

02 - Getting Started

Getting Started with JavaScript and p5.js

WE'LL COVER THE FOLLOWING ^

- Gentle Introduction to JavaScript
- Getting Started With p5.js
- More About Functions
- Coordinates in p5.js
- Summary
- Practice
- Installing p5.js Editor

Let's get this started. Before we start using p5.js to learn JavaScript, we will see couple of things on the fundamentals of JavaScript.

If you want to be testing the code on your own system you might want to install the p5.js code editor. I will provide the instructions for installing the editor at the end of this chapter.

You can also find the code examples that we will be writing throughout this course at the [Github repository](#).

Gentle Introduction to JavaScript

We can write something as simple as `1 + 1` to the screen. This is a valid JavaScript code which adds these two numbers together. If we are to execute this code, by pressing the Play button, we still won't see anything. This is kind of disappointing because we would have at least expected to see the result of this calculation.

To be able to see the results of JavaScript operations on the screen we can use

a **function** called **console.log()** .

A **function** is a programming structure that contains other code inside it that is written to perform a specific action. Functions allow us to perform complex operations by just calling them with their defined function name. When we are calling a function - which we can also refer to as *executing* the function - we would write its name, in this case **console.log**, and place brackets next to it. If the function requires an input to perform its functionality, then we would provide that input inside the brackets just like we are doing in this example.

console.log is a built-in JavaScript function that displays - or logs - the given value inside the console below the editor. When I say built-in, it means that most JavaScript execution environments would have this function. For example web browsers have a section in their interfaces called *console*, which we can access through the developer tools. p5.js also has a section that is called *console* as well below the editing area.

We can also have user defined functions that we can create for ourselves which won't be available to anyone else until we somehow share it with other people. Libraries such as p5.js have bunch of functions of their own. We will be using p5.js functions to draw shapes to the screen and create all kinds of interactive and animated visuals. We will dwell more into the concept of functions later on but for now know that there is this function that comes with JavaScript called **console.log** that accepts a value and displays that value inside the console underneath the editor. Initially the other functions that we will be learning about won't have a dot in their name. **console.log** is a bit different in that sense, but the reasons for the 'dot' usage will be explained later.

Let's add a couple of more **console.log** statements into our code.

```
console.log(1 + 1)
console.log(5 + 10)
console.log(213 * 63)
console.log(321314543265 + 342516463155)
```



When we execute this code, we will see the following results being displayed

inside the console.

```
2
15
856632392
663831006420
```

One takeaway should be that code executes from top to bottom. There are some programming structures that alter this flow, but we will see them later on. Another takeaway should be that computers don't mind working with large numbers. We can throw hard operations at them that would take days for a human to perform.

In the last **console.log** statement we have two ridiculously large numbers. What if we wanted to use the resulting number from that operation and subtract 10 from it on the next line. Right now to be able to do this we have to type that number again.

```
console.log(321314543265 + 342516463155 - 10)
```

This is obviously very wasteful. But luckily another thing that computers are great at is storing and remembering values. Therefore we can create something called a **variable** to hold on to that value. In programming languages, a variable is a name that refers to a value. So we can use a variable name to refer to that value instead of typing the value again. Here is how that works:

```
var bigNumber = 321314543265 + 342516463155
console.log(bigNumber)
console.log(bigNumber - 10)
```



We are creating a variable called **bigNumber** by using the **var** keyword. **var** is the keyword that we need to use whenever we are creating a variable. After the **var** keyword, we are giving this variable a name, which in this case is **bigNumber**.

It is important to choose a variable name that makes sense for the current

context. In this example, this might not matter too much, but as our programs get more complex, meaningful variable names can help us understand what's going on when reading our code. So naming this kind of a variable that holds a large number as **cat** wouldn't make much sense and can confuse other people that might read our code. It might even confuse us if we are to come back to our code couple of months later. Programmers always strive to make their code as readable as possible.

Once this variable is declared, we can assign a value to it by using the equal operator. This might seem unusual at first. In Math, the equal operator is used to signify equality in between two values. Here we are using it to do a value assignment to a variable. It takes the value on the right-hand side of the operation and assigns it to the variable on the left-hand side. This is a pretty common procedure that exists in many programming languages.

Now that we have a variable that points to a value, we can use this variable name in operations instead of the value itself. As mentioned earlier, it is good to have variable names that make sense. There are also some rules that govern what we can, and can't use as variable names. For example, we can't use some of the special characters such as dashes or exclamation marks or use a space character inside our variable names. Another restriction is that we can't use certain JavaScript reserved names as variable names; we can't call our variable **var** as this name is already in use by JavaScript. If we tried to use **var** as a variable name; as in `var var = 5`, JavaScript would **throw** an error.

This mention of rules might be making you uneasy at this point. After all programming is supposed to be fun right? But don't worry the reserved name list is relatively short, so you don't need to memorize it. And as you learn more of the language, you would also develop a better sense as to which names to avoid.

Regarding rules, there is another rule that should be mentioned. JavaScript needs us to place semicolons after each statement. If we don't do this our program can still work but might fail in certain edge conditions that can be hard to identify later on. So it is a good idea to use semicolons after every statement even though it means a bit more work on our part. The above code should actually be written like this:

```
console.log(1 + 1);
```

```
console.log(5 + 10);
console.log(213 * 63);
var bigNumber = 321314543265 + 342516463155;

console.log(bigNumber);
console.log(bigNumber - 10);
```



Notice that doing `bigNumber - 10` wouldn't change the initial value of the **bigNumber** variable. In this following example, the **console.log** statement would still output 10.

```
var x = 10;
x + 5;
console.log(x);
```



If we want to change the value of a variable, then we need to assign a new value to it.

```
var bigNumber = 321314543265 + 342516463155;
console.log(bigNumber);
bigNumber = 3;
console.log(bigNumber);
```



In this example, the **console.log** would display the value `3` because we override the initial value with another value on line 3.

