07 - Loops

JavaScript Loops

WE'LL COVER THE FOLLOWING

For Loop

One of the things that computers are great at are repetitions. Imagine having to create 1000 shapes on screen with varying parameters. It would take us an unreasonable amount of time to do so with our current programming knowledge. For this kind of cases where we want to repeat our code as it is or with variations we can leverage a programming structure called *loops*. A loop allows us to execute a block of code over and over again.

We are already familiar with the idea of a loop in p5.js. If you think about it, the **draw** function is a continuous loop that gets executed over and over again until we exit the p5.js program. In this chapter, we will learn how to build this kind of loops ourselves.

There are a couple of different kinds of loop structures in JavaScript, but a *for loop* is by far the most popular. It allows us to repeat an operation for a given amount of times. A *for loop* has four parts. Here is an example of how a *for loop* is constructed.

```
for (var i = 0; i < 10; i = i + 1) {
    //do something
}
```

In the first part, we initialize a variable that would keep track of the number of times that the loop gets executed - let's call this a counter variable.

```
var i = 0;
```

By convention, inside the *for loop*, we usually tend to use short variable names like **i** or **j**, especially if that variable is only in use for controlling the flow of the *for loop*. But feel free to use other names as well if it makes sense for your use case.

In the second part, we define a test condition for our loop which gets evaluated each time the loop is about to start. In this example, we are checking to see if our counter variable is smaller than the number 10.

```
i < 10;
```

In the third part, we define a way to update the counter variable which gets evaluated at the end of the loop. In this example, we get the current value of the variable **i** and add one to it.

```
i = i + 1;
```

Finally, inside curly braces we write the code that we want to have repeated. Once the counter variable doesn't satisfy the test condition, the loop terminates, and the program returns to its normal evaluation.

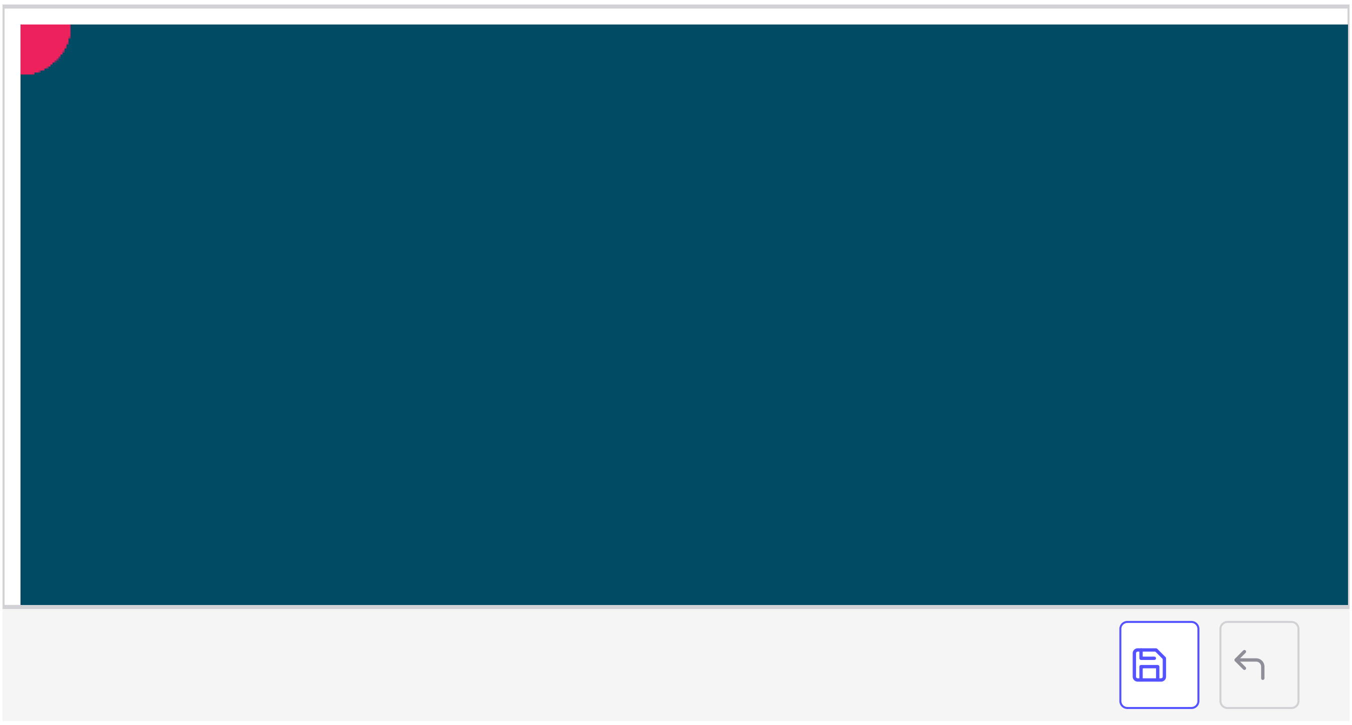
If the test condition never fails, then we would have a loop that would end up creating an *infinite loop*, a loop that doesn't have an exit condition so that it keeps going on and on until the program is terminated by external means. The **draw** function in p5.js is in an infinite loop; it keeps drawing to the screen until we close the browser window.

