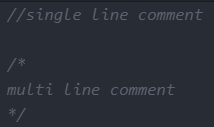
**Javascript Notes**

**Adding js file to html**



**Comments**

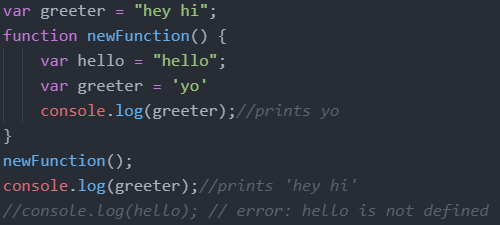
Single Line Comments // and Multi-Line Comments /\*\*/



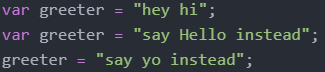
**Variables**

var

var declarations are globally scoped or function/locally scoped. If var variables are not declared a function, then it has global scope. In other words, this means any variable that is declared with var outside a function block is available for use in the whole window.

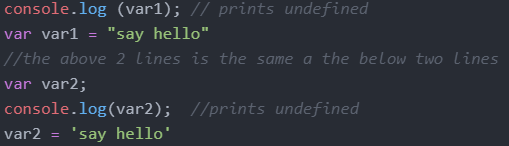


var variables can be re-declared and updated

Text

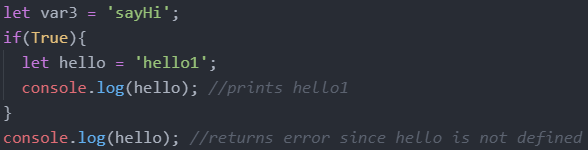
Description automatically generated

Hoisting of var. Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution

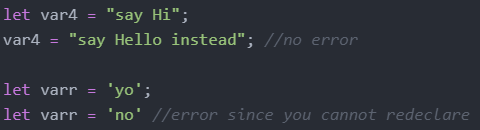


let

let is now preferred for variable declaration. let is blocked scope. a block is anything with {}When using let, you don't have to worry about if you have used a name for a variable before since a variable exists only within its scope. This fact makes let a better choice than var. Shadowing exists.



let can be updated but not re-declared



Hoisting of let. Just like var, let declarations are hoisted to the top. Unlike var which is initialized as undefined, the let keyword is not initialized. So if you try to use a let variable before declaration, you'll get a Reference Error

const

const definitions have block scope.Shadowing exists.

It does not define a constant value. It defines a constant reference to a value.

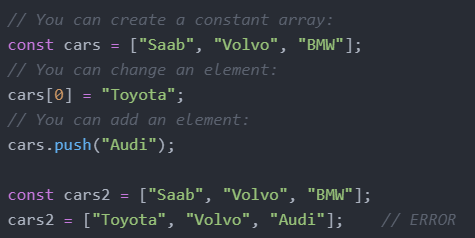
Because of this you can NOT:

* Reassign a constant value
* Reassign a constant array
* Reassign a constant object



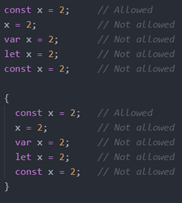
But you CAN:

* Change a constant array



* Change a constant object

Declaring a variable with const is similar to let when it comes to Block Scope. The x declared in the block, in this example, is not the same as the x declared outside the block. Redeclaring a variable with const, in another scope, or in another block, is allowed



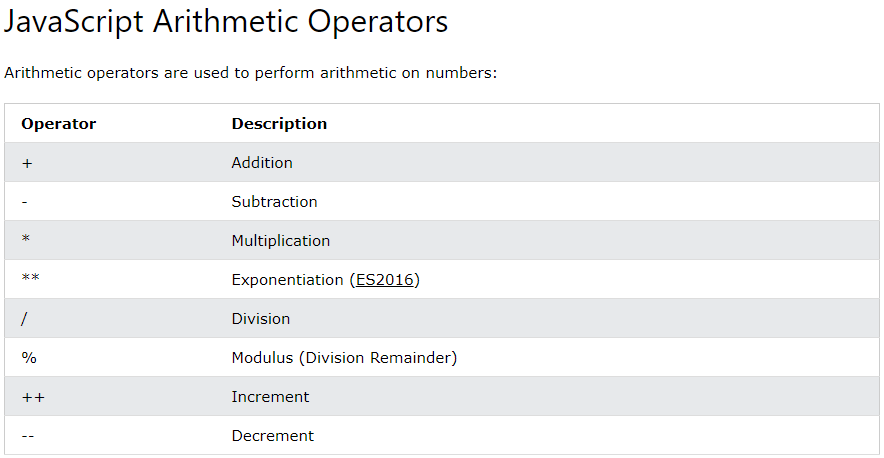
Hoisting, using a const variable before it is declared will result in a ReferenceError

