Rock, Paper, Scissors with Javascript

In this project, you will implement a browser-based implementation of Rock, Paper, Scissors in Javascript.

Learning Objectives

By the end of this lab, you will be able to

- **Describe the properties of Javascript**
- **Work with primitive and complex data types**
- **Define functions and objects**
- **Use control statements**
- **Manipulate the DOM**
- **Demonstrate the use of events/event listeners**

Introduction

Javascript is a versatile programming language that found its roots in web development. Javascript can be used for anything from making web pages dynamic to making server-side applications.
We will focus on client side or 'in-browser' Javascript which can be used to manipulate existing HTML/CSS, react to the client's action, make requests to send and retrieve data, and much more.
In this lab, we will use Javascript to implement the classic game of rock-paper-scissors. Here's a quick refresher on the rules - Rock, Paper, Scissors is a two-player game where each round both players simultaneously choose a move. If both the moves are the same, the players tie otherwise the player with the trumping move wins the round. The moves interact with each other as follows:

- **Rock beats Scissors**
- **Scissors beats Paper**
- **Paper beats Rock**

Creating the Web Page

Paste the code in index.html and save the file.

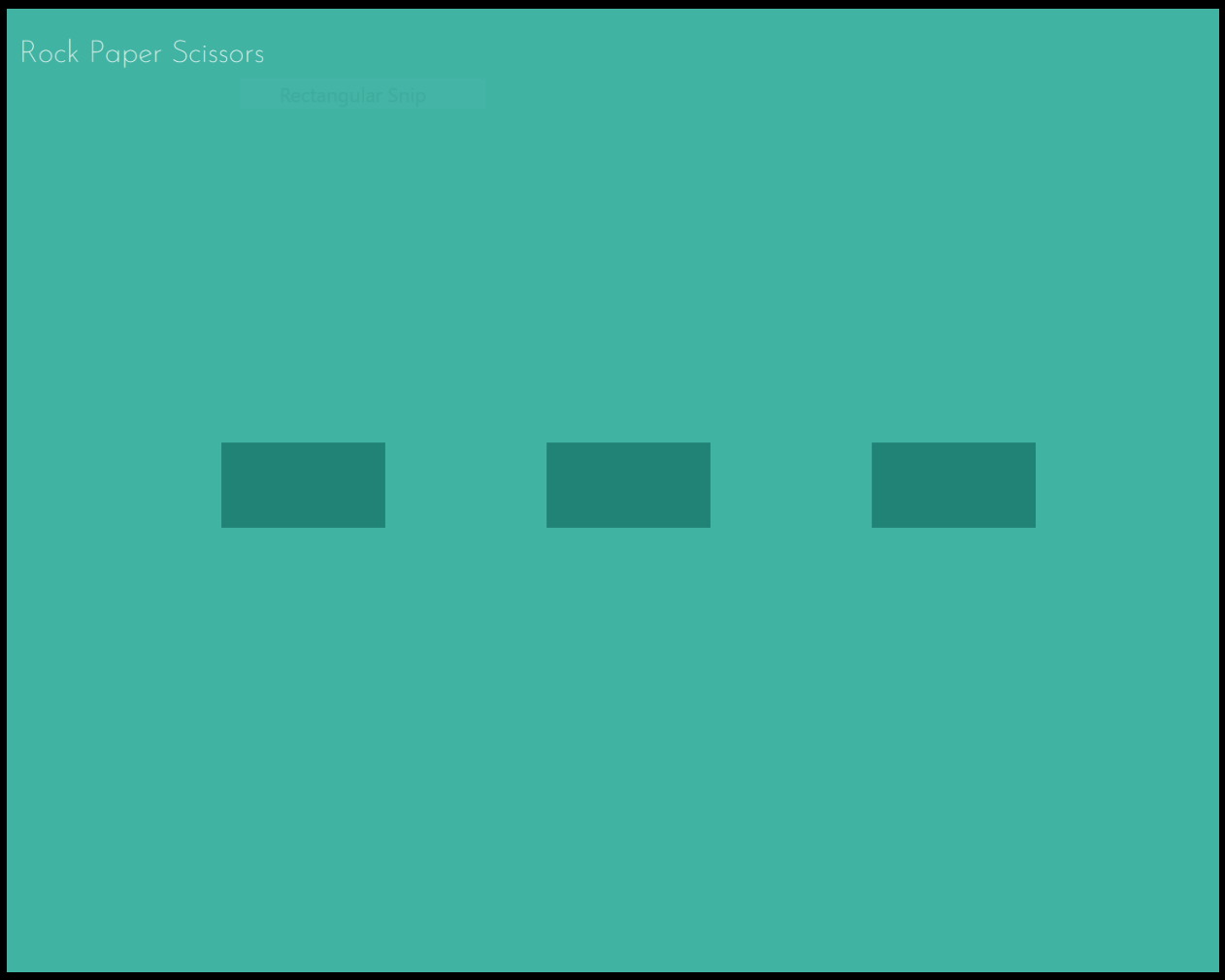
We have created a web page that allows players to interact with the game. The various elements on the web page can be used to submit inputs or display outputs. Let's start by taking a look at the source code.