Even though infinite loops are a valid use case, loops are most commonly used for executing an operation for a known amount of times. Let's create a loop that will draw given number of ellipses to the screen using a *for loop*.

Output

JavaScript

HTML

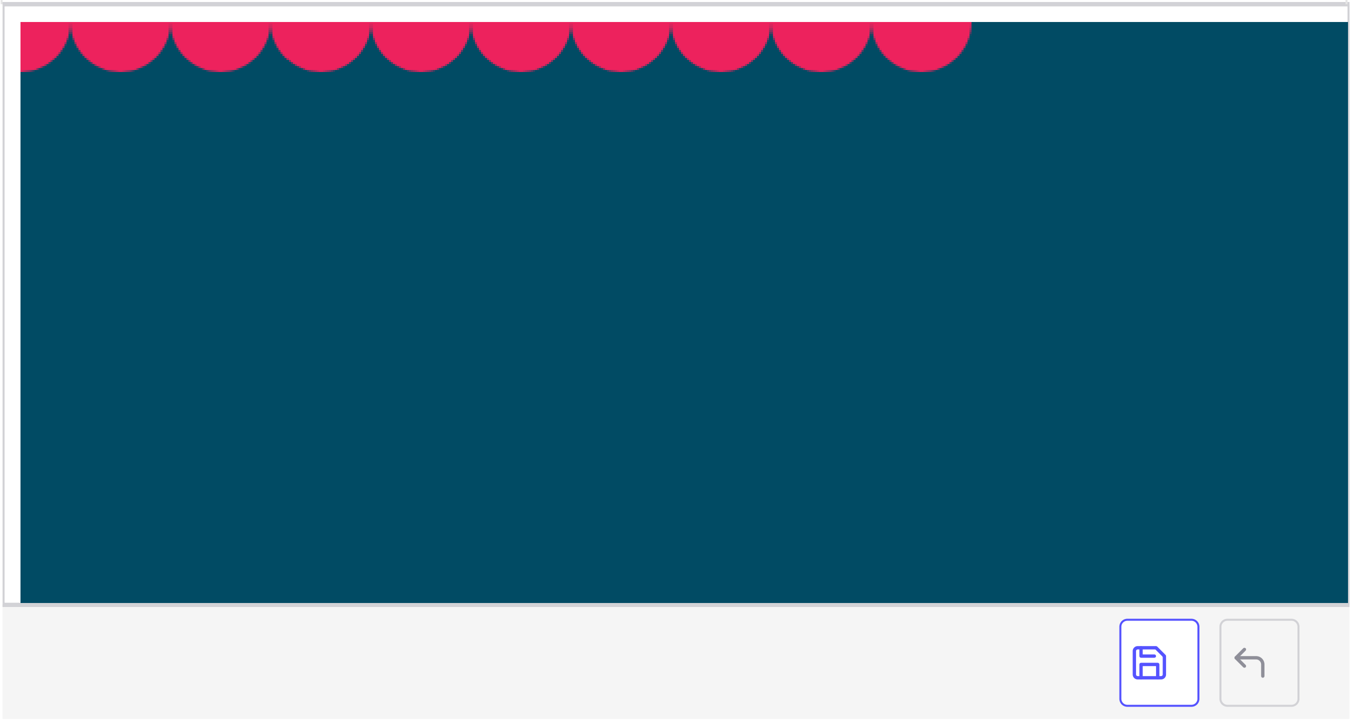


In our example, we are drawing ten circles to the screen, but there is no way of visually making that distinction since all the circles are being drawn on top of each other. This is where making use of the loop counter variable can make sense. I can basically use this variable to offset the position of circles each time the loop is called.

