# JavaScript

A JavaScript Tutorial based on this guide produced by Bro Code

<https://www.youtube.com/watch?v=lfmg-EJ8gm4>

Getting ESlint to work

<https://stackoverflow.com/a/47646706>

Configure ESLint to not mark unused variables prefixed with \_

(only uses the "no-unused-vars" section)

<https://stackoverflow.com/a/71384607>

Disable eslint for an entire file:

/\* eslint-disable no-unused-vars \*/

// eslint-disable-next-line

Some projects included in the tutorial:

* Digital Clock
* Stopwatch
* Calculator
* R.P.S.
* Image Slider
* Weather app that fetches data from an API

The basic setup for linking a JS file to an html file is very similar to linking a CSS file. The differences are where you would place the script tag for performance/parsing errors, using async/defer. You want the page to load as fast as possible, before modifying the page with JS. Loading the JS file too soon may cause issues as once the parser hits the script tag, it blocks loading all other elements until the .js file is loaded (this file may be trying to modify something else in the DOM that has not been loaded yet and that would cause an error.

One solution to this would be to place the script at the end of the html file just before the </body> tag. The problem with this would be that the user then has to wait for the JS file to load making the overall site load time longer.

There are other solutions like the async/defer attributes on scripts. These attributes tell the browser it's safe to continue parsing while the scripts are being downloaded.

## Async

<script src="path/to/script1.js" async></script>

<script src="path/to/script2.js" async></script>

https://stackoverflow.com/a/24070373

Scripts with the async attribute are executed asynchronously. This means the script is executed as soon as it's downloaded, without blocking the browser in the meantime. This implies that it's possible for script 2 to be downloaded and executed before script 1.

## Defer

<script src="path/to/script1.js" defer></script>

<script src="path/to/script2.js" defer></script>

Scripts with the defer attribute are executed in order (i.e. script1 then script2). This also does not block the browser. Unlike async scripts, defer scripts are only executed after the entire document has been loaded.

By using async or defer, script tags can be placed inside the head without impacting performance or causing issues with page loads.

When using jquery, don't use async/defer, the browser may be trying to parse before the library is loaded. You can also give the script a module attribute in ES6.

For simplicity with the small files, placing the script just before the closing body tag is fine.

An error in the JS code may also prevent html elements below it from loading.

## Writing a message to browser's Console

The simplest way the js file can interact with the browser is through a console.log, which will print output to the dev tool's console. Single Quotes, double quotes and back ticks are all usable within console.log, and back ticks are used for template literals which support substituting in values.

# My Website

A basic website with no real content, only empty html elements with Ids. We will use JavaScript to insert content into these tags instead of having it already present in the DOM.

To select a DOM element from a JavaScript file, you must call document.getElementById(""). This function is the javascript selector for an ID.

Setting the element to display a specific string. This clears any existing content previously written in that element.

document.**getElementById**("myH1").textContent = "Hello";

Append appends text to the end of the existing content instead of overwriting it.

document.**getElementById**("myH1").**append**("Hello world");

## Variables

Variables are containers that store a value. Behaves as if it were the value it contains.

Declaration is done with either let or var (var is local scoped while let is block scoped). This post goes over the differences of let/var when using an event handler and why let is generally better to use. <https://stackoverflow.com/a/30479554>. In short, take these two examples:

Html (same html between both examples):

<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>

<p>Clicking on each number will log to console:</p>

<div id="div1">1</div>

<div id="div2">2</div>

<div id="div3">3</div>

<div id="div4">4</div>

<div id="div5">5</div>

Var:

for(var i=1; i<6; i++) {

**$**("#div" + i).**click**(function () { console.**log**(i); });

  }

In this example, 5 divs area created. The Javascript function prints the value of i each time one div is clicked. Because var is locally scoped, var is always the value the value of i becomes 6 which is the stop condition for the loop.

let:

for(let i=1; i<6; i++) {

**$**("#div" + i).**click**(function () { console.**log**(i); });

  }

By changing the keyword to let, you're able to access the variable's value at that particular instance instead of having to write a closure to expose that value for that iteration.

### Closure

A closure is a nested function that allows us to access variables of the outer function after the outer function is closed. Essentially a helper function within another function exposing data.

def greet(name):

# inner function

def display\_name():

print("Hi", name)

# call inner function

display\_name()

# call outer function

greet("John")

# Output: Hi John

## Assignment

After instantiating a variable, you can finally assign a value to it.

// declaration

let x;

// assignment

x = 3;

// Can also be done in one step

let y = 3;

Trying to declare the same variable again with let within the same block will result in an error.

let age = 25;

let price = "10.99";

let gpa = "2.1";

console.**log**(age + " " + price);

console.**log**(price, age);

// template literals

console.**log**(`You are ${age} years old`);

console.**log**(`The price is $${price}`);

console.**log**(`Your gpa is ${gpa}`);

// print without trailing newline, only usable for command line, not browser console

// process.stdout.write("hello, ");

// process.stdout.write("world");

console.**log**(typeof age);

console.**log**(typeof price);

The typeof keyword can be used to identify the data type of the variable given. The variable may be placed within brackets, this does not mean a function is being invoked, only that the following data is being grouped.

## Variables and the DOM

Template literals can be used in when replacing text content within the DOM.

// Variables + the DOM

let fullName = "Bobby Tables";

let age ="25";

let isStudent = false;

document.**getElementById**("p1").textContent = `Hello ${fullName}.`;

document.**getElementById**("p2").textContent = `You are ${age}.`;

document.**getElementById**("p3").textContent = `Enrolled: ${isStudent}`;

## Taking User Input

There are a couple ways to take user input with javascript. The quick way is to call window.prompt(""), which will create an alert containing an input field. This can be used together with assignment to store the user's input.

let username = window.**prompt**("What's your username?");

console.**log**(username);

First, you need the html form element and a button with an ID and a type="button"

    <label for="myText">username</label>

    <input type="text" id="myText"><br><br>

    <button id="mySubmit" type="button">Submit</button>

Next, onto the javascript;

First, use document.getElementById("mySubmit").onclick to create an event handler.

document.**getElementById**("mySubmit").**onclick**

Next, we assign a function to this event

let username;

document.**getElementById**("mySubmit").**onclick** = function() {

    username = document.**getElementById**("myText").value;

    console.**log**(username)

}

Converting the following types:

let x = "pizza";

let y = "pizza";

let z = "pizza";

x = **Number**(x); // characters -> produces NaN, still a number

y = **String**(y); // already a string

z = **Boolean**(z); // true

console.**log**(x, typeof x);

console.**log**(y, typeof y);

console.**log**(z, typeof z);

let x = "0";

let y = "0";

let z = "0";

x = **Number**(x); // 0 number

y = **String**(y); // 0 string

z = **Boolean**(z); // 0 true

console.**log**(x, typeof x);

console.**log**(y, typeof y);

console.**log**(z, typeof z);

When using string versions of a number, 0 can be converted into a number, and evaluates to true as a Boolean.

Negatives work too, and still produce a true result for Boolean.

An empty string results in a 0 when converted to a number, empty string for a string and a false for a Boolean.

Checking for an empty string is useful for checking if someone actually filled in the input.

Declared but uninitialized variables:

let x;

let y;

let z;

x = **Number**(x);

y = **String**(y);

z = **Boolean**(z);

console.**log**(x, typeof x);

console.**log**(y, typeof y);

console.**log**(z, typeof z);

This results in Nan number, undefined string and false Boolean values.

## Constants

// constants = a variable that cant be changed\*

// objects can still be mutated

const PI = 3.14159;

let radius;

let circumference;

// throws an error, trying to assign a value to a constant

// pi = 6.28318;

radius = **Number**(window.**prompt**("Enter the radius of your circle"));

circumference = 2 \* pi \* radius;

console.**log**(circumference);

Capitalizing constants is usually only done with primitive data types like Numbers and Booleans. Reference data types such as strings, don’t normally follow this convention.

HTML prompting the user to enter a value into the textbox:

    <h1 id="myH1">Enter the radius of a circle: </h1>

    <label for="myRadius">Radius: </label>

    <input id="myRadius" type="text"><br><br>

    <button type="button" id="mySubmit">Submit</button>

index.js:

// constants = a variable that can't be changed\*

// objects can still be mutated

// typically only capitalize primitives

const PI = 3.14159;

let radius = 0;

let circumference;

// throws an error, trying to assign a value to a constant

// pi = 6.28318;

// alert prompt

// radius = Number(window.prompt("Enter the radius of your circle"));

document.**getElementById**("mySubmit").**onclick** = function() {

    // do not forget .value

    radius = **Number**(document.**getElementById**("myRadius").value);

    // debug

    // console.log(radius, typeof radius);

    circumference = 2 \* PI \* **Number**(radius);

    document.**getElementById**("myH1").textContent = `Your Circumference is ${circumference}`;

  console.**log**(circumference);

}

\*When collecting user input from a field, do not forget to use the .value property to collect the value stored in that field, otherwise, you will end up storing the html from the DOM instead.

Also remember to convert to the necessary datatypes, in this case we really wanted to work with a number, not a string representation of a number. Call the Number() function to convert the input to a number.

## Counter program

See the contents of counter/ for the example.

## Math Object

See math for examples, contains methods and other math related properties. To access, use Math.<method/property>.

## Random

Random is not cryptographically secure. It is located within the Math object.

Generates a number between 0 and 1 exclusive. If you would like to generate a random number within a range:

range = Max - Min (This is exclusive, to make it inclusive, add 1 to the range)

offset = min

Math.floor(Math.random() \* range + offset)

## Conditions

If a condition is true execute an action. The if chain exits at the first condition to match.

## Checked Property

The checked property is a value that can be checked to determine if a checkbox or radio button has been selected.

To check if a checkbox/radio button is selected, first select the element with getElementById("myId") and add .checked to the end of to access the checked property, which returns a Boolean.

## Ternary

Very similar to an if statement and can be chained like an else if. Its structure is as follows:

**condition ? if true : else do this;**

let greeting = time <= 11 ? "Good Morning"

  : time <= 14 ? "Good Afternoon"

  : "Good Night";

Chaining multiple - If time <= 11?, then greeting = "Good Moring", else: if? time <= 14 ?, then greeting = "Good Afternoon" else: greeting: "Good Night";

function **hello**(time\_of\_day) {

  return time\_of\_day <= 11 // condition

    ? "Good Morning" // true result

    : time\_of\_day <= 14 // else if (condition)

    ? "Good Afternoon" // true result

    : "Good Night"; // else

}

It's best to think of the question mark as a symbol for if, evaluating the condition preceding it, and to treat colon as an else

// if customer spends above a certain amount, give discount

let purchaseAmount = 125;

let discount = purchaseAmount >= 100 ? 10 : 0;

// expressions can be written inside a template literal, not restricted to basic substitution

console.**log**(`Your total is $${purchaseAmount - purchaseAmount \* (discount / 100)}`);

## Switches

Switches can be an efficient replacement to using many else if statements.

In Javascript, switches have a fall through behaviour instead of short circuiting like if statements, or switches in another language. It is best to end a switch with a break clause to prevent conditions from being evaluated further after a true result.

They are essentially an if statement where you must explicitly tell the program to break or all of the following code will run. (In Javascript, implementation depends on the language).

## String Methods

Some methods that can be used on strings. In JS, strings are immutable, if you wish to write a function for a string, you need to return a new string.

### String Slicing

Tuple assignment is weird in JS. Either place your variables inside square brackets, or have an extra variable to capture the array produced. (Working with .split())

let eatArray;

// tuple assignment is weird in js

firstName, lastName, eatArray = fullName.**split**(" ");

console.**log**(firstName); // SpongeBob

console.**log**(lastName);  // SquarePants

console.**log**(eatArray); // [ 'SpongeBob', 'SquarePants' ]

console.**log**("\*\*\*\*\*\*\*\*\*\*\*");

console.**log**(`Your first name is ${firstName}, and your last name is ${lastName}`);

firstName, lastName = fullName.**split**(" ");

console.**log**(firstName); // SpongeBob

console.**log**(lastName); // [ 'SpongeBob', 'SquarePants' ]

console.**log**("\*\*\*\*\*\*\*\*\*\*\*");

console.**log**(`Your first name is ${firstName}, and your last name is ${lastName}`);

[firstName, lastName] = fullName.**split**(" ");

console.**log**(firstName); // SpongeBob

console.**log**(lastName); // SquarePants

console.**log**("\*\*\*\*\*\*\*\*\*\*\*");

console.**log**(`Your first name is ${firstName}, and your last name is ${lastName}`);

In version 2 of this code, you can see that the array is actually assigned to the second parameter overwriting its previously stored value. Version 3 shows that two parameters can be used if they are placed inside square brackets, without the list overriding one of the values stored variables. Trying to use a third parameter in this example will result in that variable having undefined as its value (overwriting its previous value if it had one

# Docstring Example

<https://stackoverflow.com/questions/34205666/utilizing-docstrings>

// making use of Hoisting

// generates a random number between min and max, and optional inclusive arg

/\*\*

 \* [randomRange description]

 \* @param {Number} min Lower bound for a randomly generated integer

 \* @param {Number} max Upper bound for a randomly generated integer; exclusive by default

 \* @param {Boolean} incl Optional Boolean flag making upper-bound inclusive

 \*/

function **randomRange**(min, max, incl = false) {

    let excl = 0;

    if (incl) {

        excl = 1;

    }

    return  Math.**floor**(Math.**random**() \* (max - min + excl) + min)

}

## Note on number guessing game

Javascript is event driven. Make use of event handlers like .onclick to make something responsive, don't place code in a while loop (i.e. waiting for user input to be submitted). The loop will lock up the browser until it becomes unresponsive.

# Functions

A reusable section of code. Functions declared with the function keyword are hoisted, meaning they can be used before they are declared.

## Scopes

Where an variable is recognized and accessible (Local vs Global)

Variable names must be unique in its scope.

let x = 3; // global

function **function1**() {

    let x = 1;

    console.**log**(x);

}

function **function2**() {

    let x = 2;

    console.**log**(x);

}

**function1**();     // 1

**function2**();     // 2

console.**log**(x);  // 3

The variable x is local to those scopes.

The global variable x can be changed by a function if this is done:

function **function3**() {

    x = 7; // uses global x if called

    console.**log**(x);

}

function3();

Invoking the function is what causes the variable x to change its value in the global scope.

Functions can't observe values in other functions. i.e. if function2 had a variable y, function1 would not know function2 had a variable named y.

Variables in the global scope can be seen by everything, generally it isn't good to have global variables as you can run into naming conflicts.

## Temperature Conversion app

Creating a temperature conversion app using HTML CSS and JS.

    <form action="">

        <h1>Temperature Conversion</h1>

        <input type="number" name="" id="textBox" value="0"><br>

        <input type="radio" name="unit" id="toFahrenheit">

        <label for="toFahrenheit">Celsius ➡️ Fahrenheit</label><br>

        <input type="radio" name="unit" id="toCelsius">

        <label for="toCelsius">Fahrenheit ➡️ Celsius</label>

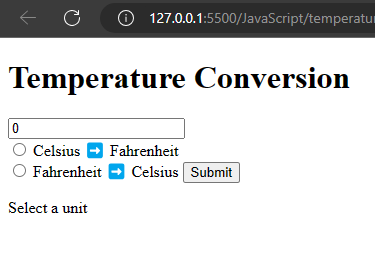
<br>

        <button type="button" onclick="convert"></button>

        <p id="result">Select a unit</p>

    </form>

Buttons have an onclick attribute, which calls a function after a click event.



### Onto the CSS

body {

    font-family: Arial, Helvetica, sans-serif;

    background-color: hsl(0, 0%, 80%);

}

h1 {

    color: hsl(219, 76%, 37%);

}

form {

    background-color: white;

    text-align: center;

    max-width: 350px;

    margin: auto;

    padding: 25px;

    border-radius: 10px;

    /\* x y blur-radius color \*/

    box-shadow: 5px 5px 15px hsla(0, 0%, 0%, 0.3)

}

#textBox {

    width: 50%;

    /\* center contents of textbox \*/

    text-align: center;

    font-size: 2em;

    border: 2px solid #000000cc;

    /\* push down radio buttons \*/

    border-radius: 4px;

    margin-bottom: 15px;

}

label {

    font-size: 1.5em;

    font-weight: bold;

}

button {

    margin-top: 15px;

    color: white;

    background-color: hsl(0, 100%, 63%);

    font-size: 1.5em;

    border: none;

    padding: 10px 15px;

    border-radius: 5px;

    cursor: pointer;

}

button:hover {

    background-color: hsl(0, 100%, 53%);

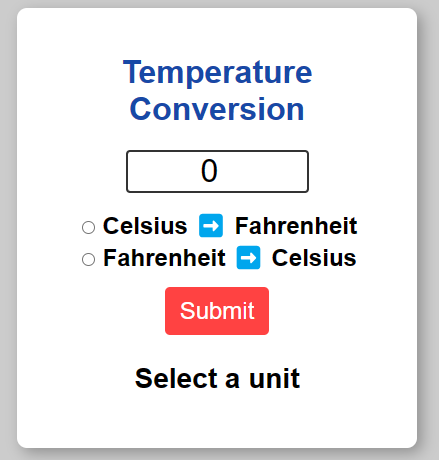
}

#result {

    font-size: 1.75em;

    font-weight: bold

}



Next, onto the Javascript

// Temperature Conversion Program

const textBox = document.**getElementById**("textBox"); // user input

const toFahrenheit = document.**getElementById**("toFahrenheit"); // radio button

const toCelsius = document.**getElementById**("toCelsius"); // radio button

const result = document.**getElementById**("result"); // final result

let temp;

function **convert**(params) {

    // use the .checked property to see if the radio button is checked

    if(toFahrenheit.checked) {

        // set temp = the value property of #textBox

        // typecast user input into a number

        temp = **Number**(textBox.value);

        temp = temp \* 9 / 5 + 32;

        result.textContent = `${temp.**toFixed**(1)}`

    } else if (toCelsius.checked) {

        temp = textBox.value

        temp = (temp - 32) \* 5 / 9;

        result.textContent = `${temp.**toFixed**(1)}℃`

    } else {

        result.textContent = "Please select a unit"

    }

}

The Javascript is pretty easy to understand in this example, with constants at the top to reduce on repetitive actions (binding document.getElementById("")) to a variable.

## Datatypes

Arrays:

// Arrays = in javascript are a variable like structure that can hold more than one value.

//          unlike arrays from statically typed languages, the data types do not need to match

//          and are dynamically sized

//          arrays are retain insertion order and individual elements can be accessed with []

//          To slice an array use the Array.prototype.slice() function, JS doesn't support [] slicing

let fruits = ["apple", "orange", "banana"];

console.**log**(fruits[0]); // apple

console.**log**(fruits[1]); // orange

console.**log**(fruits[2]); // banana

// console.log(fruits[3]); // undefined

console.**log**(fruits.**slice**(1)); // ['orange', 'banana']

## Enhanced For Loop

// enhanced for loop, keywords somewhat similar to list comprehension syntax

console.**log**("\nEnhanced for loop: ");

for (let fruit of fruits) {

    console.**log**(fruit);

}

Working with arrays, some short notes on making shallow copies and how a shallow copy behaves

// 1D arrays

let test = ["alpha", "beta", "gamma", "delta", "omega", "epsilon", "phi" ];

console.**log**("Original:", test);

let clone = test.**slice**();

console.**log**("Clone:", clone);

clone[1] = "electron";

console.**log**("Original:", test);

console.**log**("Clone:", clone);

// 2D Arrays

// Nested elements are mutated inside shallow copies

let twoD = [["electron", "proton"], ["upQuark", "downQuark"]]

let cloneTwoD = twoD.**slice**();

console.**log**(twoD);

console.**log**(cloneTwoD);

cloneTwoD[1][0] = "Left Quark";

console.**log**(twoD);

console.**log**(cloneTwoD);

## Spread Operator -> ...

The spread operator ... allows an iterable such as an array or string to be expanded into separate elements. (unpacks the elements)

Here's an example where you want to use the unpacking operator:

let numbers = [1,2,3,4,5];

let maximum = Math.**max**(numbers);

console.**log**(maximum); // NaN

Math.max() does not unpack arrays and is therefore unable to handle an array as input.

By Prefixing the numbers element with the **...** operator, the array is unpacked and the Math.max() method is able to determine the largest number in the array.

let numbers = [1,2,3,4,5];

let maximum = Math.**max**(...numbers);

console.**log**(maximum); // 5

Another use for the spread operator is to quickly create an array of single characters.

let username = "Bobby Tables";

let letters = [...username];

console.**log**(letters);

[

'B', 'o', 'b', 'b',

'y', ' ', 'T', 'a',

'b', 'l', 'e', 's'

]

The letters of "Bobby Tables" are now broken up into their individual characters within a single array. This is actually very useful for "mutating" a string, as normally a string is immutable.

The Spread Operator can also be used to make a shallow copy of an array:

// Shallow Copy

// - returning a shallow copy with the spread operator

// - a shallow copy means that it's a different data structure, but contains identical values

// - a shallow copy is will not pass changes to the original

let fruits = ["apple", "orange", "banana"]; // original

let newFruits = [...fruits];  // shallow copy of elements

newFruits[0] ="coconut";

console.**log**(fruits); // [apple, orange, banana]

console.**log**(newFruits); [coconut, orange, banana]

The spread operator can also be used to combine two arrays together.

let fruits = ["apple", "orange", "banana"]; // original

let vegetables = ["lettuce", "carrot", "radish"];

let foods = [...fruits, ...vegetables, "eggs", "milk"];

console.**log**(foods);

[ 'apple', 'orange', 'banana', 'lettuce', 'carrot', 'radish' ]

## Rest Parameters

(...rest) allow a function to work with a variable number of arguments by bundling them into an array.

spread = expands an array into separate elements (...)

rest = bundles separate elements into an array. (parameter prefixed by 3 dots) analogous to Python's packing operator.