Each HTML page usually starts with a <!DOCTYPE html> declaration to help the browser identify the document type. All content is contained within the <html></html> tags. The <head> tag contains metadata about the page; content that is not displayed by the browser. The <body> contains content that is displayed by the browser and directly viewable by the user.

In this lab, <head> contains the following elements:

- **<title>: Contains a title that is displayed by the browser on the page's tab.**
- **<script>: Contains Javascript code or points to a different file with Javascript code. In this lab, All Javascript will be written in a different file (created in a later step). Once the file/code is loaded by the browser, The Javascript is executed.**
- **<style>: Like a <script> element but for CSS, All CSS is either written in this tag or in a different file to which this tag refers. CSS is used to stylize the elements in the web page by specifying visual aspects such as margin, padding, color, etc.**

Your web page should look like the following -



There's not much to it right now but that's all right, We will be adding content and functionalities to the web page as we go along.

The Final Web Page

Some of the elements in the <body> are not visible in the web page you just saw due to them not containing any text or having their CSS display property set to hidden. So let's take a look at what the final web page will look like.

The elements in <body> can be seen in either the start or finish state of the game above. The elements can be identified by their type, class, or id. These identifiers are used in Javascript or CSS to refer to individual elements.
Go through index.html and make note of various elements along with their classes and ids.

Let's go through the elements we're working with :

- **1 - Status Display (start) - This heading has id #status-head, It displays the starting prompt.**
- **2 - Buttons (start) - Each of these is a <button> element. These need to be populated with the possible inputs. These buttons lead to the end-game.**
- **3 - Status Display (end) - The heading now displays the result of the game.**
- **4 - Buttons (end) - All but one button is visible. The button now has an updated prompt that takes the player back to the start screen.**
- **5 - Move Display (end) - These are <h2> elements in a container with class .move-display. Note how they were hidden in the starting state.**

Note: Classes are referred to using .className and ids are referred to using #idName when using CSS selectors.

Our goal will be to transition from one state to another when the appropriate action is taken. In addition, We will generate content for each state and update the web page accordingly.

Defining Some Variables

Javascript is a loosely typed and dynamically typed language.

That is when declaring a variable you don't have to specify its 'type'. Reassigning declared variables to a different type is perfectly legal in Javascript.

There are six primitive types in Javascript. In this lab, we will only be dealing with strings, numbers, and booleans. We will also see composite data types such as arrays, objects and functions.

Let's start by declaring some useful variables that we will later use in our program.