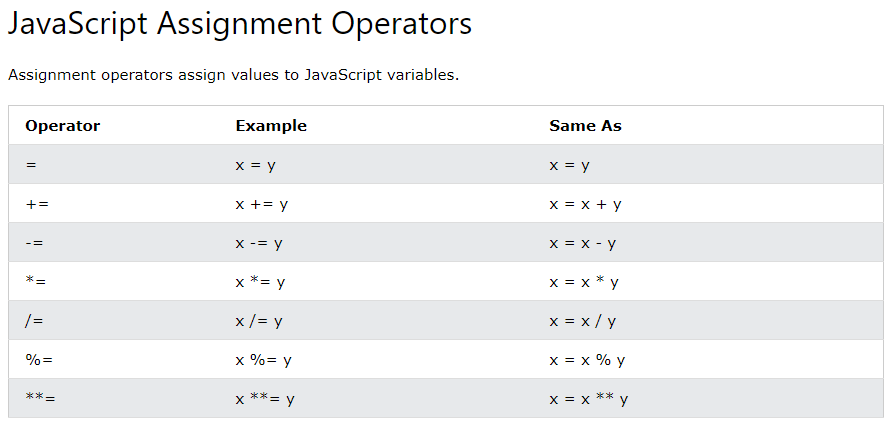


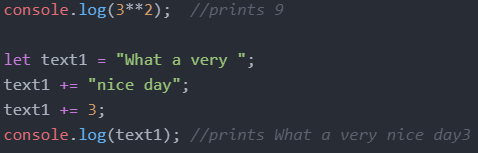
Text

Description automatically generated

**Operators**

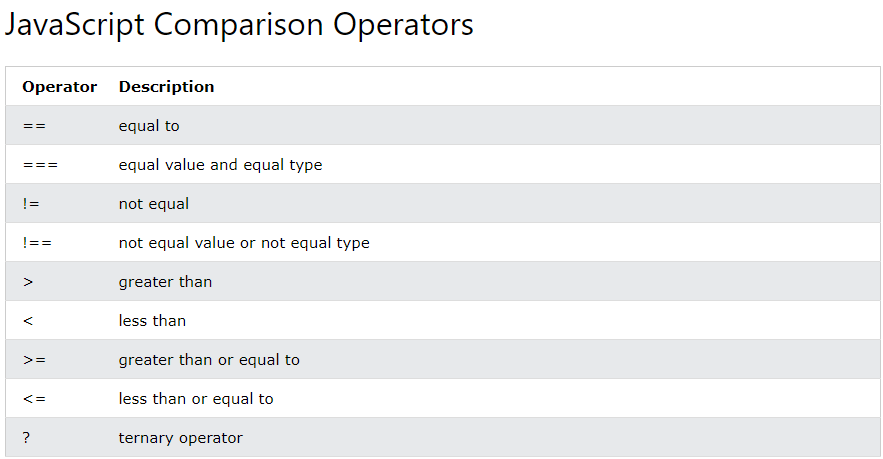


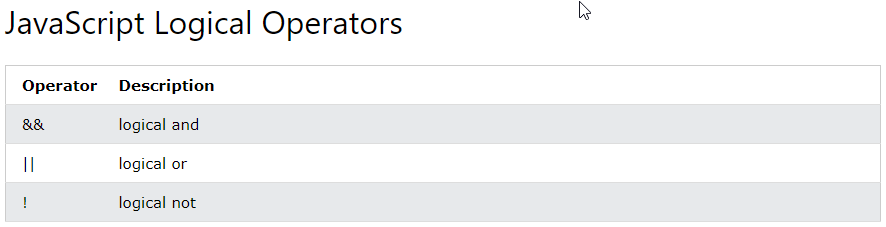


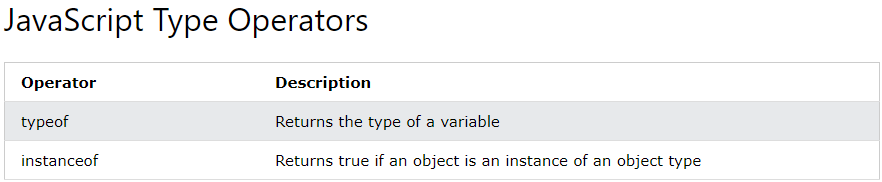


JavaScript evaluates expressions from left to right. Different sequences can produce different results:





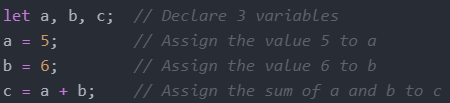




**Data Types**

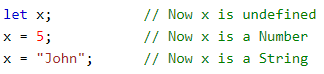
Declaring Variables

Separated by semicolons, multiple statements on one line are allowed. The two codes below are the same.





JavaScript has dynamic types. This means that the same variable can be used to hold different data types



Undefined



Strings/Empty



An empty value has nothing to do with undefined. An empty string has both a legal value and a type



Numbers



The above 3 variables will print as the following 3





Booleans

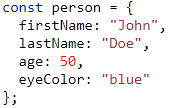
let x = true;

let y = true;

Arrays



Objects



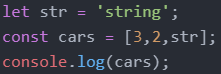
typeOf Operator



**Arrays**

Creating an Array

you can have variables of different types in the same Array such as objects, functions, arrays





Accessing Array Elements



Changing Array Elements



Access the Full Array



This will change demo to



Adding Array Elements

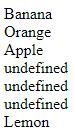
The push() method add an element to the end of an the array



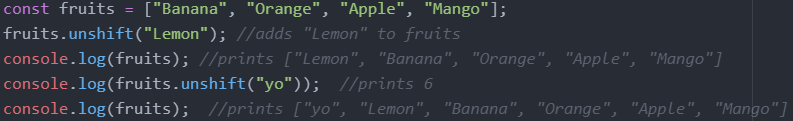
Adding large indexes can create holes



This will print the following

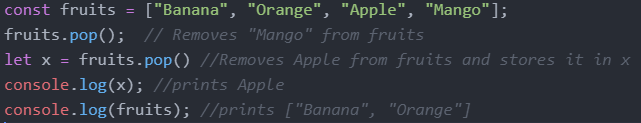


The unshift() method adds a new element to an array (at the beginning), and "unshifts" older elements:

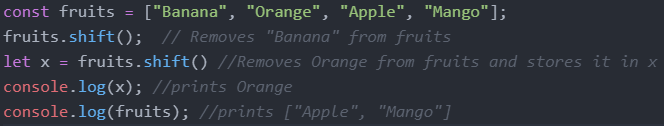


Remove Elements

The pop() method removes the last element from an array



The shift() method removes the first array element and "shifts" all other elements to a lower index



Length



typeOf and Array.isArray() and instanceof

The problem is that the JavaScript operator typeof returns "object" for arrays

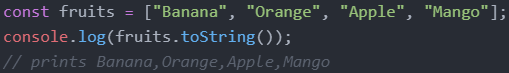


To solve this issue, we can use Array.isArray or instanceof



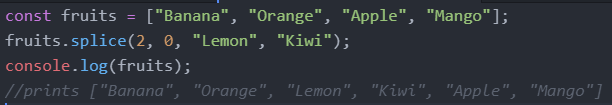


Convert Arrays to Strings



Splicing Arrays

The splice() method can be used to add new items to an array



The first parameter (2) defines the position where new elements should be added (spliced in).

The second parameter (0) defines how many elements should be removed.

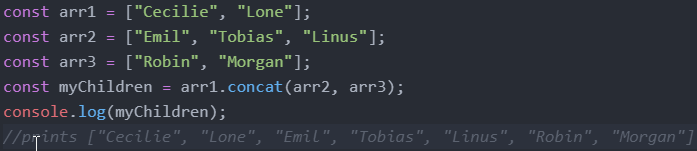
The rest of the parameters ("Lemon" , "Kiwi") define the new elements to be added.

Merging (Concatenating) Arrays

The concat() method creates a new array by merging (concatenating) existing arrays.

The concat() method does not change the existing arrays. It always returns a new array.

The concat() method can take any number of array arguments.