Output

JavaScript

HTML



We are multiplying the loop variable by 50, the diameter of the circle, before feeding into the ellipse function to be able to have the shapes not overlap with each other.

Now if we are to execute this, we will see all those circles that the *for loop* is creating for us. The great thing about this is that since we built the structure for repeating our operations, scaling it up can be as easy as changing the number that we are using inside the loop conditional to a bigger value. Rendering 100 or 1000 circles instead of 10 is just a matter of changing this one value. Though, we might start noticing performance degradation if we were to start using huge numbers.

Let's build our code so that we can fill the entire width of the screen with circles.

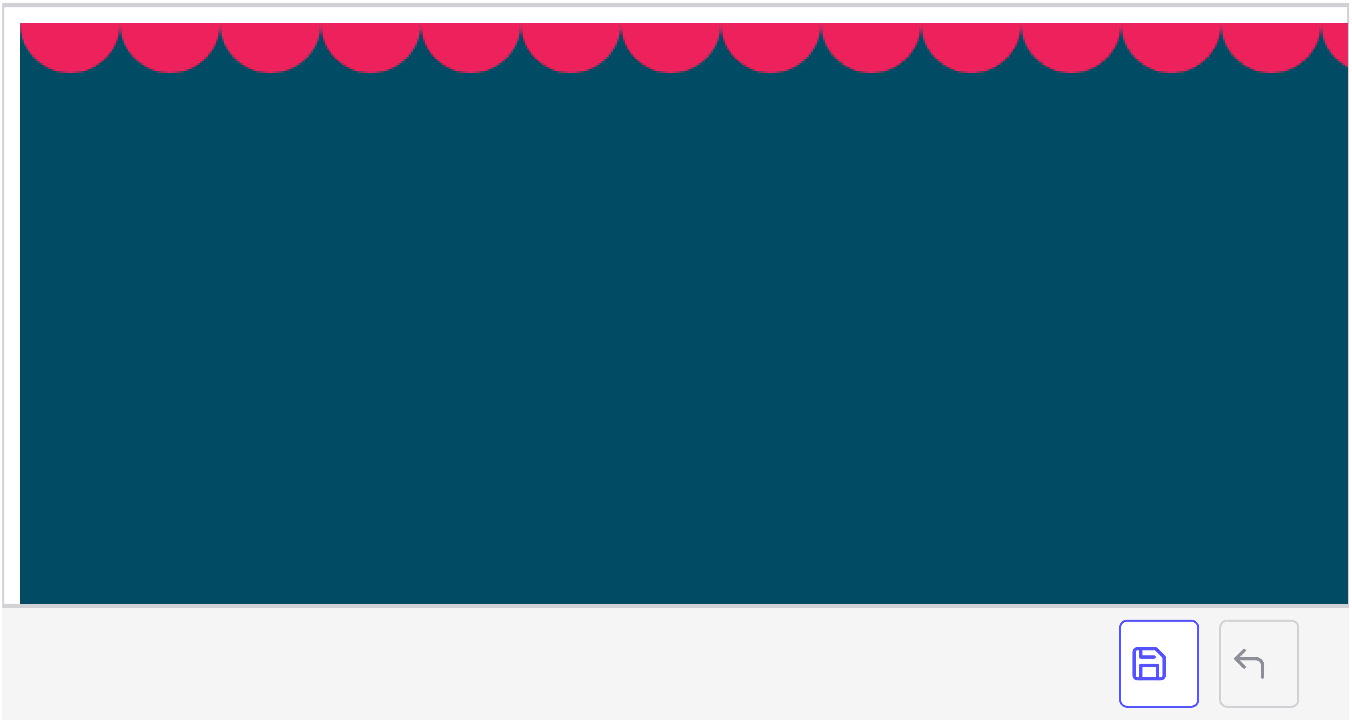
If the width of the screen is 800, and the diameter of a circle is 50 units, then it would mean that we can fill `800 / 50` circles into the width of the page. We would notice a bit of a gap at the end of the page since the first circle is a little bit outside the canvas. We can offset everything to get rid of this gap by adding 25 to the x position which is half the diameter value. As you already know, we actually don't need to do this math ourselves as we can have JavaScript calculate that value for us.

