Rock-paper-scissors game

Today we're going to code a small web page that contains a Rock Paper Scissors game. This will give you a basic understanding of the three languages of front-end web development — HTML (HyperText Markup Language), JavaScript, and CSS (Cascading StyleSheets).

This step-by-step guide is intended to go alongside the Grow workshop and is meant to help you along if you happen to need some inspiration. However, if you haven't been able to attend a Grow event, you're still welcome to follow this guide on your own.

Tools & Resources

For this exercise you'll need a computer with a modern web browser (Chrome, Safari, Firefox). In a normal work environment, you would also have your code editor installed on your machine. But to help you get straight to coding, we've set up a virtual environment for you on CodeSandbox, a free online code editor.

Here's a list of all the links that you'll find handy:

- The CodeSandbox exercise: https://codesandbox.io/s/udxo1
- The complete solution [SPOILER ALERT]: https://codesandbox.io/s/y3mq2wk67x
- A handy cheat sheet: https://drive.google.com/file/d/18hbSYsKP5d-woe0LQzQlu6YU1bx3v_od/view

HTML: Build the structure

Since we're making a rock paper scissors game, we'll need a few things:

- A title
- 3 buttons to play: rock, paper, scissors
- A line about what we played
- A line about what the opponent played
- A line about the outcome of the game

Start by opening the CodeSandbox link. You will want to write your HTML code in the **index.html** file.

Adding a title

Not to be confused with the **<title>** element inside the **<head>** section (which tells the browser what to display in the page's tab or title bar), we'd normally use a **heading** element, ranging from **<h1>** to **<h6>**, for the title. Think of headings as the titles of the sections listed in a book's index. You should never skip levels (an h1 should always be followed by an h2, not an h3), and your page should only have one h1, the same way a book would only have one title.

In the index.html file, add a main heading like so:

```
<h1>Rock Paper Scissors!</h1>
```

Adding the buttons

In order to play the game, we'll want to be able to interact with the site. There are a few HTML elements that allow that, and one of them is the **button** element.

We will need to add three buttons: one for playing "paper", one for "scissors", and one for "rock".

```
<div>
    <button>Rock</button>
    <button>Paper</button>
    <button>Scissors</button>
</div>
```

At the moment, nothing will happen when you click the buttons. We'll need to trigger a function when we click them (don't worry, we'll get to that in the Javascript section).

Adding the results

Finally, we will want to see three paragraphs that tell us what we played, what the opponent (the computer) played, and who won. For that, we'll use the **<p>** element, which stands for paragraph.

```
<p>You played scissors</p>
<p>Your opponent played paper</p>
<p>Congratulations, you won!</p>
```

Right now, these paragraphs have fixed data. Later on, we will replace their contents with what has actually been played.

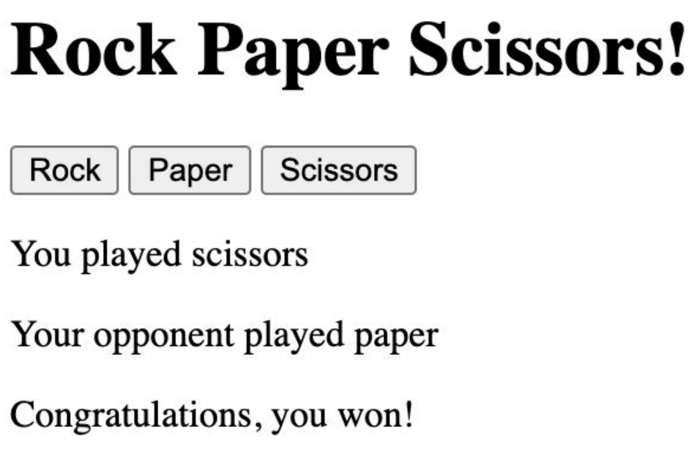
Review your HTML

Your initial HTML is done! You are now ready to start adding interaction to your game. Like a house's walls, this HTML file will be the structure of your page. Your final HTML should look something like this:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>Rock Paper Scissors</title>
  </head>

