

10 - Arrays

JavaScript Arrays

WE'LL COVER THE FOLLOWING ^

- Summary
- Practice

Arrays are another useful data structure in JavaScript, that is based on **Objects**, which makes certain operations much easier to perform.

Arrays are a sequential collection of data that is stored with a numbered index. Remember that we use curly brackets to create an empty object. We can create an empty array in a similar fashion by using square brackets.

```
var arr = [];
```

In this example, we created an empty array and used a variable called **arr** to store that array. Now if we wanted to add elements to this array, we can use the **push** method that array objects have.

```
var arr = [];  
arr.push(1);  
arr.push("hello world");  
arr.push({"name": "value"});  
console.log(arr);
```



In this example, we are pushing three new values to the previously empty array. In the first line, we are pushing in a value of **number** type, in the second line we are pushing a **string** type into the array and in the third line we are pushing an **object** type into it.

Now if we are to look at the contents of the array by using **console.log**, we will see something like this on screen:

```
[1, "hello world", {"name": "value"}]
```

Notice how we used different data types and objects to populate the Array. Arrays can contain any object, even other arrays. Just like how it is with JavaScript objects, we can populate an Array at creation time by providing desired values inside square brackets using a comma to separate them. Let's create an array with four numbers in it.

```
var arr = [15, 40, 243, 53];  
console.log(arr);
```



We can use the index number property that automatically gets generated to access the individual items in an array. One thing to know though is the indices that refer to the stored items in an array starts counting from 0. To access an individual item in an array, we can type the variable name that the array is stored in, and then use the index number in square brackets to refer to that item at that index. The number 0, will refer to the first item in the array - which is 15 -, the index number 1 will be for the second item, etc...

```
var arr = [15, 40, 243, 53];  
var firstItem = arr[0];  
console.log(firstItem);
```



If we try to access an item that doesn't exist, we will get an **undefined** value. Which makes sense because that item is not defined. Remember objects also return an **undefined** value when we try to access a property that doesn't exist.

Let's see how the array data structure can simplify things when building

programs. We will start with a simple example. Imagine we want to create five different circles of distinct sizes. To be able to do so, with our current knowledge, we would need to create five different variables and assign those variables the desired values. And then call the **ellipse** function five times, using a different variable each time.

```
var size1 = 200;
var size2 = 150;
var size3 = 100;
var size4 = 50;
var size5 = 25;

ellipse(width/2, height/2, size1, size1);
ellipse(width/2, height/2, size2, size2);
ellipse(width/2, height/2, size3, size3);
ellipse(width/2, height/2, size4, size4);
ellipse(width/2, height/2, size5, size5);
```

We are only drawing five circles to the screen, but this is already looking like a burdensome solution. What if we needed to draw 100 circles or even 1000? This is where arrays come into play and make our job much easier.

First, let's create an array of desired circle sizes. As mentioned earlier, we can use the index numbers to access the individual items in an array. We will use this knowledge to fetch the desired values from our array.

```
var sizes = [200, 150, 100, 50, 25];

ellipse(width/2, height/2, sizes[0], sizes[0]);
ellipse(width/2, height/2, sizes[1], sizes[1]);
ellipse(width/2, height/2, sizes[2], sizes[2]);
ellipse(width/2, height/2, sizes[3], sizes[3]);
ellipse(width/2, height/2, sizes[4], sizes[4]);
```

This is already looking so much better. But notice the amount of repetition that's still happening. We are essentially typing the same thing over and over again when calling the **ellipse** function, the only thing that's changing is the index numbers. A very clear pattern is emerging here if we had a structure that would create a loop for us to call **ellipse** function five times with increasing values, then we wouldn't have to repeat ourselves.

Luckily, we know how to create a *for loop* that would help us to do precisely

that. Here is the above code re-written to use a for loop.

