09 - Objects

JavaScript Objects

Introduction to Objects

JavaScript uses a data structure called **Object** that helps to organize data together. There are a couple of ways of creating an **object** in JavaScript, one way of doing it would be by using the curly brackets.

```
var colors = {};
```

These curly brackets are called *Object Initializer*. They create an empty object. We hold a reference to the object by using the variable **colors**.

Now, we can add properties to this **colors** object by providing the desired property names after a dot. This is called *dot notation*. We will also assign values to these newly created properties.

```
var colors = {};
colors.black = 0;
colors.darkGray = 55;
colors.gray = 125;
colors.lightGray = 175;
colors.white = 255;
console.log(colors);
```

If we are to look at the object at this point by using **console.log**, we would see it looks something like this.

```
{"black":0,"darkGray":55,"gray":125,"lightGray":175,"white":255}
```

We could also have created an object with the same properties from get go, by providing these properties inside curly brackets.

```
var colors = {
  black: 0,
  darkGray: 55,
  gray: 125,
  lightGray: 175,
  white: 255,
};
console.log(colors);
```

Objects are basically *key-value* pairs. Each key stores a *value* and each *key-value* pair make up a *property* on an object.

To be able to access a value on an object, we can again use the dot notation.

```
console.log(colors.gray);
```

For some situations, the dot notation doesn't work. An example is when we use numbers as our key values in an object. In that case, we can use square brackets to access values instead. Like:

```
colors[1]; // Assuming you were using numbers instead of color names as key values.
```

What do you think this above expression will return if we were to **console.log** it? We would get the value **undefined** as the key 1 doesn't exist in our current **colors** object.

We can also define functions as values for keys in an object. In that case, the resulting property would be referred to as a *method*.

Continuing from our **colors** object, let's define a method inside that object called **paintItBlack** which would make the background color to be black.

```
    var colors = {
        black: 0,
        darkGray: 55,
        gray: 125,
        lightGray: 175,
        white: 255,
        paintItBlack: function() {
            background(this.black);
        }
    };
```

Here is a p5.js code that makes use of this object

Output
JavaScript
HTML

In this example, we are initializing the **colors** variable outside the scope of **setup** and **draw** functions and then create its content inside the **setup** function. After all, we only need the content to be created once. And then we call its **paintItBlack** method if the **frameCount** is bigger than 120 which would happen after two seconds with default settings. (Remember the default value for the **frameRate** is 60, which means that, approximately, 60 frames are rendered per second.)

To be able to use a key that is defined inside the object from within, we need

to be able to refer to the object itself. In JavaScript, there is a keyword called

this which allows us to do so. Using the **this** keyword we can refer to the keys that are defined on the object itself.

```
paintItBlack: function() {
    background(this.black);
}
```

Once we defined a method on the object, we can call the method by accessing it using the dot notation. Since we are executing a function in this instance, we need to have brackets after the function name.

```
colors.paintItBlack();
```

The concept of objects in JavaScript (or in other languages that implement objects) exists so that we can model after real world objects or concepts. Just like how real world objects have properties and sometimes a behaviour, programming language objects can have properties that describe what they are and methods that specify how they behave.

Let me give you an example of a programming language object that models after a real world concept. We will create an object called **circle**. This circle object will have several properties defining how it looks and also it will have several methods that describe how it behaves.

```
var circle = {
    x: width/2,
    y: height/2,
    size: 50,
};
```

This **circle** object has an **x** and **y** property that defines its coordinates and a **size** property that defines its size. We will also create a method on it, a property that is a function, which defines a certain behaviour. In this case, the defined behaviour will be to draw the circle to the screen.

```
var circle = {
    x: width/2,
    y: height/2,
    size: 50
```

```
        size: 50,
        draw: function() {
            ellipse(this.x, this.y, this.size, this.size);
        },
    };
```

In this example, we are again using the **this** keyword to be able to access the properties on an object. **this** keyword basically refers to the object itself and allows us to call the object's properties while inside the object. We can now draw this circle on the screen by using the circle.draw() method call.

```
    circle.draw();
```

