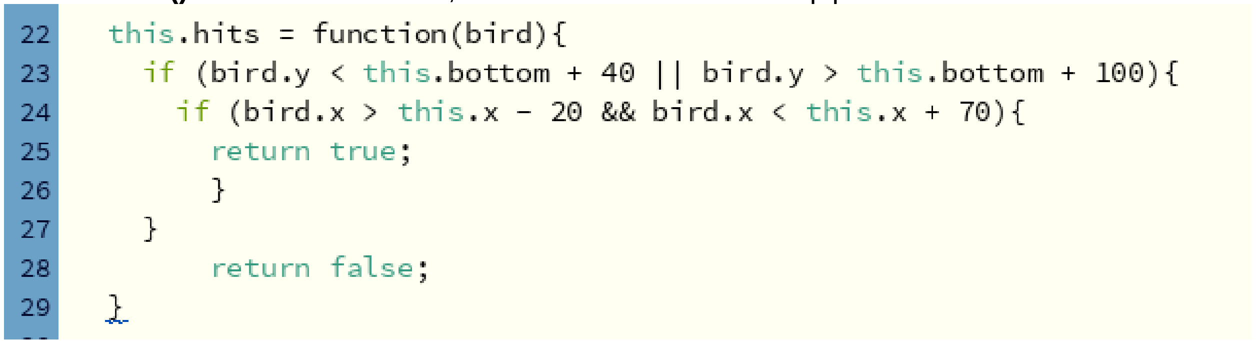


```
this.hits = function(bird){

}
```

****Disclaimer**** Since we going to be referring to the variables from the “**bird**” tab, any variable needing to be used from the tab replace the “**this.**” with “**bird.**”.

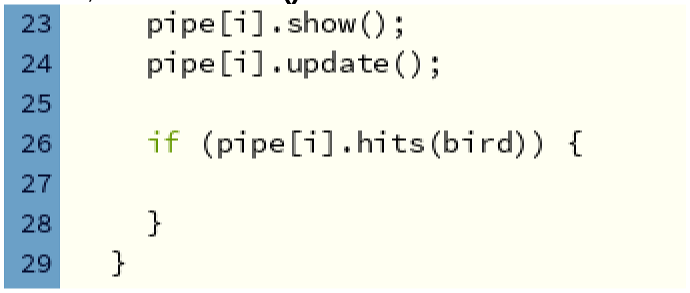
Inside this function we’re going to need to make an **if()** statement with a nested **if()** statement, an **if()** statement inside another **if()** statement. For the first **if()** statement condition, we need to check if the “bird’s” **this.y** value is less than the bottom of the third rectangle’s **(y)** value or, “ || “ is the “or” operator in a condition, if the “bird’s” **this.y** value is greater than the fourth rectangle’s **(y)** value. Right after the } bracket for the **if()** statement we need to write “return false”, because if the first **if()** statement is false that means the “bird” is in between the “pipes” and not going to hit the “pipes”. Inside of the { } brackets for the first **if()** statement, we need to make a nested **if()** statement.

For the nested **if()** statement’s condition, we need to check if the “bird’s” **this.x** value is greater than the third rectangle’s left side value, “this.x – 20”, and, “&&” is the “and” operator, if the “bird’s” **this.x** value is less than the third rectangle’s right side value, “this.x + 70”. Inside of the { } brackets for the nested **if()** statement we are going to need to write “return true”, because if both of the **if()** statements are true, then the “bird” has hit the “pipe”.



```
this.hits = function(bird){
  if (bird.y < this.bottom + 40 || bird.y > this.bottom + 100){
    if (bird.x > this.x - 20 && bird.x < this.x + 70){
      return true;
      }
  }
      return false;
}
```

Now that we are able to test if the “bird” has hit the “pipe”, we need some indication that it has hit the “pipe”. In the main sketch tab, we need to add an **if()** statement inside the **for()** loop right after the **pipe[i].update** function. The condition for the **if()** statement is going to be the **pipe.hits(bird)** function from the “**pipe”** tab, and like the **pipe[i].update** function make sure that the **pipe.hit(bird)** function also has the index **i** attached to the word **pipe**. This will see if the **if()** statement is *true* or *false*, to see if the **if()** statement will execute.

