# 04 - Operators & Variables

Using JavaScript Operators & Variables

WE'LL COVER THE FOLLOWING

## Setup

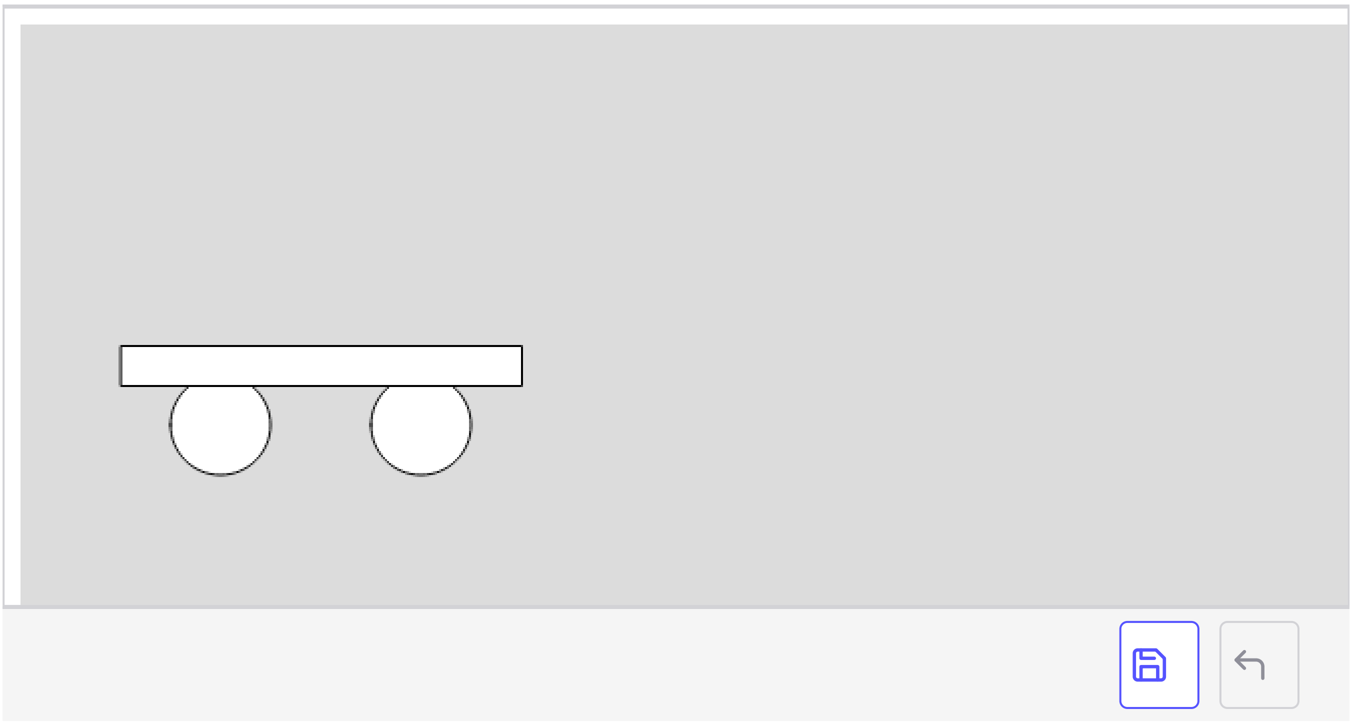
We learned about variables and math operations that we could be using in JavaScript at the beginning of this course. In this chapter, we will put that knowledge to use.

Let's first create a couple of shapes to have something to work with. Using the **ellipse** and **rect** functions let's create a shape that roughly resembles a cart.