You must be thinking to yourself: this was the most convoluted thing ever. Because why should we ever need to draw a circle this way when we can just call a function to draw it on screen?

```
    ellipse(width/2, height/2, 50, 50);
```

We are just getting started, though, let's add another method to the circle called **grow**, which would increase the size of the circle by one unit whenever it's called.

```
    var circle = {
        x: width/2,
        y: height/2,
        size: 50,
        draw: function() {
            ellipse(this.x, this.y, this.size, this.size);
        },
        grow: function() {
            if (this.size < 200) {
                this.size += 1;
            }
        },
    };
```

Now, If we are to call this function inside the **draw** function, we would see our circle keep growing as the **draw** function is continuously being called by p5.js. Here is the whole example.

Output
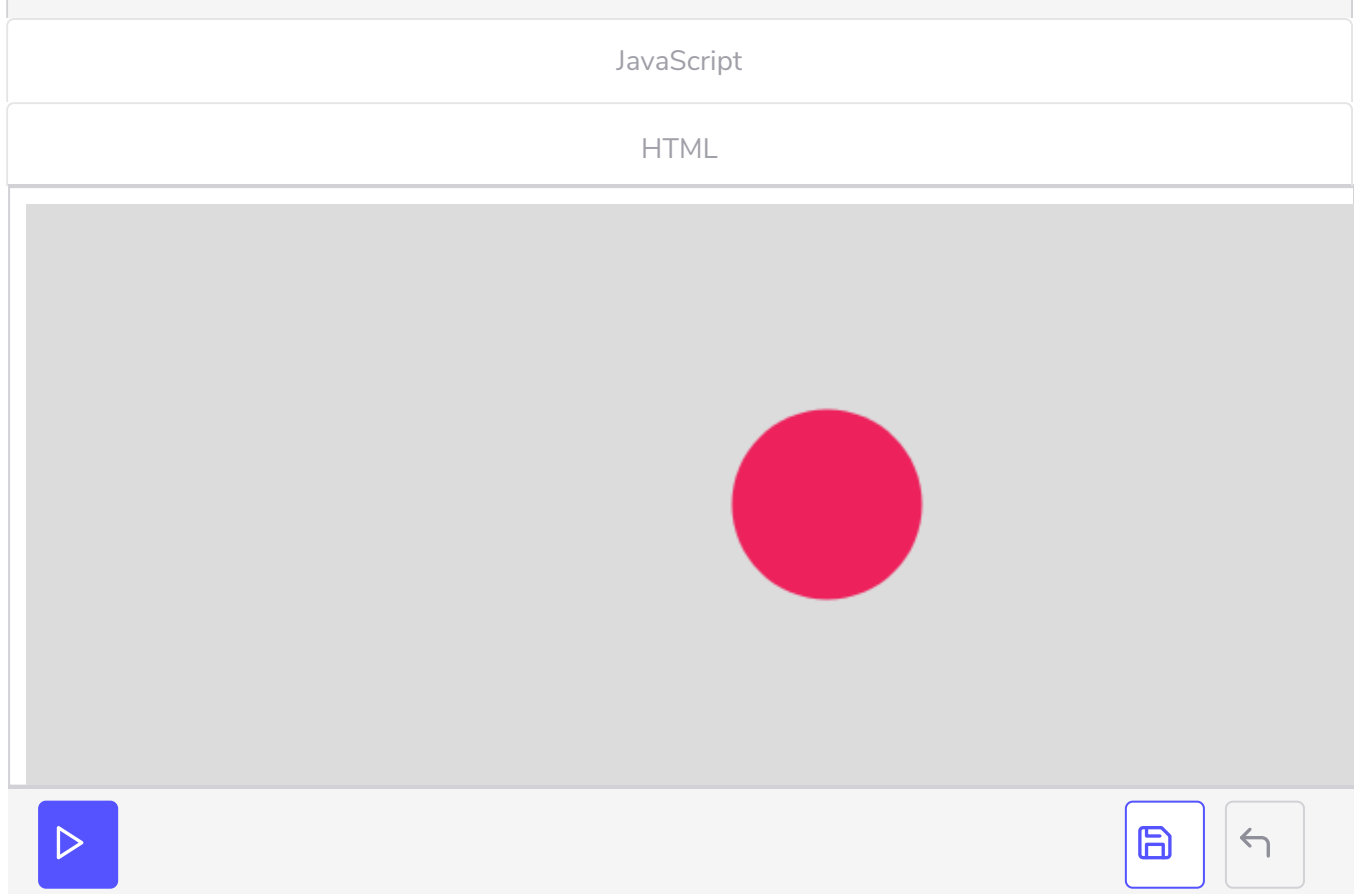
As mentioned earlier, the usage of objects is about code organization. We don't have separate functions that manipulate the circle, but a **circle** object that carries those functions and its properties within itself. This can make our code easier to reason about in certain cases.