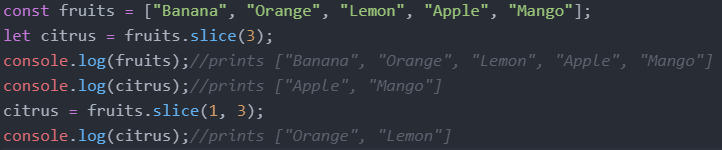
Slicing Arrays

The slice() method slices out a piece of an array into a new array.

The slice() method creates a new array. It does not remove any elements from the source array.

The slice() method can take two arguments and selects elements from the start argument, and up to (but not including) the end argument.

If the end argument is omitted, the slice() method slices out the rest of the array.

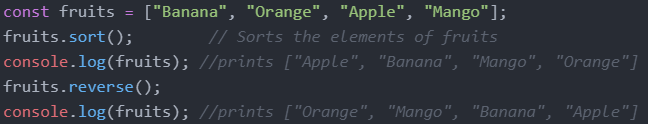


Sorting and Reversing Arrays

The sort() method sorts an array alphabetically for both strings and number elements

The reverse() method reverses the elements in an array by sorting in descending order:

Sorting Strings



Sorting Numbers

By default, the sort() function sorts values as strings

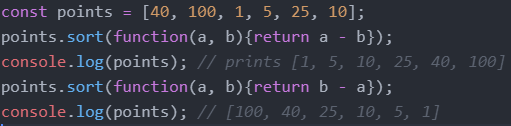
If numbers are sorted as strings, "25" is bigger than "100", because "2" is bigger than "1".

Because of this, the sort() method will produce incorrect results when sorting numbers.

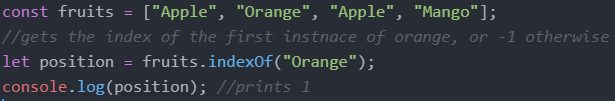
You can fix this by providing a compare function

Sort ascending: array\_name.sort(function(a, b){return a - b})

Sort descending: array\_name.sort(function(a, b){return b - a})



indexOf



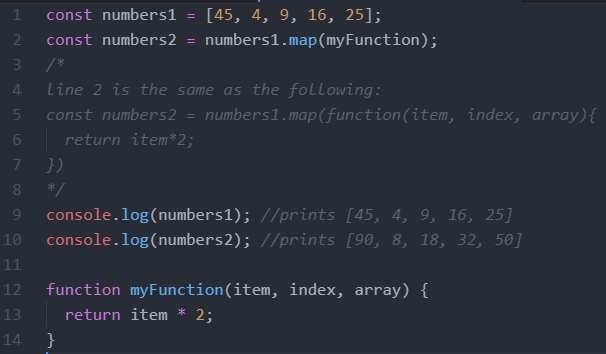
Array Iteration

map()

The map() method creates a new array by performing a function on each array element.

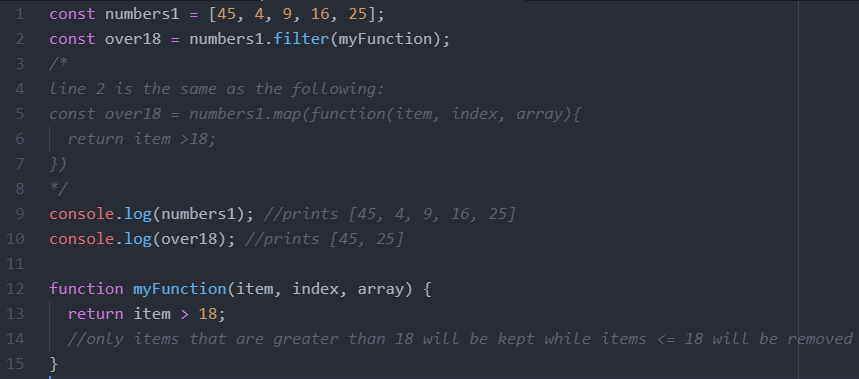
The map() method does not execute the function for array elements without values.

The map() method does not change the original array



filter()

The filter() method creates a new array with array elements that passes a test.



reduce()

The reduce() method runs a function on each array element to return a single value.

The reduce() method works from left-to-right in the array.



Array.from

Array.from() lets you create Arrays from:

array-like objects (objects with a length property and indexed elements);

or iterable objects (objects such as Map and Set)

Array.from('foo') returns the value [“f”, “o”, “o”]



Syntax: Array.from(object, mapFunction, thisValue)

Array.from() has an optional parameter mapFn, which allows you to execute a map() function on each element of the array being created. More clearly, Array.from(obj, mapFn, thisArg) has the same result as Array.from(obj).map(mapFn, thisArg), except that it does not create an intermediate array, and mapFn only receives two arguments (element, index).



Array.forEach()

* forEach() method calls a function for each element in an array.
* The forEach() method is not executed for empty elements.
* *Syntax: array*.forEach(function(currentValue, index, arr), thisValue)
* Logo

  Description automatically generated with low confidence

**Objects**

Object Definition



This Keyword

In a function definition, this refers to the "owner" of the function.

In the example above, this is the person object that "owns" the fullName function.

In other words, this.firstName means the firstName property of this object.

Accessing Object Properties

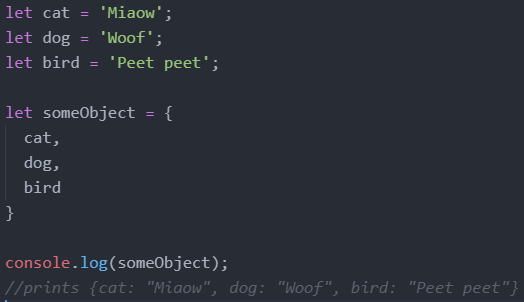
let name1 = objectName.propertyName

let name2 = objectName["propertyName"]

Accessing Object Methods

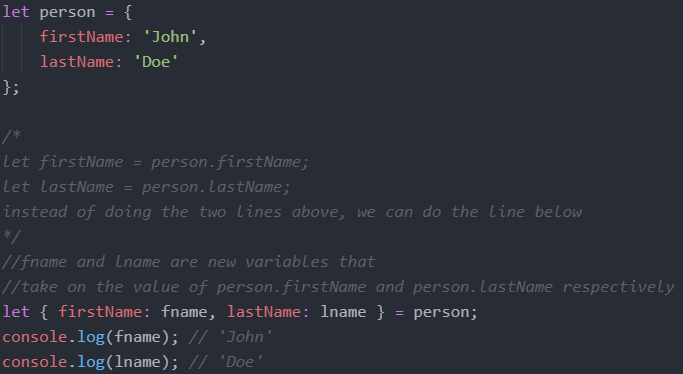
let fullname = objectName.methodName()