Output

JavaScript

HTML



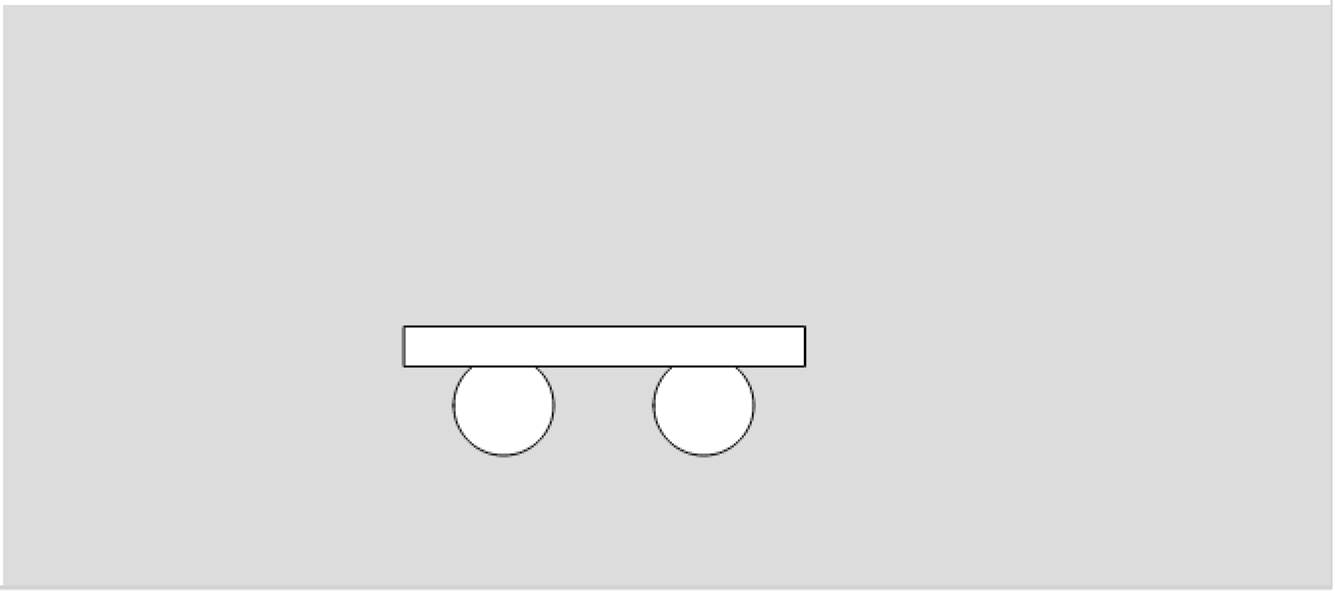
Looking at our rough drawing, I am not entirely happy with its position. I now wish that we drew it more to the right-hand side. Moving the shape now will mean that we would need to increase the value of the x position argument of each of the shape functions.

Let’s assume that we want to add 150 to all these numbers that specify the x position. We can try to do the math in our head and type the result in there but luckily we can do math operations easily with JavaScript. Instead of typing the result of addition, we can just type out the operation needed, and JavaScript will do the calculation for us.

Output

JavaScript

HTML



The same thing works with other operators as well; we can do subtraction, multiplication or division in a similar manner.

One thing that we need to keep in mind with operators is the order of operations. You might already know this from your math classes, but some operators take precedence over others. For example, if we wanted to add 2 to a number and then multiply it by 4, we might be tempted to write something like this:

```
console.log(10 + 2 * 4);
```

But in this operation multiplication will happen before addition. 2 will get multiplied with 4 before being added to 10, so this above operation will yield 18 instead of the expected value 48.

To be able to control the order of operations we can use parentheses. For example, we can write the top equation like this:

```
var x = (10 + 2) * 4;
console.log(x);
```

Anything inside parentheses will be evaluated before other operations. In the order of operations, parentheses come first, then the multiplication and division, and then addition and subtraction.

## Variables

To be able to evaluate expressions like this will make our job easier in doing calculations. But I think the real problem here, in this example, is the need to type the same number at all these three different spots. This is very repetitive and laborious. This is an instance where usage of a variable would be useful.

Whenever we need a value, and we need to use that value in multiple places, we would want to store that value in a variable. The advantage of using a variable is that, if we ever needed to update the value of the variable, we would only need to do it in a single place. Let's update this example to use a variable.

Remember how to create variables. We would start off by using the **var** keyword. Using this keyword is really important for reasons that are to be discussed later.