// Rest parameters = (...rest) allow a function to work with a variable number of arguments by

//                   bundling them into an array (parameter prefixed by three dots) analogous to pythons packing operator

//                 spread = expands an array into separate elements

//                 rest = bundles separate elements into an array

function **openFridge**(...foods) {

  console.**log**(...foods);

}

// take any number of arguments and package them into an array

// return array

function **getFood**(...foods) {

  return foods;

}

const food1 = "pizza";

const food2 = "hamburger";

const food3 = "hotdog";

const food4 = "sushi";

const food5 = "ramen";

// openFridge(food1,food2,food3,food4,food5); // [ 'pizza', 'hamburger', 'hotdog', 'sushi' ]

const foods = **getFood**(food1, food2, food3, food4, food5);

console.**log**(foods);

function **sum**(...numbers) {

  let result = 0;

  for (let number of numbers) {

    result += number;

  }

  return result;

}

function **getAverage**(...numbers) {

    let result = 0

    for (let number of numbers) {

        result += number

    }

    return result/numbers.length

}

const total = **sum**(1, 2, 3);

console.**log**(`Total: $${total}`);

const avg = **getAverage**(100, 75, 85, 90, 50);

console.**log**(`Average: ${avg}`);

## Dice Roller App

This will be an app which makes a dice roller program with HTML, CSS and Javascript.

The Html for this project is very simple:

    <div class="container">

        <h1>Dice Roller🎲</h1>

        <label for="quanity"># of Dice:</label>

        <input type="number" id="quanity" value="1" min="1">

        <button onclick="**rollDice**()">Roll Dice</button>

        <div id="diceResult"></div>

        <div id="diceImages"></div>

    </div>



A container with a label to a number input field and a button that calls a "rollDice()" function when it is clicked.

Here's the CSS for the file:

.container {

  font-family: Arial, Helvetica, sans-serif;

  text-align: center;

  font-size: 2rem;

  font-weight: bold;

}

button {

  font-size: 1.5rem;

  padding: 10px 15px;

  border-radius: 10px;

  border: none;

  background-color: hsl(209, 100%, 50%);

  color: white;

  font-weight: bold;

  cursor: pointer;

}

button:hover {

  background-color: hsl(209, 100%, 60%);

}

button:active {

  background-color: hsl(209, 100%, 70%);

}

input {

  font-size: 2rem;

  font-weight: bold;

  text-align: center;

}

Onto the Javascript

// Dice Roller

function **rollDice**() {

    // constants

    // const is a bit of a misnomer?; the variable cannot be reassigned, it doesn't

    // mean that the value contained within the object assigned to a variable is static

    // This is why we can have a const of a non-static value/arrays and modify their contents

    const numOfDice = document.**getElementById**("quantity").value;

    const diceResult = document.**getElementById**("diceResult");

    const diceImages = document.**getElementById**("diceImages");

    const values = [];

    const images =[]; // array of dice images

}

Next, create a for loop that will loop once for every dice roll.

// Dice Roller

function **rollDice**() {

  // constants

  // const is a bit of a misnomer?; the variable cannot be reassigned, it doesn't

  // mean that the value contained within the object assigned to a variable is static

  // This is why we can have a const of a non-static value/arrays and modify their contents

  const numOfDice = document.**getElementById**("quantity").value;

  const diceResult = document.**getElementById**("diceResult");

  const diceImages = document.**getElementById**("diceImages");

  const values = [];

  const images = [];

  for (let roll = 0; roll < numOfDice; roll += 1) {

    const value = Math.**trunc**(Math.**random**() \* 6) + 1;

    // push value into array of values

    values.**push**(value);

    // string representation of an html element

    // substituting the roll value to select the specific dice element

    // to be rendered

    images.**push**(`<img src="../Assets/Dice\_Images/dice${value}.png" alt="Dice ${value}">`)

  }

  diceResult.textContent =`dice: ${values.**join**(", ")}`

  // access the inner HTML to modify the DOM structure

  diceImages.innerHTML = images.**join**('');

}

The function rolls n number of dice based on the value of #quantity. It then converts these into integers of in a range of 1 - 6 and appends this value into the values[] array. At the same time, we create a custom 'HTML' string which makes use of template literals to link to a specific dice image and adds this html string to the images[] array. Finally, the #results element has the contents of values[] joined by a comma and rendered to the screen through the textContent property, while the **diceImages.innerHTML** is used to add the HTML strings containing the path to their respective dice images. This array is joined by an empty string before being added to the DOM.

values[ 3, 6 ];

// preformatted template literal using the values 3, 4 becomes

    images.**push**(`<img src="../Assets/Dice\_Images/dice${value}.png" alt="Dice ${value}">`)

images[

<img src="../Assets/Dice\_Images/dice**3**.png" alt="Dice **3**">

<img src="../Assets/Dice\_Images/dice**6**.png" alt="Dice **6**">

]

  const diceImages = document.**getElementById**("diceImages");

-> This sting array is then joined using an empty string .join("") passed to **diceImages.innerHTML** ( **^** ) which then updates the DOM with the new elements and populates it with the corresponding dice${value}.pngs. Adding alt text helps with accessibility and for debugging purposes.

# Random Password Generator

This password generator will allow the user to set multiple values to generate a password with varying levels of complexity. It will take a length of n and Booleans for character sets (upper/lower/number/symbols).

## Callback

A callback is a function that is passed as an argument to another function. Used to handle asynchronous operations:

1. Reading a file
2. Networking requests
3. Interacting with databases

"When you're done, call this next"

See <https://dev.to/bbarbour/if-javascript-is-single-threaded-how-is-it-asynchronous-56gd> for more information on how javascript is asynchronous.

function **hello** () {

**setTimeout**(() => {

        console.**log**("Hello!");

    }, 3000);

}

function **goodbye**() {

    console.**log**("Goodbye");

}

// if hello takes a while to execute, nothing stops the browser

// from executing goodbye first then completing the execution of hello

// hello is called first but goodbye is able to complete execution before hello

**hello**();

**goodbye**();

This may be a little unintuitive since Javascript is single threaded. Read through the example linked and see the browser demo for an explanation.

If you would like to guarantee that the goodbye function is called after hello is complete, you need to add a callback.

To use a callback, you pass a function to another function.

function **hello** (callback) {

        console.**log**("Hello!");

        // callback is an arbitrary name, still somewhat asynchronous

        // when used with the setTimeout function

**callback**();

}

function **goodbye**() {

    console.**log**("Goodbye!");

}

function **leave**() {

    console.**log**("Leave!");

}

// if hello takes a while to execute, nothing stops the browser

// from executing goodbye first then completing the execution of hello

// hello is called first but goodbye is able to complete execution before hello

**hello**(**leave**);

^ the callback function is passed to the initial function without brackets indicating that it is not being called. Inside the parent function definition, there is a parameter named callback (arbitrary name) and further down inside its code block, we invoke the parameter by adding parentheses to the end of it. As the code is now, an argument must be passed to hello() or it will throw an error.

Here's another example of a callback function.

// callbacks

// a sum function that calls another function after it is done executing

function **sum**(callback, x, y) {

    let result = x + y;

**callback**(result);

}

function **display**(value) {

    console.**log**(value);

}

**sum**(**display**, 10, 20);

It is somewhat similar to a helper function, but a callback is explicitly passed as an argument, allowing for some flexibility over execution and guaranteeing that the callback function will execute after the initial function is complete.

## foreach()

The forEach method is used to iterate over the elements of an array and apply a specified function (callback) to each element. It is similar to the map() function, but with a few differences. forEach() does not return anything.

stack overflow:

**forEach**: This iterates over a list and applies some operation with side effects to each list member (example: saving every list item to the database) and does not return anything.

**map**: This iterates over a list, transforms each member of that list, and returns another list of the same size with the transformed members (example: transforming list of strings to uppercase). It does not mutate the array on which it is called (although the callback function may do so).

<https://stackoverflow.com/a/34426481>



The function is applied to each element of the array, the original array should be unmutated.

let numbers = [1,2,3,4,5];

function **display**(element) {

    console.**log**(element);

}

// passing an un-invoked function as the callback function

numbers.**forEach**(**display**);

Behind the scenes, the forEach method is passing an iteration of the array into the display function, which says to print out the passed element.

// double the elements within an array

// these three args are passed into the double function

// javascript functions can take any number of arguments, which is why there was no issue for

function **double**(element, index, array) {

    array[index] = element \*2;

}

numbers.**forEach**(**double**);

console.**log**(numbers);

## Map

Map is similar to foreach, although it returns a new array instead of undefined. It is possible for the callback function to mutate the original array if items at certain indexes are reassigned like with the double function used in the forEach method.

let numbers = [1,2,3,4,5];

function **double**(element, index, array) {

    return array[index] = element \* 2;

}

let numDouble = numbers.**map**(**double**);

console.**log**(numDouble); // [ 2, 4, 6, 8, 10 ]

console.**log**(numbers); // [ 2, 4, 6, 8, 10 ]

If the function used "return element \* 2" instead, the original array would be unmodified.

# .Filter()

The .filter() method creates a new array by filtering out elements. When an element is passed to a callback function, if it returns true, filter appends the element to a list.

let numbers = []

numbers = Array.**from**({length:10}, (\_,v) => v);

console.**log**(numbers);

function **isEven**(element) {

    // return element if element mod2 === 0

    return element % 2 === 0;

}

let evenNums = numbers.**filter**(**isEven**);

console.**log**("Even Numbers:",evenNums);

// using an arrow function doing the same thing but for odd numbers, element is just arbitrary parameter name

// filter() element where=> element %2 does not!== 0;

console.**log**("Odd Numbers:", numbers.**filter**(element => element %2 !== 0));

console.**log**("Original Array:",numbers);

// more lambda practice

const ages =[16, 17, 18, 18, 19, 20, 60];

let adults = ages.**filter**(age => age >= 18);

console.**log**(adults);

function **isChild**(element) {

    return element < 18;

}

let children = ages.**filter**(**isChild**);

console.**log**(children);

const words = ["apple","orange","banana","kiwi","pomegranate", "coconut"];

// filter out words > 6

function **getShortWords**(element) {

    return element.length <= 6;

}

// get long words

function **getLongWords**(element) {

    return element.length > 6;

}

let shortWords = words.**filter**(**getShortWords**);

let longWords = words.**filter**(**getLongWords**);

console.**log**(shortWords);

console.**log**(longWords);

## .Reduce()

The Reduce() method reduces the elements of an array to a single value. Together with the reduce function, sum will continue to add the 0th element of the array to the next adjacent element and then return the result once it is the only remaining element.

let prices = [5, 30, 10, 25, 15, 20];

// continues to add elements of an array exhaustively

let total = prices.**reduce**(**sum**);

// accumulator, next\_element

function **sum**(previous, next) {

    return previous + next;

}

console.**log**(total);

# Function Expressions

Function declaration = define a reusable block of code that performs a specific task.

function expression = a way to define functions as values or variables.

const **hello** = function() {

    console.**log**("hello");

}

// still requires () to be invoked

**hello**();

A function expression does not support hoisting.

// sometimes it's easier to write the function as a declaration first

// then turn it into an expression afterwards

function **cube**(element) {

    return Math.**pow**(element, 3);

}

// expressions are essentially just stripping out the function name

// and passing this anonymous function as a callback

const cubes = numbers.**map**(function(element){

    return Math.**pow**(element, 3);

})

console.**log**(cubes);

const evenNums = numbers.**filter**(function (element) {

    return element %2 === 0;

})

// arrow function

const oddNums = numbers.**filter**((element) => element %2 !== 0);

console.**log**("Evens: ",evenNums);

console.**log**("Odds: ",oddNums);

const total = numbers.**reduce**(function(accumulator, next) {

    return accumulator + next})

console.**log**(total);

Function expressions are also used for Callbacks in asynchronous operations, Higher-order functions, closures, and event listeners.

## Arrow Functions =>

An even shorter way to write an anonymous function.

// map

const numbers = [1, 2, 3, 4, 5, 6];

const squares = numbers.**map**((base) => Math.**pow**(base, 2));

const cubes = numbers.**map**((base) => Math.**pow**(base, 3));

// filter

const evenNums = numbers.**filter**((element) => element %2 === 0);

const oddNums = numbers.**filter**((element) => element %2 !== 0);

// reduce

const sum = numbers.**reduce**((accumulator, next) => accumulator + next);

console.**log**("Squares:",squares);

console.**log**("Cubes:",cubes);

console.**log**("Evens:",evenNums);

console.**log**("Odds:",oddNums);

console.**log**("Sum:",sum);

## Adding a method to an object:

const person1 = {

    firstName: "Spongebob",

    lastName: "Squarepants",

    age: 30,

    isEmployed: true,

    // adding a method

**sayHello**: function() {

        console.**log**("Hi! I am SpongeBob!");

    },

}

To do this, we provide a key, and assign it a function expression as the value. To invoke the method, use the object's name and dot notation to specify the method. Finally use parentheses to invoke the function:

// calling method

person1.**sayHello**();

An arrow function also works:

**sayHello**: () => {console.**log**("Hey, I'm Patrick...")},

(a named function can be used, but you will not be able to invoke the function by name, calling the key will still invoke the function.

## This

This is a keyword referencing the object where THIS is used. (The object depends on the immediate context) person.name = this.name.

If we are working in person <object>, and we are accessing the name property, we can replace person with this to access the same property at this.name as long as we are within the context of the person <object>.

const person1 = {

    name: "SpongeBob",

    favFood: "hamburgers",

    // **must be a function**, arrow funcs not hoisted

    // this is a reference to self

**sayHello**: function(){console.**log**(`Hello, I'm ${this.name}`);}

}

person1.**sayHello**();

The **this** keyword allows code to be much more flexible, would replace "person1.name", which makes having multiple instances of a person class much easier to work with as they would be referencing themselves instead of the same person1 object. More in constructors.

In a browser, console.log(this); prints info about the browser window property to the console. 

May be somewhat useful for finding functions/properties.

    // in a browser context, using an arrow function with the this. keyword will reference

    // the window property, not the object. This is due to arrow funcs not being hoisted

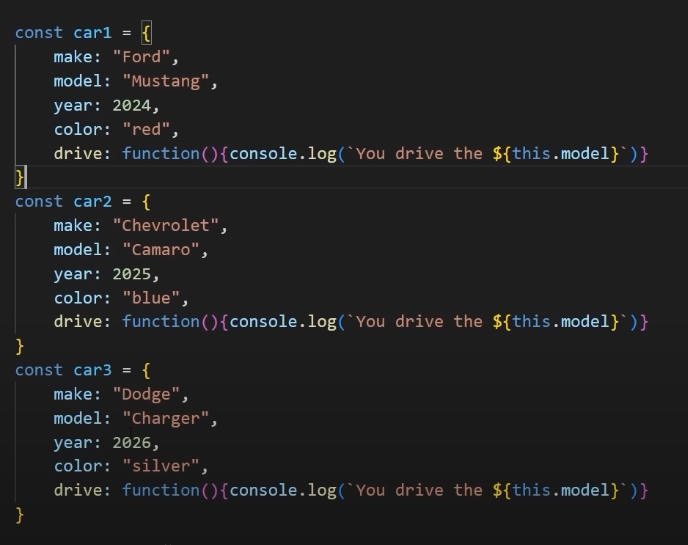
**testing**: () => console.**log**(this), // prints the window properties to console

}

If running js as node index.js, this will print an empty js object for this specific line of code, or possibly undefined if you were trying to print something else like this.name.

## Constructors

Constructors are a special method for defining the properties and methods of objects. They allow for easy creation of multiple objects, serving as a template.



Instead of manually creating 3 car objects, we can make use of a constructor to reduce the duplicated code.

// Typically, constructors begin with capital letters to differentiate them

function **Car**(make, model, year, color) {

    this.make = make;

    this.model = model;

    this.year = year;

    this.color = color;

    // class method

    this.**drive** = function() {

        console.**log**(`The ${this.make + " " + this.model} is currently driving`);

    }

}

Then to create an instance of Car, we only need to provide arguments to the Car constructor, and it will assign the properties.

Instantiating Car objects:

// construct Car objects

let car1 = new **Car**("Ford", "Mustang", 2024, "red");

let car2 = new **Car**("Chevrolet", "Camaro", 2025, "blue");

let car3 = new **Car**("Dodge", "Charger", 2026, "silver");

Accessing Car properties:

// dict representation

// console.log(car1);

console.**log**("Car1: ");

console.**log**(car1.make);

console.**log**(car1.model);

console.**log**(car1.year);

console.**log**(car1.color);

console.**log**("\nCar2:");

console.**log**(car2.make);

console.**log**(car2.model);

console.**log**(car2.year);

console.**log**(car2.color);

console.**log**("\nCar3:");

console.**log**(car3.make);

console.**log**(car3.model);

console.**log**(car3.year);

console.**log**(car3.color);

Calling class method on each instance of the object:

car1.**drive**();

car2.**drive**();

car3.**drive**();

## Classes

Classes are an ES6 Feature which provides a more structured and leaner way to work with objects compared to traditional constructor functions.

Classes add the static keyword, encapsulation and inheritance.

// class replaces this

function **Product**(name, price) {

    this.name = name;

    this.price = price;

    this.**displayProduct** = function () {

        console.**log**(`Product: ${this.name}`);

        console.**log**(`Price: $${this.price.**toFixed**(2)}`);

    };

    this.**calculateTotal** = function (salesTax) {

        return this.price + (this.price \* salesTax);

    }

}

// instantiation

const product1 = new **Product**("shirt", 19.99);

To create a class, start with the class keyword, and name your class, followed by a set of curly braces.

Classes have access to the constructor keyword, which allows for named arguments, replaces the function ObjName(**arg1**, **arg2**) {}

Inside of a class, you do not need to use the function keyword

class Product{

    constructor(name, price){

        this.name = name;

        this.price = price

    }

    // inside of a class, you do not need to use the function keyword

**displayProduct**(){

        console.**log**(`Product: ${this.name}`);

        console.**log**(`Price $${this.price.**toFixed**(2)}`);

    }

**calculateTotal**(salesTax){

        return this.price + (this.price \* salesTax);

    }

}

By using the class keyword to create the object, a lot of the bulky code can be removed (method declaration is a lot cleaner, and so are properties derived from functions).

To instantiate an instance of a class, be sure to use the new keyword in the declaration.

let car1 = new **Car**("Ford", "Mustang", 2024, "red");

## Static

The static keyword defines properties or methods that belong to a class itself rather than the objects created from that class. The class owns anything static, not the objects.

To access a static method, specify the class and use dot notation to specify the property/method.

class User {

    // track how many users have been created

    static userCount = 0;

    constructor(username) {

        this.username = username;

        // User to target the class, not an instance of the object (this.)

        User.userCount++;

    }

}

const user1 = new **User**("SpongeBob");

console.**log**(User.userCount);

console.**log**(user1.userCount); // undefined

User1 does not have a .userCount property. It is a static property belonging to the **User** class itself and not an instance of **User**.

class User {

    // track how many users have been created

    static userCount = 0;

    constructor(username) {

        this.username = username;

        // User to target the class, not an instance of the object (this.)

        User.userCount++;

    }

    // non-static method

**sayHello**(){

        console.**log**(`Hello, my username is ${this.username}`);

    }

    // static method

    static **getUserCount**() {

        console.**log**(`There are ${User.userCount} users online`);

    }

}

const user1 = new **User**("SpongeBob");

const user2 = new **User**("Patrick");

const user3 = new **User**("Sandy");

console.**log**(User.userCount);

console.**log**(user1.userCount); // undefined

user1.**sayHello**();

user2.**sayHello**();

user3.**sayHello**();

User.**getUserCount**();

## Inheritance

DRY = Don't Repeat Yourself

Inheritance allows a new class to inherit properties and methods from an existing class (parent -> child). It helps with code reusability.

To declare a relationship between parent and child, the child *extends* the parent.

class Fish extends Animal {

    name = "fish";

}

// parent

class Animal {

  alive = true;

**eat**() {

    console.**log**(`This ${this.name} is eating`);

  }

**sleep**() {

    console.**log**(`This ${this.name} is sleeping`);

  }

}

// child class

class Rabbit extends Animal {

  name = "rabbit";

}

class Fish extends Animal {

  name = "fish";

}

class Hawk extends Animal {

  name = "hawk";

}

Rabbit, hawk and fish all extend the Animal class, giving them access to an eat and sleep method without needing to implement their own.

## Super

Super is a keyword used in classes to call the constructor or access the properties and methods of a parent (superclass).

this = this object

super = the parent

If you seen an error:

ReferenceError: Must call super constructor in derived child class before accessing 'this' or returning from derived constructor ...

You are seeing this message because we are unable to use the **'this'** keyword without first calling constructor of the parent class first via super().

class Animal {

  constructor() {}

}

class Rabbit extends Animal {

  constructor(name, age, runSpeed) {

    // must call super(); within derived class's constructor

    super();

    this.name = name;

    this.age = age;

    this.runSpeed = runSpeed;

  }

}

class Fish extends Animal {

  constructor(name, age, swimSpeed) {

    super();

    this.name = name;

    this.age = age;

    this.swimSpeed = swimSpeed;

  }

}

class Hawk extends Animal {

  constructor(name, age, flightSpeed) {

    super();

    this.name = name;

    this.age = age;

    this.flightSpeed = flightSpeed;

  }

}

const rabbit = new **Rabbit**("rabbit", 1, 25);

const fish = new **Fish**("fish", 2, 12);

const hawk = new **Hawk**("hawk", 5, 40);

The child classes have a lot of repetition, name an age are present in all child classes, and could be moved into the parent instead. DRY; don't repeat yourself.

The child classes still take the name and age parameters, but the super() function handles their implementation.

class Animal {

  constructor(name, age) {

    this.name = name;

    this.age = age;

  }

}

class Rabbit extends Animal {

    // must call super(); within derived class' constructor

  constructor(name, age, runSpeed) {

    super(**name, age**); // calls Animal's constructor

    this.runSpeed = runSpeed;

  }

}

class Fish extends Animal {

  constructor(name, age, swimSpeed) {

    super(**name, age**);

    this.swimSpeed = swimSpeed;

  }

}

As can be seen in these examples, name and age are still requirements when calling the constructor for the child class, they are subsequently passed to the super() function which then calls the constructor for the **Animal** class which handles the final implementation of handling these arguments. The child classes still have their own properties which is why we created child classes and not instances of the parent class.