What you might notice at this point is that we are hard coding lots of values into our code and it would be better to use variables instead for flexibility. We will refactor our code to do so.

Output

JavaScript

HTML



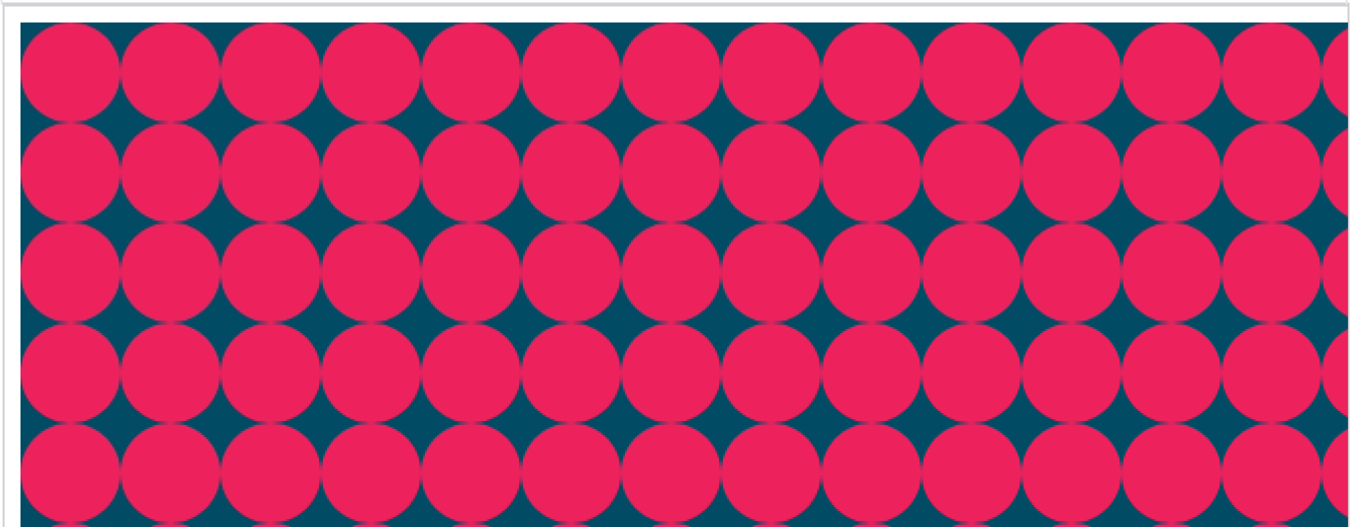
Now, if we are to change a single value, the diameter of the circle, the entire code will still draw just enough circles to fill the screen. That's a pretty impressive thing to have.

What if we wanted to fill the height of the screen with circles as well? To be able to do this, we need to write another for loop that would place circles for the entire length of the canvas for each circle that is placed for the width. This requires us to place a second loop inside the first one, effectively *nesting* a loop inside another loop.

Output

JavaScript

HTML



Notice the way we declared the **ellipse** function in this example. We are writing it over multiple lines to be able to increase the legibility. JavaScript allows us to start a new line after the comma operator.

This code is pretty useful right now. For one thing, it is robust, we could be changing the size of the drawing area or the number of circles being drawn, but things will still continue to function properly.

Something to keep in mind is; putting loops inside one another can make our program really slow due to the number of operations that need to be performed. Also, sometimes nested structures can make our programs hard to read as well.

Random and Noise Functions

Since we can now create loops that make use of a different value each time they are executed, it might be a good time to learn about the p5.js **random** function. The p5.js **random** function generates a random number every time it's called. This is useful when we want to use random values for the parameters of the shapes that we are drawing.

If we are to call the **random** function without any parameters, then it would result in a random number between 0 and 1 for each **draw** function call or each frame. If we are to provide value to a **random** function, then it would return a random value that is above 0 and below that given value. If we are to provide two values to the random function, then we would get a random value that is in between the given two numbers.

JavaScript

```
setup () {
  random(); // a random number in between 0 and 1
  random(10); // a random number in between 0 and 10
  random(100, 1000); // a random number in between 100 and 1000
}
```

Console

Clear

Here is a small script that illustrates the results of calling the random function in different ways.