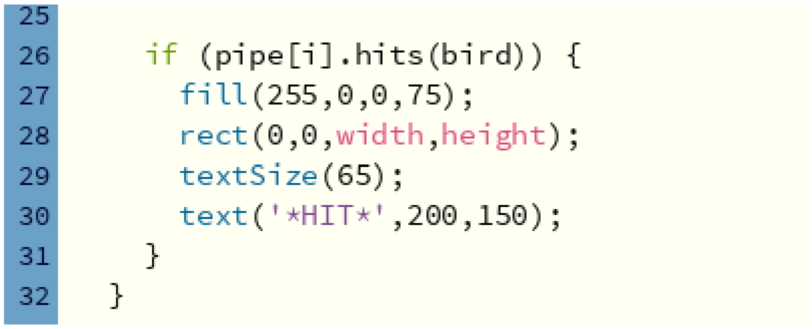


```
    pipe[i].show();
    pipe[i].update();

    if (pipe[i].hits(bird)) {

    }
  }
```

Inside the **if()** statement we’re going to add a 75% transparent red rectangle that covers the whole canvas, and text that says “*HIT*”. Just like making the “pipes” and coloring them green, we need to first put the **fill(r,g,b,a)** function. To make the color 75% transparent and red, the **(r)** value needs to be 255, the **(g)** and **(b)** value need to zero, and the **(a)** value needs to be 75. Right after the **fill()** function, we need to use the **rect(x,y,w,h)** function to make the rectangle now. The **(x)** and **(y)** value needs to be zero to have it start at the top left corner of the canvas, the **(w)** value needs to be **width**, and the **(h)** value needs to be **height**.

****Disclaimer** width** and **height** are the built-in variables from the p5.js library.

To make text appear on screen, we need to use the **textSize(theSize)** function and the **text(str,x,y)** function. Right after the **rect()** function, we need to put the **textSize()** function and make the **(theSize)** value to 65. Then we need to put the **text()** function with the **(str)** value as ‘*HIT*’, and make sure the value is in single quotation marks. The **(x)** value can be 200, and the **(y)** value can be 150.



```

    if (pipe[i].hits(bird)) {
      fill(255,0,0,75);
      rect(0,0,width,height);
      textSize(65);
      text('*HIT*',200,150);
    }
  }
```

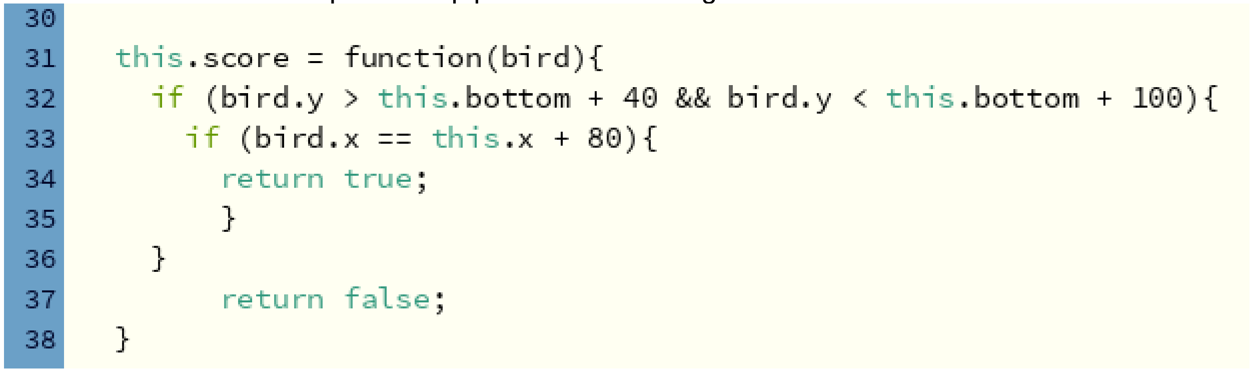
Now we have an indication that a “pipe” has been hit by the “bird”. In the mobile game when the “bird” passes through a pipe, without hitting it, a point is added to the score. Knowing that, we need to make a scoring system and even a high score system.

Section 4.4: Scoring

To make the scoring system, it’s going to be almost exactly like making the **this.hits()** function. The scoring function will be made right after the **this.hits()** function. The naming of the scoring function is going to be exactly like the **this.hits()** function, with the condition being **(bird)**, but instead of naming it **this.hits()**, we’ll name it **this.score()**. It will also have an **if()** statement with a nested **if()** statement inside.

The condition of the first **if()** statement will see if the “bird’s” **this.y** value is greater than the bottom of the third rectangle’s **(y)** value and if the “bird’s” **this.y** value is less than the top of the fourth rectangle’s **(y)** value. Right after the } bracket of this **if()** statement we need to write “return false”. If this **if()** statement is *false*, that means the “bird” is not in between the “pipes” and is going to hit a pipe. Now we need to add the nested **if()** statement in the first **if()** statement.

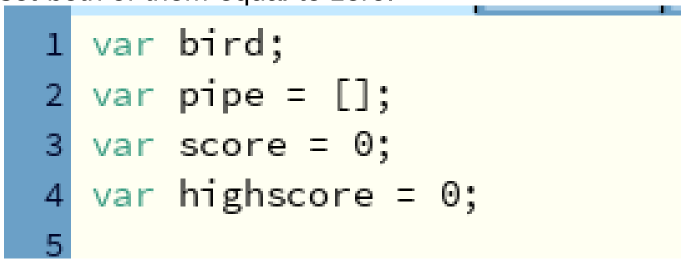
The condition for the nested **if()** statement needs to see if the “bird’s” **this.x** value is equal, “==” is the “equal” operator, to the right most side of the third rectangle’s **(x)** value. Inside the nested **if()** statement we need to write “return true”. After this, if both the **if()** statements are *true*, that means the “bird” made it past the “pipes” without hitting it.



```

  this.score = function(bird){
    if (bird.y > this.bottom + 40 && bird.y < this.bottom + 100){
      if (bird.x == this.x + 80){
        return true;
        }
    }
        return false;
  }
```

Now that we’re able to see if the “bird” has passed through the “pipe” without hitting it, we need to add the actual score and high score system. To do that we need to add two variables, an **if()** statement with a nested **if()** statement, and text for the score and high score in the main sketch tab.