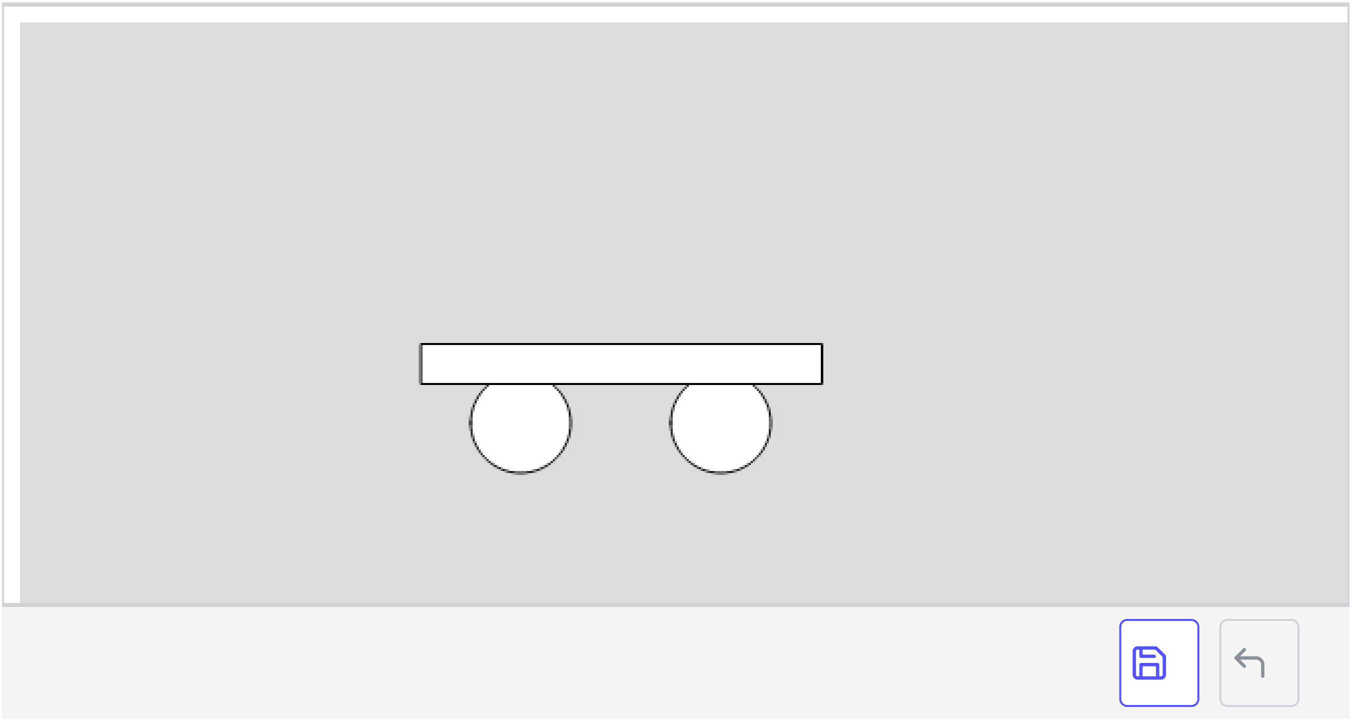
Then we would choose a name for our variable. It is also important to choose a name that makes sense. Calling this variable **offset** or **x** might make sense as I would be using to offset shapes in the x-axis. Using sensible names would help others or even us in understanding our code. We always want our programs to be as readable as possible.

Now that we have a variable that points to a value, we can use this variable in operations instead of the value itself. Doing that, we would only need to change the value of this variable from one spot to see the shapes moving.