Property Shorthand

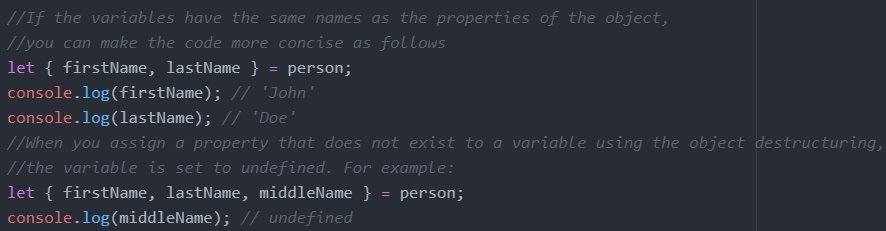


Destructuring

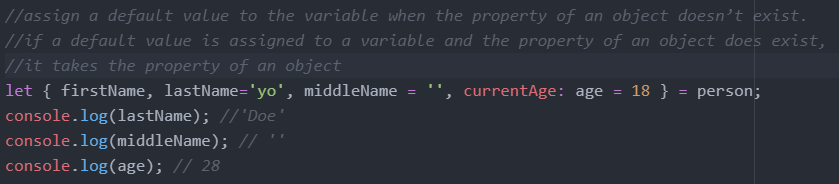
The destructuring assignment syntax is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.



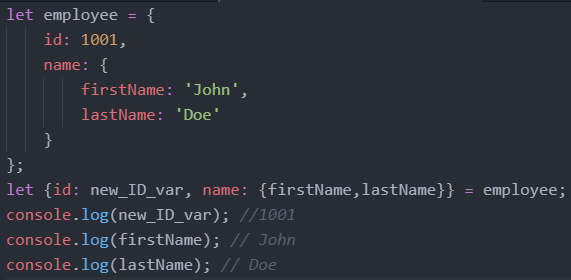
Declaring variables that have the same names as the properties of the object



Default values



Nested object destructuring



Spread Operator



**Objects vs Arrays**

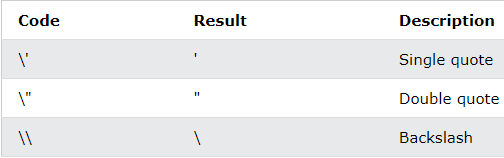
Arrays use numbered indexes while objects used name indexes

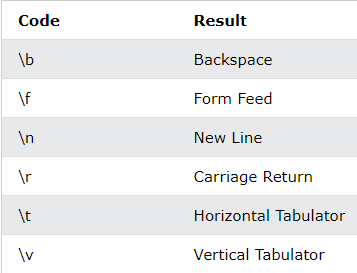
**String Methods**

Length



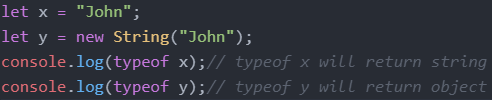
Escape Characters





Literals Vs Objects

Normally, JavaScript strings are primitive values, created from literals, but strings can also be defined as objects with the keyword new.



Comparing Strings and Objects/Strings

The == operator returns true if the values of two strings or objects are equal

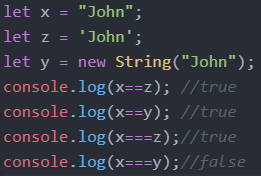
The === operator returns true if the values and data types are equal

1 ==1 is true and 1===1 is true

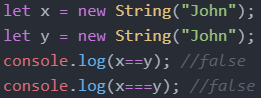
1 == ‘1’ is true since the string ‘1’ is converted to an int which is 1 and they are equal. but 1 === ‘1’ is not true since an integer and string are not of the same type.

Below is a list of weird shit. You just need to know they exist, not why that’s the case.

* + 0 == ‘’ is true
  + 0 === ’’ is false
  + 0 == false is true
  + 0 === false is false
* A good use case of the === is to differentiate between null and undefined
  + null == undefined is true
  + null === undefined is false
* In general, always use === instead of == unless we are comparing null vs undefined.

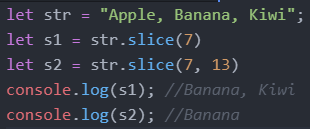


Comparing Objects and Objects



Slice()

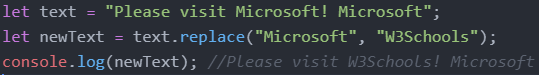
extracts a part of a string and returns the extracted part in a new string



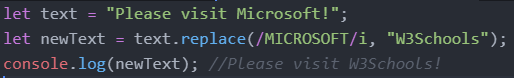
Replace()

The replace() method replaces a the first instance of a specified value with another value in a string

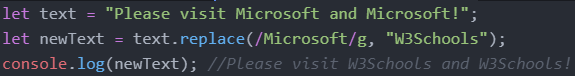
The replace() method is case sensitive



To replace case insensitive, use a regular expression with an /i flag (insensitive)



To replace all matches, use a regular expression with a /g flag (global match)



charAt()

The charAt() method returns the character at a specified index (position) in a string



indexOf()

The indexOf() method returns the index of (the position of) the first occurrence of a specified text in a string, or -1 if not found

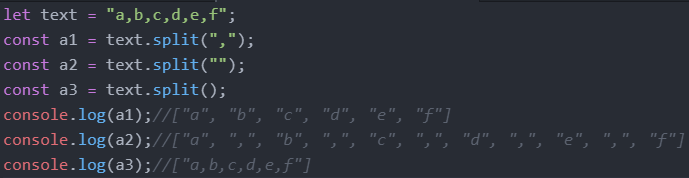


lastIndexOf()

The lastIndexOf() method returns the index of the last occurrence of a specified text in a string:



Convert String to Array



**Sets**

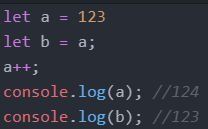
**Random**

Math.random() returns a random number between 0 (inclusive) and 1 (exclusive)

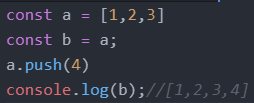
Ex: Return a random integer between 1 and 10: Math.floor((Math.random() \* 10) + 1);