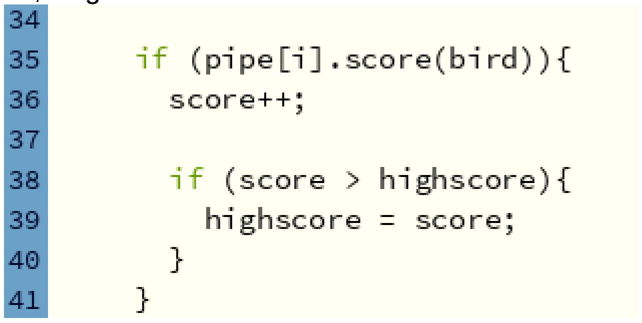
We’ll make the two variables just like the **bird** variable. Instead, we’ll name one **score** and the other **highscore** and set both of them equal to zero.



```
var bird;
var pipe = [];
var score = 0;
var highscore = 0;

```

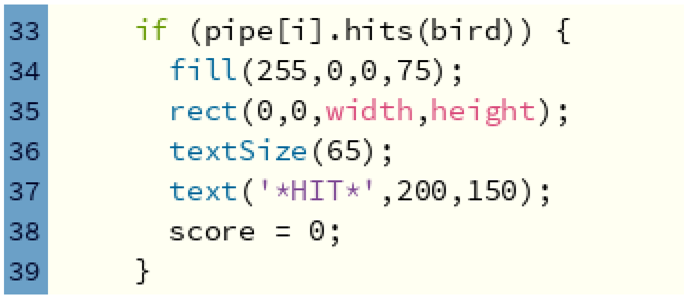
Right after the **if()** statement with the **pipe[i].hit()** function as its condition, we need to add another **if()** statement with the new **this.score()** function, which we just made in the “**pipe”** tab, as its condition. This condition will look exactly like the **pipe[i].hit()** function, just replace **hit** with **score**. Inside the **if()** statement we need to add the shorthand of adding 1 to **score**, “score++”.

Right after the shorthand, we need to add the nested **if()** statement with its condition checking to see if **score** is greater than **highscore**. Inside that **if()** statement, we need to set the value of **score** equal to **highscore**, “highscore = score”.



```
    if (pipe[i].score(bird)){
      score++;

      if (score > highscore){
        highscore = score;
      }
    }
```

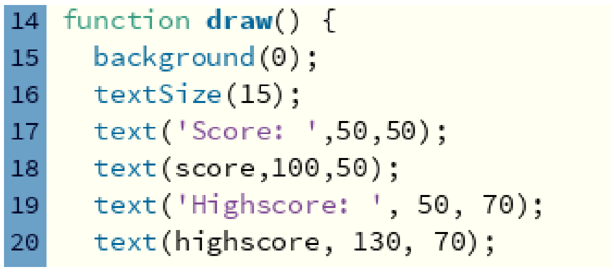
Before we move on, we need to know what happens to the **score** value if the “bird” hits a “pipe”. So, in the **pipe[i].hit()** function, right after the **text()** function, we need to set the value of **score** equal to zero, “score = 0”.



```
    if (pipe[i].hits(bird)) {
      fill(255,0,0,75);
      rect(0,0,width,height);
      textSize(65);
      text('*HIT*',200,150);
      score = 0;
    }
```

Now we need to add the text to see our score and high score. To do this we’re going to need to add one **textSize()** function, and four different **text()** functions right after the **background()** function at the beginning of the **draw()** function. For this tutorial, all the values have been tuned to make it look decent.

The text size will be 15. The first **text()** function conditions will be **(‘Score: ’, 50, 50)**.
The second **text()** function conditions will be **(score, 100, 50)** and **(score)** is the variable **score**.
The third **text()** function conditions will be **(‘Highscore: ’, 50, 70)**.
The final **text()** function conditions will be **(highscore, 130, 70)** and **(highscore)** is the variable **highscore**.



```
function draw() {
  background(0);
  textSize(15);
  text('Score: ',50,50);
  text(score,100,50);
  text('Highscore: ', 50, 70);
  text(highscore, 130, 70);
```

Congrats, you have successfully made a simplistic version of the mobile game “Flappy Bird”!

****Disclaimer****

All the values that were entered can be changed to make the game run to your likings.