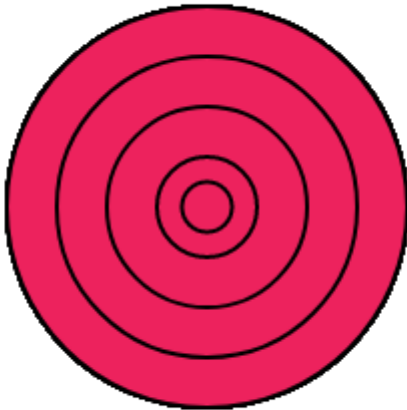
```
var sizes = [200, 150, 100, 50, 25];
for (var i = 0; i < 5; i++) {
  ellipse(width / 2, height / 2, sizes[i], sizes[i]);
}
```

Here is the usage of the code inside a p5.js example:

Output

JavaScript

HTML



The image shows a p5.js editor window. The top section is labeled 'Output' and contains the text 'JavaScript' and 'HTML'. The main canvas area displays a series of five concentric red circles centered on a white background. The circles decrease in size from the outermost to the innermost. At the bottom right of the editor, there are two icons: a blue square with a white document icon and a grey square with a white left-pointing arrow icon.

Notice the usage of the number five inside the for loop header. It is there because the array that we are using has five items in it. So if there were six items, then we should update this value to be six. But this is a bit problematic, what if we made our array bigger but forgot to update this value? Luckily we can use an array property called **length** instead, which would give us the number of items in an array. We can re-write the above code to make use of the **length** property.

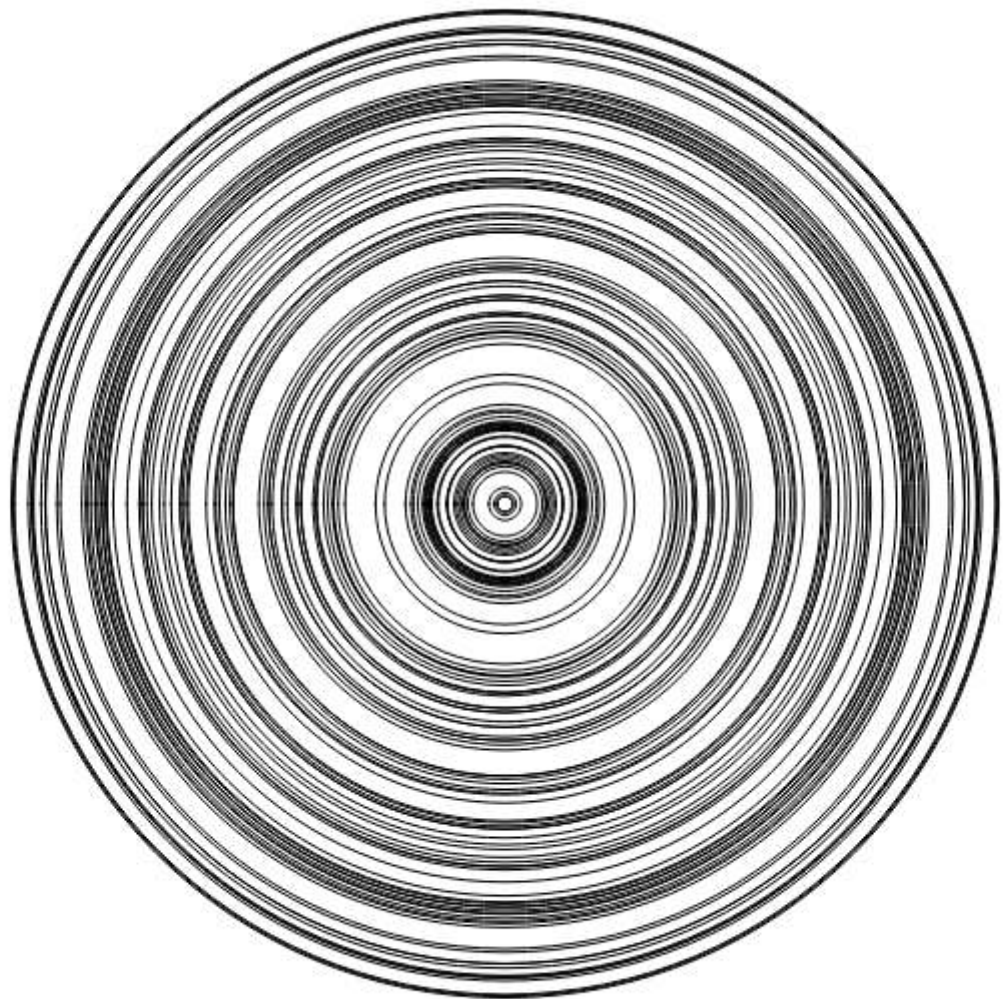
```
var sizes = [200, 156, 100, 25, 5];
for (var i = 0; i < sizes.length; i++) {
  ellipse(width / 2, height / 2, sizes[i], sizes[i]);
}
```