**Immutable Vs Mutable Data Types**

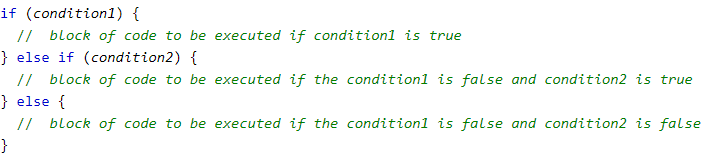
Immutable: Numbers, Strings, Null, Undefined, Booleans



Mutable: Arrays, Functions, Classes, Maps, Sets

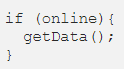


**If Statements**

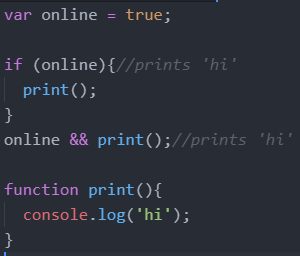


Logical Operators in Javascript such as && or || evaluate from left to right and they short circuit. Short circuiting means that in JavaScript when we are evaluating an AND expression (&&), if the first operand is false, JavaScript will short-circuit and not even look at the second operand.

Through short circuiting, the following two blocks of code are the same.

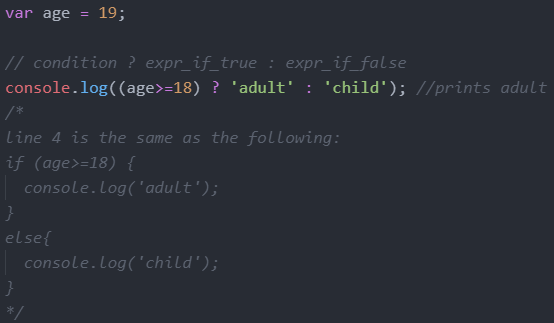
 

In the left block of code, if online is true, getData is run. If online is false, getData isn’t even looked at. In the right block of code, it reads from left to right so it first checks if online is true. If online is false, it short circuits and doesn’t even read the getData(). If online is true, it runs getData(). Note that getData() is any function and doesn’t have to return a boolean.

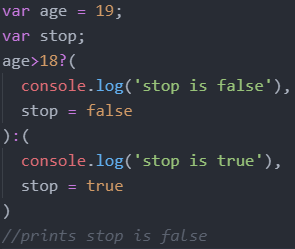


**Ternary Operator**

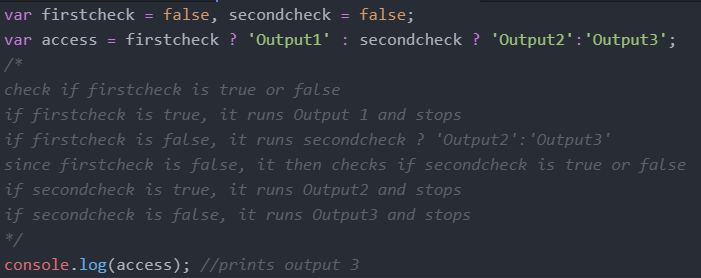
The syntax for a ternary operator is condition ? expr\_if\_cond\_is\_true : expr\_if\_cond\_is\_false



Multiple Operations Per Condition

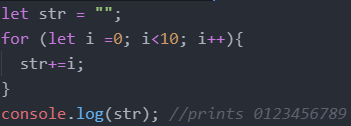


Nested Ternary Operations



**Loops**

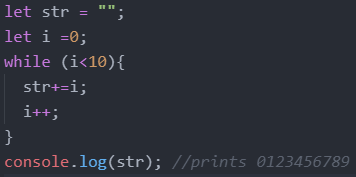
For Loops



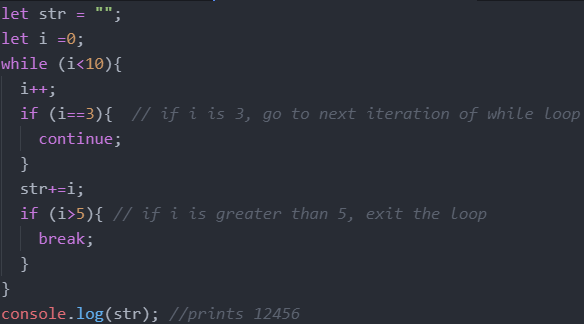
Graphical user interface, text

Description automatically generated

While Loops



Break and Continue

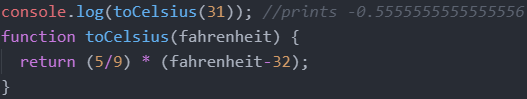


**Function**

Declaration

The function declaration (function statement) defines a function with the specified parameters.

Function declarations in JavaScript are hoisted to the top of the enclosing function or global scope. You can use the function before you declared it



Functions are pass by value, Function arguments are local variables in JavaScript.

Text

Description automatically generated

Reference values are also passed by values.

Consider the code below:

Text

Description automatically generated

The console will output the person object with an age of 26. But if the parameter is pass by value, how is it able to change the person object?

First, define the person variable that references an object with two properties name and age.

Chart

Description automatically generated

Next, define the increaseAge() function that accepts an object obj and increases the age property of the obj argument by one.

Then, pass the person object to the increaseAge() function:

Chart, diagram

Description automatically generated

Internally, the JavaScript engine creates the obj variable that points to the same memory address as the object. After that, increase the age property of the obj variable by one inside the increaseAge(). But since obj and person point to the same memory address, changing obj also changes person.

It seems that JavaScript passes an object by reference because the change to the object is reflected outside the function. However, this is not the case. In fact, when passing an object to a function, you are passing the memory address of that object, not the actual object. Therefore, the function can modify the properties of the object via its memory address.

However, you cannot change the memory address of the object that is passed as a parameter.

Text

Description automatically generated

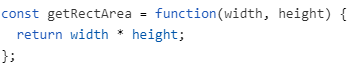
The code is the same as above. The obj variable points to the same location in memory as the person object. But the second we say obj = { name: ‘Jane’, age: 22 }, the obj variable points to a new location in memory, thus not changing any properties of the person object.

Expression/Anonymous

A function expression can be stored in a variable

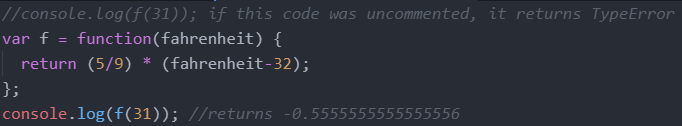
Definition

The function keyword can be used to define a function inside an expression.