### Extending a method from a parent

Another thing that can be done with the super() keyword is extend a method from the parent.

Parent:

class Animal {

  constructor(name, age) {

    this.name = name;

    this.age = age;

  }

  // method to be extended

**move**(speed) {

    console.**log**(`The ${this.name} moves at a speed of ${speed}km/h.`);

  }

}

Child:

class Rabbit extends Animal {

  // must call super(); within derived class' constructor

  constructor(name, age, runSpeed) {

    super(name, age);

    this.runSpeed = runSpeed;

  }

**run**() {

    console.**log**(`This ${this.name} can run`);

    // extending the parent method .move()

    // super indicates this is a method from the parent class

    super.**move**(this.runSpeed);

  }

}

This is more like 'syntactic sugar', we could call rabbit.move(rabbit.runSpeed) for the same output, but might not work well once getters and setters start getting used.

class Rabbit extends Animal {

  // must call super(); within derived class' constructor

  constructor(name, age, runSpeed) {

    super(name, age);

    this.runSpeed = runSpeed;

  }

**run**() {

    console.**log**(`This ${this.name} can run`);

    this.**move**(this.runSpeed);

  }

}

calling the move method this way might have issues with static properties/methods. Probably more efficient to call a parent's method than to use the child instance.

# Getters and Setters

Getter = special method that makes a property readable

Setter = special method that makes a property writable

Used to validate and modify a value when reading/writing a property.

class Rectangle {

  constructor(width, height) {

    this.width = width;

    this.height = height;

  }

}

const rectangle = new **Rectangle**(-1000000, "pizza");

console.**log**(rectangle.width); // -1000000

console.**log**(rectangle.height); // "pizza"

We do not want people being able to set a width of -1000000, or height of "pizza". This is where setters come in force a specific structure (validation).

### Setters

When setting a property, either initially or modifying an existing property, we can go through a **setter** first. Outside of the constructor, we will **set** a property.

Here is the rectangle class we setters for width and height.

class Rectangle {

  constructor(width, height) {

    this.width = width;

    this.height = height;

  }

  // setter

  set **width**(newWidth) {

    if (newWidth > 0) {

        this.\_width = newWidth;

    }

    else {

        console.**error**("Width must be a positive number");

    }

  }

  set **height**(newHeight) {

    if (newHeight > 0) {

        this.\_height = newHeight;

    } else {

   console.**error**("Height must be a positive number");

}

  }

}

In the setter function, we prefix the parameter with an underscore to indicate that this property is private and should not be accessed by anyone else. Attempting to run the program with the two previous values will print both error messages to console and trying to print the two values will show undefined. (the program continued to run even after the errors were printed to console, console.error() is not enough to stop the program from running if there is an error.

### Getters

  // getters; make sure to return the \_private property

  get **width**(){

    return this.\_width;

  }

  get **height**(){

    return this.\_height;

  }

}

Make sure the \_private property is being returned. If you see a maximum call stack exceeded error, you probably tried to return .property instead.

const rectangle = new **Rectangle**(3, 4);

console.**log**(rectangle.width); // 3

console.**log**(rectangle.height); // 4

rectangle.width = 5;

rectangle.height = 6;

console.**log**(rectangle.width); // 5

console.**log**(rectangle.height); // 6

Through getters, we can also get a "derived" property. This property isn't actually part of the object, it is just returning the result of some function being run on the object.

  // rectangle doesn't actually have an area property

  get **area**(){

    return this.\_width \* this.\_height;

  }

console.**log**(rectangle.area); // 30

console.**log**(rectangle); // Rectangle { \_width: 5, \_height: 6 }

rectangle.area doesn't actually exist, but we can access it thanks to the getter.

Both getters and setters can modify the data passed in/out, some uses might be to convert data to strings/numbers.

 get **width**(){

    return this.\_width.**toFixed**(1);

  }

return the width to a fixed precision of 1.

\*precision (toFixed) is really just a string representation of a number, not a numeric value itself.

  set **lastName**(newLastName) {

    if (newLastName.length > 1) {

    // don't accidentally assign this.lastName = this.newLastName

      this.\_lastName = newLastName;

    } else {

      console.**error**("First name must be at least 1 character");

    }

  }

This is the data that's actually stored within person:

Person { \_firstName: 'Spongebob', \_lastName: 'Squarepants', \_age: 26 }. The getter is actually an invoking of a function which fetches the specified information, which may modify it before returning it to whoever asked for the data.

## Destructuring

Destructuring = extract values from arrays and objects, then assign them to variables in a convenient way.

[] = to perform array destructuring

{} to perform object destructuring

Example 1:

Swapping the value of two variables

// -------- Example #1--------

// swap the value of two variables

let a = 1;

let b = 2;

[a, b] = [b, a];

console.**log**(`A: ${a}, B: ${b}`); // A: 2, B: 1

-makes use of [] wrapping both sets of values.

Example 2:

Swapping two elements in an array

// ---------- Example #2----------

// swap 2 elements in an array

const colors = ["cyan", "magenta", "yellow", "black"];

[colors[1], colors[3]] = [colors[3], colors[1]];

// [ 'cyan', 'black', 'yellow', 'magenta' ]

console.**log**(colors);

-makes use of [] wrapping both sets of values

Example 3:

The variables we would like to assign values to are placed within an array, and they are assigned the values from the colors array in order until there are no variables remaining. You do not need a matching number of arguments, any extra values are not assigned and if there is a surplus of variables, they will have a value of unassigned instead.

const colors = ["cyan", "magenta", "yellow", "black"];

// ---------- Example #3----------

// Assign array elements to variables

// reusing previous colors array

const [firstColor, secondColor, thirdColor] = colors;

// cyan black yellow

console.**log**(firstColor, secondColor, thirdColor);

Example 3.1:

...Rest parameters

// ...Rest parameters

const [col1, col2, col3, ...extraColors] = colors;

console.**log**(col1, col2, col3);

console.**log**(extraColors);

cyan black yellow // col1, col2, col3

[ 'magenta', 'blue', 'green', 'red' ] // extraColors

For example 3/3.1, the array we are taking values from is not placed within square brackets, the variables that we would like to assign values to still are. Think of it as grouping variables for tuple assignment.

Example 4:

Destructuring Objects:

// ---------- Example #4 ----------

// Extract values from objects

const person1 = {

    firstName: "Spongebob",

    lastName: "Squarepants",

    age: 30,

    job: "Fry Cook",

}

const person2 = {

    firstName: "Patrick",

    lastName: "Star",

    age: 30,

}

const {firstName, lastName, age, job} = person1;

const {firstName2, lastName2, age2, job2} = person2;

console.**log**(firstName);

console.**log**(lastName);

console.**log**(age);

console.**log**(job);

// job2 undefined because it is not present in person2

// no issues with mismatch in number of arguments

console.**log**(firstName2);

console.**log**(lastName2);

console.**log**(age2);

console.**log**(job2);

To destructure an object, the variables must be placed within curly braces instead of square brackets. The object assigned to the variables is **not** wrapped in curly braces. If there are more arguments than variables, they will be given a value of undefined

const person1 = {

  firstName: "Spongebob",

  lastName: "Squarepants",

  age: 30,

  job: "Fry Cook",

};

const person2 = {

  firstName: "Patrick",

  lastName: "Star",

  age: 40,

};

// the variables must be the same as the keys in person1

const { firstName,  age, lastName, job } = person1;

const { fname, lname, years, action } = person2; // undefined

console.**log**(firstName);

console.**log**(lastName);

console.**log**(age);

console.**log**(job);

The destructuring of person2 results in undefined values because the variables we are assigning it do not match the variables within the person2 object. They result in undefined values when accessing them.

To fix this, we can alias the parameter by adding a colon. {key1: alias1, key2: alias2 ...}

By using aliases, we don't run into the issue of overwriting a previously assigned variable.

const { firstName: pFirst,  lastName: pLast, age: pAge, job: pJob } = person2;

console.**log**(pFirst);

console.**log**(pLast);

console.**log**(pAge);

console.**log**(pJob); // undefined, not present in person2

console.**log**(person2);

Assigning default values is also supported:

const { firstName: pFirst,  lastName: pLast, age: pAge = 3, job: pJob = "unemployed" } = person2;

Because person2 does have an age key, it is assigned a value of 40 instead of the default of 3. Job on the other hand does not have a value (not present in the object, and job: undefined behave the same way) the default value of "unemployed" will be bound to the job/pJob variable.

Example 5:

Destructure in function parameters:

// ---------- Example #5 ----------

// Destructure in function parameters

function **displayPerson**({firstName, lastName, age, job="unemployed"}) {

    console.**log**(`Name: ${firstName} ${lastName}`);

    console.**log**(`Age: ${age}`);

    console.**log**(`Profession ${job}`);

}

**displayPerson**(person1);

**displayPerson**(person2);

**displayPerson**(person3);

Like the previous example, we can set default parameters in the event something is undefined or not present in the original object.

## Nested Objects

Nested objects = objects nested inside of another object. They allow you to represent more complex data structures.

Child object is enclosed by a parent object.

Person{Address{}, ContactInfo{}}

ShoppingCart{Keyboard{}. Mouse{}, Monitor{}}

class Character {

  constructor(name, age, ...address) {

    this.name = name;

    this.age = age;

    // Construct an Address object

    this.address = new **Address**(...address);

  }

}

class Address {

  constructor(street, city, country) {

    this.street = street;

    this.city = city;

    this.country = country;

  }

}

The constructor for the Character object calls the constructor for the Address Object which makes Address a nested object within Character.

const fruits = [

  { name: "apple", color: "red", calories: 95 },

  { name: "orange", color: "orange", calories: 45 },

  { name: "banana", color: "yellow", calories: 105 },

  { name: "coconut", color: "white", calories: 159 },

  { name: "pineapple", color: "yellow", calories: 37 },

];

console.**log**(fruits[0].name);

console.**log**(fruits[0].color);

console.**log**(fruits[0].calories);

console.**log**(fruits[1].name);

console.**log**(fruits[1].color);

console.**log**(fruits[1].calories);

console.**log**(fruits[2].name);

console.**log**(fruits[2].color);

console.**log**(fruits[2].calories);

console.**log**(fruits[3].name);

console.**log**(fruits[3].color);

console.**log**(fruits[3].calories);

console.**log**(fruits[4].name);

console.**log**(fruits[4].color);

console.**log**(fruits[4].calories);

fruits.**push**({ name: "grape", color: "purple", calories: 30 });

// console.log(fruits[5].name);

// console.log(fruits[5].color);

// console.log(fruits[5].calories);

console.**log**(fruits);

fruits.**pop**();

console.**log**(fruits);

// // splice removes elements within a certain range and returns them in a new array

// let a = fruits.splice(1, 2);

// // + doesn't work for concatenating a string to an obj

// console.log("Spliced Array: ", a);

// console.log("Remaining elements in fruits: ", fruits);

console.**log**("For Each: ");

fruits.**forEach**((fruit) => {

  console.**log**(fruit.name);

});

// map()

const fruitNames = fruits.**map**((fruit) => {

  return fruit.name;

});

const fruitColors = fruits.**map**((fruit) => {

  return fruit.color;

});

const fruitCal = fruits.**map**((fruit) => {

  return fruit.calories;

});

console.**log**(fruitNames);

console.**log**(fruitColors);

console.**log**(fruitCal);

// filter example - returns the fruit object if it's color property

//                  is === yellow

const filteredFruits = fruits.**filter**((fruit) => {

  return fruit.color === "yellow";

});

console.**log**(filteredFruits);

// making use of implicit return when calling .filter()

const lowCalFruits = fruits.**filter**((fruit) => fruit.calories < 100);

const highCalFruits = fruits.**filter**((fruit) => fruit.calories > 100);

console.**log**(lowCalFruits);

console.**log**(highCalFruits);

// reduce - compare all fruit elements and return the

//          one with the highest calories

let maxCal = fruits.**reduce**((highest, fruit) =>

  fruit.calories > highest.calories ? fruit : highest

);

let minCal = fruits.**reduce**((lowest, fruit) =>

  fruit.calories < lowest.calories ? fruit : lowest

);

// give acc an initial value of 0, use this to sum the values in an obj

let totalCal = fruits.**reduce**((acc, next) => {

    return acc + next.calories

}, 0)

console.**log**(maxCal);

console.**log**(minCal);

console.**log**(totalCal);

<https://stackoverflow.com/questions/5732043/how-to-call-reduce-on-an-array-of-objects-to-sum-their-properties>

## Sort an Array

Sort() = method used to sort elements of an array in place. Sorts elements as strings in lexicographic order, not alphabetical.

Lexicographical = (alphabet + numbers + symbols) as strings

This mutates the original array.

let fruits = [1, 10, 2, 9, 3, 8];

fruits.**sort**();

console.**log**(fruits);

Be careful with sort, you may forget that it only compares the first character by default, and requires a callback function to compare beyond the first character/value.

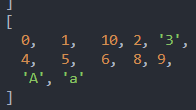
The above code incorrectly sorts the elements and produces the values 1, 10, 2, 3, 8 9 in that order. To fix the sorting provide the following arrow function:

let fruits = [1, 10, 2, 9, 3, 8];

fruits.**sort**((a,b) => a - b);

console.**log**(fruits); // [ 1, 2, 3, 8, 9, 10 ]

Terminal output may add spaces for padding which can look odd at times:



Like the previous nestedObjs examples, you're able to sort objects by a given property.

const people = [

  { name: "Spongebob", age: 30, gpa: 3.0 },

  { name: "Patrick", age: 35, gpa: 1.0 },

  { name: "Squidward", age: 51, gpa: 3.5 },

  { name: "Sandy", age: 25, gpa: 4.0 },

];

// sort is in-place, meaning you don't assign it to a new variable

// the original array is mutated

// highest to lowest

people.**sort**((a, b) => b.gpa - a.gpa);

console.**log**(people);

// lowest to highest

people.**sort**((a, b) => a.age - b.age);

console.**log**(people);

This does not work for comparing strings (values coerced to numbers will work).

// does not work for strings

people.**sort**((a, b) => a.name - b.name);

console.**log**(people);

// to compare strings use the <string.a>.localeCompare(string.b) method

people.**sort**((a,b) => a.name.**localeCompare**(b.name));

console.**log**(people);

### Shuffle an array

// shuffle the elements of an array

const cards = ['A',2,3,4,5,6,7,8,9,10,"J", "Q", "K"];

// not recommended way

// has uniformity issues, and is slower the longer the list is

cards.**sort**(() => Math.**random**() - 0.5);

console.**log**(cards);

Instead, use the Fisher-Yates algorithm

First, start at the end of the array and work your way to the beginning. A for loop will be used to decrement the array elements.

function **shuffle**(array) {

  // -1 due to 0-index ; i > 0; decrement

  for (let i = array.length - 1; i > 0; i--) {

    // get a random number between the current index and 0

    const random = Math.**floor**(Math.**random**() \* (i + 1));

    // use destructuring to swap elements within the array

    [array[i], array[random]] = [array[random], array[i]]

  }

}

**shuffle**(cards);

console.**log**(cards);

For each iteration of the loop, the current card (array[i]) is advanced by 1. This is swapped with a random number array element between the current element and one of the remaining elements. (due to this being a decrementing array, it's starting at index[12] and working its way down to index 0.

The above code shuffles the elements in place, the other way to do this is to have an empty array, randomly select one of the remaining array elements and pop it from the original array and push it into the new array. It's an algorithm for removing one of the remaining elements and inserting that element into a new list.

Below shows all the iterations as the array gets shuffled. Starting from the right, elements become "locked in" while the elements towards index 0 are able to change.

Green shows the current index while orange-yellow shows the randomly selected element, everything green and right is "finalized" when the array is sorted in place, if this went to a new array, these green elements would be popped from this array and appended to the new array removing them from the pool of elements.

[ 'A', 2, 'K', 4, 5, 6, 7, 8, 9, 10, 'J', 'Q', 3]

[ 'Q', 2, 'K', 4, 5, 6, 7, 8, 9, 10, 'J', 'A', 3]

[ 'Q', 2, 'K', 4, 5, 6, 7, 8, 9, 10, 'J', 'A', 3]

[ 'Q', 2, 'K', 4, 5, 10, 7, 8, 9, 6, 'J', 'A', 3]

[ 'Q', 2, 'K', 4, 5, 10, 9, 8, 7, 6, 'J', 'A', 3]

[ 'Q', 2, 'K', 4, 5, 10, 8, 9, 7, 6, 'J', 'A', 3]

[ 'Q', 2, 'K', 4, 8, 10, 5, 9, 7, 6, 'J', 'A', 3]

[ 10, 2, 'K', 4, 8, 'Q', 5, 9, 7, 6, 'J', 'A', 3]

[ 10, 2, 'K', 8, 4, 'Q', 5, 9, 7, 6, 'J', 'A', 3]

[ 10, 8, 'K', 2, 4, 'Q', 5, 9, 7, 6, 'J', 'A', 3]

[ 10, 8, 'K', 2, 4, 'Q', 5, 9, 7, 6, 'J', 'A', 3]

[ 10, 8, 'K', 2, 4, 'Q', 5, 9, 7, 6, 'J', 'A', 3]

In place - select one of the remaining elements and move it to the last unmodified element in the array. Decrement the element tracking the array index and repeat, selecting a new element to swap with the next unmoved element.

// return a new array

select a random element between 0 and array.length and pop that element to add it to the new array. Update array.length to set new upper bound for random and repeat until no elements are remaining in the original array.

## Date Objects

Date objects are objects that contain values that represent dates and times. Those date objects can be changed and formatted.

Tue Jan 23 2024 18:48:47 GMT-0800 (Pacific Standard Time) // from browser date formatting

2024-01-24T02:47:53.496Z // vscode date formatting

    // ( year, 0-11, 0-31, 0-23, 0-59, 0-59, 0-999)

// Date(year, month, day, hour, min, sec, ms)

const date = new **Date**(2024, 0, 1,2,3,4,5);

console.**log**(date);

-> Mon Jan 01 2024 02:03:04 GMT-0800 (Pacific Standard Time) // browser

-> 2024-01-01T10:03:04.005Z // vscode

A string is also valid input for the Date() constructor:

const dateStr = new **Date**("2024-01-02T12:00:00Z");

console.**log**(dateStr);

It's also possible to pass in the number of milliseconds since Epoch.

const fromEpoch = new **Date**(0);

console.**log**(fromEpoch);

Wed Dec 31 1969 16:00:00 GMT-0800 (Pacific Standard Time) Negative values are also valid.

No args passed to Date = current date time

const year = date.**getFullYear**();

const month = date.**getMonth**();

const day = date.**getDay**();

const today = date.**getDate**();

const hour = date.**getHours**();

const minutes = date.**getMinutes**();

const sec = date.**getSeconds**();

const ms = date.**getMilliseconds**();

console.**log**(year); // 2024

console.**log**(month); // 0 (0 = january)

console.**log**(day); // 2 (day of the week, 0-indexed)

console.**log**(today); // 1-31 day of the month

console.**log**(hour); // 19 24h format

console.**log**(minutes);

console.**log**(sec);

console.**log**(ms);

// set custom value for date

date.**setFullYear**(2035);

date.**setMonth**(15); // if > 12, month %12, year + 1

date.**setDate**(3); // set day of the month

date.**setHours**(3);

date.**setMinutes**(45);

date.**setSeconds**(3);

console.**log**(date);

// compare two dates

// these are utc values, browser will adjust accordingly

const date1 = new **Date**("2023-12-31");

const date2 = new **Date**("2024-01-01");

// date values can be compared

if (date2 > date1) {

    console.**log**("Happy New Year");

}

Javascript has poor time zone support natively, try looking for a library to handle time zones.

[https://stackoverflow.com/questions/15141762/how-to-initialize-a-javascript-date-to-a-particular-time-zone#comment111975617\_15141762](https://stackoverflow.com/questions/15141762/how-to-initialize-a-javascript-date-to-a-particular-time-zone%23comment111975617_15141762)

## Closures

javascript/functions/helper/closure.js

A closure is a function defined inside of another function. The inner function has access to the variables and scope of the outer function. Allows for private variables and state maintenance. Used frequently in JS frameworks: React, Vue, Angular

function **outer**() {

    let message = "Hello";

    // inner has access to everything within the outer scope

    function **inner**() {

        console.**log**(message);

    }

}

**outer**(); // nothing happens

To call the inner function, a call to that function needs to be added within the outer() function like so:

function **outer**() {

    let message = "Hello";

    // inner has access to everything within the outer scope

    function **inner**() {

        console.**log**(message);

    }

**inner**();

}

**outer**(); // Hello

One benefit of closures is that any variables within it are considered private. This is because the variable is nested within a different scope, preventing the variable from being accessed.

// maintain state

function **increment**() {

    let count = 0;

    count++;

    console.**log**(`Count increased to ${count}`);

}

**increment**(); // 1

**increment**(); // 1

**increment**(); // 1

Each time increment is called, count is initialized at 0 and then incremented. Each count is isolated to its own function call. By defining Count within the function, only the function is able to modify the state of count.

function **createCounter**() {

    // count scope moved outside of increment function

    let count = 0;

    function **increment**() {

        count++;

        console.**log**(`Count increased to ${count}`);

    }

    // return an object of increment, which is a reference to the function increment

    // can also be written as

    // return {increment}

    return {**customProperty**: **increment**}

}

const counter = **createCounter**();

counter.**customProperty**()

counter.**customProperty**()

counter.**customProperty**()

By moving count outside of increment, we're allowing it to save its state, but still keeping it a private variable because it's nested within another function. The customProperty name is an alias/key assigned the value of increment(), name is arbitrary and return{increment} could have been used instead.