Our code is much more concise now, and it is insanely scalable as well. We can just keep adding new values to the **sizes** array, and an equal amount of circles will be drawn for us. Just for fun, let's automate this setup even further. Currently, we are manually creating the array that has the size values. But we could create another for loop that would populate this array with any amount of random number of our choosing by using the **random** function.

Output

JavaScript

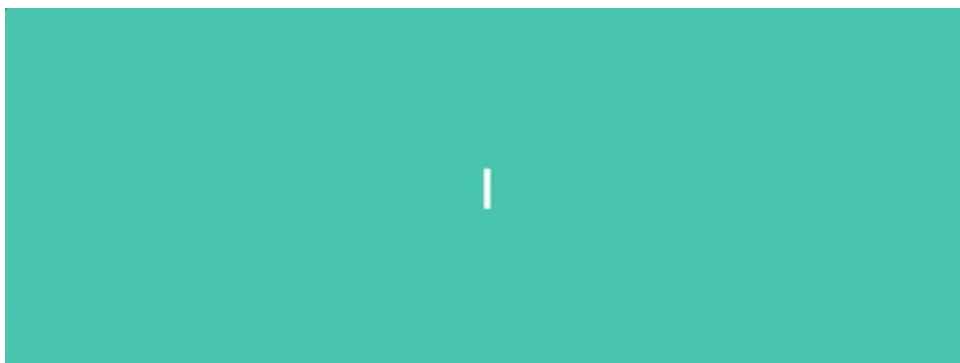
HTML



Let's walk through what's happening in this example. We are first setting the background color to be white inside the **draw** function. Also, we are calling the **noFill** function that would draw the shapes without a fill color. These are just stylistic choices. We are creating an empty **sizes** array that we will be populating with random numbers. Then we are creating a loop that is going to iterate for 100 times. Inside that loop, for each iteration, we are creating a random value in between 5 and 500 using the **random** function, and pushing that generated random value inside the **sizes** array using the **push** method.

The next step remains the same. We are creating ellipses for all the values that exist in the **sizes** array. Notice how changing a single value in this program, the amount of random numbers being generated which is at 100 right now, controls the entire outcome. This is a great example that exhibits how simple programming structures can create very robust and scalable solutions.

Let's work on another visualization using Arrays! The plan is to create an animation that is sequentially and continuously going to display the given words in a stylistic manner.



First, let's refresh our knowledge on how to create text in p5.js. We would be using the **text** function that takes three arguments: the text to display, and the x and y position of that text. Using this knowledge let's just display the word "JavaScript" on the screen on a light coloured background.

Output
JavaScript
HTML



Notice that the text we created is not vertically aligned. It doesn't look centered. It is easy to fix this using a function called **textAlign** in p5.js. Just call this function inside the **setup** function by passing the value **CENTER** to it. This would take care of the vertical alignment. We could pass **CENTER** to this function one more time to horizontally align the text as well.

```
textAlign(CENTER, CENTER);
```

Next, let's format the text so that it would look a bit better. We are going to set the text size to 45 pixels by using the **textSize** function and will make the text color to be white using the **fill** function.

Output
JavaScript
HTML

JavaScript



Perfect! In this example, we would like to create an array of words and continuously cycle through them. Let's first create the array that we would be using. We will be creating it outside the **draw** function because we only need to create this array once. If we were to declare it inside the **draw** function, then it would continuously be created and destroyed with each call to the draw function (which happens around 60 times a second by default!).