There is this concept of data types in JavaScript (and in other languages as well) to differentiate in between different kinds of values. These numbers that we have been using are of a data type called **Number**. There is another data type called **Strings** that is used to represent textual information.

In JavaScript, we can't just write a word and expect it to represent data. For example, we want to **console.log** the word **hello**. If we do this right now, we will notice that we are getting an error. JavaScript doesn't understand what **hello** means. It assumes that it is a variable that is not defined yet.

```
console.log(hello);
```



But what if we wanted to actually input the word **hello** to the computer? There are programs out there that work with textual data, which needs to process a given name or address, etc. In that case we can provide the data using quotation marks which mean that we are providing the value as a **string**.

```
console.log('hello');
```



JavaScript is not complaining this time. Anytime we are dealing with textual data we need to place it in quotation marks, this would make it registered as a string. And when I say textual data, it can be numbers as well. A string can consist of numeric values:

```
console.log('1234');
```



In that case, they are not treated as Mathematical numbers that we can perform Math operations with, but just as text.

We can perform operations on strings, but it doesn't yield the same result as when we would perform those operations using numbers. We can actually add two strings together:

```
console.log('hello' + 'world');
```



And this will just combine these two words together. And when I say we can't perform Math operations with strings that contain numeric values, this is what it meant:

```
console.log('1' + '1');
```



In this case, the numeric values are not treated as numbers but as strings, and they are not summed together but combined. This act of combining strings is commonly referred to as *concatenation* operation in programming.

String might sound like a weird name choice but it refers to ‘string of characters’. So a string is actually a collection of individual characters as far as the computer is concerned. We can define strings by using either single quotation ' or double quotation marks " but we have to finish the string with the same symbol we choose to start defining with. Also in our programs, we shouldn't use one type of quotation mark for one string and another for a different one. Consistency is very important when developing programs.

One other thing that's worth mentioning before wrapping up this section is the concept of **comments**. Comments allow us to write things into our programs that won't get executed by the computer. Such as:

```
// various examples. (this is a comment)
console.log(1 + 1);
console.log(5 + 10);
console.log(213 * 63);
var bigNumber = 321314543265 + 342516463155;
console.log(bigNumber);
console.log(bigNumber - 10);
```



The line that starts with double slashes gets ignored by JavaScript. Double slashes allow us to comment on a single line; if we needed to comment on multiple lines, we would either need to use double slashes at the beginning of each line or use the `/*` symbol like:

```
// various examples
// disabling the first 3 lines by using multiline comments:
/*
    console.log(1 + 1);
    console.log(5 + 10);
    console.log(213 * 63);
*/
```



```
var bigNumber = 321314543265 + 342516463155;  
console.log(bigNumber);  
console.log(bigNumber - 10);
```



Believe it or not, this is enough of a JavaScript primer to get us started with using p5.js. If you are using the code editor, click on the **New Project** button to be able to get a new editor window which has the template that we would be using for our p5.js code.

Getting Started With p5.js

What we see when we start a new project in the p5.js code editor are two function declarations with the names: **setup** and **draw**.

```
function setup() {  
  
}  
  
function draw() {  
  
}
```

These two function declarations need to be made for pretty much every p5.js program that we would write. p5.js finds these function definitions in our code and executes whatever is written inside them. But there is a difference in between how these functions are executed.

The block inside the **setup** function, the area in between the curly brackets, is the place where we will be writing the code that is to be executed for the initialization of our program. Code written inside the **setup** function is executed only once before the draw function.

```
function setup() {  
    // write your code for setup function inside these curly brackets  
}
```

The **draw** function is where the real magic happens. Any code that is written inside the **draw** function repeatedly executed by p5.js. This allows us to create all sorts of animated and interactive works.

p5.js makes sure to execute the **setup** function before the **draw** function. And to reiterate, p5.js executes the **setup** function only once but the **draw** function over and over again (actually close to 60 times a second). And this is how we can create interactive and animated content using p5.js.

We can actually see this in action by placing **console.log** statements at different places in our code. Place a **console.log()** statement inside the **setup** function, inside the **draw** function and outside both of these functions using different values.

```
function setup() {  
  console.log('setup');  
}  
  
function draw() {  
  console.log('draw');  
}  
  
console.log('hello');
```

Let's execute this code and immediately try to stop it. We would notice that the message **hello** is displayed as the very first thing. This is an expected behaviour. A function call that we have should be executed by JavaScript. What is rather unexpected is that **setup** and **draw** functions get executed as well. This is unexpected because these are only function declarations, they define the behaviour of a function, but we still need to execute these functions to be able to use it.

This means that if we were just using JavaScript we would need to call the **setup** and **draw** functions explicitly in order to have the console.log messages inside them to be displayed:

```
setup();  
draw();  
console.log('hello');
```

But we don't need to do this using the p5.js library. Because of how the p5.js library is architected, it looks for function declarations with the name **setup** and **draw**, and executes these functions for us. The reason why p5.js takes control of the execution of these functions is that it executes them in a very

specific manner.

p5.js executes the **setup** function only once and then goes on to execute the draw function in a repeated manner such that if we don't stop the process it will just keep working forever. This is a very standard behaviour with any graphical interface - think of the web browser, the games you play, or the operating system you interface with. These are just programs that continuously work - and display to the screen - until we explicitly close them. This is why p5.js creates an execution loop for the **draw** function so that things will persist on the screen instead of appearing for a second and then disappearing.

More About Functions

Let's talk more about functions because they will be the building blocks of the programs that we will be writing.

Function names are usually verbs. They represent the specific action that can be performed by executing that function. Hypothetically speaking, we might have a function called **drawCat** which when called can draw a cat to the screen.

```
drawCat();
```

Though this is not hypothetical at all as I actually created a cat drawing function that is called **drawCat** for this chapter. We are free to create whatever functions we want to create in JavaScript, and that gives us immense power when programming applications.