Create another file and name it script.js

Add the following to script.js. We create block-scoped variables in JavaScript using the let keyword.

```
1  let winMsg = 'Victory';
2  let loseMsg = 'Crushing Defeat';
3  let tieMsg = 'Tie';
```

Further, It would be useful to have a list of all possible moves in the game. Let's do this in the form of an array. Add the following below the previous lines of code.

```
1  let moveList =['Rock','Paper','Scissors'];
```

Now we can access each move using its corresponding index in the array. For example "Rock" can be accessed using moveList[0].

All the changes we made right now may not be visible on the web page, but the browser now knows what winMsg and loseMsg is.

Save script.js before moving to the next step.

Functions

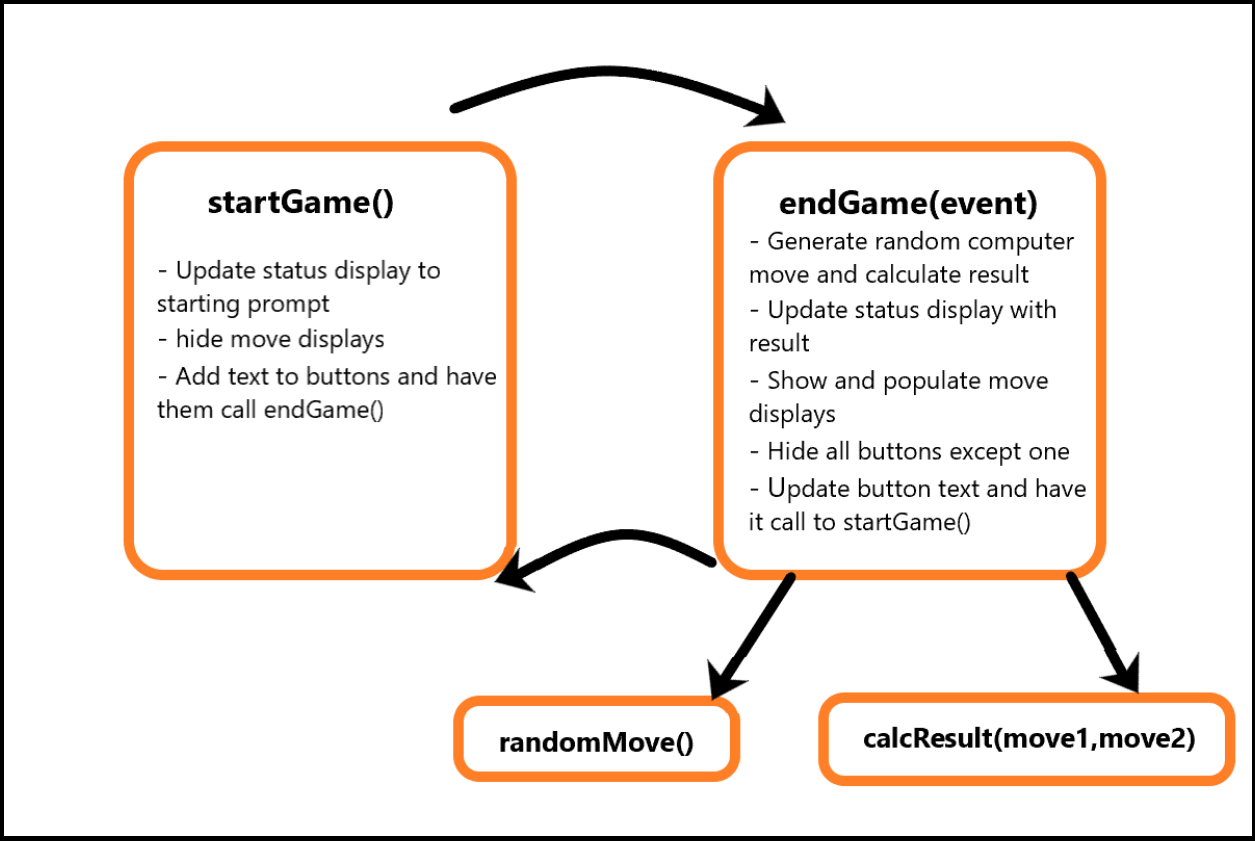
Similar to how variables allow us to organize our data, functions allow us to organize our code. Functions are clean, isolated pieces of code that can have data passed into them as inputs. We will be using functions to go from one state of the game to the other (start to finish and vice versa) as well as achieve certain functionalities in each of those states.

