Rock, Paper, Scissors with Javascript

In this guided 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 datatypes
- Define functions and objects
- Use control statements
- Manipulate the DOM
- Demonstrate the use of events/event listeners

Prerequisites

For this lab, you will need to know basic HTML and CSS syntax.

This lab assumes familiarity with fundamental programming concepts and only illustrates features/syntax unique to Javascript. For more in-depth introductory courses, visit the courses here.

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 clientside 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

To begin, select **File** -> **New File** and create a file called **index.html**:

Fig 2.1 : Creating index.html

Creating The Web Page

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

```
1.  1
2.  2
3.  3
4.  4
5.  5
6.  6
7.  7
8.  8
9.  9
10. 10
11. 11
12. 12
13. 13
14. 14
15. 15
16. 16
17. 17
18. 18
19. 19
20. 20
```

21. 21
22. 22
23. 23
24. 24
25. 25
26. 26
27. 27
28. 28
29. 29
30. 30
31. 31
32. 32
33. 33
34. 34
35. 35
36. 36
37. 37
38. 38
39. 39
40. 40
41. 41
42. 42
43. 43
44. 44
45. 45
46. 46
47. 47
48. 48
49. 49
50. 50
51. 51
52. 52
53. 53
54. 54
55. 55
56. 56
57. 57
58. 58
59. 59
60. 60
61. 61
62. 62
63. 63
64. 64
65. 65
66. 66
67. 67
68. 68
69. 69
70. 70
71. 71
72. 72
73. 73
74. 74
75. 75
76. 76
77. 77
78. 78
79. 79
80. 80
81. 81
82. 82
83. 83
84. 84
85. 85
86. 86
87. 87
88. 88
89. 89
90. 90
91. 91
92. 92
93. 93
94. 94
95. 95
96. 96
97. 97
98. 98
99. 99
100. 100
101. 101

```
102. 102
103. 103
104. 104
105. 105
106. 106
107. 107
108. 108
109. 109
110. 110
111. 111
112. 112
113. 113
114. 114
115. 115
116. 116
117. 117
118. 118
119. 119
120. 120
121. 121
122. 122
123. 123
124. 124
125. 125
126. 126
127. 127
128. 128
129. 129
130. 130
```

```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.
4.  <head>
5.    <title>Rock, Paper, Scissors</title>
6.    <script src="script.js" defer></script>
7.    <style>
8.      @import url('https://fonts.googleapis.com/css2?family=Josefin+Sans:ital,wght@0,100;1,700&family=PT+Sans&display=swap');
9.
10.     body {
11.       background-color: #41B3a3;
12.       overflow: hidden;
13.
14.     }
15.
16.     .main-container {
17.       width: 40vw;
18.       height: 70vh;
19.       margin: auto;
20.       display: grid;
21.       grid-template-columns: repeat(3, 1fr);
22.       grid-template-rows: auto auto auto;
23.       justify-content: center;
24.
25.     }
26.
27.     .title-small-wrapper {
28.       background-color: aqua;
29.
30.     }
31.
32.     h3 {
33.       size: 1em;
34.       font-family: 'Josefin Sans', sans-serif;
35.       font-weight: lighter;
36.       color: white;
37.
38.     }
39.
40.     .status-head-wrapper {
41.       grid-column-start: 1;
42.       grid-column-end: 4;
43.       grid-row-start: 2;
44.
45.       display: flex;
46.       align-items: flex-end;
47.
48.       justify-content: space-around;
49.
50.     }
51.
```

```
 52.      .status-head-wrapper h1 {
 53.        font-family: 'PT Sans', sans-serif;
 54.        font-size: 8em;
 55.        margin-bottom: auto;
 56.
 57.        color: white;
 58.      }
 59.
 60.      .game-button-wrapper {
 61.
 62.        grid-row-start: 3;
 63.        grid-column: 1/span 4;
 64.        display: flex;
 65.        align-items: flex-start;
 66.
 67.        justify-content: space-around;
 68.      }
 69.
 70.      button {
 71.        margin-top: auto;
 72.        color: #FFF;
 73.        margin: 0;
 74.        background-color: #208376;
 75.        border: 2px solid rgba(255, 255, 255, 0);
 76.        position: relative;
 77.        border-radius: 0px;
 78.        padding: 1.5em 3em;
 79.        display: inline-block;
 80.        font-family: "Lucida Console", Monaco, monospace;
 81.        font-size: 1em;
 82.        letter-spacing: 1px;
 83.        cursor: pointer;
 84.        margin-left: 1em;
 85.      }
 86.
 87.      button:hover,
 88.      button:active {
 89.        border: 2px solid rgb(255, 255, 255);
 90.      }
 91.
 92.      .move-display {
 93.        grid-row-start: 4;
 94.        grid-column: 1/span 4;
 95.
 96.        color: white;
 97.        font-family: 'PT Sans', sans-serif;
 98.        display: flex;
 99.        align-items: flex-start;
100.        justify-content: space-around;
101.      }
102.
103.      @media (max-width: 1000px) {
104.        .main-container {
105.          width: 80vw;
106.        }
107.      }
108.    </style>
109.  </head>
110.
111.  <body>
112.    <h3 id="title-small">Rock Paper Scissors</h3>
113.    <div class="main-container">
114.      <span class="status-head-wrapper">
115.        <h1 id="status-head"> </h1>
116.      </span>
117.      <div class="move-display">
118.        <h2></h2>
119.        <h2></h2>
120.
121.      </div>
122.      <div class="game-button-wrapper">
123.        <button id="rock-button" type="button"> </button>
124.        <button id="paper-button" type="button"> </button>
125.        <button id="scissors-button" type="button"> </button>
126.      </div>
127.    </div>
128.  </body>
129.
130.  </html>
```