Constructor Function

There is another way of creating objects in JavaScript, and that is by using functions. The declarations we would be making inside an object creating function is very similar to what we were doing when working with an *object initializer*.

```javascript
var Circle = function() {
    this.x = width/2;
    this.y = height/2;
    this.size = 50;
    this.draw = function() {
        ellipse(this.x, this.y, this.size, this.size);
    };
    this.grow = function() {
        if (this.size < 200) {
            this.size += 1;
        }
    };
};
```

An object creating function is called a **constructor function**. We can think of it as a template or a blueprint for creating new objects that derive their properties from this constructor function.

Let me show an example to better explain what I mean. Say we want to have a circle just like in the previous examples that exhibit the behaviour that is defined by this Circle constructor. In this case, we will not use this constructor function directly for our purposes, but we will use it to instantiate a new circle that is modelled after this template function.

```js
var myCircle = new Circle();
```

We used the **Circle** constructor function and the **new** keyword to create a *new* instance of a circle called **myCircle** that gets its properties from the constructor function. Basically, the **new** keyword allows us to create a new instance of an object from a Constructor function. We can think of the **Circle** constructor function as a blueprint and the **myCircle** as an actual circle built from that blueprint. Now we can draw this newly created circle to the screen by calling the draw method on it.
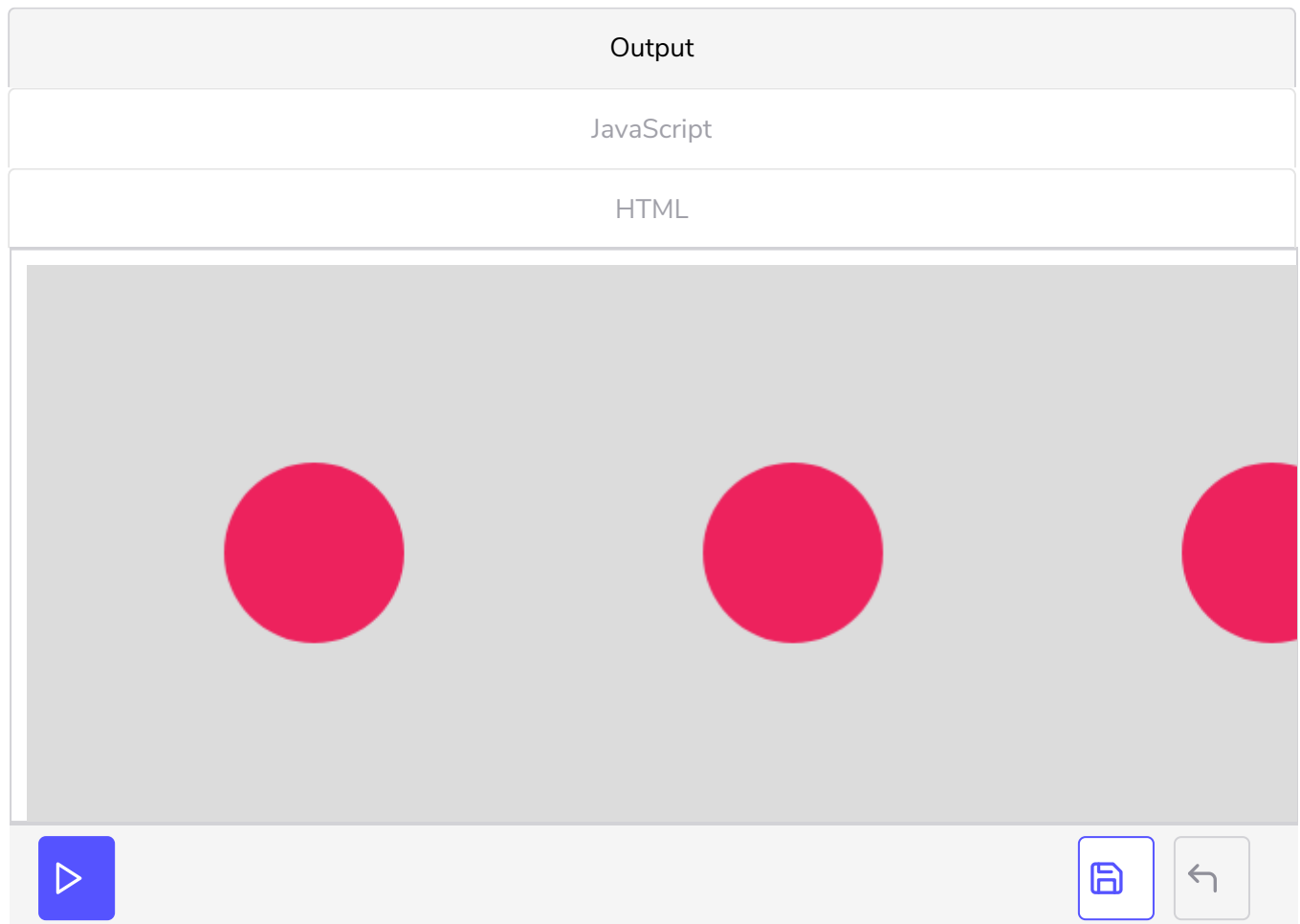
```js
myCircle.draw();
```