A getter can be added to get the current value of count, it's a "pseudo property", meaning it doesn't actually exist, it's just a value returned when calling the getter.

function **createCounter**() {

    // count scope moved outside of increment function

    let count = 0;

    function **increment**() {

        count++;

        console.**log**(`Count increased to ${count}`);

    }

    // return an object of increment, which is a reference to the function increment

    // can also be written as

    // return {increment}

    // customProperty is an arbitrary name

    function **getCount**() {

        return count;

    }

    return {**customProperty**: **increment**, **getCount**}

}

const counter = **createCounter**();

counter.**customProperty**() // 1

counter.**customProperty**() // 2

counter.**customProperty**() // 3

console.**log**(counter.**getCount**()); // 3

Closure Practice

let score = 0;

function **increaseScore**(points) {

    score += points;

    console.**log**(`+${points}pts`);

}

function **decreaseScore**(points) {

    score -= points;

    console.**log**(`-${points}`);

}

function **getScore**() {

    return score;

}

**increaseScore**(5);

**increaseScore**(6);

**increaseScore**(3);

console.**log**(`The final score is ${**getScore**()}pts`);

The problem with this is that score can be accessed at any time and modified. Use a closure to prevent modification of the score variable.

function **createGame**(){

    let score = 0;

    function **increaseScore**(points) {

        score += points;

        console.**log**(`+${points}pts`);

    }

    function **decreaseScore**(points) {

        score -= points;

        console.**log**(`-${points}pts`);

    }

    function **getScore**() {

        return score;

    }

    // return an object reference to these methods

    return {**increaseScore**, **decreaseScore**, **getScore**}

}

// create a 'createGame' object

let game = **createGame**();

game.**increaseScore**(3);

game.**increaseScore**(5);

game.**decreaseScore**(2);

// score is private and cannot be modified without a setter, this score is unrelated to the score within createGame object

game.score = 1000;

console.**log**(game);

console.**log**(`The final score is ${game.**getScore**()}pts`); // 6

## setTimeout()

setTimeout() is a function in Javascript that allows you to schedule the execution of a function after an amount of time (in ms). Times are approximate (varies based on the workload of the js runtime env.) setTimeout(callback, delay).

function **sayHello**() {

    window.**alert**("Hello");

}

**setTimeout**(() => {

**sayHello**()

}, 3000);

Anonymous functions work too:

// anonymous functions work too

**setTimeout**(() => {

    window.**alert**("Hello")

}, 3000);

There's also the clearTimeout() function which is able to cancel a timeout before it triggers. it takes a timeoutId as a parameter.

// creating timeoutId for clearTimeout() func

const timeoutId = **setTimeout**(() => {

    window.**alert**("Hello")

}, 3000);

// stops setTimeout() from running

**clearTimeout**(timeoutId);

Now to make this have an effect on a webpage.

    <button onclick="**startTimer**()">START</button>

    <button onclick="**clearTimer**()">CLEAR</button>

timer.js:

let timeoutId;

function **startTimer**() {

    // give timeoutId a value

    timeoutId = **setTimeout**(() => {

        window.**alert**("Hello");

    }, 3000);

    console.**log**("Timer started");

}

function **clearTimer**() {

**clearTimeout**(timeoutId);

    console.**log**("Timer cleared");

}

initialize a timeoutId variable which gets a value assigned to it once the start button on the website is clicked. If the CLEAR button is clicked, the callback function in startTimer will not run and the setTimeout will stop running.

# Digital Clock app

index.html:

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Digital Clock</title>

    <link rel="stylesheet" href="style.css">

</head>

<body>

    <div id="clock-container">

        <!-- placeholder time value -->

        <div id="clock">00:00:00</div>

    </div>

    <script src="index.js"></script>

</body>

</html>

style.css:

body {

    margin: 0;

    background-image: url(../Assets/clock/Waves\ Centered\ 1080\ 40000.png);

    /\* center image \*/

    background-position: center;

    background-repeat: no-repeat;

    background-size: cover;

    /\* property to create a static background image

    if the page had any scrolling \*/

    background-attachment: fixed;

}

#clock-container {

    display: flex;

    /\* center elements horizontally \*/

    justify-content: center;

    /\* vertically align items \*/

    align-items: center;

    height: 100vh;

}

#clock {

    font-family: monospace;

    font-size: 6.5rem;

    font-weight: bold;

    text-align: center;

    color: white;

    /\* add a blur behind the clock \*/

    backdrop-filter: blur(15px);

    width: 100%;

    background-color: hsla(0, 0%, 100%, 0.1);

}

index.js:

// digital clock program

function **updateClock**() {

    const now = new **Date**();

    // convert to string so that we can pad to two digits

    // 12h format

    let hours = now.**getHours**();

    const meridiem = hours >= 12 ? "PM": "AM";

    hours = hours % 12 || 12;

    hours = hours.**toString**().**padStart**(2, 0);

    const minutes = now.**getMinutes**().**toString**().**padStart**(2,0);

    const seconds = now.**getSeconds**().**toString**().**padStart**(2,0);

    // change clock text to be a template string

    const timeString = `${hours}:${minutes}:${seconds} ${meridiem}`

    // target #clock so that we can update that DOM obj

    document.**getElementById**("clock").textContent = timeString;

}

// get clock to update every second

/\* called with an arrow func

setInterval(() => {

    updateClock();

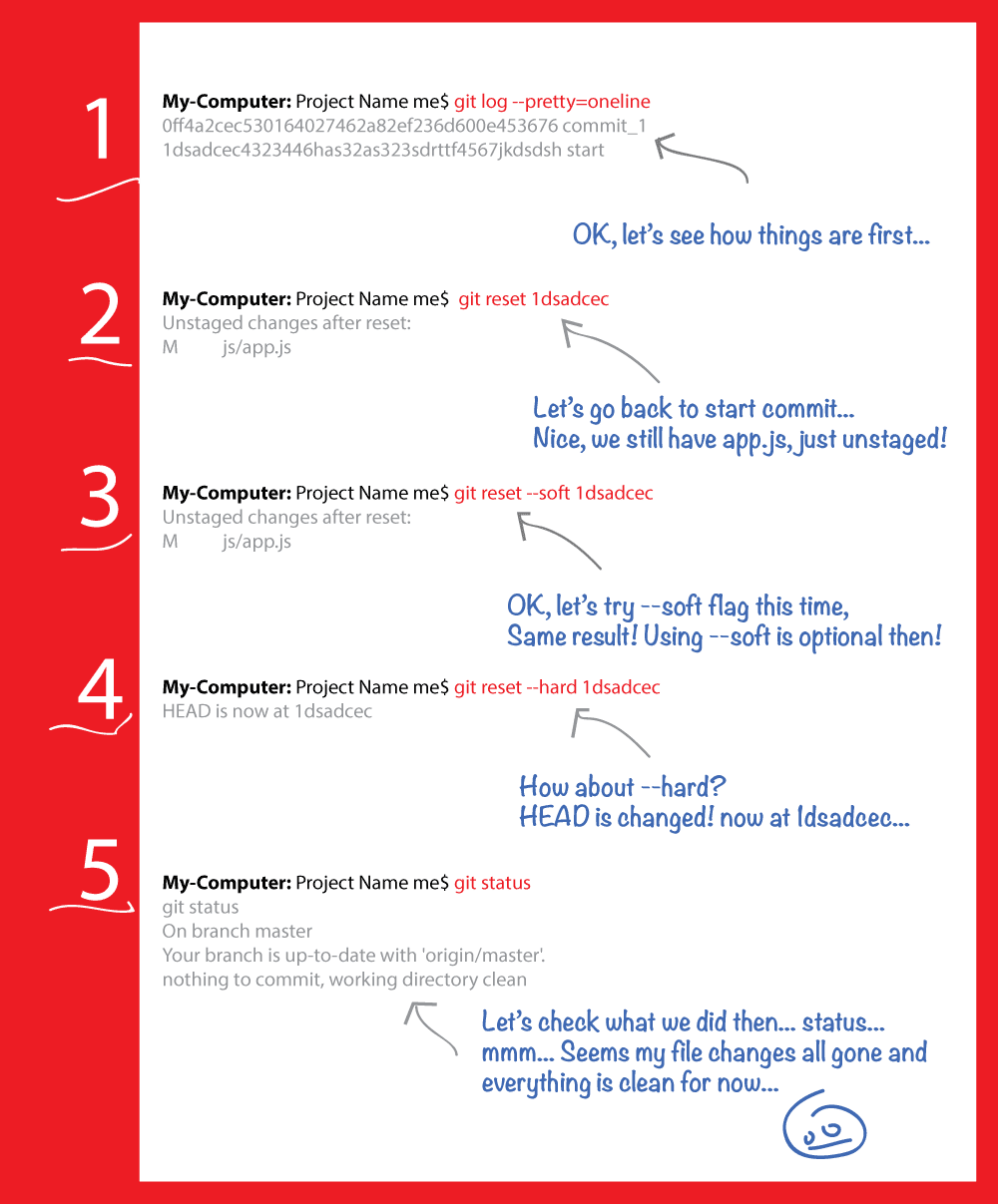
}, 1000);

\*/

// called as a callback func

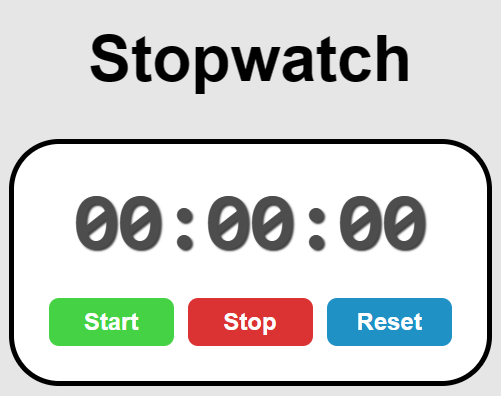
**setInterval**(**updateClock**, 1000);

Random Git --soft vs --hard



<https://stackoverflow.com/a/45219734>

## Stopwatch App



The Html is pretty minimal:

<body>

    <h1 id="myH1">Stopwatch</h1>

    <div id="container">

        <div id="display">00:00:00</div>

        <div id="controls">

            <button id="startBtn" onclick="**start**()">Start</button>

            <button id="stopBtn" onclick="**pause**()">Stop</button>

            <button id="resetBtn" onclick="**reset**()">Reset</button>

        </div>

    </div>

    <script src="index.js"></script>

</body>

Here's the CSS:

body {

  display: flex;

  flex-direction: column;

  align-items: center;

  background-color: hsl(0, 0%, 90%);

}

#myH1 {

  font-size: 4rem;

  font-family: Arial, Helvetica, sans-serif;

  color: hsl(0%, 0%, 25%);

}

#container {

  display: flex;

  flex-direction: column;

  align-items: center;

  border: 5px solid;

  padding: 30px;

  border-radius: 50px;

  background-color: white;

}

#display {

  font-size: 5rem;

  font-family: monospace;

  font-weight: bold;

  color: hsl(0, 0%, 30%);

  /\* give the text some depth \*/

  text-shadow: 2px 2px 2px hsla(0, 0%, 0%, 0.75);

  margin-bottom: 25px;

}

#controls button {

    font-size: 1.5rem;

    font-weight: bold;

    padding: 10px 20px;

    margin: 5px;

    min-width: 125px;

    border: none;

    border-radius: 10px;

    cursor: pointer;

    color: white;

    transition: background-color 0.5s ease;

}

#startBtn {

    background-color: hsl(120, 62%, 55%);

}

#startBtn:hover {

    background-color: hsla(120, 62%, 55%, 0.7);

}

#stopBtn {

    background-color: hsl(0, 70%, 53%);

}

#stopBtn:hover {

    background-color: hsla(0, 70%, 53%, 0.7);

}

#resetBtn {

    background-color: hsl(199, 72%, 45%);

}

#resetBtn:hover {

    background-color: hsla(199, 72%, 45%, 0.7);

}

index.js:

// using eslint-disable-next-line to remove warning

// when html does make user of function/var

// setting variables for targeting elements

const display = document.**getElementById**("display");

/\* start stop and reset aren't used, onclick="" handles the button elements

const startBtn = document.getElementById("startBtn");

const stopBtn = document.getElementById("pauseBtn");

const resetBtn = document.getElementById("resetBtn");

\*/

// holds the id of setInterval

let timer = null;

let startTime = 0;

let elapsedTime = 0;

let isRunning = false;

/\* eslint-disable-next-line \*/

function **start**(){

    if (!isRunning) {

        startTime = Date.**now**() - elapsedTime;

        timer = **setInterval**(**update**, 10);

        isRunning = true;

    }

}

// eslint-disable-next-line

function **pause**(){

**clearInterval**(timer);

    isRunning = false;

}

// eslint-disable-next-line

function **reset**() {

**clearInterval**(timer);

    startTime = 0;

    elapsedTime = 0;

    isRunning = false;

    display.textContent = "00:00:00:00";

}

function **update**() {

    const currentTime = Date.**now**();

    elapsedTime = currentTime - startTime;

    // convert ms to human readable time

    let hours = Math.**floor**(elapsedTime / (1000 \* 60 \* 60));

    // strip out hours using %

    let minutes = Math.**floor**(elapsedTime / (1000 \* 60) % 60)

    let seconds = Math.**floor**(elapsedTime / 1000 % 60)

    let milliseconds = Math.**floor**(elapsedTime % 1000 / 10)

    hours = **String**(hours).**toString**().**padStart**(2,0);

    minutes = **String**(minutes).**toString**().**padStart**(2,0);

    seconds = **String**(seconds).**toString**().**padStart**(2,0);

    milliseconds = **String**(milliseconds).**toString**().**padStart**(2,0);

    // update hour, min, sec, ms

    display.textContent = `${hours}:${minutes}:${seconds}:${milliseconds}`

}

# ES6 Module

A Module is an external file that contains reusable code that can be imported into other Javascript files. Write reusable code for many different apps. Can contain variables, classes, functions... and more. Introduced as part of ECMAScript 2015.

To import a module, we need to set the type attribute within the html element as module for the supplementary file.

// The file being is the source file, not the module file

    <script type="module" src="index.js"></script>

The reusable code goes into mathUtil.js.

Reusable code should be prefixed with the export keyword.

mathUtil.js

export const PI = 3.14159;

export function **getCircumference**(radius) {

    return 2 \* PI \* radius;

}

export function **getArea**(radius) {

    return PI \* radius \* radius;

}

export function **getSphereVolume**(radius) {

    return 4 \* PI \* radius\*\*2

}

export function **getSphereArea**(radius) {

    return 4/3 \* PI \* radius\*\*3

}

### Importing

To import, we need to use object destructuring. Within index.js add the following line of code:

import {} from '';

Within the curly braces, name the "keys" you would like to import:

import {PI, getCircumference, getArea, getSphereVolume} from './mathUtil.js';

The keys can then be used to call their respective functions or reference their stored values:

import {PI, getCircumference, getArea, getSphereVolume} from './mathUtil.js';

// const PI

console.**log**(PI);

const area = **getArea**(10);

const circumference = **getCircumference**(10);

const sphereVolume = **getSphereVolume**(10);

//circle

console.**log**(`The circumference of your circle is ${circumference.**toFixed**(2)}cm`);

console.**log**(`The area of your circle is ${area.**toFixed**(2)}cm^2`);

// sphere

console.**log**(`The volume of your sphere is ${sphereVolume.**toFixed**(2)}cm^3`);

# Synchronous/Asynchronous

Synchronous executes line by line consecutively in a sequential manner.

Asynchronous allows multiple operations to be performed concurrently without waiting. Does not block the execution flow and allows the program to continue. (I/O operations, network requests, fetching data). Handled with **Callbacks**, **Promises**, **Async**/**Await**.

Task 1 is asynchronous while tasks 2-4 are synchronous. This simulates waiting on input which is asynchronous and then executing the rest of the program once the input is received.

function **func1**(callback) {

**setTimeout**(() => {

    console.**log**("Task 1");

    // don't forget () to invoke the callback

**callback**()

  }, 3000);

}

function **func2**() {

  console.**log**("Task 2");

  console.**log**("Task 3");

  console.**log**("Task 4");

}

Only after task 1 is complete will func2 execute.

## Error Handling

Error = an object that is created to represent a problem that occurs. Occur often with user input or establishing a connection.

An error that has not been caught will interrupt the execution of the program.

// comment me

let x = "Hello";

try {

  console.**log**(x);

  //    network errors

  //    promise rejection

  //    security errors

} catch (error) {

  console.**error**(error);

} finally {

  // close files

  // close connections

  // release resources

  console.**log**("This always executes");

}

console.**log**("You have reached the end!");

// handle bad user input

try {

    const dividend = **Number**(window.**prompt**("Enter a dividend: "));

    // dangerous

    const divisor = **Number**(window.**prompt**("Enter a divisor: "));

    if (divisor == 0) {

        throw new **Error**("You can't divide by zero!");

    }

    if (**isNaN**(dividend) || **isNaN**(divisor)){

        throw new **Error**("Values must be a number")

    }

    if (!dividend) {

        throw new **Error**("No dividend entered.");

    }

    if (!divisor) {

        throw new **Error**("No Divisor entered.");

    }

    const result = dividend / divisor;

    console.**log**(result);

} catch (error) {

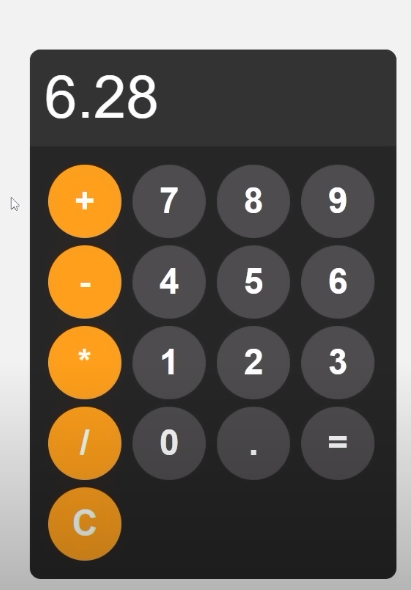
    console.**log**(error);

}

console.**log**("This is the end of the program!");

Very similar to Pythons Try Except, minor differences like being able to alias the error with as keyword instead of naming it in the catch() block.

## Calculator App



Here's the general HTML structure for the calculator:

    <div id="calculator">

        <input type="text" name="" id="display" readonly>

        <div id="keys">

            <button onclick="**appendToDisplay**('+')">+</button>

            <button onclick="**appendToDisplay**('7')">7</button>

            <button onclick="**appendToDisplay**('8')">8</button>

            <button onclick="**appendToDisplay**('9')">9</button>

            <button onclick="**appendToDisplay**('-')">-</button>

            <button onclick="**appendToDisplay**('4')">4</button>

            <button onclick="**appendToDisplay**('5')">5</button>

            <button onclick="**appendToDisplay**('6')">6</button>

            <button onclick="**appendToDisplay**('\*')">\*</button>

            <button onclick="**appendToDisplay**('1')">1</button>

            <button onclick="**appendToDisplay**('2')">2</button>

            <button onclick="**appendToDisplay**('3')">3</button>

            <button onclick="**appendToDisplay**('/')">/</button>

            <button onclick="**appendToDisplay**('0')">0</button>

            <button onclick="**appendToDisplay**('.')">.</button>

            <button onclick="**calculate**()">=</button>

            <button onclick="**clear**()">C</button>

        </div>

    </div>

The structure is very simple, it's a container for all of the contents, an input field with the readonly attribute as a display (this could also be a div) and a set of buttons, where anything that is a symbol will call an "appendToDisplay('value')" function, a calculate button ('=') and a clear button.



Calculator CSS:

Here's the general button styling:

button {

    width: 100px;

    height: 100px;

    border-radius: 50px;

    border: none;

    background-color: hsl(0, 0%, 30%);

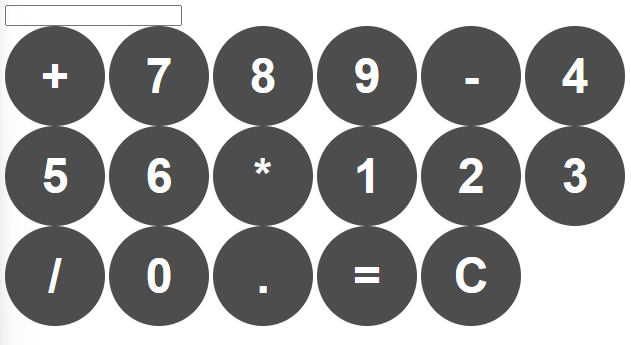
    color: white;

    font-size: 3rem;

    font-weight: bold;

    cursor: pointer;

}



Next, target the #keys container and set the display mode to grid.

#keys {

    display: grid;

    /\* create 4 columns 1 fraction wide each\*/

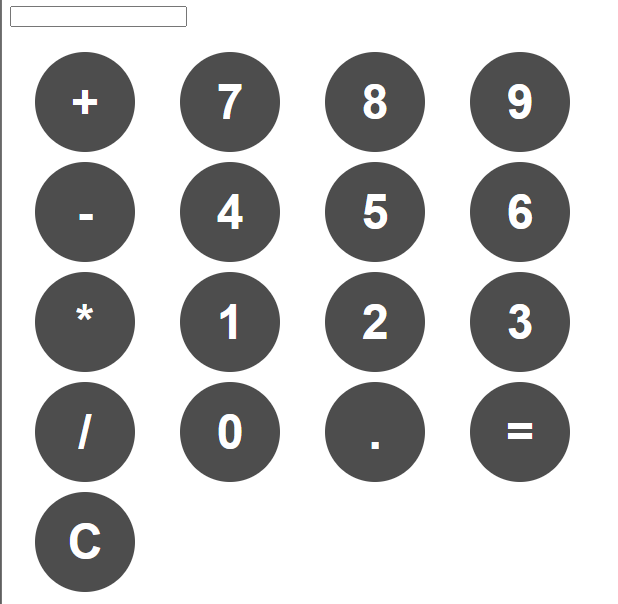
    grid-template-columns: repeat(4, 1fr);

    gap: 10px;

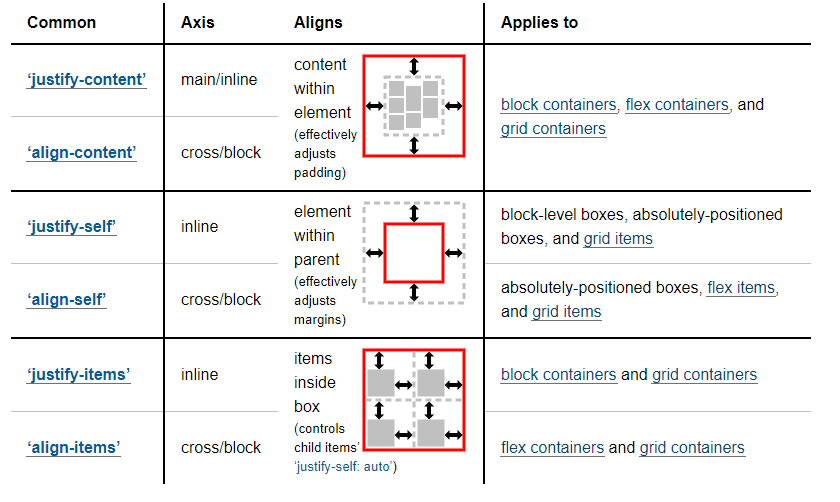
    padding: 25px;

}

Without specifying a column width, the keys do not get distributed into the four columns, and remain in one monolithic column. The elements are distributed left to right.