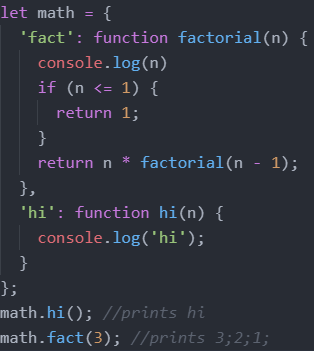
****

Hoisting

Function expressions in JavaScript are not hoisted, unlike function declarations. You can't use function expressions before you create them.



Named Function Expression



Arrow Functions

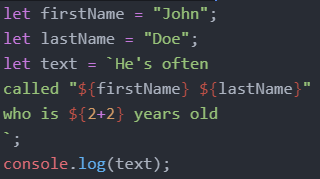
Arrow functions allow us to write shorter function syntax



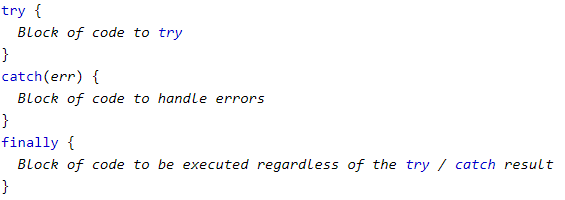
**Template Literals**

Template Literals use back-ticks (``) rather than the quotes ("") to define a string.

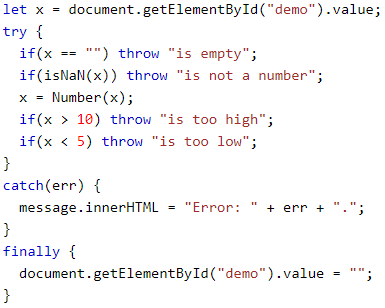
With template literals, you can use both single and double quotes inside a string, multiline strings, and interpolate variables and expressions into strings.



**Error Handling**

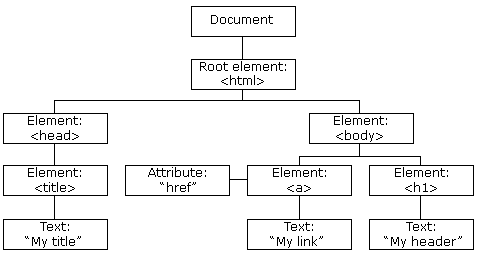


Ex:



**DOM (Document Object Model)**

The HTML DOM model is constructed as a tree of Objects



HTML DOM properties are values (of HTML Elements) that you can set or change.

HTML DOM methods are actions you can perform (on HTML Elements)

When we target an element using one of the selectors below, we get an Element object that represents the element.

|  |
| --- |
| **Finding HTML Elements** |
| **Method** |
| document.getElementById("id\_name") |
| document.getElementsByTagName("p") |
| document.getElementsByClassName("class\_name") |
| document.querySelector(“.className”) (This only gives us one element despite many elements possibly having the class name of className. This returns the first element with the class name of className) |
| document.querySelectorAll(“.className”) (this gives us all of the elements with the class name of className). |
| document.querySelector(“#idName”) |

Consider the HTML below:

Text

Description automatically generated

Selecting Children

Suppose we are given a DOM element and we want to get all the elements right underneath it. (not to the side of it or above, but right below), we can use .children.



The .children will give us all the children of a given element. In this case, the children would be the two divs with class of parent. The datatype of parents is an HTMLCollection which is a live, ordered collection of the DOM elements which are children of grandparent. You can access the individual child nodes in the collection by using either the item() method on the collection, or by using JavaScript array-style notation. If the element has no element children, then children is an empty list with a length of 0.

Text

Description automatically generated

To make it easier for us to use the collection of children elements, we couldve said the following instead which makes it an array



Thus, parents[0] would give us the Element object that represents the first element in parents.

Selecting Descendants

Suppose we are given a DOM element and we want to find a certain element below it (not to the side of it or above it, but below), we can use .querySelector



Notice we are apply the querySelectorAll on a single element object. This is no different that what we were doing before since before, we applied querySelector on document which is the root node. Grandparent is now just some other node so we can also apply querySelector on it no differently.

In the above example, we are looking for the all the elements that have a class of “child” that are also descendants of the Element object stored in the grandparent variable.

Selecting Parents

Suppose we are given a DOM element and we want to find the element right above it (not to the side of it or below it, but right above), we can use .parentElement. We could also use .parentNode but that might sometimes select a node which is not what we want.

Text

Description automatically generated

Selecting Ancestors

Suppose we are given a DOM element and we want to find a certain element above it (not to the side of it or below it, but above), we can use .closest which works the same as querySelector but closest goes upwards while querySelector goes down the tree.



When we do childOne.closest(“.grandparent”), we will go to childOne’s parent to get an element with class “parent”. Since “parent” is not “grandparent”, we go look at “parent”’s parent which does have a class of “grandparent”.

Selecting Siblings

Suppose we are given a DOM element and we want to find the element right after it with the same hierarchy. We can use .nextElementSibling. We can use previousElementSibling to get the previous element with the same hierarchy.





childTwo is highlighted below in black.

Text

Description automatically generated



 this returns the element highlighted in red.



|  |  |
| --- | --- |
| **Changing HTML Elements** | |
| **Property** | **Examples** |
| element.innerHTML = new html content | document.getElementById("p1").innerHTML = "New text!"; |
| element.attribute = new value | document.getElementById("myImage").src = "landscape.jpg"; |
| element.style.property = new style | document.getElementById("p2").style.color = "blue"; |
| **Method** |  |
| element.setAttribute(attribute, value) |  |

|  |
| --- |
| **Adding and Deleting Elements** |
| **Method** |
| document.createElement(element) |
| document.removeChild(element) |
| document.appendChild(element) |
| document.replaceChild(new, old) |

|  |
| --- |
| **Adding Events Handlers** |
| **Method** |
| document.getElementById(id).onclick = function(){code}  Examples:  document.getElementById("myBtn").onclick = displayDate; |