[Copied!]

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.

Make sure you save the file after making changes. Created files can be saved in the following manner -

Fig 3.1 : Saving a File

Take a look at what the page currently looks like by saving your file, right clicking on it in the File Explorer & selecting **Open with Live Server** .

This should show a notification mentioning that the server has started on port 5500.

Click the Skills Network button on the left to open the "Skills Network Toolbox". Click "Other" then "Launch Application". From there, enter the port number as **5500** and launch your application.

Fig 3.2 : Previewing index.html

Your web page should look like the following -

Fig 3.3 : initial web page

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.

Fig 4.1 : The finished game

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. 1
2. 2
3. 3
```

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

Copied!

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. 1
```

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

Copied!

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 now browser 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.

```
1.  1
2.  2
3.  3
4.  4
5.  5
6.  6
7.  7
8.  8
9.  9
10. 10
11. 11
12. 12
13. 13
14. 14
```

```
1.  // Option 1
2.  function startGame(argument) {
3.     return;
4.  }
5.  // Option 2
6.
7.  let startGame = function (argument) {
8.     return;
9.  };
10.
11. // Option 3
12. let startGame = (argument) => {
13.     return;
14. };
```

Copied!

Fig 6.1 : Game Functions

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.

```
1.  1
2.  2
3.  3
4.  4
5.  5
6.  6
7.  7
8.  8
9.  9
10. 10
11. 11
12. 12
13. 13
14. 14
15. 15
16. 16
17. 17
18. 18
19. 19
20. 20
21. 21
22. 22
23. 23
24. 24
25. 25
26. 26
27. 27
28. 28
29. 29
30. 30
31. 31
```

```
1.  /**
2.   * Computes result of the game. returns victory message if move 1 wins.
3.   * @param  {Number}   move1  move 1
4.   * @param  {Number}   move2  move 2
5.   * @return {String}   result result of the game
6.   */
7.
8.  function calcResult(move1, move2) {}
9.
10. /**
11.  * @return {Number}    random number between 0 and 2
12.  */
13.
14. function randomMove() {
15.   return Math.floor(Math.random() * 3);
16. }
17.
18. /**
19.  * Displays start state of game
20.  */
21.
22. function startGame() {}
23.
24. /**
25.  * Displays end state of game
26.  * @param {Event} event event containing information of users input.
27.  */
28.
29. function endGame(event) {}
30.
31. startGame();
```

Copied!

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.

Fig 7.1 : The DOM Tree

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. 1

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

Copied!

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.

1. 1
2. 2

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

Copied!

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. 1

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

Copied!

`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.

```
1  for (let i = 0; i < buttons.length; i++) {
2    buttons[i].textContent = moveList[i];
3    buttons[i].style.display = "inline-block";
4  }
```

Copied!

Here, we initalize 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

```
1  buttons.forEach((button, index) => {
2    button.textContent = moveList[index];
3    button.style.display = "inline-block";
4  });
```

Copied!

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 summarise:

```
1  arrayName.forEach(
2    // The function defined here will execute once for each element of the array
3    (element, counter) => {
4      // Any change made to element in this function will occur to each element of the array
5      // counter counts the iteration of the loop ( starting from 0)
6    }
7  );
```

Copied!

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()`.

▼ Click here for the solution
Add the following to `startGame()` in **script.js** and save.

```
1  1
```

```
  1. moveDisplays.forEach(moveDisplay => moveDisplay.style.display='none');
```

[Copied!]