### Helpful flex/grid diagram from W3



https://stackoverflow.com/a/48553129

[https://www.w3.org/TR/css-align-3/ - overview](https://www.w3.org/TR/css-align-3/#overview)

Back to calculator:

For the operator keys (+, -, /, \*, C), add a class named .operator-btn so that these keys can be styled separately from numeric keys. This is the final CSS for styling the calculator:

body {

    margin: 0;

    display: flex;

    /\* center contents horizontally \*/

    justify-content: center;

    /\* requires a document height property set to vertically center

       with align-items using height: 100vh achieves this \*/

    align-items: center;

    height: 100vh;

    background-color: hsl(0, 0%, 95%);

}

#calculator {

  font-family: Arial, Helvetica, sans-serif;

  background-color: hsl(0, 0%, 15%);

  border-radius: 15px;

  max-width: 500px;

  overflow: hidden;

}

#display {

    width: 100%;

    padding: 20px;

    font-size: 5rem;

    text-align: left;

    border: none;

    background-color: hsl(0, 0%, 20%);

    color: white;

}

#keys {

  display: grid;

  /\* create 4 columns 1 fraction wide each\*/

  grid-template-columns: repeat(4, 1fr);

  gap: 10px;

  padding: 25px;

}

button {

  width: 100px;

  height: 100px;

  border-radius: 50px;

  border: none;

  background-color: hsl(0, 0%, 30%);

  color: white;

  font-size: 3rem;

  font-weight: bold;

  cursor: pointer;

}

button:hover {

    background-color: hsl(0, 0%, 40%);

}

button:active {

    background-color: hsl(0, 0%, 50%);

}

.operator-btn {

    background-color: hsl(35, 100%, 55%);

}

.operator-btn:hover {

    background-color: hsl(35, 100%, 65%);

}

.operator-btn:active {

    background-color: hsl(35, 100%, 75%);

}i

### Calculator Javascript

First, get the display element.

const display = document.**getElementById**("display");

// negative expression [^...], check if characters are outside of this set

// do not execute if invalid char detected

// eslint-disable-next-line

const re = new **RegExp**("[^0-9-+\*/.]");

// if previous symbol was "[./\*-+]", replace with new input

// eslint-disable-next-line

const repeatSymbol = new **RegExp**("[-./\*+]$");

display.value = 0;

// eslint-disable-next-line

function **appendToDisplay**(input) {

  // overwrite duplicate symbol

  if (

    display.value.**charAt**(display.value.length - 1).**match**(repeatSymbol) &&

    input.**match**(repeatSymbol)

  ) {

    // console.log("match Detected");

    display.value = display.value.**slice**(0, display.value.length - 1) + input;

  } // replace leading zero

  // need to fix this section

  else if (input > 0 && display.value === "0") {

    display.value = input;

  } else {

    // "bug" doesn't effect result, but allows for padding of leading 0s

    // would require regex to backtrack to last /\*-+ and check if there were a number > 0

    display.value += input;

  }

  console.**log**(display.value);

}

// eslint-disable-next-line

function **calculate**() {

  // try to prevent bad user input

  if (display.value.**match**(re)) {

    throw **Error**(`InvalidInput: ${display.value}`);

  }

  try {

    display.value = **eval**(display.value);

  } catch (error) {

    display.value = "Error";

  }

}

// eslint-disable-next-line

function **clearDisplay**() {

  display.value = "0";

}

/\*

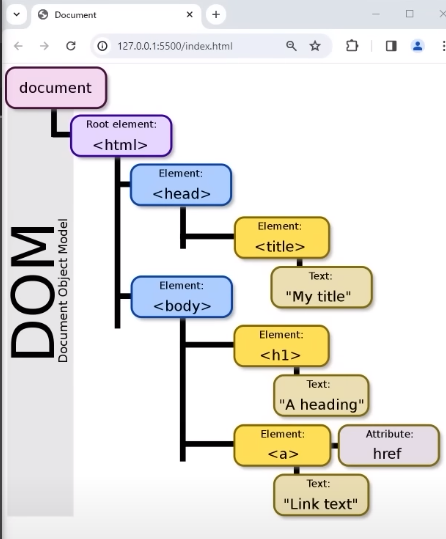
    There are a couple of "issues" that could be fixed, currently it's

    possible to have multiple 0s instead of being limited to 1 0.

    Floating points are left as is, could apply .toFixed(n) for better precision,

\*/

# What is the DOM



DOM = Document Object Model

Object{} that represents the page you see in the web browser and provides you with an API to interact with it. Web browser constructs the DOIM when it loads an HTML document, and structures all the elements in a tree-like representation. Javascript can access the DOM to dynamically change the content, structure, and style of a webpage.

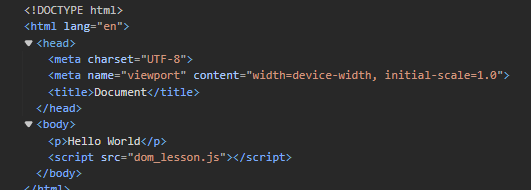
Depending on how the website is loaded, the two lines will print different content to the console.

console.**log**(document);

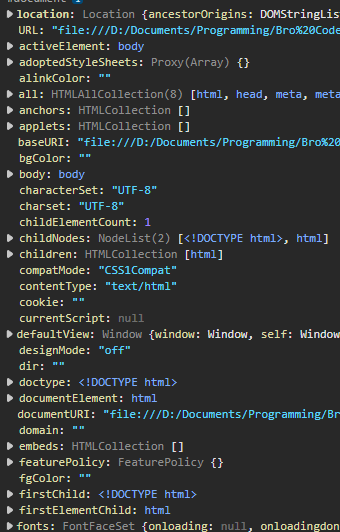
console.**dir**(document);

Directly launching the webpage as "D:/Documents...index.html" console.log(document); will print out the raw html to the console while console.dir(document) will print out all the properties.

Console.log(document):



Console.dir(document): (console.directory)

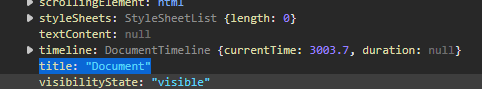


When launching the web page from live server, both console.log and dir appear to print the console.dir(document) contents.

Console.log(window)/.dir(window) behave the same regardless of which way the site was launched, printing properties to the console.

The objects listed in this view can be selected and have their properties altered (console.dir(document) 's view) . For example, changing the title's contents through javascript.

Console.dir(document);:



We can see that the document object has a title property and that its current value is "Document". In index.js, select document.title and set the value to "My Website"

document.title = "My Website";

Here's the same section of the document which has been modified after the previously shown line of Javascript:



Here's another example, where the background colour is modified:

document.body.style.backgroundColor = "hsl(0 0% 20%)";



Browser represents this as rgb, but the value is correct.

Here's another example where a welcome message either appends a username or guest to the word welcome depending on if username is an empty string or not.

let username = "Bobby Tables";

// username = "";

const welcomeMsg = document.**getElementById**("welcome-msg")

// mixing in ternary to check for empty string

welcomeMsg.textContent += username === "" ? ` Guest`: ` ${username}`;

HTML:

    <h1 id="welcome-msg">Welcome</h1>



# Element Selectors

Element selectors = Methods used to garget and manipulate HTML elements

They allow you to select one or multiple HTML elements from the DOM.

    1. document.getElementById()       // ELEMENT OR NULL

    2. document.getElementsClassName() // HTML COLLECTION

    3. document.getElementsByTagName() // HTML COLLECTION

    4. document.querySelector()        // First ELEMENT OR NULL

    5. document.querySelectorAll()     // NODELIST

document.getElementById():

    <h1 id="my-heading">Food R Us</h1>

// uses camelCase when accessed through JS, spine-case in HTML/CSS

myHeading.style.backgroundColor = "yellow";

myHeading.style.textAlign ="center";

console.**log**(myHeading); // inspect the console



The code directly modified the HTML and added inline style properties.

\*Trying to get an element my ID that does not exist will print null to the console if you console.log it. (i.e. typo in my-heading and then console.log(myHeading).

\*If you attempted to modify an element that didn't exist before using console.log, the resulting error message would stop the rest of the program from executing before making it to console.log.

### Get Elements by class name

This returns a collection, which is similar to an array, but is limited in its built-in methods.

HTML:

    <div class="fruits">Apple</div>

    <div class="fruits">Orange</div>

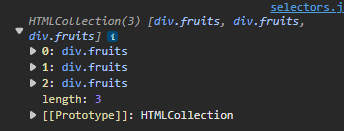
    <div class="fruits">Banana</div>

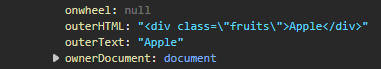
Javascript:

const fruits = document.**getElementsByClassName**("fruits");

console.**log**(fruits);

Viewing fruits in console:



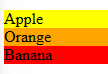
Accessing an element is the same as in an array.

// access and modify an HTML collection

fruits[0].style.backgroundColor = "yellow";

fruits[1].style.backgroundColor = "orange";

fruits[2].style.backgroundColor = "red";

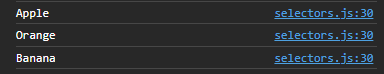


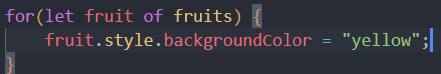
If you would like to iterate over the elements of an HTML collection, you can use an enhanced for loop to do it.

for(let fruit of fruits) {

    console.**log**(fruit.textContent);

}







HTML Collections **do not** have a For Each method. You will see a TypeError: fruits.forEach is not a function in the console if you attempt to use this method on one.

To get around this, you can typecast the HTML collection as an array.

// typecast an HTML collection as an array to gain access to .forEach()

Array.**from**(fruits).**forEach**(fruit => {

    fruit.style.backgroundColor = "orange";

})



### Get Elements by Tag Name

HTML

<h4>Root Vegetables</h4>

    <ul>

        <li>Beets</li>

        <li>Carrots</li>

        <li>Potatoes</li>

    </ul>

<h4>Non-Root Vegetables</h4>

    <ul>

        <li>Broccoli</li>

        <li>Celery</li>

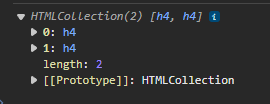
        <li>Onion</li>

    </ul>

JS:

const h4Elements = document.**getElementsByTagName**("h4");

console.**log**(h4Elements);



Everything that applied to getElementsByClassName applies here too. It's a selection by HTML <tag>

One thing to note is that you can't apply a rule to everything in a collection without some form of loop (including single element collections). Bracket Notation must be used to access the element and modify the element, or a loop to access an iteration of the collection.

h4Elements.style.backgroundColor = "yellow";

h4Elements[0].style.backgroundColor = "yellow"; // target single

for (const h4Element of h4Elements) {

    h4Element.style.backgroundColor = "yellow";

}

loop through all elements and color them yellow.

More practice:

const liElements = document.**getElementsByTagName**("li");

for (const liElement of liElements) {

    liElement.style.background = "lightgreen";

}

Array.**from**(h4Elements).**forEach**(

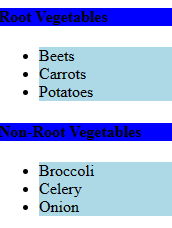
  (h4Element) => (h4Element.style.backgroundColor = "blue")

);

Array.**from**(liElements).**forEach**(

  (liElement) => (liElement.style.backgroundColor = "lightBlue")

);



### Query Selector

The Query Selector returns the first matching element or NULL if there is no match.

// use a . to access class

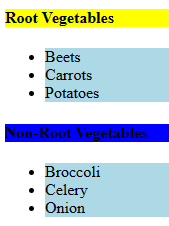
const element = document.**querySelector**(".fruits");

element.style.backgroundColor ="yellow";

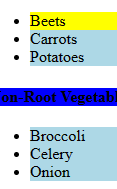


const element = document.**querySelector**("h4"); // root vegetables

element.style.backgroundColor ="yellow";



const element = document.**querySelector**("li");



const element = document.**querySelector**("ul");

// light blue comes from a different rule targeting list items. The yellow actually extends behind the list elements.



Returns null if there are no matches (and nothing attempts to use the variable).

### Query Selector All

This returns Node List, which is similar to an HTML Collection, has built in methods similar to an array but are static, unlike an HTML collection which is Live.

What this means is that an HTML Collection can update automatically in the DOM, while a NodeList will not update in the DOM automatically.

// query selector all

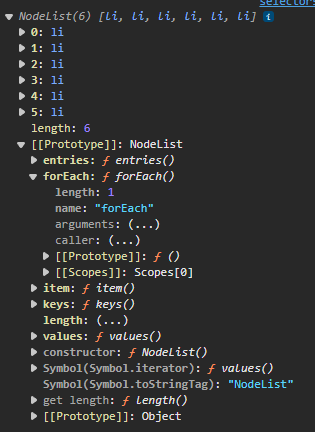
const berries = document.**querySelectorAll**(".fruits");

console.**log**(berries);

Like an HTML Collection, only an instance can be modified at a time, not the entire set.

berries[1].style.backgroundColor = "cyan";





NodeList > [[Prototype]] > methods

From here, we can see that the NodeList has a foreach method already, instead of requiring us to typecast as an Array.

// query selector all returns a NodeList (static)

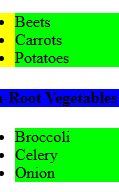
const foods = document.**querySelectorAll**("li");

console.**log**(foods);

// foods[1].style.backgroundColor = "cyan";

foods.**forEach**(food =>

    food.style.backgroundColor ="lime");



# DOM Navigation

The Process of navigation through the structure of an HTML document using Javascript.

These properties can be used to traverse the DOM structure.

    .firstElementChild

    .lastElementChild

    .nextElementSibling

    .previousElementSibling

    .parentElement

    .children

HTML structure:

    <ul id="fruits">

        <li>apple</li>

        <li>orange</li>

        <li>banana</li>

    </ul>

    <ul id="vegetables">

        <li>carrots</li>

        <li>onions</li>

        <li>potatoes</li>

    </ul>

    <ul id="desserts">

        <li>cake</li>

        <li>pie</li>

        <li>ice cream</li>

    </ul>

JS

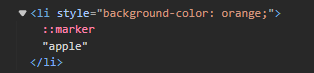
.firstElementChild

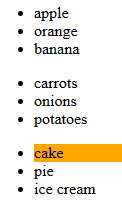
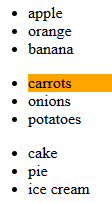
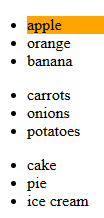
The unordered lists are elements, they each have their own children (child elements). A child element is any element found nested within the targeted element. The three parent elements are the unordered lists and their child elements are apple, orange, banana; carrots, onions potatoes; cake, pie, ice cream respectively. The "first born" elements would be apple, carrots and cake, second child orange, onions, pie and the last born would be banana, potatoes and ice cream.

const element = document.**getElementById**("fruits");

const firstChild = element.firstChild;

console.**log**(firstChild);



 // fruits, vegetables, desserts respectively.

// using a NodeList with querySelectorAll and selecting first child

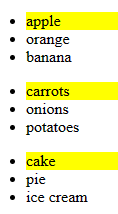
const ulElements = document.**querySelectorAll**("ul");

ulElements.**forEach**((ulElement) => {

    const firstChild = ulElement.firstElementChild;

    firstChild.style.backgroundColor = "yellow";

});



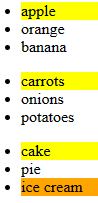
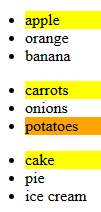
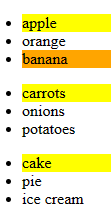
// Selecting the last child element

const lastChildElement = document.**getElementById**("fruits");

const lastChild = lastChildElement.lastElementChild;

console.**log**(lastChild);

lastChild.style.backgroundColor = "orange";

 // fruits, vegetables, desserts

// select all ul elements and return a NodeLIst

const ulElement = document.**querySelectorAll**("ul");

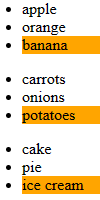
// styling last child from a Node List

ulElement.**forEach**(ulElement => {

    const lastChild = ulElement.lastElementChild;

    lastChild.style.backgroundColor ="orange";

})



### Next Element Sibling

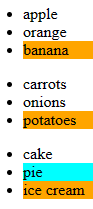
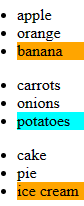
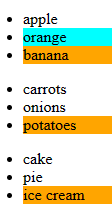
The HTML doc was updated to give each li an id corresponding to it's value ( <li id="apple>apple</li>

// --------------- .nextElementSibling ---------------

const listItem = document.**getElementById**("apple");

const nextSibling = listItem.nextElementSibling;

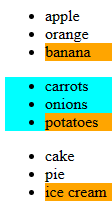
nextSibling.style.backgroundColor ="cyan";

 apple, onion, pie

const listItem = document.**getElementById**("fruits");

const nextSibling = listItem.nextElementSibling;

nextSibling.style.backgroundColor ="cyan";

 The next sibling element is now colored cyan (uls: fruits, vegetables desserts)

### Previous element sibling

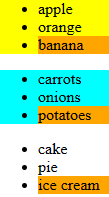
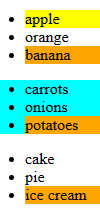
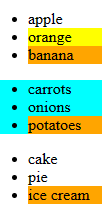
Same as above but selects the previous sibling in the DOM.

// --------------- .previousElementSibling ---------------

const prevItem = document.**getElementById**("banana");

const prevSibling = prevItem.previousElementSibling;

prevSibling.style.backgroundColor = "yellow";

 banana, orange, vegetables.

### Parent Element

This selects the parent element of currently selected element.

const childElement = document.**getElementById**("banana");

const parentElement = childElement.parentElement;

parentElement.style.border = "5px solid black";



### Children

const parent = document.**getElementById**("desserts");

const children = parent.children;

console.**log**(children);

This returns an html collection, cast as an array if you would like to modify multiple elements.

const parent = document.**getElementById**("desserts");

const children = parent.children;

// console.log(children); // children is an HTML Collection, not a NodeList

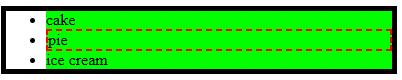
Array.**from**(children).**forEach**(child =>{

    child.style.backgroundColor = "lime";

})



children[1].style.border = "2px dashed red";



# Add and change HTML elements using JavaScript

/\* Adding elements to the DOM

    Step 1: Create the element

    Step 2: Add Attributes/Properties

    Step 3: Append the element to the DOM

    Remove the HTML element

\*/

To create a new element, call the createElement() method as shown:

const newH1 = document.**createElement**("h1");

Step 2: Set any attributes/properties:

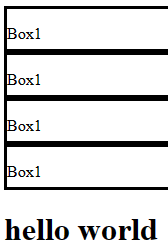
// Step 2: add Attribute/Properties

newH1.textContent = "hello";

newH1.textContent += " world";

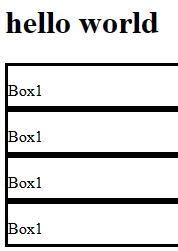
Step 3: Append Element to the DOM:

document.body.**append**(newH1);



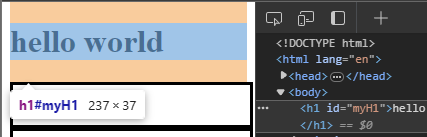
This element was appended to the very end of the DOM. If you would like it to be the first child, use prepend instead:

document.body.**prepend**(newH1);



// adding a property

newH1.id = "myH1";



We can see that the h1 is given an id of myH1 after setting the id property.

Place newH1 into a box:

document.**getElementById**("box1").**append**(newH1);



Prepend can be used the same way, to place the generated HTML element before any other contents of the parent element:

document.**getElementById**("box1").**prepend**(newH1);



To insert an element between two divs instead of inside, first create a reference to one of the nodes you would like the element to come before or after. Next call the following method:

document.body.**insertBefore**(node, child)

Provide the method with a Node you just created and the sibling element you would like to insert it next to. (new element, current element).

### Selecting an element that does not have an ID:

To do this, use the .querySelectorAll() method, it will return a NodeList which you can use to anchor an insertion to.

// inserting an element when you do not have an ID to work with:

// select anything with a class of box, creates a NodeList

const boxes = document.**querySelectorAll**(".box");

document.body.**insertBefore**(newElement, existingElement);

document.body.**insertBefore**(newH1, boxes[2]);

 (0-indexed)

// Remove an HTML element

// element is somewhere in the body

document.body.**removeChild**(newH1);

Just need to understand how to target the location of the element then call .removeChild(reference) to remove part of the DOM.