  <body>
    <h1>Rock Paper Scissors!</h1>
    <div>
      <button>Rock</button>
      <button>Paper</button>
      <button>Scissors</button>
    </div>
    <p>You played scissors</p>
    <p>Your opponent played paper</p>
    <p>Congratulations, you won!</p>
  </body>
</html>
```

You might've chosen different words, so don't worry if it doesn't look exactly the same. The main thing to watch out for is that a tag you're opening (e.g. <h1>) is always being closed by its counterpart (e.g. </h1>). Don't forget to save the index.html file! Depending on your browser, your page should look something like this:


Rock Paper Scissors!

You played scissors

Your opponent played paper

Congratulations, you won!

Javascript: Add interaction

Now comes the more challenging part. If HTML is like the structure of a house — the walls, floor, and ceiling — then JavaScript is like the dishwasher, TV remote, heat pump, washing machine, microwave, and so on. It's what gives us the ability to make a website behave in a certain way — to open a popup when a button is clicked, to change what is shown on the page when different users are logged in, to reveal what's hidden when you click a square on a game board, etc.

Including JavaScript in HTML

First of all, we need to have a way of including JavaScript in our HTML file. We can do this by adding the HTML <**script>** element to our index.html file, and then either writing JavaScript code directly between the opening and closing <script> tags, or adding the location of our JavaScript file as the source for the script element using the **src** attribute:

```
<script src="./script.js"></script>
```

The best place to add a <script> is generally right before the closing tag of the HTML file's <**body>** element.

Using the onclick attribute

As we mentioned in the HTML section, we're going to need to have a way to trigger something to happen when we click one of our rock, paper, or scissors buttons. We'll need to:

1. Tell JavaScript what we've played
2. Generate the computer's move
3. Update our HTML to show what we played and what the computer played
4. Compare the computer's move to what we played, and decide whether we won, lost, or tied
5. Update our HTML to say what the result of the game was.

If that seems like a lot, don't worry, we're going to take this step by step. The first thing we need to do — 'Tell JavaScript what we've played' — seems like it will need some kind of interaction between our HTML buttons and our JavaScript. We can do that by adding a special attribute to our HTML buttons — the **onclick** attribute. The syntax for this is as follows:

```
<button onclick="play('rock')">Rock</button>
```

What we're doing here is saying that when this button is clicked, we want to call a JavaScript function called **play()**, and pass it the argument **'rock'**. The browser will go and look in our JavaScript file for a function with this name, and call it — that is, run the code inside it — with this argument.

So what is a **function**, exactly? A function is a reusable piece of JavaScript code that allows us to repeat a piece of functionality as many times as we want. We write a **function definition** and then later **call** the function when we want to use that piece of code.
When we write a function definition, we can optionally give the function one or more **parameters** — these are placeholders for the **arguments** that we'll pass into the function

when we call it (in the above example, ‘rock’ is the argument we’re passing into the play function). We add parameters to our function definition to stand in for values that the function needs to use, but that will vary each time we call the function.

Writing our play() function

Now if we open up our **script.js** file, we’ll see that the function **play()** is already there, waiting for us to add some code inside it:

```
function play(myMove) {
 // Determine what the result of the game is
 const result = 'win';

 /* This is showing the result in the console
 How do we show the result in a HTML element instead? */
 console.log(result);
}
```

What this function (as it’s currently written) will do is set a variable named **result** to the string ‘win’, and then log this to the console. If you open up the console from the bottom right of the CodeSandbox browser window, and then click on one of the three buttons we have on our page, you should see the word ‘win’ appear in the console. Click another button, and another ‘win’ appears below it. Logging to the console is a very handy tool for checking that the code we’ve written is working as expected, or trying to figure out what the problem is if it’s not working as expected (this is called ‘debugging’).

But we don’t need the code that logs ‘win’ to the console, as this isn’t what we want our play() function to do. You can remove all the code inside the function, and we’ll write some new code in its place.

You might have also noticed that our play() function takes a **parameter** named **myMove**. Depending on which button we click to call the play() function, our function call could look like one of the following: **play(‘rock’)**, **play(‘paper’)**, **play(‘scissors’)**.

Although we’re not using myMove anywhere yet inside the body of our function, we know we’ll want to use it, because we’ll need to write some code to decide whether we won, lost, or tied depending on what we played and what the computer played.

Generating the computer’s move

But before we can compare what we played with what the computer played, we’ll need to generate the computer’s move. We’ll do this by randomly setting a variable, **opponentMove**, to be either ‘rock’, ‘paper’, or ‘scissors’.

How do we do this? Well, JavaScript has a few features that can help us out here. It has a built-in **Math** object with a couple of functions on it that could be useful:

- **Math.random()**, which generates a random number between 0 and 1.
- **Math.floor()**, which rounds any number that's passed in as an argument down to the nearest integer below it. So calling Math.floor(5.78), for example, will give us the integer 5 as the returned value.

Although Math.random() generates a number between 0 and 1, we can make it generate a number between 0 and any positive number simply by multiplying our Math.random() function call by that number:

```
Math.random() * 3;
```

This will give us a number somewhere between 0 and 3 — it could be 0, it could be 1, it could be 2, or it could be 0.173994932 or 2.747737822 or 1.590096549. So that we only get one of three possible whole integers returned — 0, 1, or 2 — we can use Math.floor() to round the number that Math.random() * 3 generates down to the nearest integer below it:

```
Math.floor(Math.random() * 3);
```

Here we're calling Math.floor(), and passing in as an argument the number that is generated by calling Math.random() and multiplying its returned value by three. (Note that a **returned value** is the value that is passed out of the function when it finishes running.)

So how can we decide what move the computer has played, depending on whether the piece of code above returns 0, 1, or 2?

There are various ways we could do this, but one way is to use an **array**. An array is a collection of values, and the syntax for it looks like this:

```
const moves = ['rock', 'paper', 'scissors'];
```

Here we're setting the variable **moves** to be an array containing the strings 'rock', 'paper', and 'scissors'. The feature of arrays that's really useful is that we can access a particular value inside an array using its **index**. Indices in arrays start from 0, and items are numbered in order (with the last item in the array having an index that's equal to the length of the array minus one). The syntax for accessing a value inside an array looks like this:

```
moves[0];
```

The value of the above piece of code will be the string ‘rock’, because ‘rock’ is the first item in the moves array and has an index of 0. So if we think about it, we can use the code we wrote earlier (to randomly generate 0, 1, or 2) to pull the item out of our moves array whose index corresponds to our random integer:

```
const opponentMove = moves[Math.floor(Math.random() * 3)];
```

This will set the variable **opponentMove** to randomly be either ‘rock’, ‘paper’, or ‘scissors’. Yay! Now we have a value for both myMove (which we’re passing into our play() function) and opponentMove.

Updating our HTML to show what we played and what the computer played

Next we want to display something on our web page to say what we played and what our opponent, the computer, played. We’ve seen an example already of how JavaScript and HTML can interact, with the **onclick** attribute that we added to our HTML button elements. But how do we directly access and change our HTML from JavaScript? As you’ll remember, at the moment our HTML elements that say what we played and what our opponent played look like this:

```
<p>You played scissors</p>
<p>Your opponent played paper</p>
```

But these are just hardcoded — at the moment we have no way of changing these paragraphs to say ‘You played rock’ or ‘Your opponent played scissors’. What we’ll need to do is find a way to select these particular paragraph elements using JavaScript, and then change their contents so that they actually say what we played and what the computer played.

There’s an HTML attribute that we can add to our code that will be useful here. It’s called the **id** attribute, and it’s basically what it sounds like — an id for that particular HTML element that we can use to distinguish it from others. An id with a particular value should only be used once on a page. We can add id attributes to the above HTML as follows:

```
<p id="me"></p>
<p id="opponent"></p>
```

We can then select these paragraph elements using JavaScript’s inbuilt **document** object, which refers to the HTML code of the web page we’re interacting with. There’s a function on the document object called **getElementById()**, which we can use to select the above paragraph elements:

```
document.getElementById('me');
document.getElementById('opponent');
```

But how do we change the contents of these paragraph elements? There's a property called **innerHTML** that will allow us to set their contents:

```
document.getElementById('me').innerHTML = 'You played ' + myMove;
document.getElementById('opponent').innerHTML = 'Your opponent
played: ' + opponentMove;
```

What we're doing here is saying, 'Go and get the element in the HTML document that has an id of "me", and set its contents to a string that combines the words "You played " with whatever the myMove variable is set to.' Note that we're using something here called **string concatenation**, which is when two separate strings are combined into a single string by adding them together using the **+** symbol. Because myMove evaluates to a string — either 'rock', 'paper', or 'scissors', depending on which button we clicked — we can use the name of the variable here. Note that we want a space between the word 'played' and the move we played, so we need to add this to the end of the 'You played ' string — otherwise this string concatenation would evaluate to something like 'You playedpaper' or 'You playedrock'.

With this code written, our **play()** function in our script.js file should look like this so far:

```
function play (myMove) {
  const moves = ['rock', 'paper', 'scissors'];
  const opponentMove = moves[Math.floor(Math.random() * 3)];
  document.getElementById('me').innerHTML = 'You played ' +
myMove;
  document.getElementById('opponent').innerHTML = 'Your opponent
played: ' + opponentMove;
}
```

Deciding whether we won, lost, or tied

The next thing we need to do is figure out, based on the values of myMove and opponentMove, whether we won, lost, or tied in our rock paper scissors game.

We're going to create a separate function to get the rock paper scissors game's result, and put this above our **play()** function in our script.js file. We'll call this function **getResult()**. First of all, inside our **play()** function, let's create a variable, **result**, which we'll set to the value returned by **getResult()**:

```
const result = getResult(myMove, opponentMove);
```

Then, above the **play()** function, let's add our **getResult()** function:

```
function getResult (myMove, opponentMove) {

}
```

As you can see, we’re going to be using two parameters, **myMove** and **opponentMove**. This is because we’ll need to know what both of these variables are in order to determine whether we won, lost, or tied.

So how are we going to do that? A feature of JavaScript that will come in useful here is known as a **conditional**.

The way conditionals work is that we can check **if** a particular condition evaluates to true, and if it does, run a piece of code. An example we can use in our **getResult()** function is as follows:

```
function getResult(myMove, opponentMove) {
  if (myMove === 'rock') {
    if (opponentMove === 'rock') {
      return 'tie';
    }
    if (opponentMove === 'paper') {
      return 'lose';
    }
    if (opponentMove === 'scissors') {
      return 'win';
    }
  }
}
```

So what’s happening here? First of all, with our first conditional **if** statement, we’re checking whether the value of myMove is equal to ‘rock’. That’s what the strange-looking triple equals sign (**===**) is doing — it means ‘is equal to’. Don’t get this confused with the single equals sign (**=**) we use when assigning a variable. Then after the conditional statement in parentheses (**()**), we open a set of curly braces (**{}**), which is where the code that we want to run only if the conditional statement preceding it evaluates to true goes.

What you’ll notice is that the code inside the first conditional block is actually another series of **if** statements, this time starting with **if (opponentMove === ‘rock’)**. This is called a nested conditional. We’ll only run any of these additional **if** statements if the condition **myMove === ‘rock’** is true, because of how our code is structured.

Because we’ve only written the code that we want to run when **myMove** is equal to ‘rock’, you’ll need to add further conditional statements below this whole block to tell the browser what to do if myMove is equal to ‘paper’ or ‘scissors’. You’ll want to use nested conditionals like we have here (since you’ll need to check the values of both **myMove** and

opponentMove in order to decide what to return), but remember that you'll need to change what's returned based on the rules of rock paper scissors (rock beats scissors, paper beats rock, and scissors beats paper).

Inside each of our nested conditionals, we have some more syntax you might be unfamiliar with. What does **return 'tie'** mean? This is a **return statement**. It means that when we reach this line, we exit out of the whole function (without running any more of the code inside it) with a **return value** of the string 'tie'. (Note that because we'll only run each of these return statements if certain conditions are met, we'll actually only be running at most one return statement each time we run the function, depending on what the values of **myMove** and **opponentMove** are. Any code that's inside a block whose conditions aren't met will just be skipped over.)

Updating our HTML to show the result

Okay. So now that we've written our **getResult()** function, we'll need to return to our **play()** function to finish writing the code we want to run each time the user clicks one of our three buttons.

You'll remember that we just added a line where we set the variable **result** to equal the result of calling **getResult()** with our **myMove** and **opponentMove** variables passed in. From the code we wrote in our **getResult()** function, you'll know that the value returned will be either 'tie', 'win', or 'lose' — so this is what our **result** variable will be set to after **getResult(myMove, opponentMove)** is evaluated.

Now we need to update our HTML to show this result on the page. Again, we're going to want to use the **document.getElementById()** method to get access to the HTML element we want to update, and then set the **innerHTML** property to the string we want to display. But first, we'll need to add an **id** to this HTML element in our index.html file so we can access it:

```
<p id="result"></p>
```

Then, inside our **play()** function in our script.js file, we can assign a new variable, **resultElement**, and set it to our HTML element with the id of 'result':

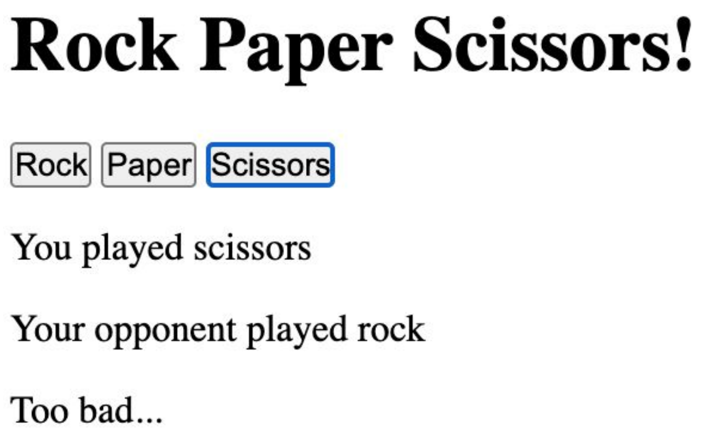
```
const resultElement = document.getElementById('result');
```

We can then set the **innerHTML** property of this result element to display something different depending on the value of **result**, again using conditionals:

```
if (result === 'win') {
  resultElement.innerHTML = 'You win!';
} else if (result === 'lose') {
  resultElement.innerHTML = 'Too bad...';
} else if (result === 'tie') {
  resultElement.innerHTML = "It's a tie!";
}
```

Note that **else if** simply means 'If the preceding conditionals didn't evaluate to true but this one does, run the following code.' It's another piece of syntax that can be used in conditionals, along with the **else** statement (which simply means, 'If the preceding conditionals didn't evaluate to true, run the following code').

Great! We've now written all of our JavaScript code. Go ahead and save your script.js file, and try clicking one of the three buttons to play either rock, paper, or scissors. If everything is working as expected, the page in your CodeSandbox browser should look something like this:


Rock Paper Scissors!

You played scissors

Your opponent played rock

Too bad...

Debug your code using console.log

If your page doesn't look like this, you may want to go through the instructions in this document again to check that you've done everything correctly. But another way to figure out what might be going wrong is to use the **console**.

Inside either of your functions, you can add **console.log()** statements with whatever you want to log inside the parentheses, e.g. console.log(myMove), console.log(result), or console.log('Opponent move inside play function: ' + opponentMove). This last example uses string concatenation to make the log statement clearer. You'll want to make sure you add any console.log statements after any relevant variables have been defined in your code.

Then open up the console (at the bottom of the CodeSandbox browser), and try clicking the buttons to see what's logged. If nothing is logged to the console, this might mean you need to check that the **onclick** attributes on your buttons are correct.

Console.log is an incredibly useful way for JavaScript developers to check what's happening in the code at any given point in time and that the values of different variables are as expected. It's used a lot by professional developers, and once you've played around with it a bit you'll see why it's so handy!

CSS: Make it look good

Phew! JavaScript is definitely the most challenging part of this exercise. Now we get to move onto something a bit more straightforward: CSS.

If HTML is the structure of our house, and JavaScript is the interactivity, then CSS (Cascading StyleSheets) is the paint, the decorations, the carpets, etc. It's what allows us to make our site look good. HTML comes with styles built into it, but they're incredibly basic. Using CSS, we can make our rock paper scissors game a lot more fun to look at! We'll run through an example here, but feel free to style your game any way you like.

Adding a <style> tag

First off, we're going to need to add a **<style>** HTML tag inside the **<head>** element in our index.html file. Alternatively we could add a link to a separate CSS file, but since we won't be adding many styles, we can add our CSS directly into our HTML. So go ahead and add the opening and closing tags for our CSS styles below the other elements inside your page's **<head>** tag:

```
<head>
  <meta charset="UTF-8" />
  <title>Rock Paper Scissors</title>
  <style>

  </style>
</head>
```

CSS selectors

In CSS code, we use **selectors** to identify HTML elements, and then add styling rules to them. Common types of selectors are **tag names** (like p for <p> elements, body for the <body> element, h1 for <h1> elements), **ids** (which apply the CSS to any HTML element with that id), and **classes** (which apply the CSS to any HTML element with that class attribute).

The syntax for writing tag name selectors is as follows:

```
body {
  font-size: 16px;
  font-family: san
}
```

The syntax for writing id selectors is the name of the id with a hash (#) symbol in front of it:

```
#result {
  color: red;
}
```

And finally, the syntax for class selectors is the name of the class with a dot (.) in front of it:

```
.play-button {
  background-color: purple;
}
```

Tag name selectors will apply their styles to every HTML element with that tag name, whereas id and class selectors allow us to target particular HTML elements with much more specificity. And because of the **cascade**, the order of the selectors in our CSS file matters, with selectors at the top of the file having their style rules added first, and so on down to the bottom of the file. This means that if we add a selector at the end of our CSS file that targets the same HTML element as an earlier selector, any properties from the earlier selector that are repeated will be overridden by those properties in the later selector. For example, we might have the following HTML code in our index.html file:

```
<p class="result">You win!</p>
```

And we might have the following CSS in our style.css file:

```
p {
  color: green;
  font-family: Arial;
  font-size: 12px;
}

.result {
  color: black;
  font-weight: bold;
}
```

In this case, the **color** property (which sets the font colour) on any <p> elements with the class ‘result’ will be black instead of green, because the CSS we’ve written for our .result class will override any repeated properties from the p selector.

As you can see from the above examples, the syntax for adding styles to a selector is simple: inside the curly braces for that selector, add a **property name** (e.g. ‘font-family’) followed by a colon, and then the **value** you want to set that property to followed by a semicolon. There are plenty of references online that give lists of CSS properties and the possible values they can be set to, e.g. this one from MDN.

Styling our rock paper scissors game

Now for the fun part! We’re going to add some example styles to our rock paper scissors game, but you can play around with different CSS properties and style your game any way you like.

First, let’s have a look at our text. At the moment our <h1> title and paragraph (<p>) elements have the default font of Times New Roman. Let’s set the font to Arial for our whole page:

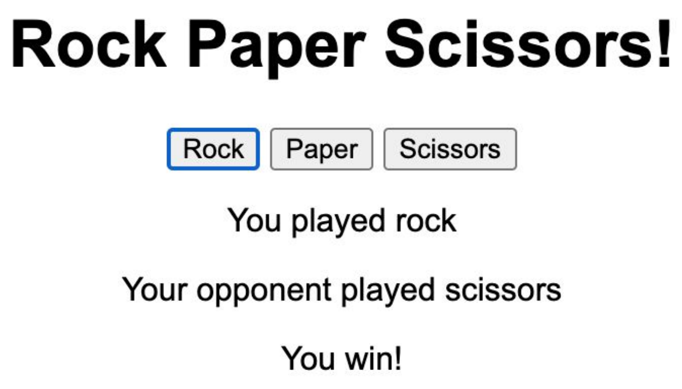
```
body {
  font-family: Arial, Helvetica, sans-serif;
}
```

Note that we’ve also added Helvetica and sans-serif to our **font-family** property — this just means that if our browser can’t use Arial, it will next try Helvetica, and if that doesn’t work, it will default to a generic sans serif font.

And let’s horizontally centre the contents of our page:

```
body {
  font-family: Arial, Helvetica, sans-serif;
  text-align: center;
}
```

Our page should now look something like this:


Rock Paper Scissors!

You played rock

Your opponent played scissors

You win!

Hmm, our buttons aren't looking that great. Let's add some padding to them so their outlines are further away from the text inside, and while we're at it let's add a few other styles to them too:

```
button {
  padding: 10px;
  border: black solid 1px;
  border-radius: 5px;
  font-weight: bold;
}
```

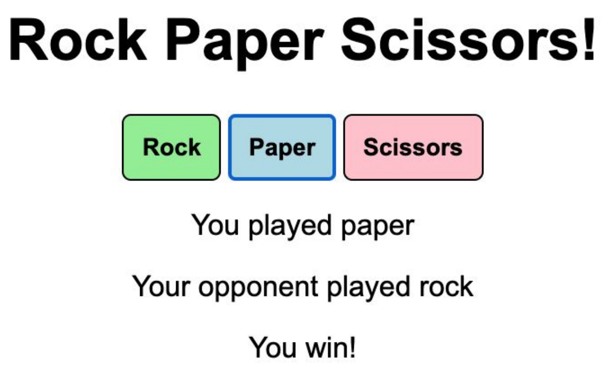
Let's also spice up this very colourless page by making our buttons all different colours. To do this, we'll first need to add some class attributes to them in our HTML so we can target them individually with our CSS:

```
<button class="btn-rock" onclick="play('rock')">Rock</button>
<button class="btn-paper" onclick="play('paper')">Paper</button>
<button class="btn-scissors"
onclick="play('scissors')">Scissors</button>
```

And now let's add some CSS:

```
.btn-rock {
  background-color: lightgreen;
}
.btn-paper {
  background-color: lightblue;
}
.btn-scissors {
  background-color: pink;
}
```

Much better! Our page should now look something like this after we click one of the buttons:


Rock Paper Scissors!

You played paper

Your opponent played rock

You win!

What else can we do to make this page more interesting? We might want to draw some more attention to our win/lose/tie message so it stands out from the other paragraphs above it:

```
#result {
  font-weight: bold;
  text-transform: uppercase;
}
```

Finally, how about we do something really cool, and add an image of what we played and what our opponent played to the page? How would we do that?

First, we'll need to go and find some images and then upload them to CodeSandbox by dragging and dropping them into our files area. Let's name them uniformly, with 'rock', 'paper', or 'scissors' followed by the .png file type. Then we'll need to add some extra HTML at the bottom of our page, inside the <body> element.

```
<img id="img-me"></img>
<img id="img-opponent"></img>
```

Now let's add some CSS to keep the images a reasonable size:

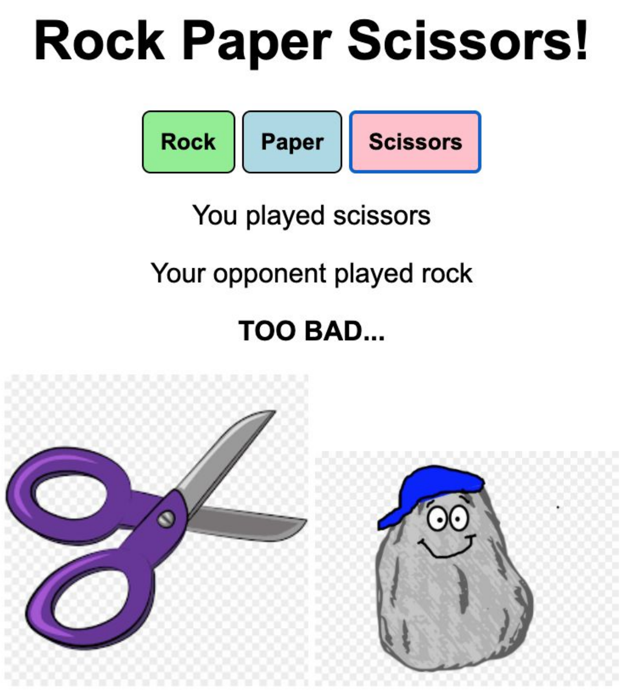
```
img {
  max-width: 40%;
}
```

And finally, we'll need to add some JavaScript inside our **play()** function to set each of our images to the correct file, depending on what we and our opponent played. For this we'll use the **document.getElementById()** method, along with the **setAttribute()** method. You can add this code anywhere inside the play() function as long as it's below our variable declaration for **opponentMove**:

```
document.getElementById('img-me').setAttribute('src', myMove +
'.png');
document.getElementById('img-opponent').setAttribute('src',
opponentMove + '.png');
```

Here what we're doing is selecting the element with the id of 'img-me' or 'img-opponent', and then setting the **src** attribute (which tells the browser which source file to use for the <img> tag) to either 'rock.png', 'paper.png', or 'scissors.png', depending on the value of the myMove or opponentMove variable. Note that we're once again using string concatenation to combine the value of the variable with the '.png' ending.

Your page should now look something like this after you make a play:



Rock Paper Scissors!

You played scissors

Your opponent played rock

TOO BAD...

And that's it! Feel free to continue to play around with the styling, or try out different things with how you display the images. With CSS, the possibilities are truly endless for all the different ways you could choose to style your page.

Wrapping it all up

We hope you found this tutorial helpful and interesting! In summary, your finished index.html file should look something like this:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>Rock Paper Scissors</title>
    <style>
      body {
        text-align: center;
        font-family: Arial, Helvetica, sans-serif;
      }
      button {
        padding: 10px;
        border: black solid 1px;
        border-radius: 5px;
        font-weight: bold;
      }
      .btn-rock {
        background-color: lightgreen;
      }
      .btn-paper {
        background-color: lightblue;
      }
      .btn-scissors {
        background-color: pink;
      }
      #result {
        font-weight: bold;
        text-transform: uppercase;
      }
      img {
        width: 40%;
      }
    </style>
    <script src="./script.js"></script>
  </head>

  <body>
    <h1>Rock Paper Scissors!</h1>
    <div>
      <button class="btn-rock"
onclick="play('rock')">Rock</button>
      <button class="btn-paper"
onclick="play('paper')">Paper</button>
      <button class="btn-scissors"
onclick="play('scissors')">Scissors</button>
    </div>
    <p id="me"></p>
    <p id="opponent"></p>
    <p id="result"></p>
    <img id="img-me"></image>
    <img id="img-opponent"></image>
  </body>
</html>
```

And your finished script.js file should look something like this:

```
function getResult(myMove, opponentMove) {
  if (myMove === 'rock') {
    if (opponentMove === 'rock') {
      return 'tie';
    }
    if (opponentMove === 'paper') {
      return 'lose';
    }
    if (opponentMove === 'scissors') {
      return 'win';
    }
  }
  if (myMove === 'paper') {
    if (opponentMove === 'rock') {
      return 'win';
    }
    if (opponentMove === 'paper') {
      return 'tie';
    }
    if (opponentMove === 'scissors') {
      return 'lose';
    }
  }
  if (myMove === 'scissors') {
    if (opponentMove === 'rock') {
      return 'lose';
    }
    if (opponentMove === 'paper') {
      return 'win';
    }
    if (opponentMove === 'scissors') {
      return 'tie';
    }
  }
}

function play(myMove) {
  const moves = ['rock', 'paper', 'scissors'];
  const opponentMove = moves[Math.floor(Math.random() * 3)];

  document.getElementById('me').innerHTML = 'You played ' +
myMove;
  document.getElementById('opponent').innerHTML =
    'Your opponent played ' + opponentMove;

  document.getElementById('img-me').setAttribute('src', myMove +
'.png');
  document
    .getElementById('img-opponent')
    .setAttribute('src', opponentMove + '.png');

  const result = getResult(myMove, opponentMove);

  const resultElement = document.getElementById('result');
  if (result === 'win') {
    resultElement.innerHTML = 'You win!';
  } else if (result === 'lose') {
    resultElement.innerHTML = 'Too bad...';
  } else if (result === 'tie') {
    resultElement.innerHTML = "It's a tie!";
  }
}
```

If your code doesn’t look exactly the same as ours, don’t worry — as long as it works, you’ve done it right!

We really hope you enjoyed this exercise. We’re always open to feedback, questions, and suggestions, so if you have any, please don’t hesitate to email us at grow@springload.co.nz. Thanks for learning with us!