Output

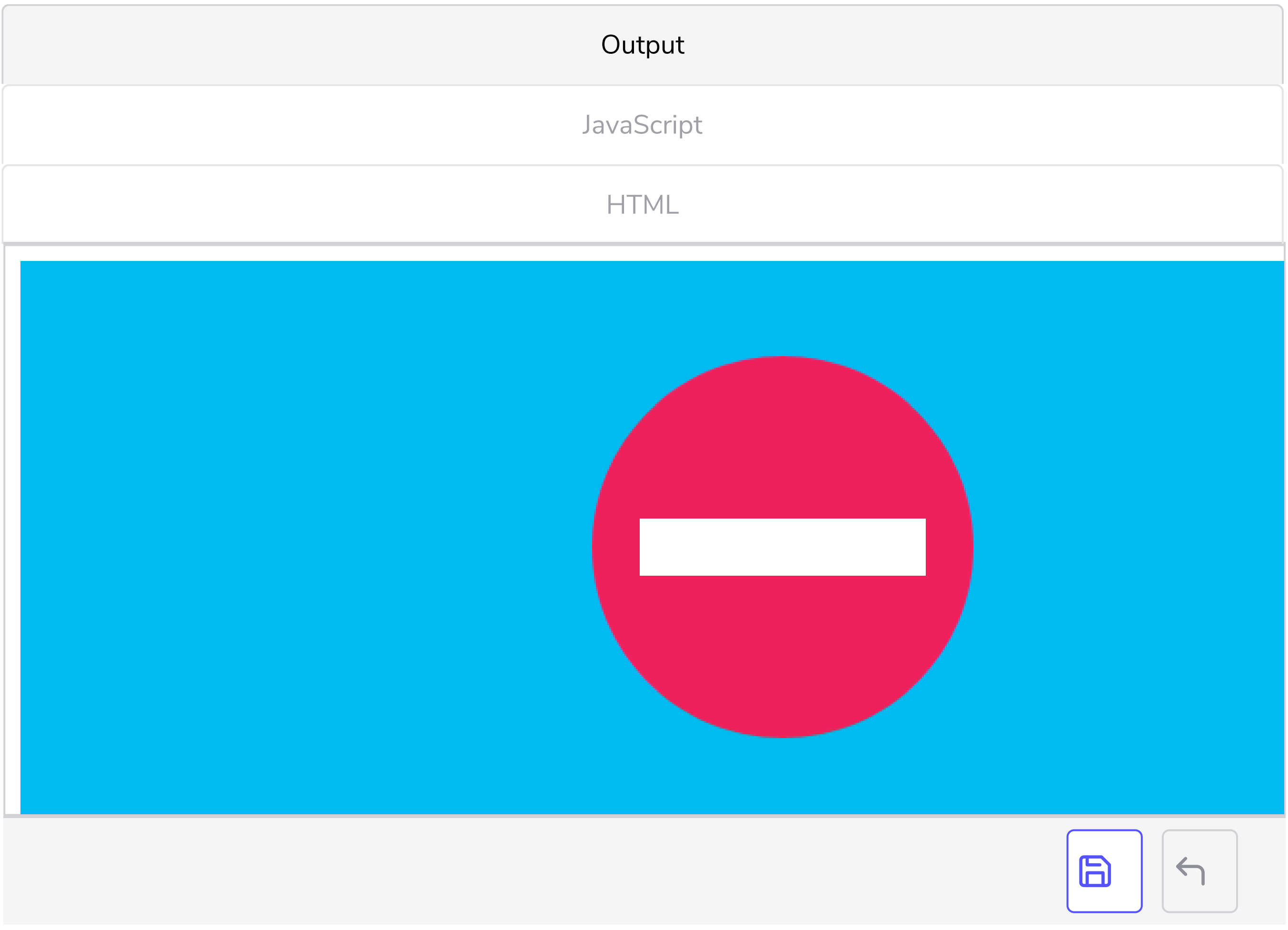
JavaScript

HTML

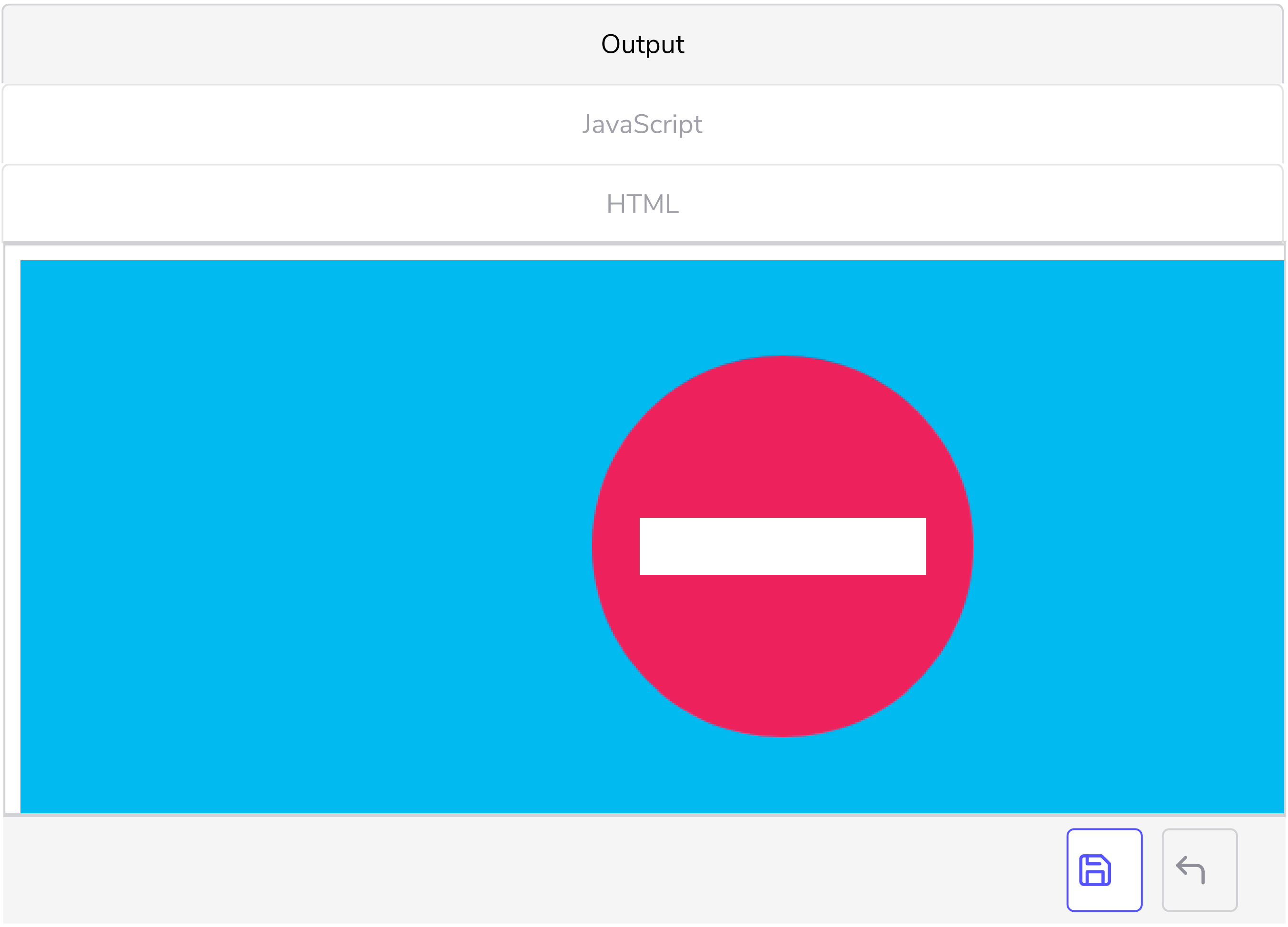


## Variables Continued #

I would like to illustrate another behaviour of variables in a different example. Let’s just draw a single circle in the middle of the screen and a rectangle in the middle.



Can you think of one optimization that we could do to the above program? Notice how we are repeating the x and y position values for the shapes. Let’s use a variable instead.



Since these shapes are not being positioned relative to the canvas size, if we are to change the size of the canvas, the relative position of the shapes will change as well. For a square canvas, the shape is currently at the center, but for a wider canvas, the shape might start falling to the left-hand side. To have the shapes close to the center for any given canvas size, we can start off by using a variable to set the width and height values for the canvas. Then we can utilize the same variable to control the position of the shapes.