Let's create a variable called **words** outside the **draw** and **setup** functions. Since the variable is initialized outside of both **setup** and **draw** functions, it would be accessible from both of them.

```
var words = ['I', 'love', 'programming', 'with', 'JavaScript'];
```

Next, we need to devise a way that would continuously generate a value in between 0 and the length of this array to be able to refer to the individual items in an array. To do so, we can use the **remainder** (%) operator.

The remainder operator is a bit different than all the operators we have seen previously, such as plus or minus, so it might be beneficial to see how it works. Given two values, the **remainder** operator returns the remainder left over when the first value is divided by the second value. The % operator symbolizes it.

As we can see in this example, given an incrementally increasing first value,

the **remainder** operator allows us to cycle through second value minus one.

```
console.log(1 % 6) // returns 1.  
console.log(2 % 6) // returns 2.  
console.log(3 % 6) // returns 3.  
console.log(4 % 6) // returns 4.  
console.log(5 % 6) // returns 5.  
console.log(6 % 6) // returns 0.  
console.log(7 % 6) // returns 1.  
// etc..
```



You might find yourself thinking: “how would you even know this?” As in, this can be something that is really hard to think of, if all we knew was what **remainder** operator did but didn’t have any practice using it. This is perfectly normal. You get to understand what kind of an operator or structure you can use for a certain purpose by seeing other people use it. It is sometimes a matter of experience and practice rather than knowledge.

If I am to provide a constant supply of incremental values to a **remainder** operator alongside with the length of my array, I will be able to generate values cycling in between 0 and that length.

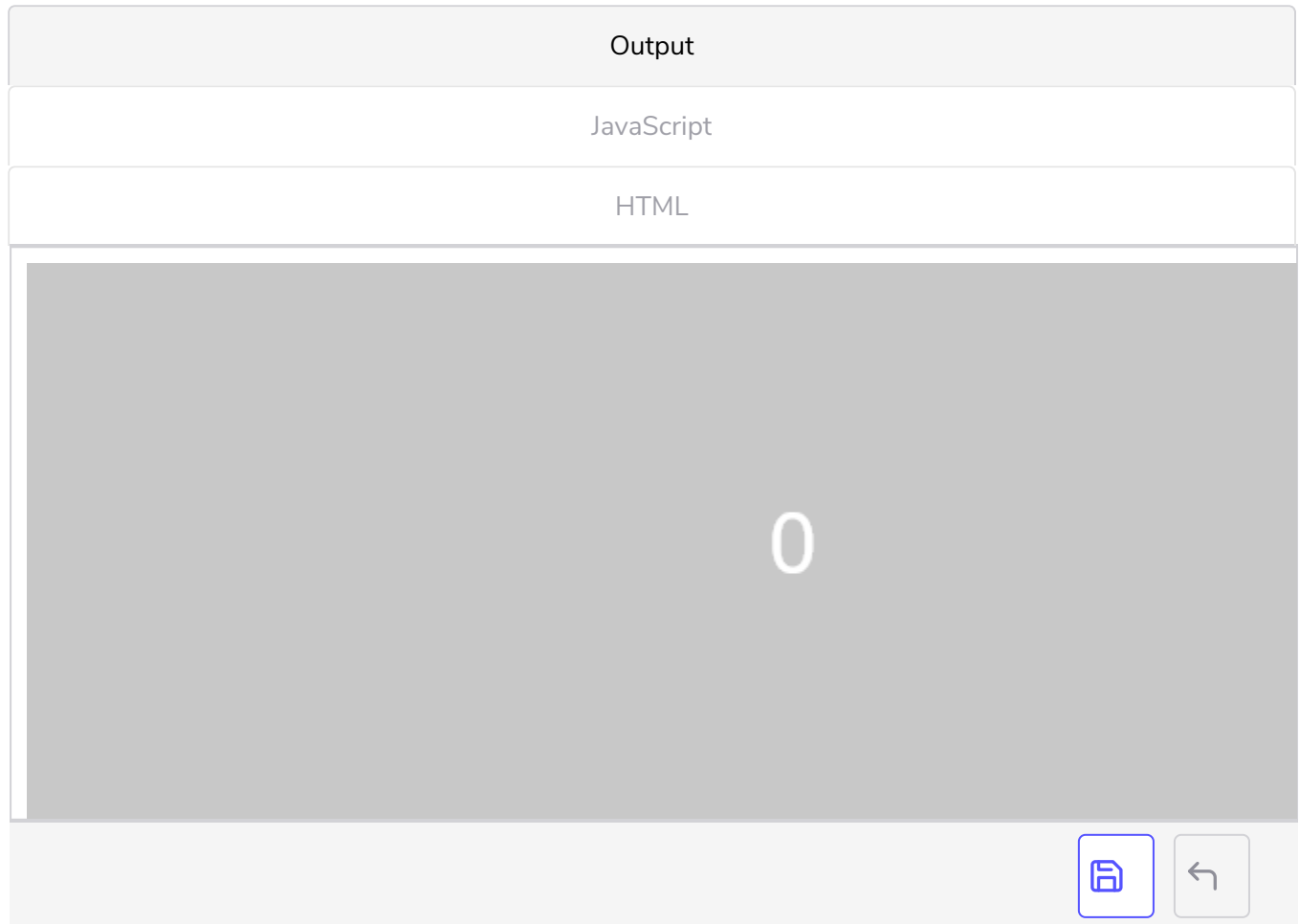
In the p5.js context that constant supply of values could be the **frameCount** variable. Remember **frameCount** tells us how many times the **draw** function has been called so far. Let’s create a variable inside the **draw** function with the name **currentIndex**, which uses the **remainder** operator, the **frameCount** p5.js variable and the length of the words array to create values in between 0 and the length of the array minus one.

```
var currentIndex = frameCount % words.length;
```