Here is the full example:

Output
JavaScript
HTML

The beauty of this method is that we can keep creating new circles from the same blueprint. And since these circles are separate entities -instances-, they can have different properties from each other. Let's see an example of that.

Output
JavaScript
HTML

In this example, we are creating three variables outside the p5.js functions called **circle_1**, **circle_2** and **circle_3**. These variables are created outside the p5.js functions so that they would be in scope for both of those functions.

We are making these variables to be Circle instances by assigning the **Circle** constructor function using the **new** keyword to them. Now that we have three separate circle objects, we can change their properties in the **draw** function, and get different results from each one of them.

One thing that is important to note is how we are using a function name that starts with a capital letter for the constructor function. We use a capital letter to remind ourselves and others that this function is a constructor function and needs to be called with the **new** keyword.

It is important to call a constructor function with the **new** keyword. If we don't it won't work properly, as the **this** keyword inside the constructor function wouldn't refer to the instance object but the global object.

Usage of the capital letter is not a rule but a convention. No one forces us to do it. But it is expected that we follow this convention since not realizing a function is a constructor function and then calling it without the **new** keyword will have unintended consequences.

Summary

In this chapter, we learned about the JavaScript objects. Simply put, objects are a way of organizing code.

We learned about the dot notation and the square bracket notation that are used to access the properties on an object.

We also learned about the **this** keyword which allows us to refer to the properties of the object from within the object itself.

We learned about two ways of creating objects. One of them is using an **object initializer**, and the other one is using **constructor functions**. While we are creating a single object using an object initializer, constructor functions act as a blueprint from which we can create many object instances using the **new** keyword.

There is a whole programming paradigm called *Object Oriented Programming* across different programming languages that leverage the usage of objects for code organization and clarity. Using p5.js, we don't necessarily need to create objects to organize our code, but I wanted to introduce objects for two reasons:

First of all, objects are a fundamental part of the JavaScript Language. If you would like to learn more about the language at a later time, you would need to get comfortable with how objects work.

Secondly, JavaScript has other built-in structures that are based on objects that we will be using so it was important for us to familiarize ourselves more with objects.

Now if you are just starting out, the takeaway from this chapter should be to know that there is this data structure that exists in JavaScript called **Objects** that helps with organizing code. It also allows for code to be represented in a certain way that makes it easier to work with and reason about in certain cases. As we are starting out, we won't make use of objects all that much, but it is important to know of them for your later studies.