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 execute 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 match the following code and save the file.

```
 1. 1
 2. 2
 3. 3
 4. 4
 5. 5
 6. 6
 7. 7
 8. 8
 9. 9
10. 10
11. 11
12. 12
```

```
 1. /**
 2.  * Displays start state of game
 3.  */
 4. function startGame() {
 5.    statusDisplay.textContent = "Choose!";
 6.    buttons.forEach((button, index) => {
 7.      button.textContent = moveList[index];
 8.      button.style.display = "inline-block";
 9.      buttons[index].addEventListener("click", endGame);
10.    });
11.    moveDisplays.forEach((moveDisplay) => (moveDisplay.style.display = "none"));
12. }
```

[Copied!]

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 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.

```
1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. 9
```

```
1. if (condition_1 || conditon_2) {
2.   // condition 1 or condition 2
3.   // Code goes here
4. } else if (condition_3 || conditon_4) {
5.   // condition 1 and condition 2
6.   // Code goes here
7. } else {
8.   // Code goes here
9. }
```

Copied!

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?

▼ Click to show

Here is an implementation of `calcResult()` -

1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. 9
10. 10
11. 11
12. 12
13. 13
14. 14
15. 15
16. 16
17. 17

```
1. /**
2.  * Computes result of the game. returns victory message if move 1 wins.
3.  * @param  {Number}   move1  move 1
4.  * @param  {Number}   move2  move 2
5.  * @return {String}   result result of the game
6.  */
7. function calcResult(move1, move2) {
8.    var result = move1 - move2;
9.
10.   if (result == 1 || result + 3 == 1) {
11.      return winMsg;
12.   } else if (result == 2 || result + 3 == 2) {
13.      return loseMsg;
14.   } else {
15.      return tieMsg;
16.   }
17. }
```

Copied!

Note there are multiple ways to implement this function.

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

End the Game

We are almost at the end! Finally let's implement the ending state of the game. Here we bring together everything we've implemented in the previous steps.
Update your `endGame()` to match the following and save it.

1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. 9
10. 10
11. 11
12. 12
13. 13
14. 14
15. 15

```
16. 16
17. 17
18. 18
19. 19
20. 20
21. 21
22. 22
23. 23
24. 24
25. 25
26. 26
27. 27
```

```
 1. /**
 2.  * Displays end state of game
 3.  * @param {Event} event event containing information of users input.
 4.  */
 5.
 6. function endGame(event) {
 7.   let playerMove = moveList.indexOf(event.target.textContent);
 8.   let computerMove = randomMove();
 9.
10.   statusDisplay.textContent = calcResult(playerMove, computerMove);
11.
12.   moveDisplays.forEach(
13.     (moveDisplay) => (moveDisplay.style.display = "inline-block")
14.   );
15.   moveDisplays[0].textContent = `You played ${moveList[playerMove]}`;
16.   moveDisplays[1].textContent = `Computer played ${moveList[computerMove]}`;
17.
18.   buttons.forEach((button, index) => {
19.     if (index == 1) {
20.       button.textContent = "Play Again";
21.       button.removeEventListener("click", endGame);
22.       button.addEventListener("click", startGame);
23.     } else {
24.       button.style.display = "none";
25.     }
26.   });
27. }
```

Copied!

There is a lot to unpack here. Let's go through a few key pieces of code

- `function endGame(event)` - The argument event contains the event object that triggered the `endGame()` function. In this case, the event is the user clicking a button corresponding to the move
- `let playerMove = moveList.indexOf(event.target.textContent);` - event.target refers to element associated with the event i.e the clicked button. Consequently, `event.target.textContent` refers to the text of the clicked button. Finally, we use `moveList.indexOf()` to find where the users move lies in the moveList and hence get the move as a number between 0-2.
- `let computerMove = randomMove();` - The computers move is generated randomly using the earlier implemented function. -`statusDisplay.textContent = calcResult(playerMove,computerMove);` - The result of the game as returned by `calcResult()` is assigned to the `textContent` of the `statusDisplay` element, thereby displaying the result in the DOM.
- `moveDisplays[1].textContent = "Computer played ${moveList[computerMove]}";` - The move played by the computer is displayed by setting the `textContent` of the `moveDisplays` to template strings. Template strings allow variables to be embedded in a string. They use back-ticks `\...` instead of regular quotes and any expression placed inside`${}` is evaluated and substituted.
- `buttons.forEach((button,index)=>{...});` - All buttons except one (index 1) are hidden by setting their display to `none`. The previous event listener on the remaining button is removed and one corresponding to `startGame()` is added.