We can **console.log** this statement to verify we are indeed creating values in the desired range. But a better way of doing things might be just to use the **text** function that we already have to display this value using p5.js. We are visual learners after all.

One thing to notice at this point is that the display of the numbers is simply too fast, it is really hard to understand what’s going on. We should slow p5.js down or else our text would be very hard to read. One way of doing it could

be to decrease the frame rate using the **frameRate** function. Inside the setup function let us change this value to 3.



Awesome! Using this code we should be able to see a range of numbers being displayed on the screen. But we are not interested in displaying numbers to the screen but the words inside the array. That's very easy to do using our knowledge. We will use the square bracket notation to access individual items inside the array.

Let's create another variable called **currentWord**, which would store the current word as determined by the **currentIndex** variable. Now we can use this variable instead of **currentIndex** inside the text function.

```
var currentWord = words[currentIndex];
```

We are almost done. But one another thing that I would like to do is to change the background color per word since this is not aesthetically pleasing at all right now.

We will create another array called **colors** that would contain color information. It turns out that we can pass an array into p5.js color functions and it is the same as passing values one by one to it.

So these two expressions will create the same color as each other.

```
fill(255, 0, 0);  
fill([255, 0, 0]);
```

We will create **colors** array that contain arrays of colors that we will be using. We can try to come up with color values by ourselves, but it is hard to find good looking colors that way.

Adobe has a webpage where we can find color themes to use in our designs. I will use it to find a theme that would go with my visualization:

<https://color.adobe.com/>

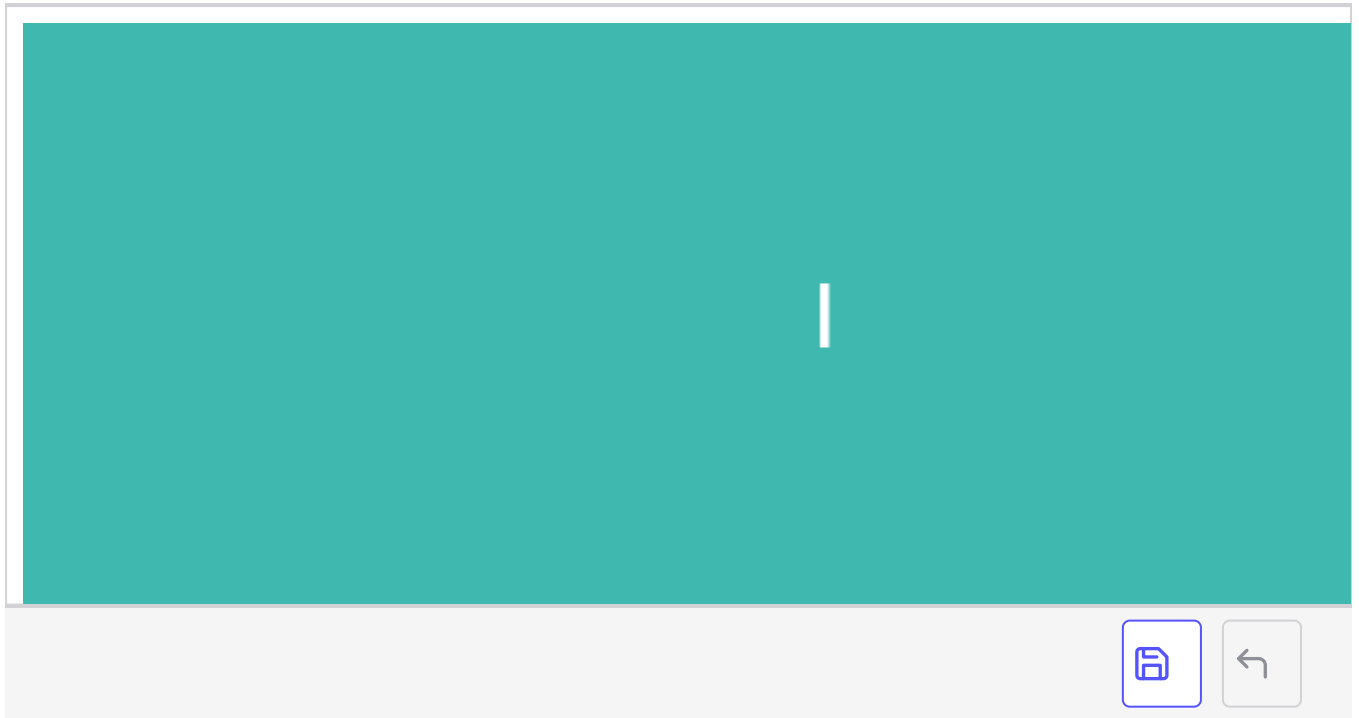
Under the explore tab, once we find a desirable theme, we can click the edit button and be able to see the RGB values for the colors. We can pick a palette of our liking. Here is a sample of colors picked from that website.

```
var colors = [  
  [63, 184, 175],  
  [127, 199, 175],  
  [218, 216, 167],  
  [255, 158, 157],  
  [255, 61, 127],  
];
```

Notice my formatting for the data is a bit different because I didn't prefer the line length to be too long as it might hamper the legibility of our code. This is just a stylistic choice.

Now we can use these color values inside the **fill** function to change the color of the background with each frame. Here is what the final code looks like:

Output
JavaScript
HTML



Summary

In this chapter we learned about a JavaScript data structure called Arrays. Arrays allow us to store multiple values of any type in a sequential fashion. The values that are stored in an array can be accessed using the square bracket notation.

We can populate an array with the desired values when they are first created or after their creation using the **push** method.

Arrays are particularly useful when used with loops. Loops let us access the items in an array in a very easy manner.

We also learned about the **remainder** operator. Remainder operator returns the remainder from a division operation in between two numbers. Using this operator, we can derive sequential values that cycle in between zero and the desired value.

Practice

Build a function called **countdown** that would get two arguments, a **number** and a **message**, and would create a visualization that is similar to above that would display a countdown from the given number to number 0. At the end of the countdown, it should display the given message, the second argument, to the screen

the screen.

Feel free to add another parameter to the function that would control how long each number would stay on the screen.

```
countdown(10, "Launch!");
```

Output

JavaScript

HTML

CSS (SCSS)



Console

Clear