Inside the **setup** function, we are going to create two new variables called **canvasWidth** and **canvasHeight** with the value of 800 and 300. And we will pass these variables to the **createCanvas** function instead of using hard-coded values as before. The plan is that we can be using these same variables inside the **draw** function as well so that even if we are to change the size of the canvas, the relative position of the shapes will remain the same. So let’s put these variables into use in the **draw** function. We will be dividing them by 2 so that we can get the half point of width and height of the canvas.

Output

JavaScript

HTML

CSS (SCSS)

Console

Clear

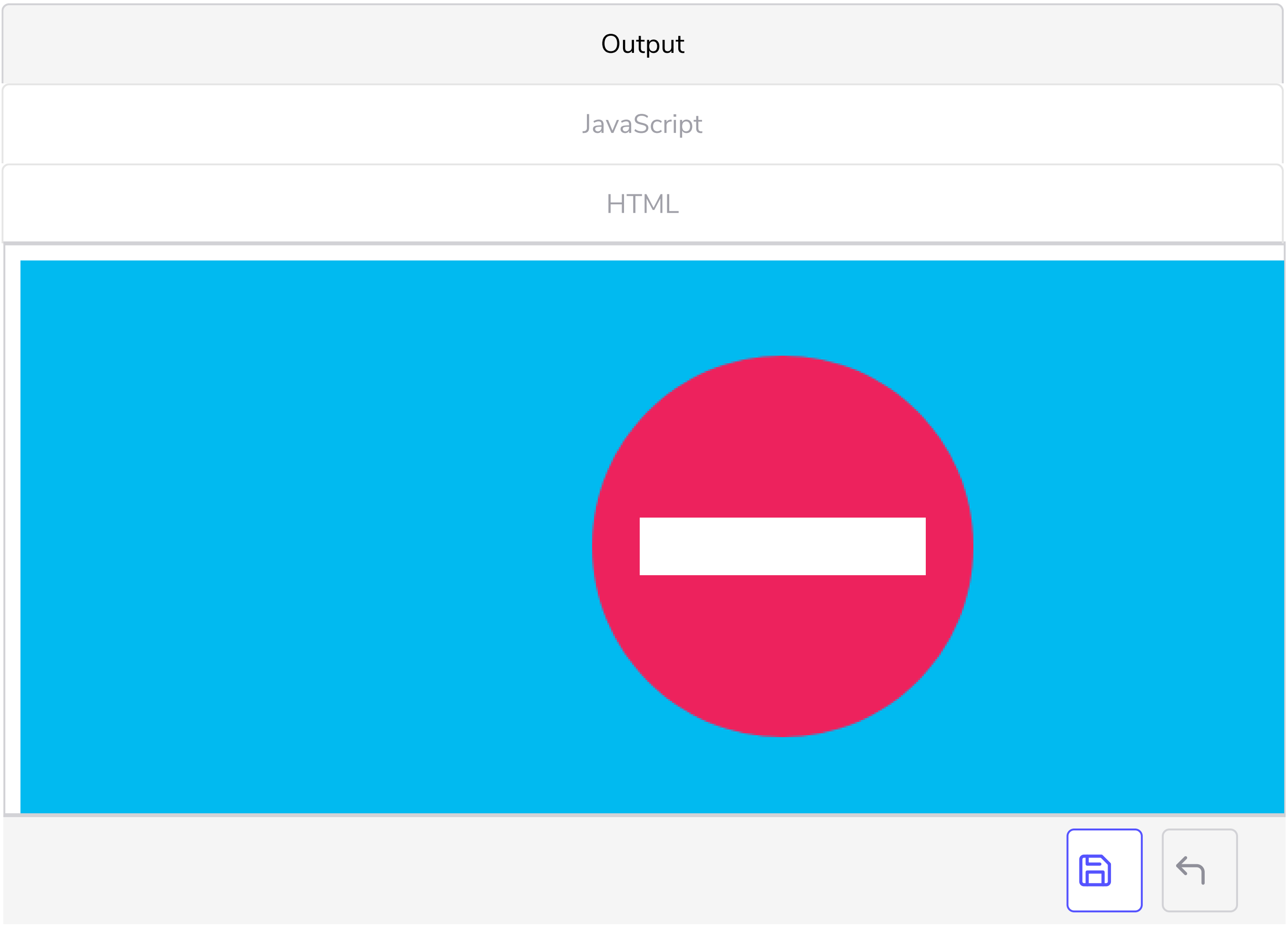
Executing the code, you will notice that we are getting an error. If we are to look at the error message inside the console, it says something about the variable name not being defined.

```
Uncaught ReferenceError: canvasHeight is not defined (sketch: line 14)
Uncaught ReferenceError: canvasWidth is not defined (sketch: line 14)
```