Final code

You should now have a working game! Now take look at **index.html** one final time and make sure the game works as expected. Play a few rounds and make sure you get the right result.
You can cross-reference your final code with the one below.

```
 1. 1
 2. 2
 3. 3
 4. 4
 5. 5
 6. 6
 7. 7
 8. 8
 9. 9
10. 10
```

11. 11
12. 12
13. 13
14. 14
15. 15
16. 16
17. 17
18. 18
19. 19
20. 20
21. 21
22. 22
23. 23
24. 24
25. 25
26. 26
27. 27
28. 28
29. 29
30. 30
31. 31
32. 32
33. 33
34. 34
35. 35
36. 36
37. 37
38. 38
39. 39
40. 40
41. 41
42. 42
43. 43
44. 44
45. 45
46. 46
47. 47
48. 48
49. 49
50. 50
51. 51
52. 52
53. 53
54. 54
55. 55
56. 56
57. 57
58. 58
59. 59
60. 60
61. 61
62. 62
63. 63
64. 64
65. 65
66. 66
67. 67
68. 68
69. 69
70. 70
71. 71
72. 72
73. 73
74. 74
75. 75
76. 76
77. 77
78. 78
79. 79
80. 80
81. 81

```
 1. let winMsg = "Victory";
 2. let loseMsg = "Defeat";
 3. let tieMsg = "Tie";
 4.
 5. let moveList = ["Rock", "Paper", "Scissors"];
 6.
 7. let statusDisplay = document.querySelector("#status-head");
 8. let moveDisplays = document.querySelectorAll(".move-display h2");
 9. let buttons = document.querySelectorAll("button");
```

```
 10.
 11.  /**
 12.   * Computes result of the game. returns victory message if move 1 wins.
 13.   * @param  {Number}    move1  move 1
 14.   * @param  {Number}    move2  move 2
 15.   * @return {String}    result result of the game
 16.   */
 17.
 18.  function calcResult(move1, move2) {
 19.    var result = move1 - move2;
 20.
 21.    if (result == 1 || result + 3 == 1) {
 22.      return winMsg;
 23.    } else if (result == 2 || result + 3 == 2) {
 24.      return loseMsg;
 25.    } else {
 26.      return tieMsg;
 27.    }
 28.  }
 29.
 30.  /**
 31.   * @return {Number}    random number between 0 and 2
 32.   */
 33.
 34.  function randomMove() {
 35.    return Math.floor(Math.random() * 3);
 36.  }
 37.
 38.  /**
 39.   * Displays start state of game
 40.   */
 41.
 42.  function startGame() {
 43.    statusDisplay.textContent = "Choose!";
 44.    buttons.forEach((button, index) => {
 45.      button.textContent = moveList[index];
 46.      button.style.display = "inline-block";
 47.
 48.      buttons[index].removeEventListener("click", startGame);
 49.      buttons[index].addEventListener("click", endGame);
 50.    });
 51.    moveDisplays.forEach((moveDisplay) => (moveDisplay.style.display = "none"));
 52.  }
 53.
 54.  /**
 55.   * Displays end state of game
 56.   * @param {Event} event event containing information of users input.
 57.   */
 58.
 59.  function endGame(event) {
 60.    let playerMove = moveList.indexOf(event.target.textContent);
 61.    let computerMove = randomMove();
 62.
 63.    statusDisplay.textContent = calcResult(playerMove, computerMove);
 64.    moveDisplays.forEach(
 65.      (moveDisplay) => (moveDisplay.style.display = "inline-block")
 66.    );
 67.    moveDisplays[0].textContent = `You played ${moveList[playerMove]}`;
 68.    moveDisplays[1].textContent = `Computer played ${moveList[computerMove]}`;
 69.
 70.    buttons.forEach((button, index) => {
 71.      if (index == 1) {
 72.        button.textContent = "Play Again";
 73.        button.removeEventListener("click", endGame);
 74.        button.addEventListener("click", startGame);
 75.      } else {
 76.        button.style.display = "none";
 77.      }
 78.    });
 79.  }
 80.
 81.  startGame();
```

[Copied!]

Congratulations! You have finished the lab. In this lab, you have implemented a version of Rock, Paper, and Scissors. There are several ways to do this so feel free to iterate on this implementation and add your features using the concepts you just learned.

Summary

In this lab, you have learned how to -

- Describe Javascript as a dynamically typed web-oriented language
- Work with primitive data types such as strings, numbers, and booleans and complex data types such as functions and objects
- Define functions and objects and use them in various contexts
- Use control statements such as `for` loops and `if{..} else{..}` statement
- Manipulate the DOM using the DOM API and navigate the DOM tree
- Demonstrate the use of event handlers to execute functions and manipulate event objects

Next Steps

In this lab you learnt Javascript and worked with primitive data types. You used functions and objects and control statements. You manipulated and navigated the DOM using the DOM API and used event handlers to execute functions and manipulate event objects. If you are interested in continuing to learn Javascript, you can check out the https://www.edx.org/course/introduction-to-cloud-development-with-html5-css3-and-javascript course.