# **JavaScript language overview**

JavaScript (often shortened to JS) is a lightweight, interpreted, object-oriented language with first-class functions, and is best known as the scripting language for Web pages, but it's used in many non-browser environments as well. JavaScript is a dynamic language with types and operators, standard built-in objects, and methods. Its syntax is based on the Java and C languages — many structures from those languages apply to JavaScript as well. JavaScript supports object-oriented programming with object prototypes (When it comes to inheritance, JavaScript only has one construct: objects. Each object has a private property which holds a link to another object called its . That prototype object has a prototype of its own, and so on until an object is reached with null as its prototype) and classes. It also supports functional programming since functions are first-class (A programming language is said to have when functions in that language are treated like any other variable. For example, in such a language, a function can be passed as an argument to other functions, can be returned by another function and can be assigned as a value to a variable.) that can be easily created via expressions and passed around like any other object.

This section serves as a quick overview of various JavaScript language features, written for readers with background in other languages, such as C or Java.

## **Data Types**

Let's start off by looking at the building blocks of any language: the types. JavaScript programs manipulate values, and those values all belong to a type. JavaScript offers seven data types:

* Number: used for all number values (integer and floating point) except for big integers.
* BigInt: used for arbitrarily large integers.
* String: used to store text.
* Boolean: true and false — usually used for conditional logic.
* Symbol: used for creating unique identifiers that won't collide.
* Undefined: indicating that a variable has not been assigned a value.
* Null: indicating a deliberate non-value.

Everything else is known as an Object. Common object types include:

* Function
* Array
* Date
* RegExp
* Error

Functions aren't special data structures in JavaScript — they are just a special type of object that can be called.

### **Numbers**

JavaScript has two built-in numeric types: Number and BigInt.

The Number type is a IEEE 754 64-bit double-precision floating point value, which means integers can be safely represented between -(253-1) and (253-1) without loss of precision, and floating point numbers can be stored all the way up to 1.79 × 10308. Within numbers, JavaScript does not distinguish between floating point numbers and integers.

console.log(3 / 2); // 1.5, not 1

### **BigInt**

The BigInt type is an arbitrary length integer. Its behavior is similar to C's integer types (e.g. division truncates to zero), except it can grow indefinitely. BigInts are specified with a number literal and an n suffix.

console.log(-3n / 2n); // -1n

### **Strings**

Strings in JavaScript are sequences of Unicode characters (Unicode includes characters from most of today's languages, punctuation marks, diacritics, mathematical symbols, technical symbols, arrows, emoji, and more). Strings can be written with either single or double quotes — JavaScript does not have the distinction between characters and strings. If you want to represent a single character, you just use a string consisting of that single character.

console.log("Hello"[1] === "e"); // true

### **Variables**

Variables in JavaScript are declared using one of three keywords: let, const, or var.

let allows you to declare block-level variables. The declared variable is available from the block it is enclosed in.

code:

let a;

let name = "Simon";

// myLetVariable is \*not\* visible out here

for (let myLetVariable = 0; myLetVariable < 5; myLetVariable++) {

// myLetVariable is only visible in here

}

// myLetVariable is \*not\* visible out here

const allows you to declare variables whose values are never intended to change. The variable is available from the block it is declared in.

code:

const Pi = 3.14; // Declare variable Pi

console.log(Pi); // 3.14

var declarations can have surprising behaviors (for example, they are not block-scoped), and they are discouraged in modern JavaScript code. The scope of var goes global and leads to poor memory management and leaks.

If you declare a variable without assigning any value to it, its value is undefined. You can't declare a const variable without an initializer, because you can't change it later anyway.

## **Operators**

JavaScript's numeric operators include +, -, \*, /, % (remainder), and \*\* (exponentiation). Values are assigned using =. Each binary operator also has a compound assignment counterpart such as += and -=, which extend out to

x = x operator y.

code:

x += 5;

x = x + 5;

You can use ++ and -- to increment and decrement respectively. These can be used as prefix or postfix operators.

The + operator also does string concatenation:

"hello" + " world"; // "hello world"

If you add a string to a number (or other value) everything is converted into a string first. This might trip you up:

"3" + 4 + 5; // "345"

3 + 4 + "5"; // "75"

Adding an empty string to something is a useful way of converting it to a string itself.

Comparisons in JavaScript can be made using <, >, <= and >=, which work for both strings and numbers. For equality, the double-equals operator performs type coercion if you give it different types, with sometimes interesting results. On the other hand, the triple-equals operator does not attempt type coercion, and is usually preferred. = in JavaScript is used for assigning values to a variable. == in JavaScript is used for comparing two variables, but it ignores the datatype of the variable. === is used for comparing two variables, but this operator also checks datatype and compares two values.

123 == "123"; // true

1 == true; // true

123 === "123"; // false

1 === true; // false

The double-equals and triple-equals also have their inequality counterparts: != and !==.