This might come as a surprise since we clearly declared these variables inside the **setup** function. The reason for this error has to do with something called the **scope**. The scope of a variable determines where a variable will be accessible. JavaScript variables when using the **var** keyword to declare them have a *function scope*.

How *function scope* works is that any variable that is declared inside a function won't be visible from outside the function. It is only available to the function that it lives in and other functions that might be nested inside this function. Likewise, if we were to have a variable that is at the top level, this variable would be visible to everything that is at that level and at levels that are nested inside that level, like the functions that might be defined in there. The problem that we are faced with right now is that the variables that are defined inside the **setup** function are not visible from the **draw** function, and if were to declare variables inside the **draw** function; they wouldn't be visible inside other functions at the same level.

The solution to this problem is; instead of declaring our variables inside the **setup** function, we should declare them at this top level so they would be accessible from everything else that is declared inside at the top level.



A variable that is declared at the top level is called a *global variable*. It is usually not the best idea to declare variables at this top level since we run our code in a browser, where other things that are working in the browser such as plugins, add-on's, etc. might cause conflicts by defining variables with the same name for their purposes. Whenever two variable declarations share the same name, the one that gets declared later overwrites the other one since the code is executed from top to bottom. This might result in programs not behaving as expected. But it is not something that you should necessarily

worry about a lot as a beginner since other, more experienced, developers - having the same concern - would have safeguards in place to ensure their variables are not being overwritten. For now, we can put our variables in the top section and be able to share them in different functions that are defined at the same level or below.

In this case, we are initializing the necessary variables outside the setup function so that those variables would be accessible from both the **setup** and **draw** functions. Now we can try setting the **canvasWidth** and **canvasHeight** variables to different values and notice how the shape always remains at the center because its position is derived using the same variables as the canvas.