// element is within box1

document.**getElementById**("box1").**removeChild**(newH1);

### Working with an ordered list

**HTML**

<!DOCTYPE html>

<!-- Part2 of inserting additional elements -->

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

    <link rel="stylesheet" href="additionalElements.css">

</head>

<body>

    <ol id="fruits">

        <li id="apple">apple</li>

        <li id="orange">orange</li>

        <li id="banana">banana</li>

    </ol>

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

</body>

</html>

**CSS**

/\* Part 2 of inserting elements \*/

#fruits {

    border: 3px solid;

    font-size: 2rem;

}

// step 1: create an element

const newListItem = document.**createElement**("li");

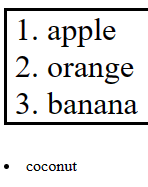
// step 2: Add attributes/properties

newListItem.textContent = "coconut";

// step3: append element to DOM

// element needs to be inserted into the list not appended to the DOM

document.body.**appendChild**(newListItem);



Some Changes:

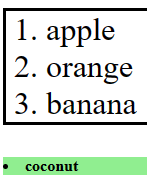
// step 2: Add attributes/properties

newListItem.textContent = "coconut";

newListItem.id = "coconut";

newListItem.style.fontWeight = "bold";

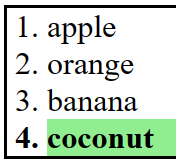
newListItem.style.backgroundColor = "lightgreen";



// inserting element to correct location

// Target the ordered list and append the newListItem there

document.**getElementById**("fruits").**append**(newListItem)



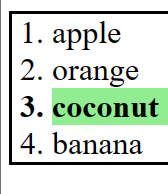
Inserting an element between a list item:

// insert between elements:

const orange = document.**getElementById**("orange");

orange.**append**(newListItem);

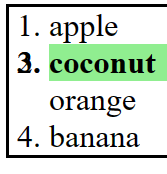
First create a reference to the target element, then append your new element to the existing.



Alternatively parent.insertBefore(target, Orange).

(the append/**prepend** method may have issues where numbers will overlap)

orange.**prepend**(newListItem);



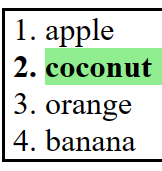
It is probably safer to call the insertBefore method:

// insert between elements: (parent.insertBefore(new, existing))

const fruits = document.**getElementById**("fruits");

const orange = document.**getElementById**("orange");

fruits.**insertBefore**(newListItem, orange);



### Inserting without IDs to target

Create a NodeList from the ordered list by using the .querySelectorAll method.

// inserting without IDs to target

// calling query selector all, get all the elements within fruit and

// return all of the list items within it as a NodeList

const boxes = document.**querySelectorAll**("#fruits li")

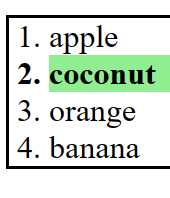
// inserting without IDs to target

// calling query selector all, get all the elements within fruit and

// return all of the list items within it as a NodeList

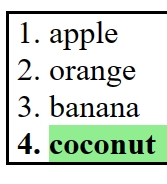
const existingListItem = document.**querySelectorAll**("#fruits li")

document.**getElementById**("fruits").**insertBefore**(newListItem, existingListItem[1]);



Targeting an index out of range places the element at the end of the list.

document.**getElementById**("fruits").**insertBefore**(newListItem, existingListItem[6]);



// remove without an ID

document.**getElementById**("fruits").**removeChild**(newListItem);

## Mouse Event Listeners

Event Listeners listen for specific events to create interactive web pages.

Types of events:

* click
* mouseover
* mouseout

To add an event listener in js, use .addEventListener(event, callback).

// create a reference

const myBox = document.**getElementById**("myBox");

// Adding an event listener

myBox.**addEventListener**(typeOfEvent, Callback);

The typeOfEvent is a string value, and the callback is a function signature (a function that is not being invoked)

// event is automatically provided to us by the browser

function **changeColor**(event) {

    console.**log**(event);

}

// create a reference

const myBox = document.**getElementById**("myBox");

// Adding an event listener

myBox.**addEventListener**("click", **changeColor**);



The data contained within the PointerEvent contains details about what happened.

Access the **.target** property of the event object to specify the DOM Element that was actually clicked on.

// event is automatically provided to us by the browser

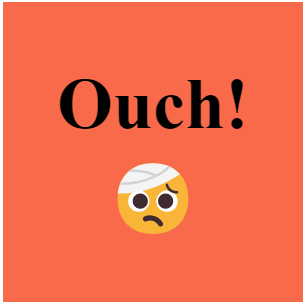
function **changeColor**(event) {

    // console.log(event);

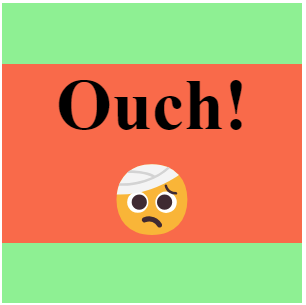
    event.target.style.backgroundColor = "tomato";

    event.target.textContent = "Ouch! 🤕"

}



There is a bit of an issue when doing it this way, clicking on the text content makes that the target which prevents the background color of the div from changing:



This is because we are selecting a target (DOM object which was clicked) rather than the element itself.

    event.target.style.backgroundColor = "tomato";

It has a very easy fix though, target the reference to the parent element instead of the event.target:

    myBox.style.backgroundColor = "tomato";

The same as before but using an anonymous function instead:

// all in one anonymous function

myBox.**addEventListener**("click", function(event){

    event.target.style.backgroundColor = "tomato";

    event.target.textContent = "Ouch! 🤕"

})

The same as above, but using an Arrow Function:

// arrow function, if an arrow func has one parameter, it doesn't need parentheses

myBox.**addEventListener**("click", event => {

    event.target.style.backgroundColor = "tomato";

    event.target.textContent = "Ouch ! 🤕";

})

The final JS looks like:

const myBox = document.**getElementById**("myBox");

// arrow function, if an arrow func has one parameter, it doesn't need parentheses

myBox.**addEventListener**("click", event => {

    event.target.style.backgroundColor = "tomato";

    event.target.textContent = "Ouch ! 🤕";

    // console.log(event);

})

// enter bounds of an element

myBox.**addEventListener**("mouseover", (event) => {

    event.target.style.backgroundColor = "yellow";

    // for some reason, changing the text content here

    // prevents the bug mention earlier from occurring

    event.target.textContent = "Don't do it! 😰";

    // console.log(event);

})

// exit bounds of element

myBox.**addEventListener**("mouseout", event => {

    event.target.style.backgroundColor = "lightgreen";

    event.target.textContent = "Click Me 😃";

})

// use an anonymous func/arrow func as the callback to avoid eslint warning

// or eslint-disable-next-line, or a rule to ignore \_prefixes for unused args

// Same as above, but using a button element instead of event.target

myButton.**addEventListener**("click", \_event => {

    myBox.style.backgroundColor = "tomato";

    myBox.textContent = "Ouch ! 🤕";

})

myButton.**addEventListener**("mouseover", () => {

    myBox.style.backgroundColor = "yellow";

    myBox.textContent = "Don't do it! 😰";

})

myButton.**addEventListener**("mouseout", () => {

   myBox.style.backgroundColor = "lightGreen";

    myBox.target.textContent = "Click Me 😃";

})

These are the settings in .eslintrc.js for this project:

module.exports = {

  env: {

    browser: true,

    es2021: true,

    node: true,

  },

  extends: "eslint:recommended",

  overrides: [

    {

      env: {

        node: true,

      },

      files: [".eslintrc.{js,cjs}"],

      parserOptions: {

        sourceType: "script",

      },

    },

  ],

  parserOptions: {

    ecmaVersion: "latest",

    sourceType: "module",

  },

  rules: {

    // https://stackoverflow.com/a/71384607

    // note you must disable the base rule

    // as it can report incorrect error

    "no-unused-vars": [

      "warn",

      {

        argsIgnorePattern: "^\_",

        varsIgnorePattern: "^\_",

      },

    ],

  },

};

## Key Event Listeners

This section discusses keydown and keyup events.

document.**addEventListener**("keydown", (event) =>{

    console.**log**(event);

})

Looking in the console, we can see a basic overview of the keyEvent in the collapsed view:



and a much more in-depth view in the expanded view. Keys can be identified in a few different ways, as key: 'f', or as code: 'keyf'. It even differentiates between lower case and upper case letters (There are even true/false values for modifier keys like ctrl and shift).

Key: does not differentiate between numpad numbers and Digit, while code: will inform you if the numpad version of a key was used or Digit version. An '@' symbol is registered as a Digit2 keypress.

If you don't need to differentiate between a key that appears multiple times on a keyboard, you can access the .key property.

document.**addEventListener**("keydown", (event) =>{

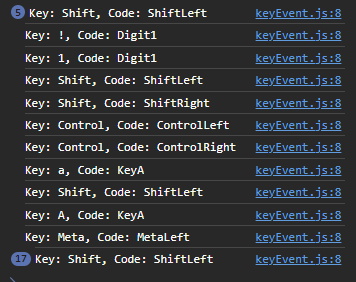
    // console.log(event);

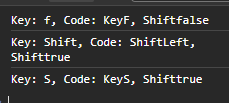
    console.**log**(`Key: ${event.key}, Code: ${event.code}`);

    // console.log(event.code);

})

Here are some comparisons between how the event.key and event.code properties behave, and event.shiftKey returns a value of shifttrue/shiftfalse depending on if it was pressed down or not.





Also, a key pressed down for a prolonged period of time will continue to produce keydown events as shown by the final line displaying the number 17. These are somewhat discrete events, if I hold down shift, then press some other key and press shift again, the counter next to shift will stop incrementing. If I instead press shift, release it and then press it again, it will instead continue to increment that counter. (pretty pointless but might be helpful to know this behaviour for debugging).

// keydown

document.**addEventListener**("keydown", (event) =>{

    // console.log(event);

    //console.log(`Key: ${event.key}, Code: ${event.code}, Shift${event.shiftKey}`);

    // console.log(event.code);

    console.**log**(`Key down = ${event.key}`);

})

// keyup

document.**addEventListener**("keyup", (event) => {

    console.**log**(`Key up = ${event.key}`);

})

## Moving content based on key presses

HTML, very basic, only has an id named myBox with an emoji as its content

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

    <link rel="stylesheet" href="keyEvent.css">

</head>

<body>

    <div id="myBox">😀</div>

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

</body>

</html>

CSS

body {

  margin: 0;

}

#myBox {

  background-color: lightblue;

  width: 200px;

  height: 200px;

  display: flex;

  justify-content: center;

  align-items: center;

  position: relative;

  font-size: 7.5rem;

}



Creates a box around the character, uses flex box to center the element vertically (cross axis)

Javascript:

// a reference to id="myBox"

const myBox = document.**getElementById**("myBox");

document.**addEventListener**("keydown", (\_event) => {

  myBox.textContent = "😲";

  myBox.style.backgroundColor = "tomato";

});

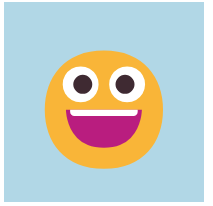
// revert style/content changes

window.**addEventListener**("keyup", (\_event) => {

    myBox.textContent = "😀";

    myBox.style.backgroundColor = "lightblue";

});



Next, we're going to create a function which moves the elements while a key is held down.

// part 2, move the element based on user input

document.**addEventListener**("keydown", (event) => {

    // debug, print .key property to console: ArrowUp etc.

    // console.log(event.key);

    // only do something if the event starts with the string Arrow

    if (event.key.**startsWith**("Arrow")) {

        // using a switch to examine the event

        switch (event.key) {

            case "ArrowUp":

                y -= moveAmount;

                break;

            case "ArrowDown":

                y += moveAmount;

                break;

            case "ArrowLeft":

                x -= moveAmount

                break;

            case "ArrowRight":

                x += moveAmount;

                break;

            default:

                break;

        }

        // debug

        // console.log(`X: ${x}, Y: ${y}`);

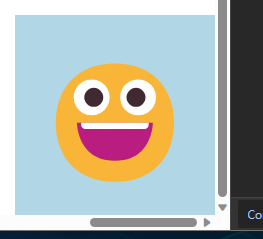
        // don't add semicolon to string literals

        myBox.style.top = `${y}px`;

        myBox.style.left = `${x}px`;

    }

})

One noticeable effect of this program is that once the object goes out of bounds to the bottom or to the right, a scroll bar is added and the arrow keys also begin to pan the window. 

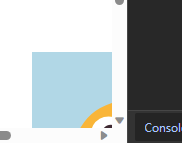
To stop the window from scrolling with the arrow keys, call the .preventDefault() method on the event within the if statement checking for an arrow key press. (This allows other keys to retain their default functionality but will prevent any key press starting with "Arrow" from executing default functionality)

document.**addEventListener**("keydown", (event) => {

    if (event.key.**startsWith**("Arrow")) {

        // stop arrow keys from panning when there's a scroll bar

        event.**preventDefault**()

 Still doesn't prevent the element from going off screen.

# Hide and show Elements in HTML

HTML:

<body>

    <button id="myButton">Hide</button>

    <img id="fractal" src="..\apps\Assets\clock\Waves Centered 1080 40000.png" alt="fractal" >

    <script src="toggle.js"></script>

CSS:

#fractal {

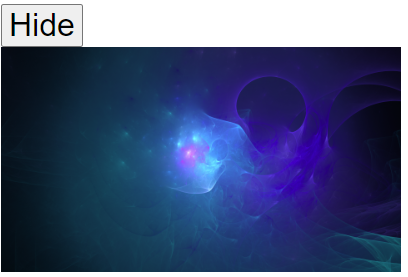
    width: 400px;

}

#myButton {

    font-size: 2rem;

}



\* Button and Image have been inverted, this creates a situation where the button can be displaced by presence of the image.

    <img id="fractal" src="..\apps\Assets\clock\Waves Centered 1080 40000.png" alt="fractal" >

    <button id="myButton">Hide</button>

myButton.**addEventListener**("click", (\_event) => {

    console.**log**(fractal.style);

    // use an if statement to toggle states

    if (fractal.style.display === "none") {

        // unhide image with display: block

        fractal.style.display = "block";

        myButton.textContent = "Hide";

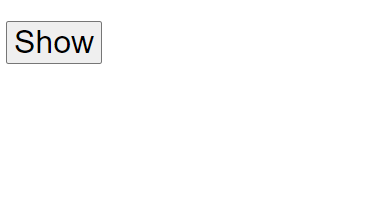
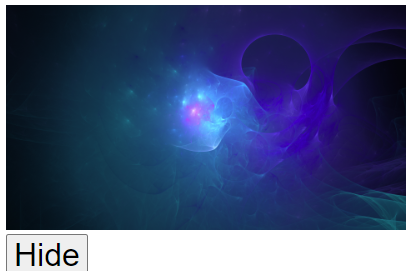
    } else {

        fractal.style.display = "none";

        myButton.textContent = "Show";

    }

})



Setting style.display = "none" removes the element from the document flow. This causes the "show" button to be displaced, instead you should target the .visible property so that the image can be hidden without removing it from the flow of the document:

// tie events to myButton

myButton.**addEventListener**("click", (\_event) => {

    console.**log**(fractal.style);

    // use an if statement to toggle states

    if (fractal.style.visibility === "hidden") {

        // unhide image with display: block

        fractal.style.visibility = "visible";

        myButton.textContent = "Hide";

    } else {

        fractal.style.visibility = "hidden";

        myButton.textContent = "Show";

    }

})

# NodeLists

A NodeList is a static collection of HTML elements by (id, class, element). It can be created by using the querySelectorAll(). Similar to an array, but no map, filter or reduce. NodeList won't update automatically to reflect changes.

// create a NodeList using querySelectorAll()

// . = class, # = id, tags seem to work too

let buttons = document.**querySelectorAll**(".myButtons");

console.**log**(buttons);

// create a NodeList using querySelectorAll()

// . = class, # = id, tags seem to work too

let buttons = document.**querySelectorAll**(".myButtons");

// console.log(buttons);

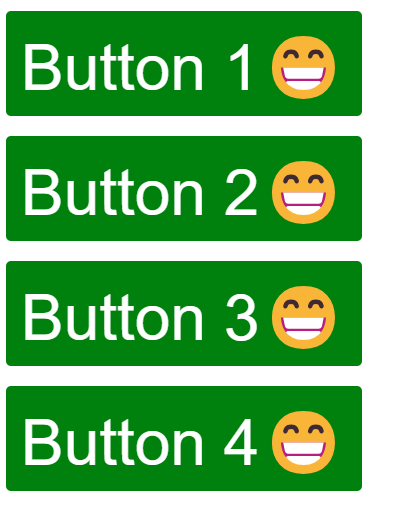
// Add HTML/CSS Properties by using forEach on a NodeList

buttons.**forEach**((button) => {

    button.style.backgroundColor = "green";

    button.textContent += "😁";

})



/\* Click Event Listener

    .target makes it so that only the clicked element changes

    \*/

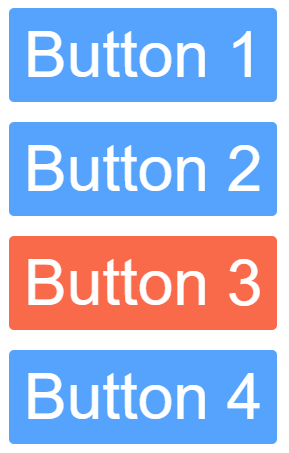
    buttons.**forEach**((button) => {

        button.**addEventListener**("click", (event) => {

            event.target.style.backgroundColor = "tomato";

        })

    })



/\* mouseover and mouseout

\*/

buttons.**forEach**(button => {

    button.**addEventListener**("mouseover", \_event => {

        button.style.backgroundColor = "blue";

    })

})

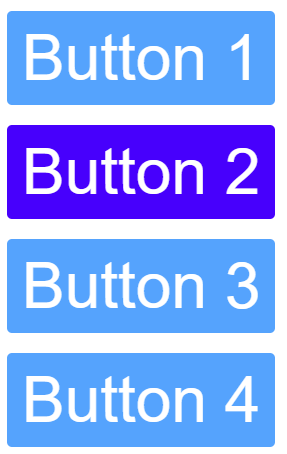
buttons.**forEach**(button => {

    button.**addEventListener**("mouseout", \_event => {

        button.style.backgroundColor = "hsl(205, 100%, 60%)";

    })

})



### Add an element:

/\* Add an element

\*/

// Step 1, create the element

const newButton = document.**createElement**("button");

// Step 2, set content/properties

newButton.textContent = "Button 5";

// classList works with the existing list of classes

// className clears existing classes and starts fresh

newButton.classList = "myButtons";

// Step 3, add element to the DOM

// access the parent

document.body.**appendChild**(newButton);

### Update Node List

To update the Node list, all we have to do is call document.querySelectorAll() again to fetch a new node list.

// is static collection, only shows 4 buttons

console.**log**(buttons);

// call document.querySelectorAll again to update the NodeList

// Now contains 5 nodes

buttons = document.**querySelectorAll**(".myButtons");

console.**log**(buttons);

### Remove an element from the DOM and update Node List

/\* remove an element from the DOM

\*/

buttons.**forEach**(button => {

    button.**addEventListener**("click", (event) => {

        event.target.**remove**();

        buttons = document.**querySelectorAll**(".myButtons");

        // console.log(buttons);

    })

})



## classLists

ClassList = Element property in Javascript used to interact with an element's list of classes (CSS Classes). They allow you to make reusable classes for many elements across your webpage.

Methods for class lists:

add(), remove(), toggle(remove if present, add if not), replace(oldClass, newClass), contains()

HTML:

<body>

    <button id="myButton">My Button</button>

    <script src="index.js"></script>

</body>

CSS:

#myButton {

    font-size: 4rem;

    margin: 10px;

    border: none;

    border-radius: 5px;

    padding: 10px 15px;

}

/\* class to be applied through JS \*/

.enabled {

    background-color: hsl(240, 100%, 50%);

    color: white;

}

Javascript will add the .enabled class to the element.

const myButton = document.**getElementById**("myButton");

// add a class an element using the .add() method

myButton.classList.**add**("enabled");



Viewing the html, we can see that the enabled class has now been applied to the element.

### Remove a class with .remove() method

// remove a class

myButton.classList.**remove**("enabled");

 (enabled was the only existing method on the class)

You can see that the class no longer has the enabled class after calling the remove method.

### Dynamically add and remove a class on hover

// mouseover => add

myButton.**addEventListener**("mouseover", event => {

    event.target.classList.**add**("hover");

})

// mouseout => remove

myButton.**addEventListener**("mouseout", event => {

    event.target.classList.**remove**("hover");

})

### Toggle

myButton.**addEventListener**("mouseover", event => {

    event.target.classList.**toggle**("hover");

})

// mouseout => remove

myButton.**addEventListener**("mouseout", event => {

    event.target.classList.**toggle**("hover");

})

Works the same, but can also be used to create a "latching" effect if one of the event listeners are removed.

### Replace()

myButton.classList.**add**("enabled"); // set to enabled

// change enabled to disabled after click

myButton.**addEventListener**("click", event => {

    event.target.classList.**replace**("enabled", "disabled");

    console.**log**(event.target.classList);

})

### contains()

myButton.classList.**add**("enabled");

myButton.**addEventListener**("click", event => {

    if (event.target.classList.**contains**("disabled")) {

        event.target.textContent +="💢";

    }

    else {

        event.target.classList.**replace**("enabled", "disabled");

    }

})

## Reusable Code

We can append these classes to more elements if we want to reuse them, we aren't forced to start styling from scratch.