JavaScript also has bitwise operators and logical operators. Notably, logical operators don't work with boolean values only — they work by the "truthiness" of the value.

const a = 0 && "Hello"; // 0 because 0 is "falsy"

const b = "Hello" || "world"; // "Hello" because both "Hello" and "world" are "truthy".

The && and || operators use short-circuit logic, which means whether they will execute their second operand is dependent on the first. This is useful for checking for null objects before accessing their attributes:

const name = o && o.getName();

Or for caching values (when falsy values are invalid):

const name = cachedName || (cachedName = getName());

Below is the example of the Short circuiting operators.

<script>

function gfg() {

// AND short circuit

document.write(false && true)

document.write("</br>");

document.write(true && true)

document.write("</br>");

// OR short circuit

document.write(true || false)

document.write("</br>");

document.write(false || true)

}

gfg();

</script>

false

true

true

true

## **Grammar**

JavaScript grammar is very similar to the C family. There are a few points worth mentioning:

* Identifiers can have Unicode characters, but they cannot be one of the reserved words.
* Comments are commonly // or /\* \*/, while many other scripting languages like Perl, Python, and Bash use #.
* Semicolons are optional in JavaScript — the language automatically inserts them when needed. However, there are certain caveats to watch out, since unlike Python, semicolons are still part of the syntax.

## **Control structures**

JavaScript has a similar set of control structures to other languages in the C family. Conditional statements are supported by if and else; you can chain them together:

let name = "kittens";

if (name === "puppies") {

name += " woof";

} else if (name === "kittens") {

name += " meow";

} else {

name += "!";

}

name === "kittens meow";

JavaScript has while loops and do...while loops. The first is good for basic looping; the second is for loops where you wish to ensure that the body of the loop is executed at least once:Example 1: Display Numbers from 1 to 5

// program to display numbers from 1 to 5

// initialize the variable

let i = 1, n = 5;

// while loop from i = 1 to 5

while (i <= n) {

console.log(i);

i += 1;

}

1

2

3

4

5

Example 3: do while, Display Numbers from 1 to 5

// program to display numbers

let i = 1;

const n = 5;

// do...while loop from 1 to 5

do {

console.log(i);

i++;

} while(i <= n);

1

2

3

4

5

JavaScript also contains two other prominent for loops: for, for...of, loops through the values of an iterable object, and for...in, which loops through the properties of an object.

//for loop

let text = "";

for (let i = 0; i < 5; i++) {

text += i + " ";

}

console.log(text);

Output

1 2 3 4 5

//for in

const numbers = [45, 4, 9, 16, 25];

let text = "";

for (let x in numbers) {

text += numbers[x]+"<br>";

}

Output

45

4

9

16

25

The switch statement can be used for multiple branches based on equality checking.

switch (action) {

case "draw":

drawIt();

break;

case "eat":

eatIt();

break;

default:

doNothing();

}

Similar to C, case clauses are conceptually the same as labels, so if you don't add a break statement, execution will "fall through" to the next level. However, they are not actually jump tables — any expression can be part of the case clause, not just string or number literals, and they would be evaluated one-by-one until one equals the value being matched. Comparison takes place between the two using the === operator.

## **Errors**

JavaScript errors are handled using the try...catch statement.

The try statement defines a code block to run (to try).

The catch statement defines a code block to handle any error.

The finally statement defines a code block to run regardless of the result.

The throw statement defines a custom error.

In general, you can't tell the type of the error you just caught, because anything can be thrown from a throw statement. However, you can usually assume it's an Error instance, as is the example below. There are some subclasses of Error built-in, like TypeError and RangeError, that you can use to provide extra semantics about the error. There's no conditional catch in JavaScript — if you only want to handle one type of error, you need to catch everything, identify the type of error using instanceof, and then rethrow the other cases.

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Error Handling</h2>

<p>How to use <b>catch</b> to display an error.</p>

<p id="demo"></p>

<script>

try {

adddlert("Welcome guest!");

}

catch(err) {

document.getElementById("demo").innerHTML = err.message;

}

/\*

function adddlert (str){

document.getElementById("demo").innerHTML= str;

}

\*/

</script>

</body>

</html>

If an error is uncaught by any try...catch in the call stack, the program will exit.

## **Objects**

JavaScript objects can be thought of as collections of key-value pairs.

JavaScript objects are hashes. Unlike objects in statically typed languages, objects in JavaScript do not have fixed shapes — properties can be added, deleted, re-ordered, mutated, or dynamically queried at any time.

Spaces and line breaks are not important. An object definition can span multiple lines:

Code:

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Objects</h2>

<p id="demo"></p>

<script>

// Create an object:

const person = {

firstName: "John",

lastName: "Doe",

age: 50,

eyeColor: "blue"

};

// Display some data from the object:

document.getElementById("demo").innerHTML = person.firstName + " is " + person.age + " years old.";

</script>

</body>

</html>

Output

John is 50 years old.