## Predefined Variables in p5.js

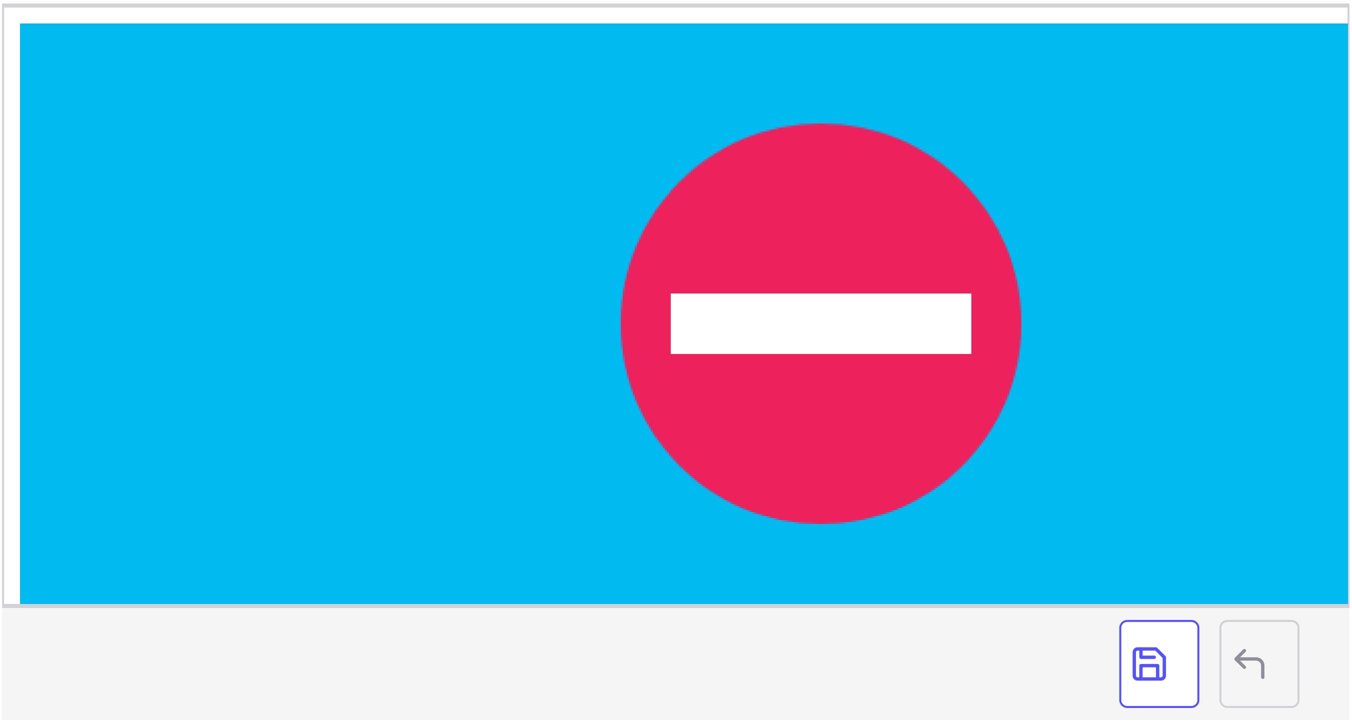
p5.js, being a super helpful library, has a couple of predefined variables that we can use to obtain certain values. Two such variable names that we can use are **width** and **height**. By using these variable names inside the **setup** or **draw** function; we can get the current canvas size. This allows us to do the same thing that we were trying to do by defining our own variable names. p5.js developers must have realized this is something that a lot of developers would try to do by themselves and hence provided an easier solution to the problem.

With this knowledge, the above code could be written like this:

Output

JavaScript

HTML



You should note that **width** and **height** are p5.js variables which mean that they won't be available outside the setup or draw functions.

Now that we know how to use variables, we can animate our shapes! The trick to animation in p5.js is remembering that **draw** function is constantly being executed for us by p5.js. Whatever we are putting inside this function is actually being redrawn each time the **draw** function is executed again.

The number of times that this **draw** function is executed (can be thought as rendered to the screen) is called a frame rate. By default, p5.js has a frame rate of 60. This means that it tries to re-draw (or render) the content of the **draw** function 60 times a second. If we had a way to change the values of the variables that we are using in between each of these **draw** calls, then we would be able to create animations.

This should remind you of flipbook animations. Each call to a **draw** function results in a static image, but since it happens 60 times a second when each of these images is slightly different, you perceive it to be animated.

To be able to create an animation, we are going to initialize a variable outside the **draw** function called **count**. And inside the **draw** function, we will use this simple expression which will increment the **count** variable by one every time the **draw** function is called.

```
    count = count + 1;
```

Now if we are to make use of this variable in a position argument, we can make a shape move. This is an amazing step forward in our p5.js adventure.

Output

JavaScript

HTML

What if instead of making the shape move, we wanted to make it bigger? Easy! We will first create a **size** variable and use that inside our shapes instead of hard-coded values to be able to update the size easier.

Output

JavaScript

HTML

## Summary #

In this chapter, we revisited operators that we have seen before and talked a bit about operator precedence. Then we looked at variables again and learned more about their behaviour, especially regarding their scope. We also learned about some of the built-in variables that p5.js comes with such as the **width** and **height** that are only available inside the **setup** and **draw** functions.

And finally we created our first animation!

## Practice #

Create an animation where five rectangles that are initially off-screen are animated to enter the screen from left-hand side and exit from the right-hand side. They should also be moving at different speeds.

Output

JavaScript

HTML