Output

JavaScript

HTML

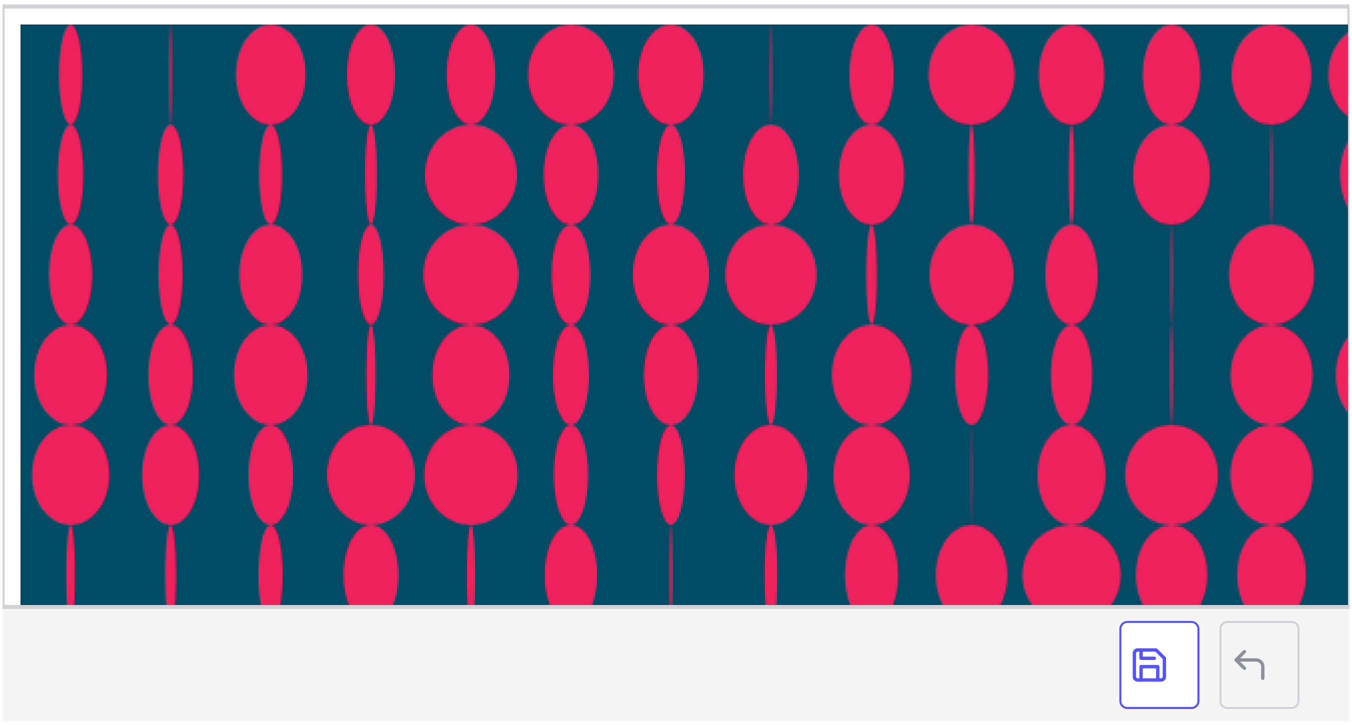
```
0.9116300851164847
1.7119215076798078
726.5390748471174
```

Let’s update our previous code to make use of **random** function.