Events

onclick

onmousedown

onmouseup

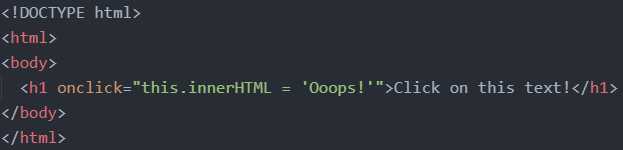
onload

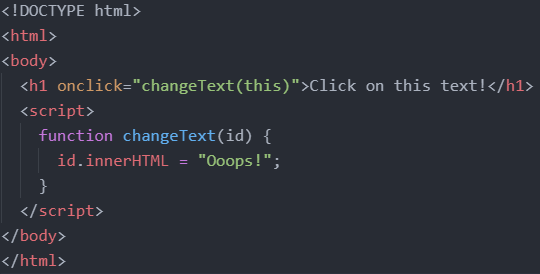
onunload

onchange

onmouseover

onmouseout





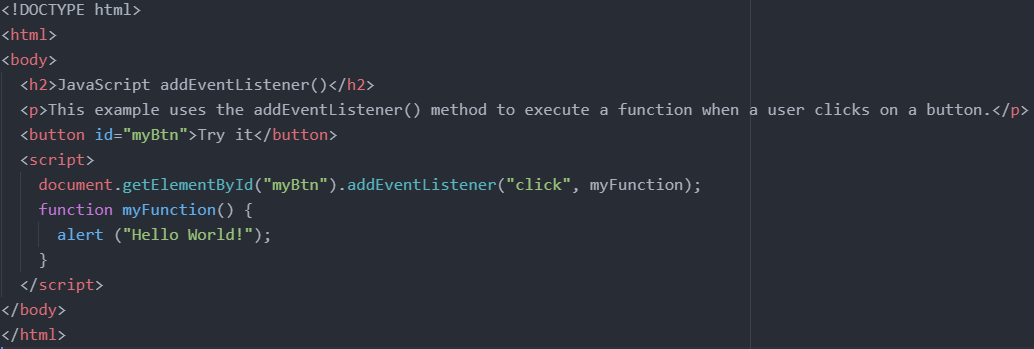
Event Listener

Syntax: element.addEventListener(event, function, useCapture);

- The first parameter is the type of the event (like "click" or "mousedown" or any other HTML DOM Event.). Note that you don't use the "on" prefix for the event; use "click" instead of "onclick".

- The second parameter is the function we want to call when the event occurs.

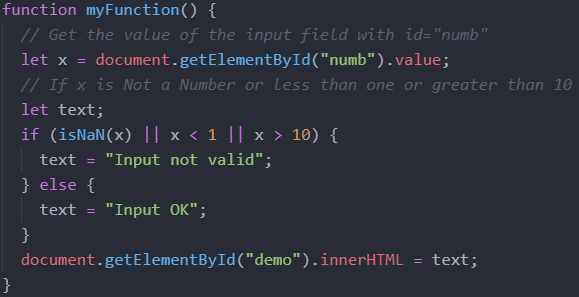
- The third parameter is a boolean value specifying whether to use event bubbling or event capturing. This parameter is optional.



Forms

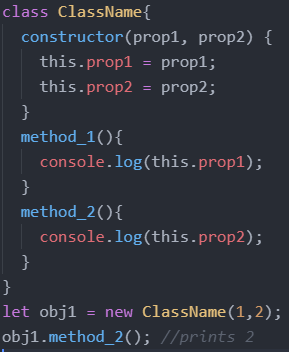
Get the value of an input field through document.getElementById("input\_field\_id").value;





**Classes**

A JavaScript class is not an object. It is a template for JavaScript objects.



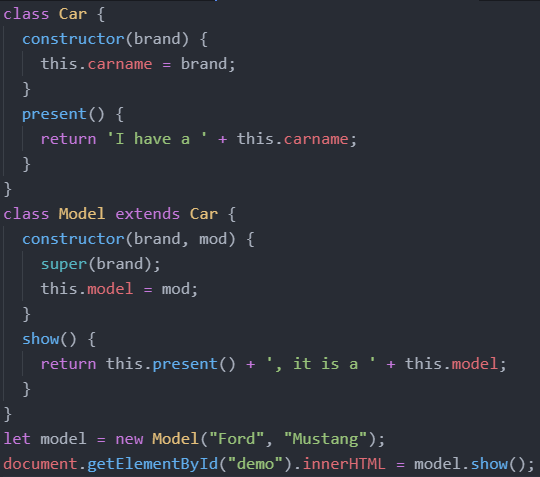
It has to have the exact name "constructor". The constructor method is called automatically when a new object is created. If you do not define a constructor method, JavaScript will add an empty constructor method.

Inheritance

To create a class inheritance, use the extends keyword.

A class created with a class inherits and inherits all the methods from another class.

The super() method refers to the parent class. By calling the super() method in the constructor method, we call the parent's constructor method and gets access to the parent's properties and methods.



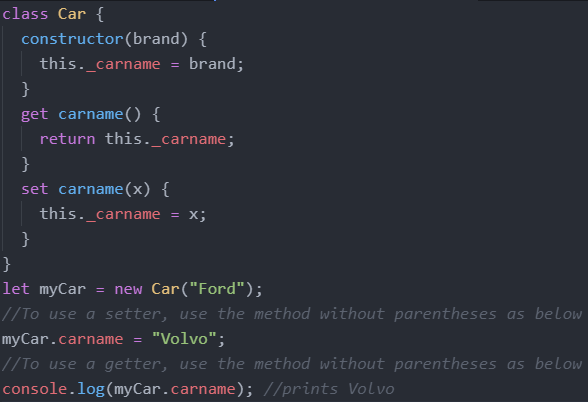
Getters and Setters

To add getters and setters in the class, use the get and set keywords.

The name of the getter/setter method cannot be the same as the name of the property, in this case carname.

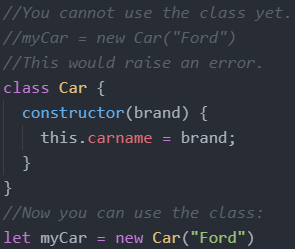
Many programmers use an underscore character \_ before the property name to separate the getter/setter from the actual property. Notice that setters and getter methods can have the same name.

To use a getter or setter, use the same syntax as when you set a property value, without parentheses.



Hoisting

Cass declarations are not hoisted so you must declare a class before you can use it.



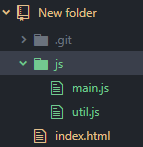
Static Class Properties and Methods



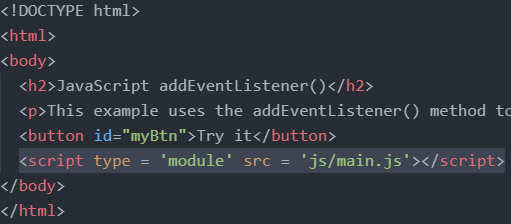
**Modules**

Modules can only be used in live servers, by directly clicking the file on our desktop.