OK, to be fair, this function doesn't do a great job in drawing a cat.

To use a function, we call it by its name and then put brackets next to it to have the function executed. Sometimes functions, depending on how they are created or defined, are parameterized. This means they can accept input values which would effect the outcome of a function. For example a **drawCat** function might get a number input, which would determine the size of the cat that is drawn. Or maybe the number input determines the amount of cats that would be drawn to the screen. It really depends on how this function is constructed.

In our example, this function that I created can get an input which allows us to change the size of the cat head that gets drawn on the screen.

```
drawCat(2);
```





Unfortunately, p5.js doesn't come with a **drawCat** function - I had to create my own - but it has lots of other useful functions that allow us to perform complicated tasks in an easy manner. To be able to do anything using the p5.js library, we will be using the functions that come with it, that are coded by the smart people who created this library.

Here is a function from p5.js library that probably all the sketches we will be writing will require: **createCanvas** function. What **createCanvas** function does is that it creates a drawing area, canvas, inside the web page for us to work. But for this function to work, we need to provide it with two comma separated values. A width and height for the drawing area. We should be calling the **createCanvas** function inside the **setup** function because it only needs to get executed once and it needs to be executed before we can do any

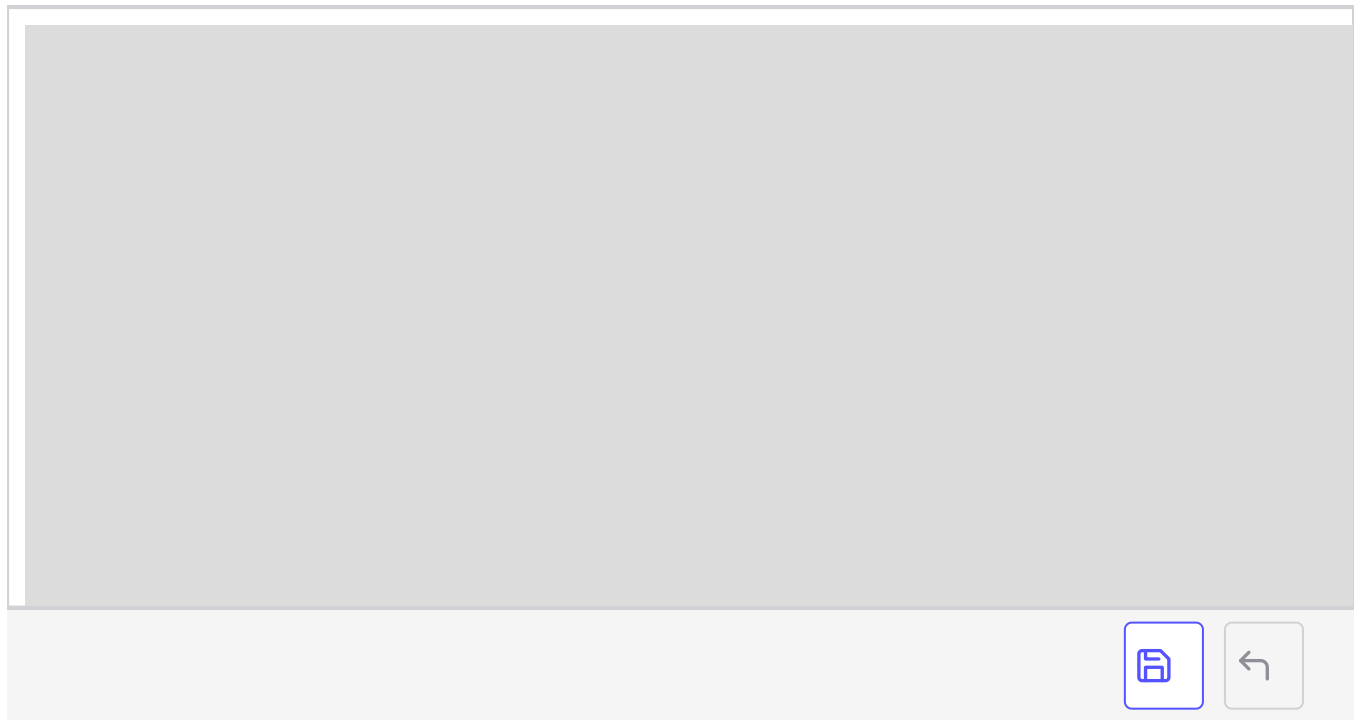
needs to get executed once and it needs to be executed before we can do any drawing.

Let's provide this function with the values **800** and **300** and execute our sketch to see what's happening. It seems like not much have changed, but the size of the browser window that gets launched seems to have increased. It is now using the dimensions that we have provided. Let's change the dimensions again to see the window size updating. (You might have to try this inside the p5.js editor to see a change. Since it's a white screen we are drawing the change might not be observable inside this web page.)

Output
JavaScript
HTML
<div></div>
<div></div>

There is another function that we will frequently be using which is called **background**. **background** function sets the color of the canvas using the given value. We will look at how color values are represented in p5.js in another chapter, but for now, we can just provide this function with the value **(220,220,220)** to see the background become light gray.

Output
JavaScript
HTML



As we can see again, the code is executed from top to bottom. p5.js first creates the canvas for us and then sets the background to be gray.

It is worth emphasizing this once more: the **setup** and **draw** are function definitions that we need for p5.js to work correctly. Our job when we are using p5.js is to determine what is placed inside these functions that are executed by p5.js. This is due to how p5.js is architected. The creators of p5.js wanted to make sure some of the code we will be writing will only be executed once for initialization and setup purposes, while some will be executed all the time for drawing, animation and interactivity purposes.

We used functions that come with p5.js library such as **createCanvas** and **background** inside these function definitions. These functions are already defined by someone else so we don't actually know what code is contained inside them. But we don't really need to have this knowledge anyway since all we care about is knowing what they do and how to use them.