Output

JavaScript

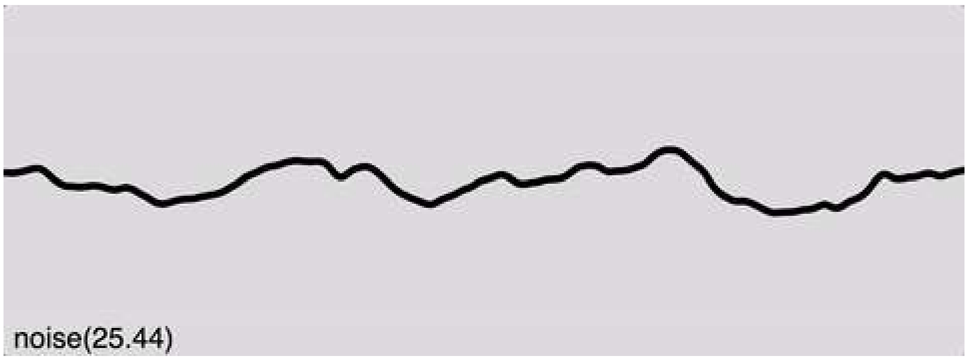
HTML



We are using the result of the **random** function to multiply the width of the ellipse with a random number that would be a value in between 0 and 1 each time the **random** function is called. Since **random** function can assume any value in its range in any frame, the animation looks pretty aggressive. If we want randomness that changes gradually, and hence looks a bit more organic, then we should look into the **noise** function.

We can feed any numeric value to the **noise** function and it would return a semi-random value in between 0 and 1. It would always return the same output for the given value. The good thing about **noise** function is that if the value we are feeding to the **noise** function changes only incrementally then the output value will only change incrementally as well. This will result in a smooth transition between the random values we are getting back.

To be able to conceptualize how the **noise** function works, we can think of an infinite amount of random values that are changing gradually like a wave and the values that we provide to the **noise** function are like coordinates for these random values. Essentially we are just sampling an already existing noise. Whenever we provide the noise function with the same values, we are going to receive the same semi-random value in return.

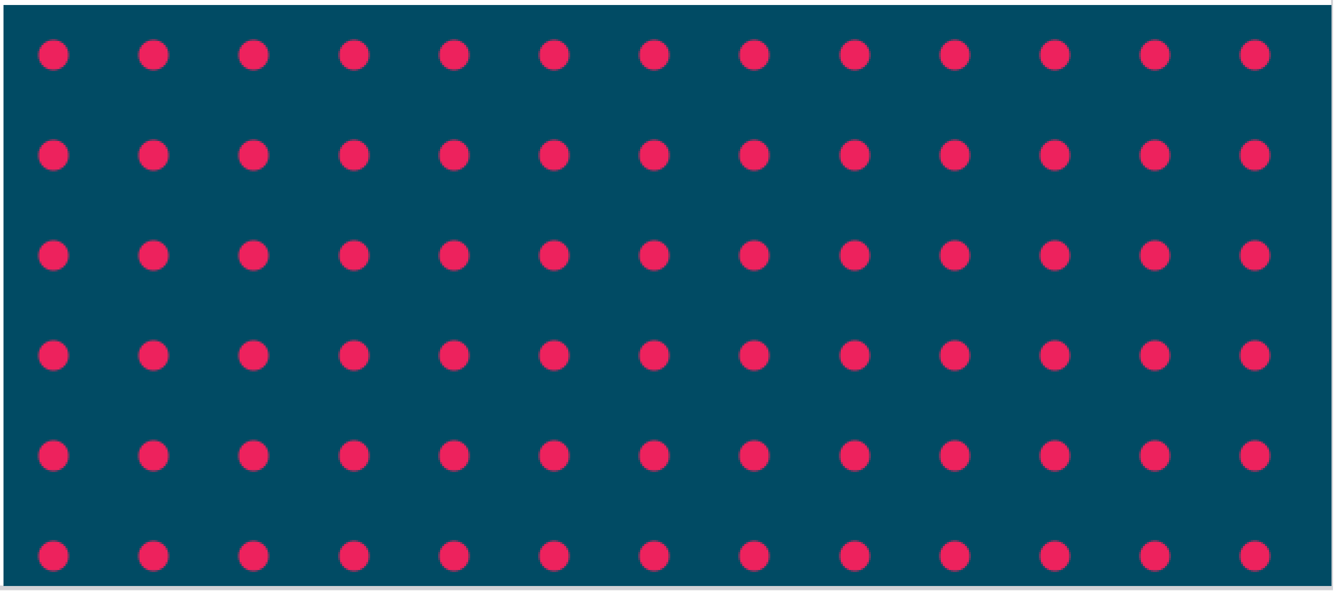


We will write the above program to make use of **noise** function instead. We will feed the noise function with the **frameCount** variable since it is a good way of getting sequential numbers in p5.js. But we will divide the **frameCount** with 100 to be able to slow down the change of values and hence the resulting animation a bit.