// reusable classes

myH1.classList.**add**("enabled");

myH1.**addEventListener**("click", event => {

    if (event.target.classList.**contains**("disabled")) {

        event.target.textContent +="💢";

    }

    else {

        event.target.classList.**replace**("enabled", "disabled");

    }

})

## Example 2, working with a NodeList:

    <button class="myButtons">Button 1</button>

    <button class="myButtons">Button 2</button>

    <button class="myButtons">Button 3</button>

    <button class="myButtons">Button 4</button>

.myButtons {

    font-size: 4rem;

    margin: 10px;

    border: none;

    border-radius: 5px;

    padding: 10px 15px;

}

.enabled {

    background-color: hsl(204, 100%, 50%, 1);

    color: white;

}

.hover {

    box-sizing: 0 0 10px hsl(0, 0%, 0%, 0.2);

    font-weight: bold;

}

.disabled {

    background-color: hsl(0, 0%, 60%, 1);

    color: hsl(0, 0%, 80%, 1);

}

Javascript

const buttons = document.**querySelectorAll**(".myButtons");

// check that the button elements are selected

// console.log(buttons);

buttons.**forEach**(button => {

    button.classList.**add**("enabled");

})

// add a class on mouseover

buttons.**forEach**(button => {

    button.**addEventListener**("mouseover", event => {

        event.target.classList.**add**("hover");

    })

})

// remove a class

buttons.**forEach**(button => {

    button.**addEventListener**("mouseout", event => {

        event.target.classList.**remove**("hover");

    })

})

// replace a class

buttons.**forEach**(button => {

    button.**addEventListener**("click", event => {

        // if classList is already disabled

        if (event.target.classList.**contains**("disabled")) {

            event.target.textContent += "🤬";

        }

        else {

            event.target.classList.**replace**("enabled", "disabled");

        }

    })

})



# Rock Paper Scissors game

HTML:

    <h1>Rock - Paper - Scissors</h1>

    <!-- on click call playGame func and pass string as arg -->

    <div id="choices">

        <button id="rock" onclick="**playGame**('rock')">✊ </button>

        <button id="paper" onclick="**playGame**('paper')">✋ </button>

        <button id="scissors" onclick="**playGame**('scissors')">✌️</button>

    </div>

    <div id="playerDisplay">Player: </div>

    <div id="computerDisplay">Computer: </div>

    <!-- placeholder -->

    <div id="resultDisplay">It's a Tie!</div>

CSS:

body {

    font-family: Arial, Helvetica, sans-serif;

    font-weight: bold;

    margin: 0;

    display: flex;

    flex-direction: column;

    /\* align-items:center left justifies content once width is too low \*/

    /\* align-items: center; \*/

    text-align: center;

}

h1 {

    font-size: 3.5rem;

    color: hsl(0, 0%, 20%, 1);

}

.choices {

    margin-bottom: 3px;

}

.choices button {

    font-size: 7.5rem;

    min-width: 160px;

    margin: 0px 10px;

    border-radius: 200px;

    background-color: hsl(200, 100%, 50%, 1);

    cursor: pointer;

    transition: background-color 0.5s ease;

}

.choices button:hover {

    background-color: hsl(200, 100%, 70%, 1);

}

#playerDisplay, #computerDisplay {

    font-size: 2.5rem;

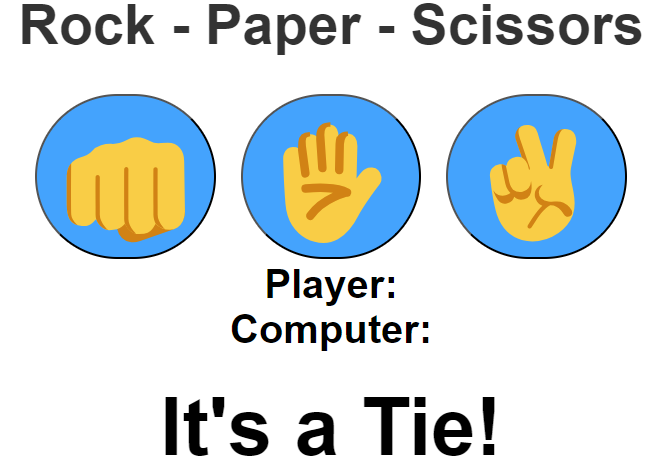
}

#resultDisplay {

    font-size: 5rem;

    margin: 30px 0;

}



JS:

// Rock Paper Scissors

const choices = ["rock", "paper", "scissors"];

// references to player, computer, and results

const playerDisplay = document.**getElementById**("playerDisplay");

const computerDisplay = document.**getElementById**("computerDisplay");

const resultsDisplay = document.**getElementById**("resultDisplay");

// playGame function, handles string input of rock, paper or scissors

function **playGame**(playerChoice) {

  // select a random element from choices array

  const computerChoice = choices[Math.**trunc**(Math.**random**() \* 3)];

  // console.log(computerChoice); // debug

  console.**log**(playerChoice);

  let result = "";

  if (playerChoice === computerChoice) {

    result = "IT'S A TIE!";

  } else {

    switch (playerChoice) {

      case "rock":

        result = computerChoice === "scissors" ? "YOU WIN!" : "YOU LOSE!";

        break;

      case "paper":

        result = computerChoice === "rock" ? "YOU WIN!" : "YOU LOSE!";

        break;

      case "scissors":

        result = computerChoice === "paper" ? "YOU WIN!" : "YOU LOSE!";

        break;

    }

  }

  playerDisplay.textContent = `Player: ${playerChoice}`;

  computerDisplay.textContent = `Computer: ${computerChoice}`;

  resultsDisplay.textContent = result;

}

That's the basics of the program, next, add some extra features like styling elements based on winning/losing the game.

.greenText {

    color: hsl(120, 79%, 62%);

}

.redText {

    color: #ff6347;

}

## Image Slider

HTML

    <div class="slider">

        <!-- image container -->

        <div class="slides">

            <img alt="image1.jpg" class="slide" src="..\Assets\Fractals\red star 583.png">

            <img alt="image2.jpg" class="slide" src="..\Assets\Fractals\green star 109.png">

            <img alt="image3.jpg" class="slide" src="..\Assets\Fractals\blue star 116.png">

        </div>

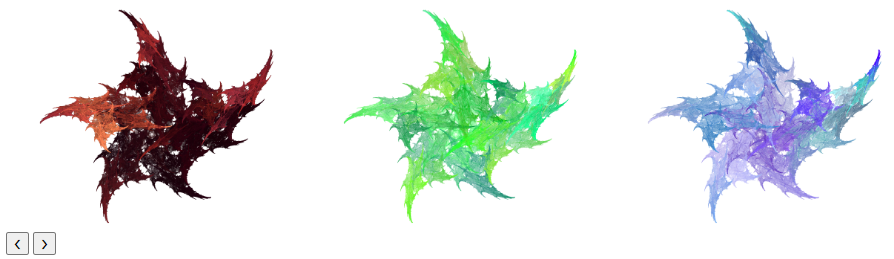
        <button class="prev" onclick="**prevSlide**()">&#10094;</button>

        <button class="next" onclick="**nextSlide**()">&#10095;</button>

    </div>

    <script src="index.js"></script>

</body>



### Optional: Wait for DOM content to load before running certain sections of code

To do this, add an eventListener to the DOM in the .js file. Give it the "DOMContentLoaded" string as the event and provide it a callback function for whatever you want to run only after the DOM has loaded.

document.**addEventListener**("DOMContentLoaded", **initializeSlider**);

This is the final HTML:

    <div class="slider">

        <!-- image container -->

        <div class="slides">

            <img alt="image1.jpg" class="slide correction" src="..\Assets\Fractals\red star 583.png">

            <img alt="image2.jpg" class="slide" src="..\Assets\Fractals\green star 109.png">

            <img alt="image3.jpg" class="slide" src="..\Assets\Fractals\blue star 116.png">

        </div>

        <button class="prev" onclick="**prevSlide**()">&#10094;</button>

        <button class="next" onclick="**nextSlide**()">&#10095;</button>

    </div>

CSS:

.slider {

  /\* Elements will move relative to their normal position \*/

  position: relative;

  margin: auto;

  overflow: hidden;

  width: 60%;

}

.slider img {

  width: 100%;

  /\* width: 40%; \*/

  display: none;

  background-color: black;

  border-radius: 20px;

}

img.displaySlide {

  display: block;

  /\* calling our fade animation, causes the white fade in \*/

  animation-name: fade;

  animation-duration: 1.5s;

}

.correction {

  filter: brightness(1.75);

}

.slider button {

  /\* positions an element relative to its parent (.slider element)\*/

  position: absolute;

  /\* moves top of button down 50% of parent's height \*/

  top: 50%;

  /\* translateY up 50% of the button's height to reach the center of the parent container \*/

  transform: translateY(-50%);

  font-size: 2rem;

  /\* remove bit of padding above text character \*/

  line-height: 2rem;

  background: hsl(0, 0%, 100%, 0.2);

  color: white;

  border: none;

  cursor: pointer;

  padding: 10px 8px;

  vertical-align: baseline;

}

.prev {

  left: 0;

}

.next {

  right: 0;

  color: black;

  background-color: black;

}

/\* animation is named fade, need to call it \*/

@keyframes fade {

  from {

    opacity: 0.5;

  }

  to {

    opacity: 1;

  }

}

Javascript:

// create a NodeList of all images within the slide class

// this selects the img elements inside a parent container with a class of .slides

const slides = document.**querySelectorAll**(".slides img");

// console.log(slides[0]); // debug

let slideIndex = 0;

let intervalId = null;

// optional, use an eventListener which prevents js from running

// until DOM has loaded

// initializeSlider();

document.**addEventListener**("DOMContentLoaded", **initializeSlider**);

// call this func to populate browser with first img

function **initializeSlider**() {

  // check that the slides NodeList is not empty, if it has elements

  // run code

  if (slides.length > 0) {

    slides[slideIndex].classList.**add**("displaySlide");

    // setInterval returns a process Id to identify it, we then

    // store it in intervalId

    // every 5s, the function calls the nextSlide() function

    intervalId = **setInterval**(**nextSlide**, 5000);

    console.**log**(intervalId);

  } else {

    console.**log**("check querySelectorAll() statement");

  }

}

// index of the next slide to preview

function **showSlide**(index) {

  // check if slide index is out of bounds

  if (index >= slides.length) {

    slideIndex = 0;

    // if someone clicks prev on slide[0], wrap to slide on the end

  } else if (index < 0) {

    slideIndex = slides.length - 1;

  }

  slides.**forEach**((slide) => {

    // strip the displaySlide class so that the slide can be swapped

    slide.classList.**remove**("displaySlide");

  });

  // add displaySlide to the new element we want to display

  slides[slideIndex].classList.**add**("displaySlide");

}

/\* eslint-disable-next-line \*/

function **prevSlide**() {

  // interrupt auto cycle so that it gets full 5s loop

**clearInterval**(intervalId);

  slideIndex--;

**showSlide**(slideIndex);

  // resume auto cycle

  intervalId = **setInterval**(**nextSlide**, 5000);

}

function **nextSlide**() {

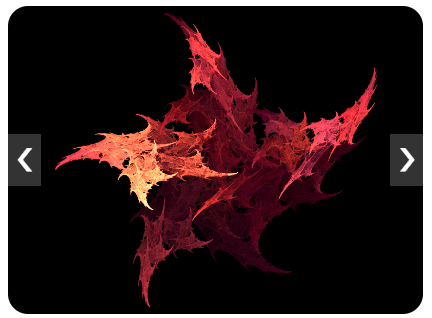
**clearInterval**(intervalId);

  slideIndex++;

**showSlide**(slideIndex);

  intervalId = **setInterval**(**nextSlide**, 5000);

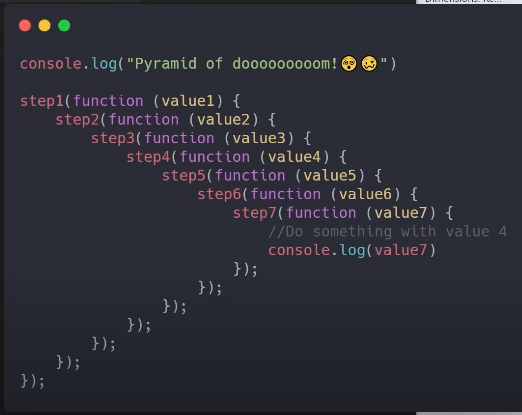
}



pagination

## Callback Hell

Situation in javascript where callbacks are nested within other callbacks to the point where the code becomes difficult to read. Old pattern to handle asynchronous functions. Uses Promises + async/await to avoid callback hell.



Very difficult to read.

// synchronous funcs

function **task1**() {

  console.**log**("Task 1 complete");

}

function **task2**() {

  console.**log**("Task 2 complete");

}

function **task3**() {

  console.**log**("Task 3 complete");

}

function **task4**() {

  console.**log**("Task 4 complete");

}

**task1**();

**task2**();

**task3**();

**task4**();

console.**log**("All tasks complete");

**$ node index.js**

**Task 1 complete**

**Task 2 complete**

**Task 3 complete**

**Task 4 complete**

**All tasks complete**

**// asynchronous funcs**

function **task1**() {

**setTimeout**(() => {

    console.**log**("Task 1 complete");

  }, 2000);

}

function **task2**() {

**setTimeout**(() => {

    console.**log**("Task 2 complete");

  }, 1000);

}

function **task3**() {

**setTimeout**(() => {

    console.**log**("Task 3 complete");

  }, 3000);

}

function **task4**() {

**setTimeout**(() => {

    console.**log**("Task 4 complete");

  }, 1500);

}

**task1**();

**task2**();

**task3**();

**task4**();

console.**log**("All tasks complete");

**$ node index.js**

**All tasks complete**

**Task 2 complete**

**Task 4 complete**

**Task 1 complete**

**Task 3 complete**

Asynchronous code can complete at any time, nothing halts execution. If it is important for tasks to be completed in order, make use of Async/Await.

// callback hell

**task1**(() => {

**task2**(() => {

**task3**(() => {

**task4**(() => {

        console.**log**("All tasks complete");

      });

    });

  });

});

$ node index.js

Task 1 complete

Task 2 complete

Task 3 complete

Task 4 complete

All tasks complete

The code now executes sequentially, but it is very hard to work with.

Four levels generally isn't too bad, but it can get much worse to the point the code becomes very difficult to read and unmanageable.

# Promises

/\* Promise

    An object that manages asynchronous operations.

    Wrap a Promise Object around {asynchronous code}

    "I promise to return a value"

    PENDING -> RESOLVED or REJECTED

    new Promise ((resolve, reject) => {asynchronous code})

    DO THESE CHORES IN ORDER

    1. walk the dog

    2. clean the kitchen

    3. take out the trash

\*/

// without Promises to handle asynchronous code

// relying on callbacks to keep code synchronous

function **walkDog**(callback) {

**setTimeout**(() => {

    console.**log**("You walk the dog");

**callback**();

  }, 1500);

}

function **cleanKitchen**(callback) {

**setTimeout**(() => {

    console.**log**("You clean the kitchen");

**callback**();

  }, 2500);

}

function **takeOutTrash**(callback) {

**setTimeout**(() => {

    console.**log**("You take out the trash");

**callback**();

  }, 500);

}

**walkDog**(() => {

**cleanKitchen**(() => {

**takeOutTrash**(() => {

      console.**log**("You finished all the chores");

    });

  });

});

By wrapping the code in a Promise instead, we don't need to use callbacks. Instead of callbacks, we can use method chaining. To do this, we need to modify our functions to return a Promise object instead.

Here is a comparison between a function making use of a Promise and one that uses a callback:

// return a Promise Object

function **walkDog**() {

  return new Promise((**resolve**, **reject**) => {

    // Asynchronous Code goes inside the Promise Obj

**setTimeout**(() => {

**resolve**("You walk the dog");

    }, 1500);

  });

}

function **cleanKitchen**(callback) {

**setTimeout**(() => {

    console.**log**("You clean the kitchen");

**callback**();

  }, 2500);

}

The Callback parameter is no longer present. We also call resolve("some value") instead of just executing some code on completion.

function **walkDog**() {

  return new Promise((**resolve**, **reject**) => {

    // Asynchronous Code goes inside the Promise Obj

**setTimeout**(() => {

**resolve**("You walk the dog");

    }, 1500);

  });

}

**walkDog**().**then**((value) => console.**log**(value)); // you walk the dog

Somewhat similar to Rust's option

// method chain instead

// resolve message stored passed to .then()

**walkDog**().**then**(value => {console.**log**(value); return **cleanKitchen**()})

         .**then**(value => {console.**log**(value); return **takeOutTrash**()})

         .**then**(value => {console.**log**(value); console.**log**("Tasks complete");})

Sometimes with asynchronous code, the task will fail (such as locating a file). If you can't locate that file, you do not want to resolve that promise.

Instead, you want to make use of the reject property.

### Before Reject

function **walkDog**() {

  return new Promise((**resolve**, **reject**) => {

**setTimeout**(() => {

**resolve**("You walk the dog");

    }, 1500);

  });

}

### After Reject

function **walkDog**() {

  return new Promise((**resolve**, **reject**) => {

**setTimeout**(() => {

      const dogWalked = true;

      if (dogWalked) {

**resolve**("You walk the dog");

      } else {

**reject**("You didn't walk the dog");

      }

    }, 1500);

  });

}

Here is the full program:

// return a Promise Object

function **walkDog**() {

  return new Promise((**resolve**, **reject**) => {

**setTimeout**(() => {

      const dogWalked = true;

      if (dogWalked) {

**resolve**("You walk the dog");

      } else {

**reject**("You didn't walk the dog");

      }

    }, 1500);

  });

}

function **cleanKitchen**() {

  return new Promise((**resolve**, **reject**) => {

**setTimeout**(() => {

      const kitchenCleaned = false;

      if (kitchenCleaned) {

**resolve**("You clean the kitchen");

      } else {

**reject**("You did not clean the kitchen");

      }

    }, 2500);

  });

}

function **takeOutTrash**() {

  return new Promise((**resolve**, **reject**) => {

**setTimeout**(() => {

      const trashRemoved = true;

      if (trashRemoved) {

**resolve**("You take out the trash");

      } else {

**reject**("You did not take out the trash");

      }

    }, 500);

  });

}

// method chain instead

// resolve message stored passed to .then()

// resolve message stored passed to .then()

**walkDog**()

.**then**((value) => {

    console.**log**(value);

    return **cleanKitchen**();

})

.**then**((value) => {

    console.**log**(value);

    return **takeOutTrash**();

})

.**then**((value) => {

    console.**log**(value);

    console.**log**("Tasks complete");

})

// add a catch to the end of the chain

.**catch**((error) => console.**error**(error));

A catch block must be added to the end which handles errors in the case that a rejection occurs. The program will exit where it encountered the error (this is due to how we caught the error, handling this a different way may allow a program to continue, but generally it is preferred to exit gracefully.

It's a bit like some extra syntax for a try catch block.

# Async/Await

Async/Await

-Async = makes a function return a promise.

-Await = makes an async function wait for a promise.

Allows you to write asynchronous code in a synchronous manner.

Async doesn't have resolve or reject parameters.

Everything after Await is placed in an event queue.

Async/Await removes the need to method chain .then() statements, everything must be contained within an Async Function.

Await will result in a syntax error if it is not used within an async function.

### Handling uncaught reject requires a try block:

// return a Promise Object

function **walkDog**() {

  return new Promise((**resolve**, **reject**) => {

**setTimeout**(() => {

      // false to cause reject

      const dogWalked = false;

      if (dogWalked) {

**resolve**("You walk the dog");

      } else {

**reject**("You didn't walk the dog");

      }

    }, 1500);

  });

}

$ node index.js

node:internal/process/promises:289

triggerUncaughtException(err, true /\* fromPromise \*/);

^

[UnhandledPromiseRejection: This error originated either by throwing inside of an async function witho

ut a catch block, or by rejecting a promise which was not handled with .catch(). The promise rejected

with the reason "You didn't walk the dog".] {

code: 'ERR\_UNHANDLED\_REJECTION'

}

Place all of your function calls within the async function within a try block, then catch any resulting errors.

// code contained within an async function

async function **doChores**() {

    // code promise may return reject

    try {

        const walkDogResult = await **walkDog**();

        console.**log**(walkDogResult);

        const cleanKitchenResult = await **cleanKitchen**();

        console.**log**(cleanKitchenResult);

        const takeOutTrashResult = await **takeOutTrash**();

        console.**log**(takeOutTrashResult);

        console.**log**("you finished all the chores");

    } catch (error) {

        console.**error**(error);

    }

}

**doChores**();

Async is somewhat like a main() function which forces code to be run top to bottom. Requires the helper functions to be called after the keyword Await. Place the code to be run in a try catch block and handle errors in the catch portion of the code.

# Files/JSON

JavaScript Object Notation, is a data-interchange format used for exchanging data between a server and a web application. JSON files use a {key:value} or [val1, val2, val3] format.

JSON.stringify() converts a JS object to a JSON string.

JSON.parse() converts a JSON string into a JS object.

Here are some sample json files:

names.json:

["Spongebob", "Patrick", "Sandy", "Squidward"]

person.json:

{

    "name": "Spongebob",

    "age" : 30,

    "isEmployed" : true,

    "hobbies" : ["Jellyfishing", "Karate", "cooking"]

}

people.json:

[{

    "name": "Spongebob",

    "age" : 30,

    "isEmployed": true

},

{

    "name": "Patrick",

    "age" : 35,

    "isEmployed": false

},

{

    "name": "Squidward",

    "age" : 40,

    "isEmployed": true

},

{

    "name": "Sandy",

    "age" : 27,

    "isEmployed": false

}]

Comments are not valid JSON, which can make leaving notes a little annoying.

The first JSON file is an array of strings, second is a "dictionary", and the third is an array of "dictionaries".

JSON formats are one long string to represent an object or an array. Using the json.stringify() method, we can convert a js object or an array into a JSON string.

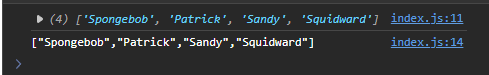
const names = ["Spongebob", "Patrick", "Sandy", "Squidward"];

console.**log**(names);

const jsonString = JSON.**stringify**(names);

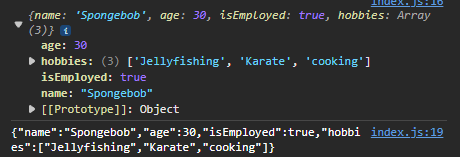
console.**log**(jsonString); // ["Spongebob","Patrick","Sandy","Squidward"]

These have two different representations in the browser console:

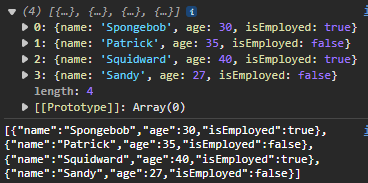


The first is an actual javascript array while the second is a string representation of the array.

using the data from person.json



More complex data from people.json:



// fetch a file/url, returns a promise

**fetch**("./names.json")

.**then**(response => response.**json**())

.**then**(value => console.**log**(value));

**fetch**("./person.json")

.**then**(response => response.**json**())

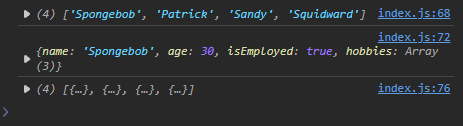
.**then**(value => console.**log**(value));

**fetch**("./people.json")

.**then**(response => response.**json**())

.**then**(value => console.**log**(value));

Using the then() method allows our program to behave synchronously rather than the program moving onto the next piece of code before the previous fetch operation completed.