Functions in Javascript can be declared in any of the following ways.

```
// Option 1
function startGame(argument) {
  return;
}
// Option 2

let startGame = function (argument) {
  return;
};

// Option 3
let startGame = (argument) => {
  return;
};
```



We will be implementing the following functions through the course of this lab.

- **startGame() - Starts the game and displays input prompts.**
- **endGame(event) - Triggered by user input. Computes and displays the end of the game.**
- **calcResult(move1,move2)- Called in endGame() to compute the result and return it as a string. The moves are a number between 0 - 2 corresponding to the index of the moveList. A victory is returned when move 1 wins over move 2.**
- **randomMove() - A function that returns a random number between 0 - 2 . Used to generate the computer move in calcResult().**

Add the following below the variable declarations in script.js and save the file.

```
/**
 * Computes result of the game. returns victory message if move 1 wins.
 * @param  {Number}   move1  move 1
 * @param  {Number}   move2  move 2
 * @return {String}   result result of the game
 */

function calcResult(move1, move2) {}

/**
 * @return {Number}   random number between 0 and 2
 */

function randomMove() {
  return Math.floor(Math.random() * 3);
}

/**
 * Displays start state of game
 */

function startGame() {}

/**
 * Displays end state of game
 * @param {Event} event event containing information of users input.
 */

function endGame(event) {}

startGame();
```

In the above block of code, we've outlined what our final script will look like.

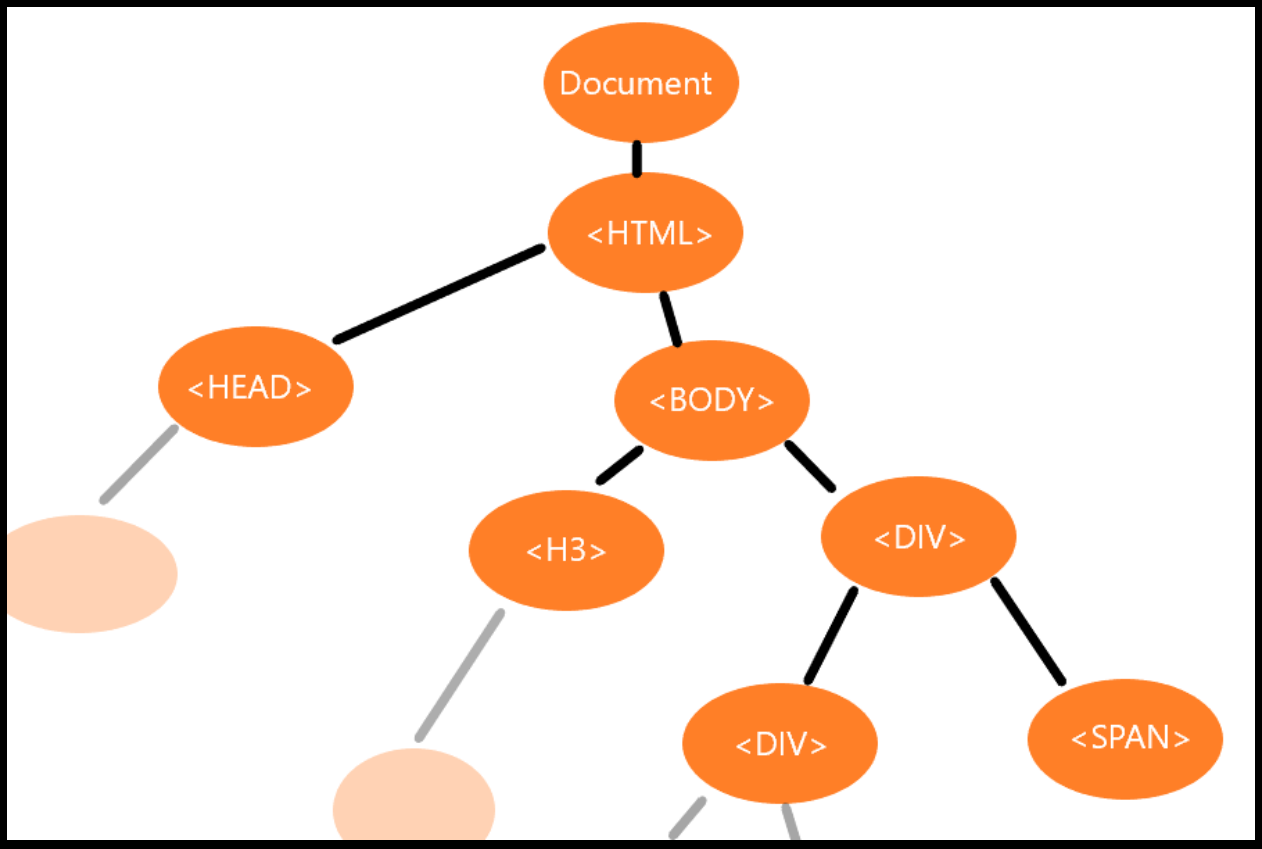
randomMove() has already been implemented. It simply uses the predefined Math.floor() (performs the floor operation on a number) and Math.random() (returns a random number between 0 and 1) functions to generate a value between 0 and 2.