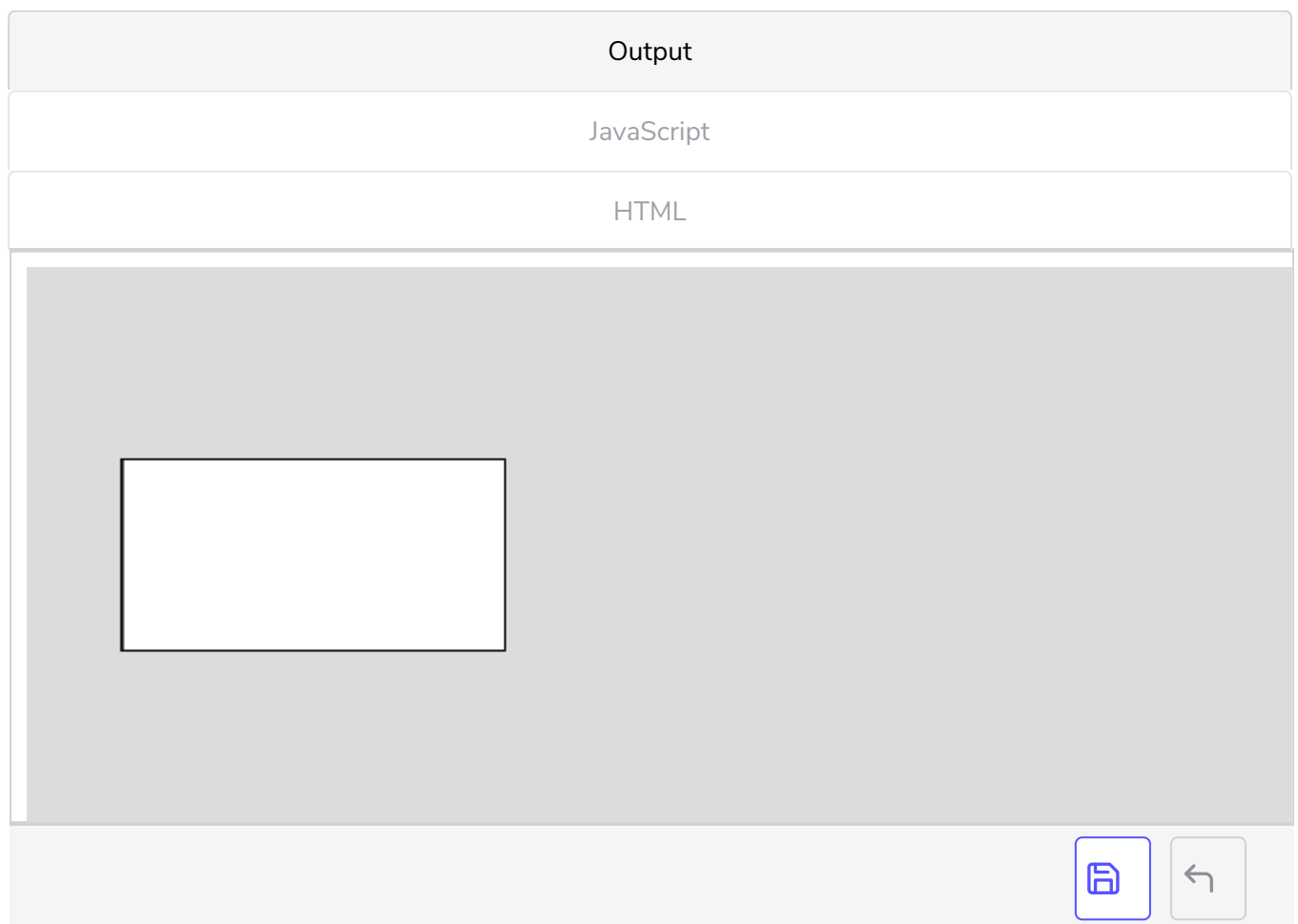
Functions allow us to perform complicated tasks in an easy manner. By using the **createCanvas** function we don't need to know what kind of work goes into creating a canvas element in a page. These details are hidden away, abstracted, from us. We just need to know how to call this function to make it work for us.

Finally, we will be calling one more function, this time inside the **draw**

function definition, to draw a rectangle on the page.

To draw a rectangle we will be utilizing a function called **rect**. **rect** function requires us to provide it with four input values. The x and y position of the rectangle inside the canvas drawing area, and the width and height values for the rectangle.

Without knowing anything about how the coordinates work in p5.js, we will just provide this function with the x value of 50, y of 100, the width of 200 and height of 100.