Output

JavaScript

HTML

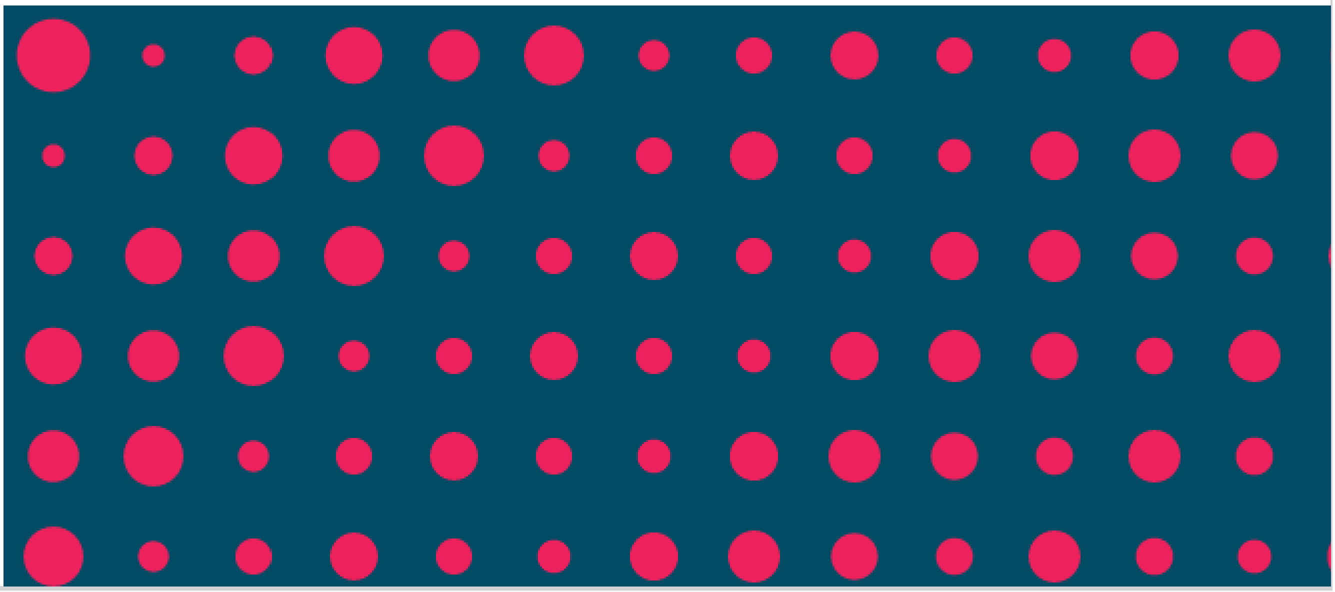


Notice how all the shapes are using the same animation right now. What if we wanted to get a different noise value for each one of these shapes? Currently we have the values repeating since the **noise** function when provided with same values returns the same output. To be able to get a different output value for each of the shapes, we might want to rewrite the above function to make use of **i** and **j** values of the **for loop** to adjust where the noise is being make use of **i** and **j** values of the **for loop** to adjust where the noise is being sampled from.

Output

JavaScript

HTML



The value 10000 that we are using as a multiplier above is completely arbitrary. We are just trying to make sure that the coordinates that we are providing to the noise function are farther apart from each other.