Towards the end, we call the function startGame() to move our game to its starting state. startGame() will be executed as soon as the script loads.

The DOM

Modern browsers have a lot of useful functionalities built into them in the form of browser APIs.

One such incredibly useful interface is the Document Object Model (or the DOM). The DOM represents the HTML document as a tree where each tag or element is represented as a node of the tree. Further, we can modify the content or attributes of each node.



We can traverse the whole document by accessing the parents, siblings, or children of an element.

Alternatively, We can fetch certain elements using some useful searching functions.

To get a desired element, We can use document.querySelector which allows elements to be selected using CSS selectors. Add the following to script.js below your variable declarations and save.

```
1   let statusDisplay = document.querySelector('#status-head');
```

Here, we declare a new variable called **statusDisplay** and store the first element that matches the CSS query **#status-head** (i.e. an element with the id **status-head**). Further, if we want to access multiple elements matching a criteria, for example all the buttons or all the move displays, we can use **document.querySelectorAll**. Add the following below the variable declarations in script.js and save.

```
let moveDisplays = document.querySelectorAll('.move-display h2');
let buttons = document.querySelectorAll('button');
```

Here, we take all **h2** elements which are children of an element with class **.move-display** and store it in **MoveDisplays**. Similarly, we store all buttons in the DOM under **buttons**.

Now that we have access to elements of the nodes, let's see how we can manipulate them.

Note: moveDisplays and buttons are a list of elements. These lists are called 'node lists' and are similar to arrays. For the purposes of this lab, you can treat them identical to arrays.

Objects

Before we move further, Let's define what the nodes are - Objects.

An Object is a collection of properties - key-value pairs. Each key has a designated value that can store a data type such as a Number, another object, or even a function. Each property can be accessed using the dot notation like so - objectName.keyName. Similarly, A method can be called using objectName.methodName().
Recall using the document object in the DOM. We accessed the body node using document.body and used the document.querySelector method to select elements.
Similarly, all nodes in the DOM have various properties that can be manipulated to change the DOM.
Add the following to the startGame() function in script.js and save the file.

```
1   statusDisplay.textContent = 'Choose!';
```

statusDisplay.textContent refers to the text stored in the <h1> tag that stausDisplay refers to. We updated the status display text to show the starting prompt.