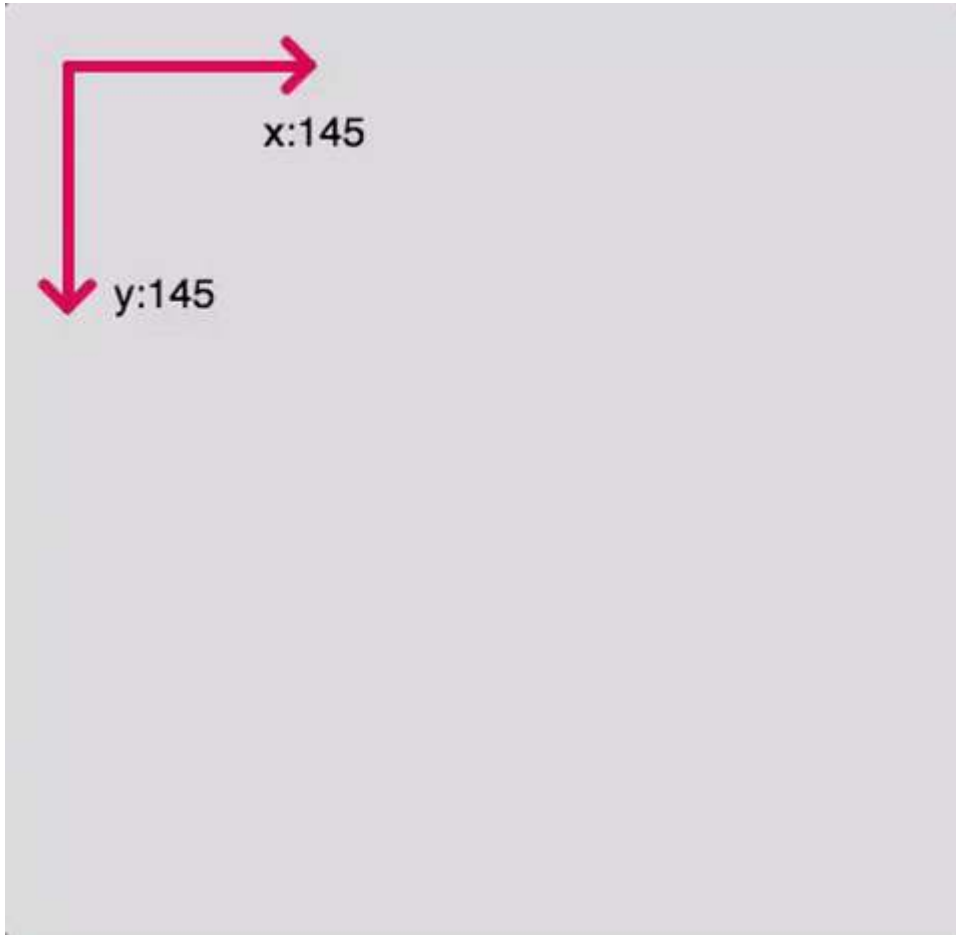
By calling this function we drew our first shape to the screen!

Coordinates in p5.js


At this point, let's take some time to explain how the coordinate system works in p5.js.

To locate any point on a flat surface we use a two-axis coordinate system. The vertical axis is called the Y-axis, and the horizontal one is the X-axis. The point where these two axes meet is called the *origin*. In canvas, where we draw our shapes, the origin point is at the top left of the canvas. From there below, the Y

shapes, the origin point is at the top left of the canvas. From there below, the Y values increases and to the right, the X values increases.

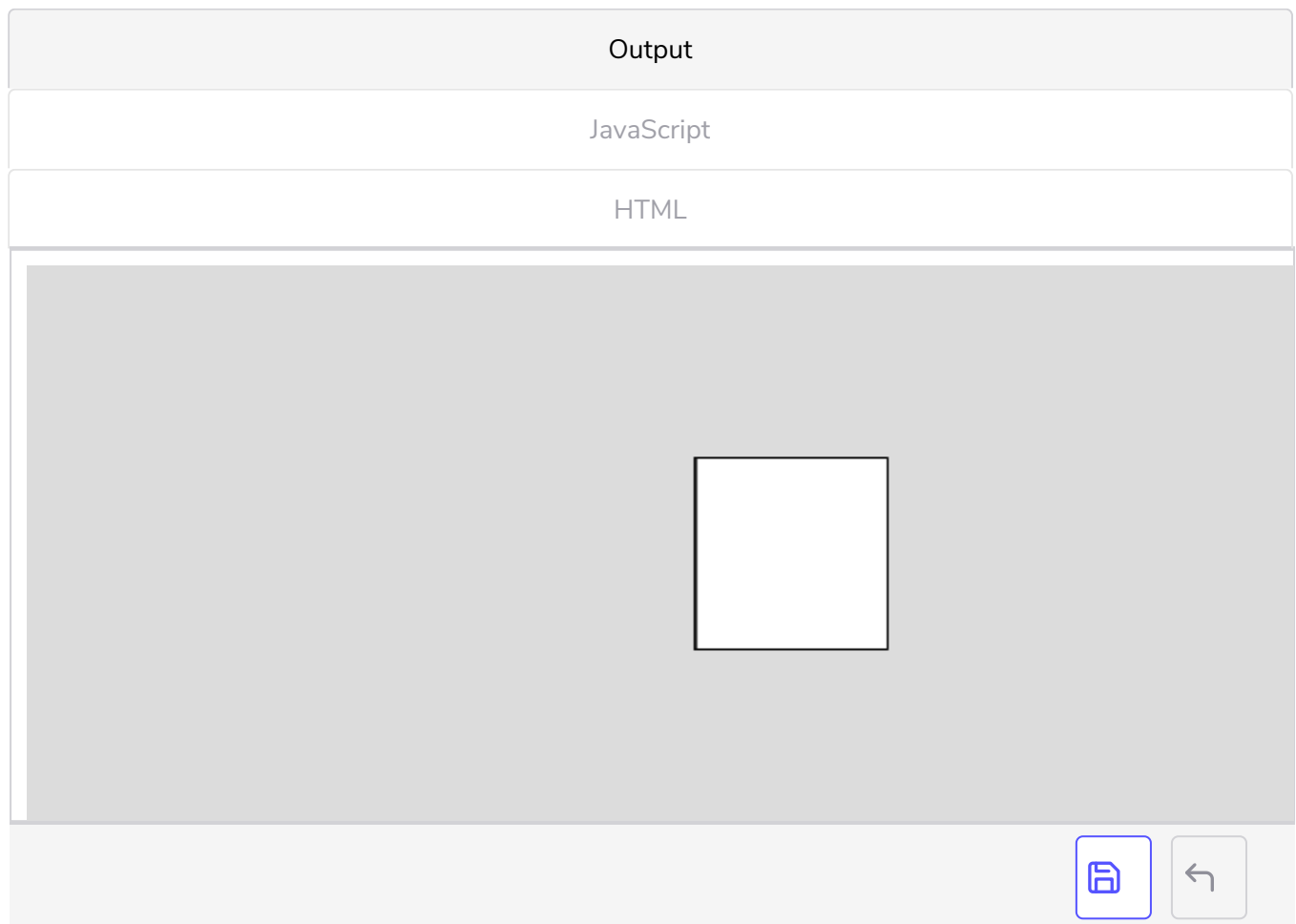


Feel free to tinker with the values we have previously provided to the rectangle shape to see this concept in action. One thing that you might notice is that even when we provide half of the canvas width and half of the height as the coordinates for the rectangle, it doesn't seem like the rectangle is drawn at the center of the canvas. The reason for that is by default; the rectangle shape is being drawn from its top left corner.