Summary

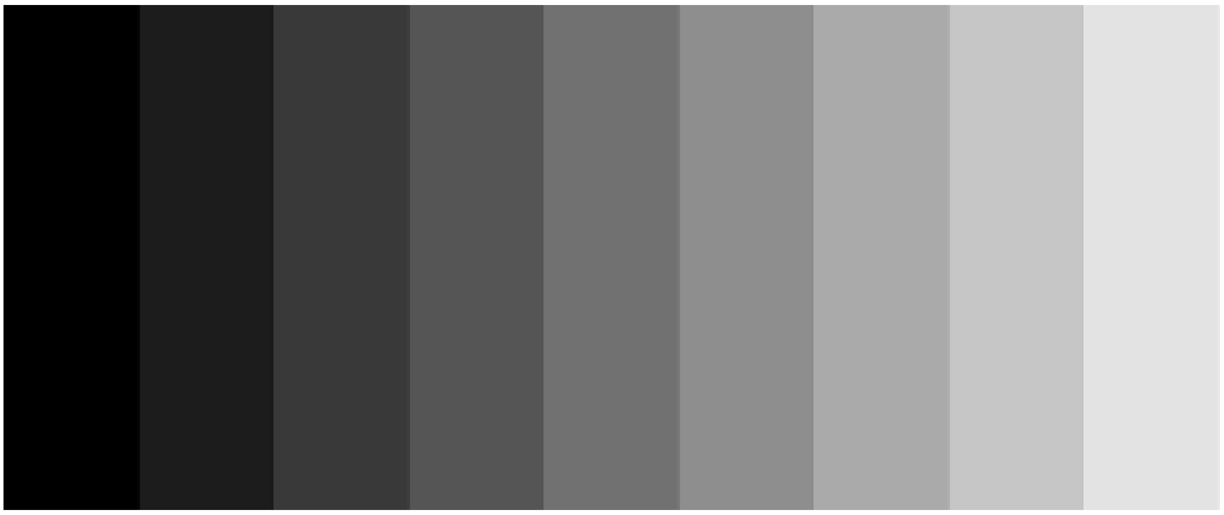
Loops are one of the most powerful structures in programming. It allows us to tap into the true computational power of computers, repeating operations on a larger scale that could be impossible for a human to perform in a reasonable amount of time.

In this chapter we learned about how to build **for loops** and how to nest loops in each other to be able to get a grid of repeating shapes instead of just a line of them.

We also learned about the p5.js **random** and **noise** functions and the difference between them.

Practice

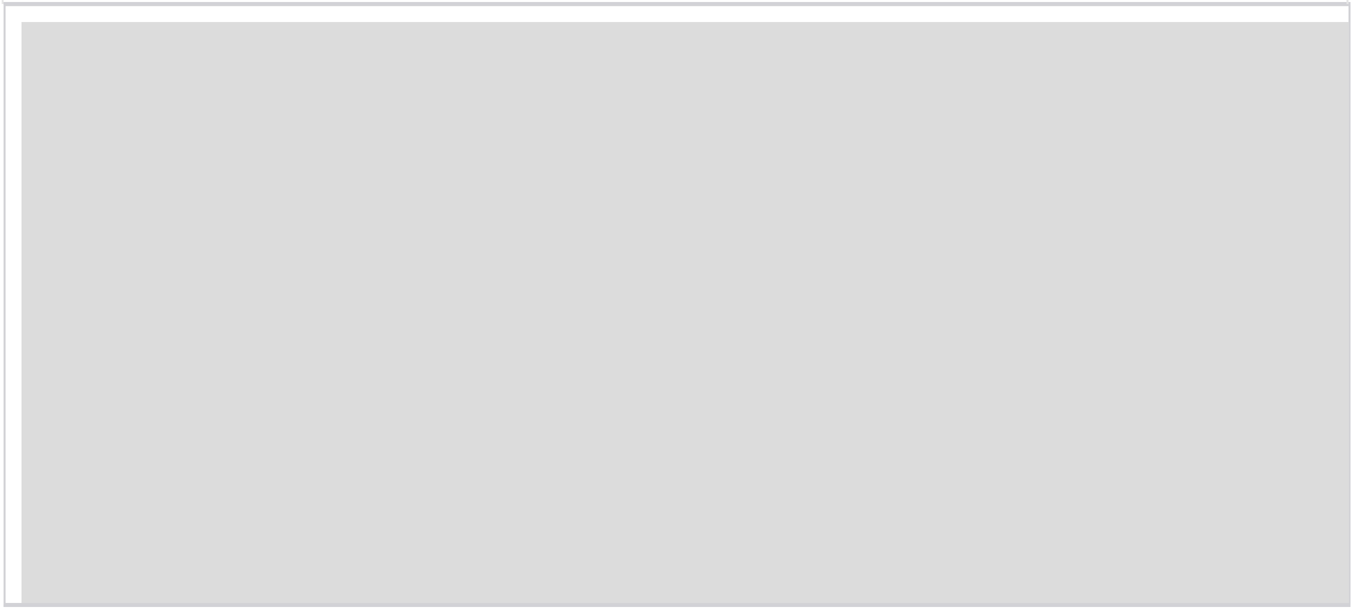
Create a loop that would create an array of rectangles that have their color changed gradually from black to white. You should build the loop in such a way that a single variable would control the number of rectangles drawn.



Output

JavaScript

HTML



Console

Clear