Take a look at index.html by previewing it with Live Server. You should now see some text.

Loops

Our starting screen is coming together. Now let's add some text to the buttons. Considering the operations we have to perform for each button are fairly similar, It makes sense to use a loop for them.

A loop in Javascript can be defined as follows.

```
for (let i = 0; i < buttons.length; i++) {
  buttons[i].textContent = moveList[i];
  buttons[i].style.display = "inline-block";
}
```

Here, we initialize a variable i with the value 0. We pass in a condition that runs if i is lower than buttons.length. Finally, after the code is complete we increment i. We use the previously defined move list to add text to each button. We also make sure the display of each button is set to inline-block to make sure all the buttons are visible at this stage.

Alternatively, the following syntax can be used for the loop

```
buttons.forEach((button, index) => {
  button.textContent = moveList[index];
  button.style.display = "inline-block";
});
```

This achieves the same functionality as described above. Let's break down the syntax. buttons refers to our list of button elements. Arrays (and other iterable objects) can access the method Array.forEach() to iterate over themselves. The argument for Array.forEach() is an arrow function (Option 3 in function syntax) which is executed over each element of the array. This arrow function further has two arguments - an element of the array and a counter that increments each loop. The second argument, i.e. the counter, is optional

To summarize:

```
arrayName.forEach(
  // The function defined here will execute once for each element of the array
  (element, counter) => {
    // Any change made to element in this function will occur to each element of the array
    // counter counts the iteration of the loop ( starting from 0)
  }
);
```

The latter syntax might be harder to unpack but it does result in more readable (and consequently, easier to debug) code.
Add one of the above loops that interact with the buttons to the startGame() function in script.js and save the file.

As a challenge, make a loop that sets the CSS display property of each element in the moveDisplays to none in startGame().

Take a look at what index.html looks like at this point, It should match the image in earlier steps. However, you might note that the buttons still do not perform any function.

Event Listeners

In Javascript, we can have functions executed in response to certain actions that take place in a browser. Such actions are called 'events'. Examples include scrolling, the document loading, or in our case, a button being clicked.

An event listener is 'attached' to an element. It continues to check for event execution and calls a function when that event is triggered. The triggering event itself is an object with useful information.

Update your startGame() in script.js to your logic and save the file.

```
/**
 * Displays start state of game
 */
function startGame() {
    //Your Code logic
 }
```

Event listeners are attached to elements where the expected action takes place using - element.addEventListener(eventType,listener). Here, listener is a function that receives the triggering event as an object. Notice how when functions are arguments the parenthesis next to their name is omitted.

Game Logic

As a challenge, try and implement calcResult() on your own. Recall, this function is supposed to calculate the result (did you win, tie, or lose), and return the value as a string. The parameters being passed into the function are move1 and move2.

if{..} else{..} blocks have the following syntax.

```
if (condition_1 || conditon_2) {
  // condition 1 or condition 2
  // Code goes here
} else if (condition_3 || conditon_4) {
  // condition 1 and condition 2
  // Code goes here
} else {
  // Code goes here
}
```

In addition, equality may be checked with == and inequality may be checked with !=.

Hint

Remember that the moveList is an array ["Rock", "Paper", "Scissors"]. What condition, using moveList's indices, will evaluate to the correct result when compared. For example, what condition creates the *win* scenario. How can you represent that condition using the indices?

Make sure you complete calcResult() in script.js and save it before moving further.

End The Game

We are almost at the end! Finally lets implement the ending state of the game. Here we bring together everything we've implemented in the previous steps.

Update your endGame() to create your logic and save it.

```
/**
 * Displays end state of game
 * @param {Event} event event containing information of users input.
 */

function endGame(event) {
    // Your code
 }
```

There is a lot to unpack here.

—End of Document—