Output
JavaScript
HTML


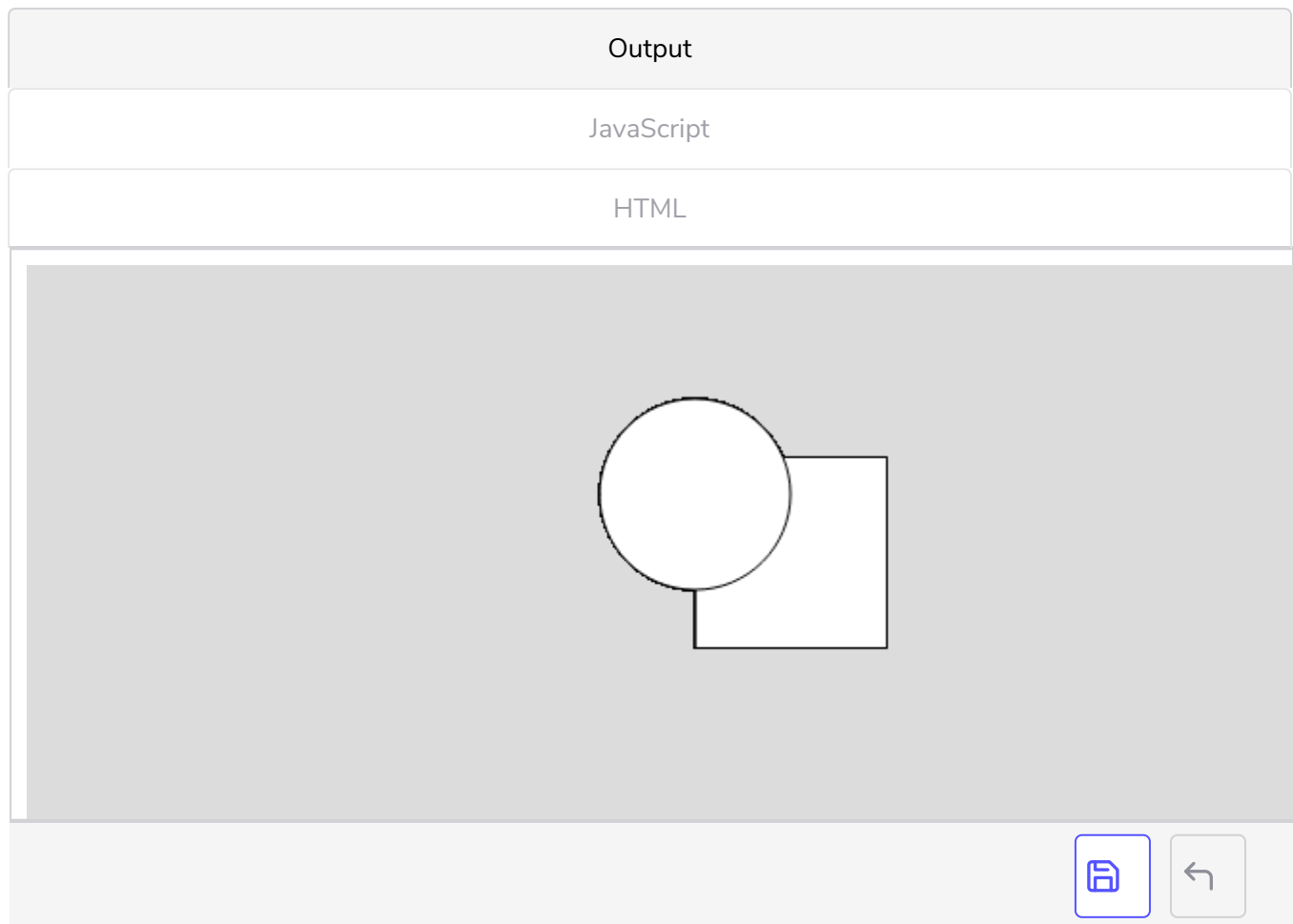


If this is not the behaviour that you want, we can make a call to another p5.js function called **rectMode** and provide it with the value **CENTER** to change how rectangles are drawn in our program. Since this function is more like a setup & initialization related function we will be placing it under the **setup** function definition.



There is also an **ellipse** function in p5.js to draw circular shapes. How **ellipse** works is very similar to the **rect** function. First, two arguments are x and y coordinates, the 3rd argument is the horizontal radius, and the 4th one is the vertical radius. So to be able to draw a circle with the **ellipse** function, we need to provide equal horizontal and vertical radius values to it.

If you are experimenting with drawing these shapes to the screen, you might have noticed at this point that, whenever a shape function is called, it draws itself on top of the previous shapes. We can change the order of the function calls to affect the stacking order of the shapes.



One more drawing function that I want to introduce is the **line** function. As the name implies, the **line** function draws a line to the screen. We need to provide four arguments to the **line** function, the starting x and y coordinates and the ending x and y coordinates. Play with the line function a bit; it would give you a good sense of how the coordinate system works in p5.js. You can for example try drawing an X that spans the entire canvas.

Summary

In this chapter we made a quick start with using p5.js and actually drew shapes on the screen.

We have seen that we need to write our code in two function definition blocks that go with the name **setup** and **draw**. Anything that only needs to be executed once is placed under the **setup** function, and anything that we might like to animate or interact with goes into the **draw** function. Writing our code into these two functions is something that p5.js requires us to do. It is not a general programming principle, convention or anything like that. We could have been using a different library that doesn't require this kind of a structuring to our code. This requirement has to do with how p5.js is

architected as a library. We will need to start all of our p5.js sketches with these two function definitions.