Assume we have the following directory structure where main.js imports code from util.js.

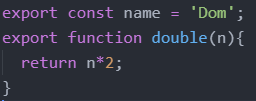


In the html, make sure to link to the main js file with type = 'module'.

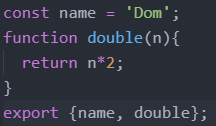


Single Export

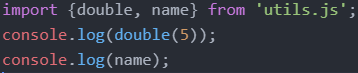
In the utils.js file, we have the following. We have a const and function declaration as normal, but we stick the word export in front. This makes the const and function exportable.



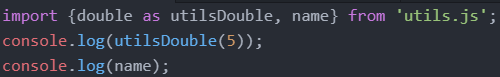
We can also do the following



In the main.js file, we do import {item1, item2, item3, …} from 'filePath.js';



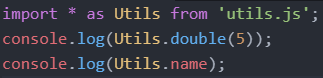
We can also change the names of imported items



Multiple Export

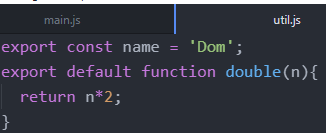
We still have to put export for each thing we want to export in the utils.js file.

But in the html file, we can do \* instead of listing each item we want to import.

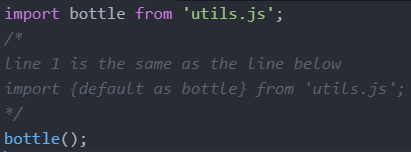


Default Export

The export default keyword means the item is the default export for the module. One module can only have 1 default export.



In the main.js folder, we can do the following. Notice there is no item called bottle in the utls.js module. Thus, bottle will be used for the default export which is the function.



**Callbacks**

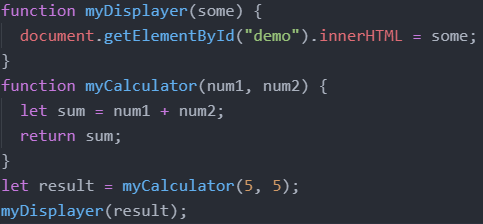
A callback is a function passed as an argument to another function.

This technique allows a function to call another function.

A callback function can run after another function has finished.

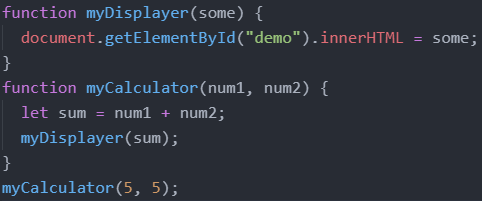
Consider the following 2 scenarios that don’t use callbacks that allow us to do a calculation, and then display the result.

Scenario 1: You could call a calculator function (myCalculator), save the result, and then call another function (myDisplayer) to display the result:



However, the problem is that you have to call two functions to display the result.

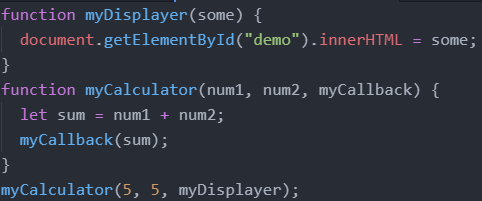
Scenario 2: You could call a calculator function (myCalculator), and let the calculator function call the display function (myDisplayer):



The problem is that you cannot prevent the calculator function from displaying the result.

Now, we will use callbacks.

You could call the calculator function (myCalculator) with a callback, and let the calculator function run the callback after the calculation is finished.



In the example above, myDisplayer is the name of a function and is passed to myCalculator() as an argument.

When you pass a function as an argument, remember not to use parenthesis.

Right: myCalculator(5, 5, myDisplayer);

Wrong: myCalculator(5, 5, myDisplayer());

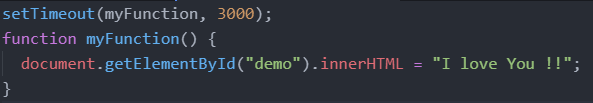
Where callbacks really shine are in asynchronous functions, where one function has to wait for another function (like waiting for a file to load).

**Asynchronous Functions**

Functions running in parallel with other functions are called asynchronous.

setTimeout()

A good example is JavaScript setTimeout() to specify a callback function to be executed on time-out.



In the example above, myFunction is used as a callback.

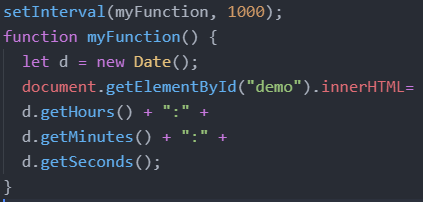
The function (the function name) is passed to setTimeout() as an argument.

3000 is the number of milliseconds before time-out, so myFunction() will be called after 3 seconds.

When you pass a function as an argument, remember not to use parenthesis.

setInterval()

you can specify a callback function to be executed every few seconds(the interval time)



In the example above, myFunction is used as a callback.

The function (the function name) is passed to setInterval() as an argument.

1000 is the number of milliseconds between intervals, so myFunction() will be called every second.

Waiting for Files

If you create a function to load an external resource (like a script or a file), you cannot use the content before it is fully loaded.

This is the perfect time to use a callback.

This example loads a HTML file (mycar.html), and displays the HTML file in a web page, after the file is fully loaded:



In the example above, myDisplayer is used as a callback.

The function (the function name) is passed to getFile() as an argument.

**Promise**

A promise in JS is like a promise in real life. You commit to something by promising to do something and you either fulfill your promise (resolve) or fail to keep your promise (reject)

In the code below, in the promise we call a function that does something(which can succeed or fail). In that function, we will also either call the resolve function if the task was successful and the reject function if the task failed. The .then method does what should happen if the resolve function was called and the .catch method does what should happen if the reject function is called.

Promises are really good when you need to do something that takes a long time(and might fail) in the background. For example, if we have to download an image from a different server.

Text

Description automatically generated

Promises are meant to replace callbacks

The following two functions are the same. The one on the left uses callbacks. The one on the right uses promises.

For the left code, if the task is unsuccessful(there is an error), the function will execute the errorCallback function and if the task is successful, then the function will execute the callback method.

For the right code, if the task is unsuccessful(task failed), the reject function will be executed. Otherwise, the task is successful, and the resolve function will be executed.

Text

Description automatically generatedText

Description automatically generated

**Async Await**

* Different syntax that can replace promises and make it look cleaner

Text

Description automatically generated