If you would like to iterate through the array of objects, it's very easy to add in a forEach loop to iterate over the elements.

**fetch**("./people.json")

.**then**(response => response.**json**())

.**then**(values => values.**forEach**(value => {

    console.**log**(value);

}));

If there are any specific values you would like to access, use dot notation.

**fetch**("./people.json")

.**then**(response => response.**json**())

.**then**(values => values.**forEach**(value => {

    console.**log**(value.**name**);

}));



Don't forget to add a catch method to catch any errors that might occur, the fetch method returns a Promise.

  // there is an implicit try catch block

  // try is done in the background, but you need to method

  // chain a catch statement to handle errors which may occur

**fetch**("./people.json")

  .**then**((response) => response.**json**())

  .**then**((values) =>

    values.**forEach**((value) => {

      console.**log**(value.isEmployed).**catch**((error) => console.**error**(error));

    })

  );

[https://javascript.info/promise-error-handling](https://javascript.info/promise-error-handling#:~:text=Implicit%20try%E2%80%A6&text=The%20code%20of%20a%20promise,and%20treated%20as%20a%20rejection.)

In the background, a Promise incorporates a try catch block. The try encompasses anything you tell the code to do up to the .catch() method.

# Fetch Data from an API

fetch

    A function used for making HTTP requests to fetch resources.

    (JSON style data, images, files)

    Simplifies asynchronous data fetching in Javascript and

    used for interacting with APIS to retrieve and send data

    asynchronously over the web.

    fetch(url, {options})

The API that this project will use is the one hosted at pokeapi.co. It is a RESTful API.

Here's an example URL for Pikachu: <https://pokeapi.co/api/v2/pokemon/pickachu>.

**fetch**("https://pokeapi.co/api/v2/pokemon/pickachu")

  .**then**(/\* do something \*/)

  .**catch**((error) => console.**error**(error));

The fetch function is promise based, it will either resolve or reject, so you should method chain a catch clause onto the end as shown.

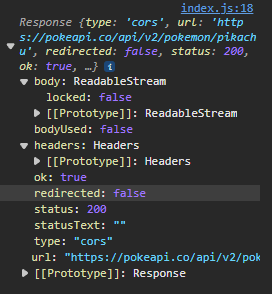
// https://pokeapi.co/ is the api that will be used

**fetch**("https://pokeapi.co/api/v2/pokemon/pikachu")

  .**then**(resolve => console.**log**(resolve))

  .**catch**((error) => console.**error**(error));

This prints the following data to the terminal:



[Symbol(realm)]: null,

[Symbol(state)]: {

aborted: false,

rangeRequested: false,

timingAllowPassed: true,

requestIncludesCredentials: true,

type: 'default',

status: **200,**

timingInfo: {

startTime: 57.996399998664856,

redirectStartTime: 0,

redirectEndTime: 0,

postRedirectStartTime: 57.996399998664856,

finalServiceWorkerStartTime: 0,

finalNetworkResponseStartTime: 0,

finalNetworkRequestStartTime: 0,

endTime: 0,

encodedBodySize: 59,

decodedBodySize: 0,

finalConnectionTimingInfo: null

},

cacheState: '',

statusText: 'OK',

headersList: HeadersList {

cookies: null,

[Symbol(headers map)]: [Map],

[Symbol(headers map sorted)]: null

},

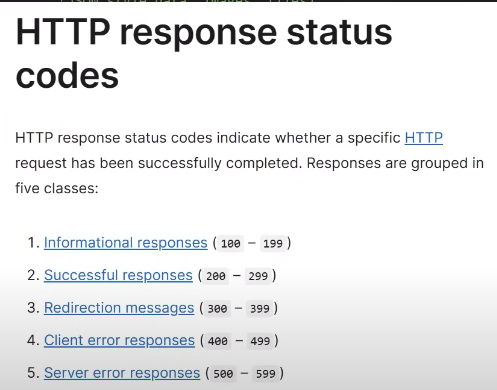
urlList: [ URL {} ],

body: { stream: undefined }

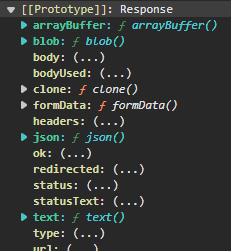
},

Note the status: 200 and the ok: true value in the screenshot. This means that it was a successful response.

Other HTTP response status codes:



As it is currently, this is not a readable format for us. There are a couple of methods to handle this. Array Buffer, Blob, Text and JSON. Look in the [[Prototype]] section for methods that can be used on this API object.



For this example, we're interested in the JSON method.

The next step is to take the response object and convert it to a JSON object.

**fetch**("https://pokeapi.co/api/v2/pokemon/pikachu")

  .**then**(response => response.**json**())

The .json() method is also promise based, chain a .then() method call to work with the data after it resolves.

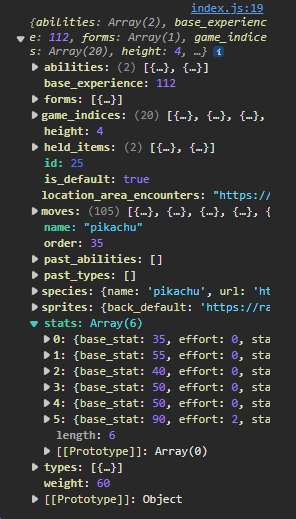
// https://pokeapi.co/ is the api that will be used

**fetch**("https://pokeapi.co/api/v2/pokemon/pikachu")

  .**then**(response => response.**json**())

  .**then**(data => console.**log**(data))

  .**catch**((error) => console.**error**(error));



We can see the structure of the returned object, there's it's id: 25, name: "Pikachu", some stats, its weight, etc.

Use dot notation to access a specific property.

**fetch**("https://pokeapi.co/api/v2/pokemon/pikachu")

  .**then**(response => response.**json**())

  .**then**(data => console.**log**(data.name))

  .**catch**((error) => console.**error**(error));



Handling an error:

// example of a url which will return a 404

**fetch**("https://pokeapi.co/api/v2/pokemon/spongebob")

// https://pokeapi.co/ is the api that will be used

// fetch("https://pokeapi.co/api/v2/pokemon/pikachu")

  .**then**((response) => {

    // if the response is not ok, handle the error

    if (!response.ok) {

      throw new **Error**("Sorry, we could not fetch resource");

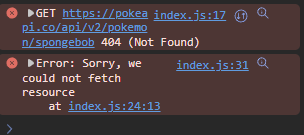
    }

    return response.**json**();

  })

  .**then**((data) => console.**log**(data.name))

  .**catch**((error) => console.**error**(error));



The if statement triggers, and the code jumps to the .catch() statement which then prints the error to the console.

Changing the url back to Pikachu shows us that the program is able to receive a response correctly and printing data.id gives us the .id property of the object which is 25.

Remember that calling the .json() method returns a Promise which either resolves or rejects depending on if it is successful. Within an arrow function, create a named parameter to handle the resolve case and chain the catch method to handle a rejection.

This fetch request can also be rewritten to make use of async/await instead.

First, create an async function. It will take 0 parameters and place a try catch block within it. Catch will be a simple console.error(error);

// Async/Await

async function **fetchData**() {

  try {

    // await the Promise returned by fetch("");

    const response = await **fetch**(

      "https://pokeapi.co/api/v2/pokemon/typhlosion"

    );

    // check if response (ok: true/false) resolves or 404s

    if (!response.ok) {

      throw new **Error**("Sorry, could not fetch resource");

    }

    // if it is ok: true

    // json returns a promise, which is why we use the await keyword

    const data = await response.**json**();

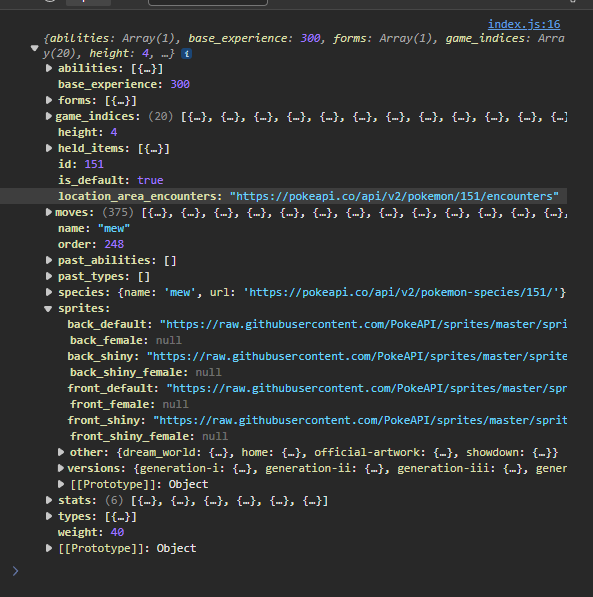
    console.**log**(data);

  } catch (error) {

    console.**error**(error);

  }

}



We want to fetch the front\_default sprite for our web page.

HTML:

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

    <link rel="stylesheet" href="style.css">

</head>

<body>

    <input type="text" id="pokemonName" placeholder="Enter Pokemon name">

    <button onclick="**fetchData**()">Fetch Pokemon</button><br>

    <!-- probably better to create an image element in js and append it -->

    <img src="" alt="Pokemon Sprite" id="pokemonSprite" style="display: none">

    <script src="index.js"></script>

</body>

</html>

Javascript:

async function **fetchData**() {

  try {

    const pokemonName = document

      .**getElementById**("pokemonName")

      .value.**toLowerCase**();

    const response = await **fetch**(

      `https://pokeapi.co/api/v2/pokemon/${pokemonName}`

    );

    if (!response.ok) {

      throw new **Error**("Could not fetch resource");

    }

    const data = await response.**json**();

    // console.log(data);

    const pokemonSprite = data.sprites.front\_default;

    // console.log(data.sprites.front\_default);

    const imgElement = document.**getElementById**("pokemonSprite");

    imgElement.src = pokemonSprite;

    imgElement.style.display = "block";

  } catch (error) {

    console.**error**(error);

  }

}

# Creating a weather app

This project makes use of OpenWeather's API. Do not share your API key. It looks like the API key can be deactivated from the home.openweathermap.org/api\_keys address once we are finished with it.

Not sure about loading an API key from a file, seems to result in an async await block which can't return results and store it outside of that scope.

Here's the HTML for the project with some placeholder elements to create a card:

    <form action="" class="weatherForm">

        <input type="text" class="cityInput" placeholder="Enter City">

        <button type="submit">Get Weather</button>

    </form>

    <!-- display data as a card -->

    <div class="card">

        <h1 class="cityDisplay">Vancouver</h1>

        <p class="tempDisplay">0°C</p>

        <p class="humidityDisplay">Humidity: 75%</p>

        <p class="descDisplay">Cloudy</p>

        <p class="weatherEmoji">☁️</p>

        <p class="errorDisplay">Please enter a City</p>

    </div>

CSS:

body {

  font-family: Arial, Helvetica, sans-serif;

  background-color: hsl(0, 0%, 95%, 1);

  margin: 0;

  display: flex;

  flex-direction: column;

  align-items: center;

}

.weatherForm {

  margin: 20px;

}

.cityInput {

  padding: 10px;

  font-size: 2rem;

  font-weight: bold;

  border: 2px solid hsl(0, 0%, 20%, 0.3);

  border-radius: 10px;

  margin: 10px;

  width: 300px;

}

/\* buttons [property = "value"] selector \*/

button[type="submit"] {

  padding: 10px 20px;

  font-weight: bold;

  font-size: 2rem;

  background-color: hsl(122, 39%, 50%, 1);

  color: white;

  border: none;

  border-radius: 5px;

  cursor: pointer;

}

button[type="submit"]:hover {

  background-color: hsl(122, 39%, 40%, 1);

}

.card {

  background: linear-gradient(180deg, hsl(210, 100%, 75%), hsl(40, 100%, 75%));

  padding: 50px;

  box-shadow: 2px 2px 5px hsla(0, 0%, 0%, 0.5);

  min-width: 300px;

  display: flex;

  flex-direction: column;

  align-items: center;

  border-radius: 10px;

}

/\* styling city names \*/

h1 {

  margin-top: 0;

  margin-bottom: 25px;

}

/\* styling data \*/

p {

  font-size: 1.5rem;

  margin: 5px 0;

}

.city,

.tempDisplay {

  font-size: 3.5rem;

  font-weight: bold;

  color: hsl(0, 0%, 0%, 0.75);

  margin-bottom: 25px;

}

.humidityDisplay {

  font-weight: bold;

  margin-bottom: 25px;

}

.descDisplay {

  font-style: italic;

  font-weight: bold;

  font-size: 2rem;

}

.weatherEmoji {

  margin: 0;

  font-size: 7.5rem;

}

.errorDisplay {

  font-size: 2.5rem;

  font-weight: bold;

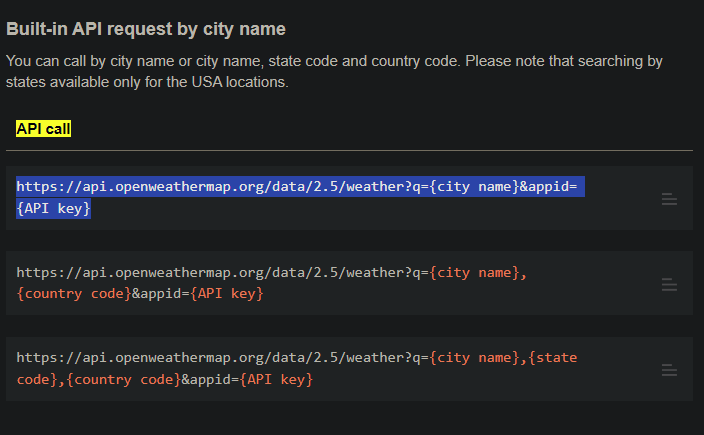
  color: hsl(0, 0%, 0%, 0.75);

}

Remove all of the contents within the card element so that it is only an empty container, then it's time to start working on the JS. (Also set the card's style to display: none so that it is hidden until the user submits a city.

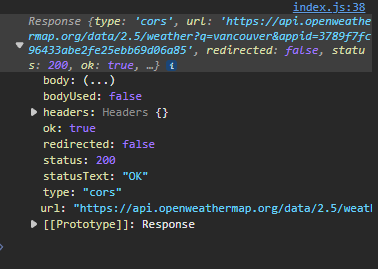
### Creating the getWeatherData() funciton

This Function will need to create a URL from the one listed in openweather's API. Navigate to https://openweathermap.org/current and find the section labeled "API Call". Copy the Url given to you.



and paste it into a string literal within getWeatherData. Replace the city name with a place holder and add your apiKey to into the second set of curly braces.

This is the response given to us:



We can see that it has a status of 200 and that it is ok.

Creating a fake city gives a response of ok: false and a 404 error.

Add an if statement to handle !response.ok.

### Important

Anything serving a file like apikeys.json requires the page to be run from live server. In offline mode (**D:\Documents\Programming\...\simulateLoading\loadingKey.html**) you will not be able to access the .json file due to cors policy.

Here is the Final HTML for the file:

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

    <link rel="stylesheet" href="style.css">

</head>

<body>

    <form action="" class="weatherForm">

        <input type="text" class="cityInput" placeholder="Enter City">

        <button type="submit">Get Weather</button>

    </form>

    <!-- hide card until city entered -->

    <div class="card" style="display: none">

    </div>

    <script src="index.js"></script>

</body>

</html>

CSS:

body {

  font-family: Arial, Helvetica, sans-serif;

  background-color: hsl(0, 0%, 95%, 1);

  margin: 0;

  display: flex;

  flex-direction: column;

  align-items: center;

}

.weatherForm {

  margin: 20px;

}

.cityInput {

  padding: 10px;

  font-size: 2rem;

  font-weight: bold;

  border: 2px solid hsl(0, 0%, 20%, 0.3);

  border-radius: 10px;

  margin: 10px;

  width: 300px;

}

/\* buttons [property = "value"] selector \*/

button[type="submit"] {

  padding: 10px 20px;

  font-weight: bold;

  font-size: 2rem;

  background-color: hsl(122, 39%, 50%, 1);

  color: white;

  border: none;

  border-radius: 5px;

  cursor: pointer;

}

button[type="submit"]:hover {

  background-color: hsl(122, 39%, 40%, 1);

}

.card {

  background: linear-gradient(180deg, hsl(210, 100%, 75%), hsl(40, 100%, 75%));

  padding: 50px;

  box-shadow: 2px 2px 5px hsla(0, 0%, 0%, 0.5);

  min-width: 300px;

  display: flex;

  flex-direction: column;

  align-items: center;

  border-radius: 10px;

}

/\* styling city names \*/

h1 {

  margin-top: 0;

  margin-bottom: 25px;

}

/\* styling data \*/

p {

  font-size: 1.5rem;

  margin: 5px 0;

}

.cityDisplay,

.tempDisplay {

  font-size: 3.5rem;

  font-weight: bold;

  color: hsl(0, 0%, 0%, 0.75);

  margin-bottom: 25px;

}

.humidityDisplay {

  font-weight: bold;

  margin-bottom: 25px;

}

.descDisplay {

  font-style: italic;

  font-weight: bold;

  font-size: 2rem;

}

.weatherEmoji {

  margin: 0;

  font-size: 7.5rem;

}

.errorDisplay {

  font-size: 2.5rem;

  font-weight: bold;

  color: hsl(0, 0%, 0%, 0.75);

}

Javascript:

// Weather App

// Using query selector to get first element matching .<class>

// do not forget the . when trying to query for a class

const weatherForm = document.**querySelector**(".weatherForm");

const cityInput = document.**querySelector**(".cityInput");

const card = document.**querySelector**(".card");

// Key can now be accessed from an external file

// adding async to event so that getWeather can use await

weatherForm.**addEventListener**("submit", async (event) => {

  // forms have a default function which refreshes the page

  event.**preventDefault**();

  const city = cityInput.value;

  if (city) {

    // try: user input, code could fail

    try {

      const weatherData = await **getWeatherData**(city);

**displayWeatherInfo**(weatherData);

      // 404 errors, city not in OpenWeather (likely result of typos by user)

    } catch (error) {

      console.**error**(error);

**displayError**(error);

    }

  } else {

**displayError**("Please enter a city");

  }

});

async function **getWeatherData**(city) {

  // load API key from file

  let filehandle = await **fetch**("../Assets/APIs/OpenWeather.json");

  const fileData = await filehandle.**json**()

  .**catch**(error => {

    console.**error**(error)

  })

  const apiKey = fileData.key;

  // console.log(apiKey);

  // debug, is key loaded or undefined

  // console.log(apiKey);

  const apiUrl = `https://api.openweathermap.org/data/2.5/weather?q=${city}&appid=${apiKey}`;

  // is url being created correctly

  // console.log(apiUrl);

  // pass the url created and stored within getWeatherData

  const response = await **fetch**(apiUrl);

  console.**log**(response);

  // handle bad fetch request

  //"Invalid API key. Please see https://openweathermap.org/faq#error401 for more info."

  // API key may take a few hours for a key to become valid, this also applies reactivated keys

  if (!response.ok) {

    throw new **Error**("Could not fetch weather data");

  }

  // convert response to json data and return it to the event listener

  return await response.**json**();

}

function **displayWeatherInfo**(data) {

  // debug see if data was retrieved correctly

  // console.log(data);

  // use object destructuring

  // access name, return city, access main return object

  const {

    name: city,

    main: { temp, humidity },

    weather: [{ description, id }],

  } = data;

  card.textContent = "";

  card.style.display = "flex";

  const cityDisplay = document.**createElement**("h1");

  const tempDisplay = document.**createElement**("p");

  const humidityDisplay = document.**createElement**("p");

  const descDisplay = document.**createElement**("p");

  const weatherEmoji = document.**createElement**("p");

  cityDisplay.textContent = city;

  tempDisplay.textContent = `${(temp - 273.15).**toFixed**(1)}°C`;

  humidityDisplay.textContent = `Humidity: ${humidity}%`;

  descDisplay.textContent = `${description}`;

  weatherEmoji.textContent = **getWeatherEmoji**(id);

  cityDisplay.classList.**add**("cityDisplay");

  tempDisplay.classList.**add**("tempDisplay");

  humidityDisplay.classList.**add**("humidityDisplay");

  descDisplay.classList.**add**("descDisplay");

  weatherEmoji.classList.**add**("weatherEmoji");

  card.**appendChild**(cityDisplay);

  card.**appendChild**(tempDisplay);

  card.**appendChild**(humidityDisplay);

  card.**appendChild**(descDisplay);

  card.**appendChild**(weatherEmoji);

}

function **getWeatherEmoji**(weatherId) {

  // codes can be found here:

  // https://openweathermap.org/weather-conditions

  // use a switch to determine the emoji

  // examine the bool value of true

  // does the case evaluate to true, return that value

  switch (true) {

    case weatherId >= 200 && weatherId < 300:

      return "🌩️";

    case weatherId >= 300 && weatherId < 400:

      return "🌦️";

    case weatherId >= 500 && weatherId < 600:

      return "🌧️";

    case weatherId >= 600 && weatherId < 700:

      return "❄️";

    case weatherId >= 700 && weatherId < 800:

      return "🌫️";

    case weatherId == 800:

      return "☀️";

    case weatherId >= 801 && weatherId < 810:

      return "☁️";

    default:

      return "❓";

  }

}

function **displayError**(message) {

  // create a paragraph to display error

  const errorDisplay = document.**createElement**("p");

  errorDisplay.textContent = message;

  errorDisplay.classList.**add**("errorDisplay");

  card.textContent = "";

  card.style.display = "flex";

  card.**appendChild**(errorDisplay);

}

Go to ./simulateLoading to see a small example of how the API key is being loaded from a file.