Code like this which needs to be written repetitively with little or no alteration is called **boilerplate** code. Having lots of boilerplate is never a good thing since we would find ourselves having to repeat our work a lot but in this case the amount of boilerplate is very manageable.

Inside these function definitions we made use of functions that come with p5.js library such as **createCanvas**, **background** and some shape functions such as **rect**. As mentioned earlier, functions are general programming structures that allow us to bundle code together for reusability purposes. Functions also abstract away a great deal of complexity from us. We don't need to know how a function works; we just need to know how to use it. We can absolutely have no idea how the **createCanvas** actually creates a canvas element inside a web page. It doesn't matter as long as we know how to use this function. Think of driving a car; we don't necessarily need to know how an internal combustion engine works to be able to drive it. We just need to know how to interface with the car using the steering wheel, the pedals, etc. The similar idea applies to the functions as well.

Later on, we will be creating our functions as well to manage the complexity of our programs and to create reusable pieces of code.

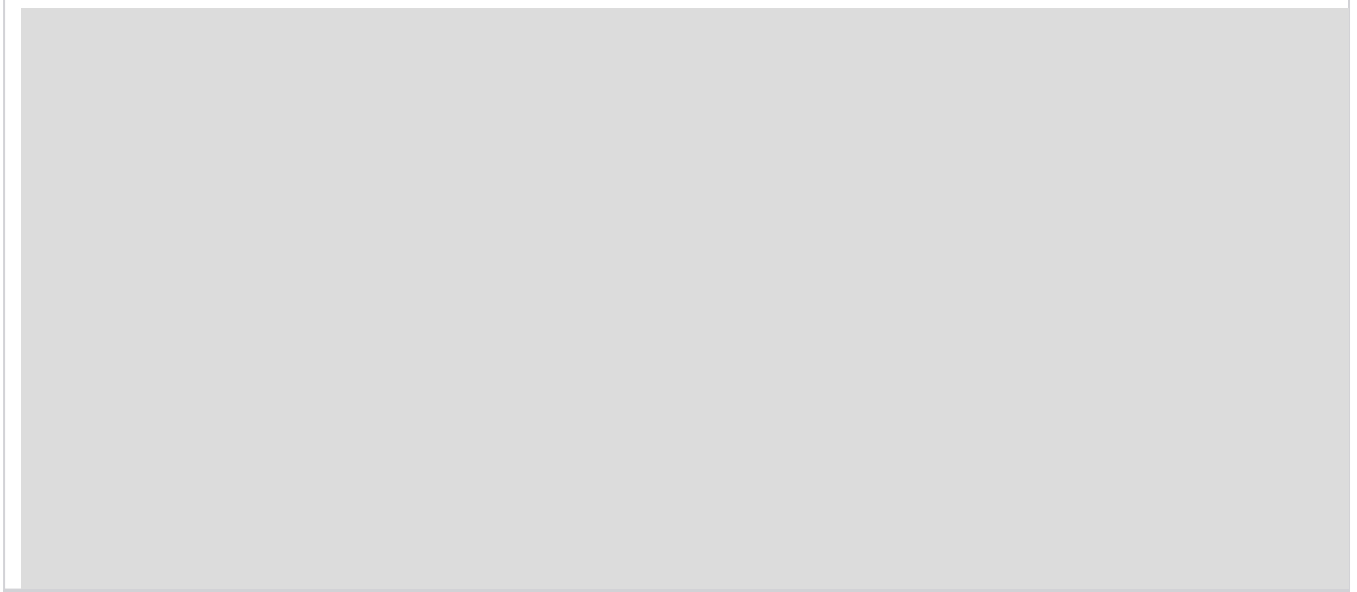
Practice

Try to recreate the image below.



Output

JavaScript



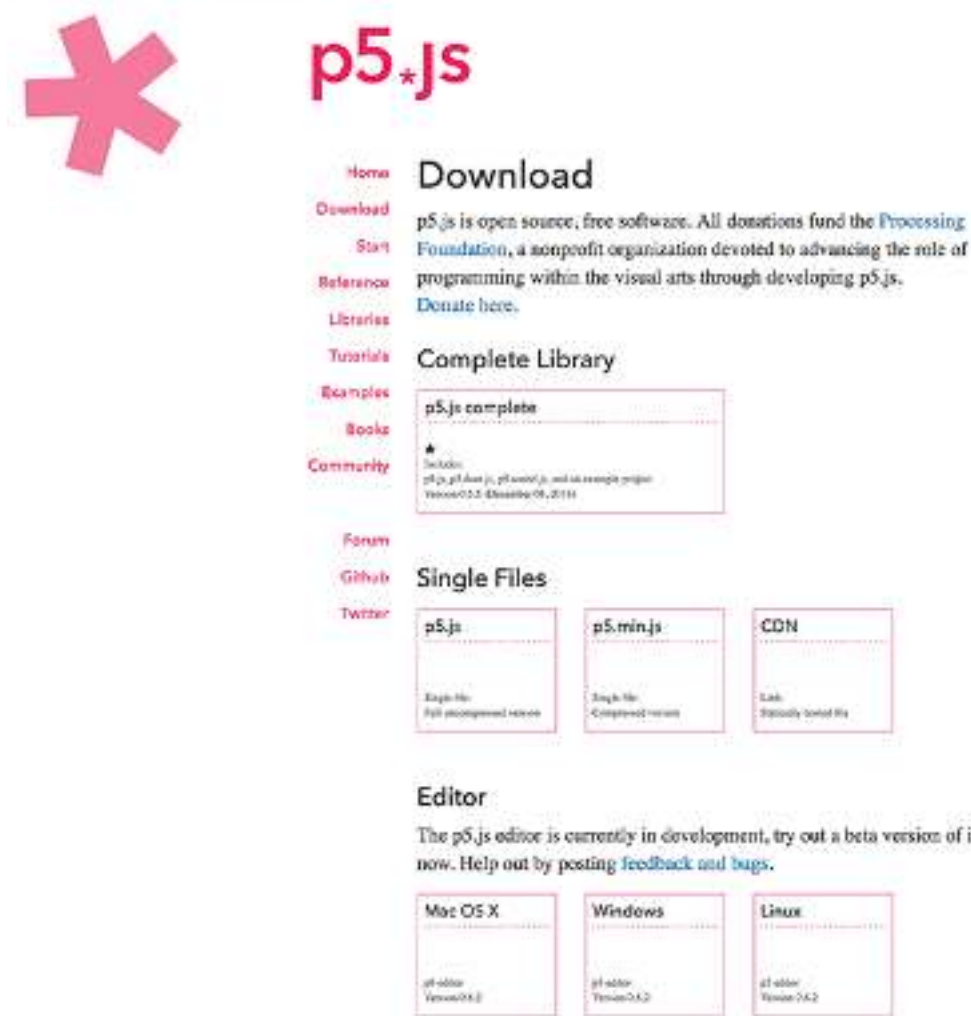
Console

 Clear

Installing p5.js Editor

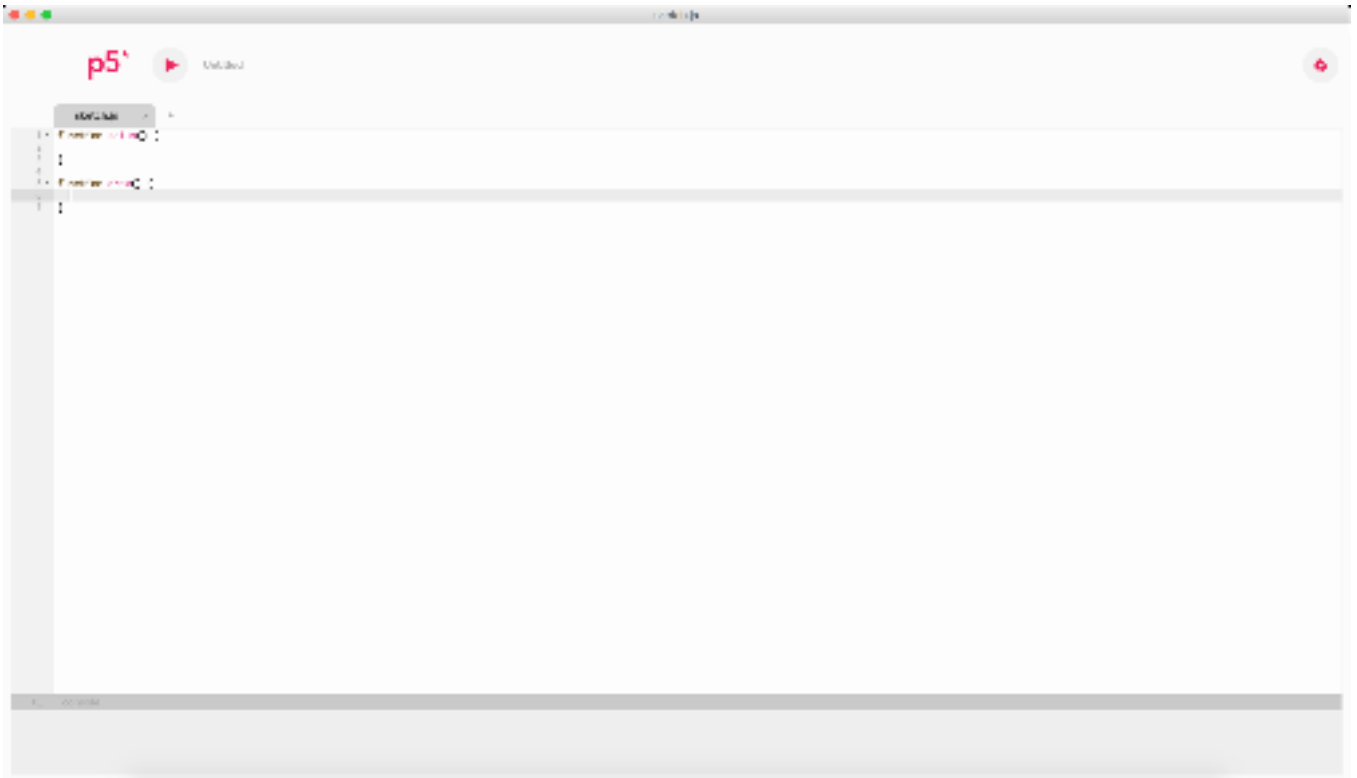
To install the p5.js editor we can go to their web page and download the editor for our operating system, whether we are using Mac, Windows or Linux, by following the links on the download page.

[p5.js](#) | [download](#)



We can also use an online editor to code with p5.js. One of the easy to use options that is available at the time of the preparation of this course can be found in this link: [p5.js online editor - alpha](#)

Once we launch the editor, we will see an area where we can write our code. A code editor is pretty similar to a text editor, like **Notepad** or **Word**, but it has special features that make coding much easier such as highlighting of special words for a given programming language, which in this case that language is JavaScript.



How this specific editor works is that, whenever we have some code ready to be executed we will press the *play* button at the top of the page. This play button will launch another page that displays the visual result of our code - that is if our code was doing any drawing to the screen. Online Editor works in a similar manner but it shows the results on the right hand-side panel instead of launching a new page. Pressing the Play button at this point wouldn't do much as we didn't write any code that draws to the screen. We will just see an empty screen get generated. But as we can see, this editor has some code already written into it. This code that we see is needed for almost all the p5.js programs that we will be writing so it is included here for our convenience.

```
function setup() {  
  
}  
  
function draw() {  
  
}
```