Object properties can be accessed using dot (.) or brackets ([]). When using the dot notation, the key must be a valid identifier. Brackets, on the other hand, allow indexing the object with a dynamic key value.

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Objects</h2>

<p>There are two different ways to access an object property.</p>

<p>You can use person.property or person["property"].</p>

<p id="demo"></p>

<script>

// Create an object:

const person = {

firstName: "John",

lastName : "Doe",

id : 5566

};

//access property using .

/\*// Display some data from the object:

document.getElementById("demo").innerHTML =

"Using . "+person.firstName + " " + person.lastName;

\*/

// access property using []

// Display some data from the object:

document.getElementById("demo").innerHTML =

"Using [] "+person["firstName"] + " " + person["lastName"];

</script>

</body>

</html>

Output

## **JavaScript Objects**

There are two different ways to access an object property.

You can use person.property or person["property"].

Using [] John Doe

## **Arrays**

Arrays in JavaScript are actually a special type of object. They work very much like regular objects (numerical properties can naturally be accessed only using [] syntax) but they have one magic property called length. This is always one more than the highest index in the array.

Arrays are usually created with array literals:

const a = ["dog", "cat", "hen"];

a.length; // 3

JavaScript arrays are still objects — you can assign any properties to them, including arbitrary number indices. The only "magic" is that length will be automatically updated when you set a particular index.

const a = ["dog", "cat", "hen"];

a[100] = "fox";

console.log(a.length); // 101

console.log(a); // ['dog', 'cat', 'hen', empty × 97, 'fox']

The array we got above is called a sparse array because there are uninhabited slots in the middle, and will cause the engine to deoptimize it from an array to a hash table. Make sure your array is densely populated!

Out-of-bounds indexing doesn't throw. If you query a non-existent array index, you'll get a value of undefined in return:

const a = ["dog", "cat", "hen"];

console.log(typeof a[90]); // undefined

Arrays can have any elements and can grow or shrink arbitrarily.

const arr = [1, "foo", true];

arr.push({});

// arr = [1, "foo", true, {}]

Arrays can be iterated with the for loop, as you can in other C-like languages:

for (let i = 0; i < a.length; i++) {

// Do something with a[i]

}

Or, since arrays are iterable, you can use the for...of loop, which is synonymous to C++/Java's for (int x : arr) syntax:

for (const currentValue of a) {

// Do something with currentValue

}

Code:

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Array.forEach()</h2>

<p>Calls a function once for each array element.</p>

<p id="demo"></p>

<script>

const numbers = [45, 4, 9, 16, 25];

let txt = "";

numbers.forEach(myFunction);

document.getElementById("demo").innerHTML = txt;

function myFunction(value, index, array) {

txt += value + "<br>";

}

</script>

</body>

</html>

Output

**JavaScript Array.forEach()**

Calls a function once for each array element.

45

4

9

16

25

Arrays come with a plethora of array methods.

### **map()**

Many of them would iterate the array — for example, map() would apply a callback to every array element, and return a new array:

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.

Code:

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Array.map()</h2>

<p>Creates a new array by performing a function on each array element.</p>

<p id="demo"></p>

<script>

const numbers1 = [45, 4, 9, 16, 25];

const numbers2 = numbers1.map(myFunction);

document.getElementById("demo").innerHTML = numbers2;

function myFunction(value, index, array) {

return value \* 2;

}

</script>

</body>

</html>

Output

**JavaScript Array.map()**

Creates a new array by performing a function on each array element.

90,8,18,32,50

Note that the function takes 3 arguments:

The item value

The item index

The array itself

When a callback function uses only the value parameter, the index and array parameters can be omitted to:

const numbers1 = [45, 4, 9, 16, 25];

const numbers2 = numbers1.map(myFunction);

function myFunction(value) {

return value \* 2;

}

### **Destructuring**

To illustrate destructuring, we'll make a sandwich. Do you take everything out of the refrigerator to make your sandwich? No, you only take out the items you would like to use on your sandwich. Destructuring is exactly the same. We may have an array or object that we are working with, but we only need some of the items contained in these. Destructuring makes it easy to extract only what is needed.

const vehicles = ['mustang', 'f-150', 'expedition'];

const [car, truck, suv] = vehicles;

When destructuring arrays, the order that variables are declared is important. If we only want the car and suv we can simply leave out the truck but keep the comma:

const vehicles = ['mustang', 'f-150', 'expedition'];

const [car,, suv] = vehicles;

Destructuring comes in handy when a function returns an array:

<!DOCTYPE html>

<html>

<body>

<script>

function calculate(a, b) {

const add = a + b;

const subtract = a - b;

const multiply = a \* b;

const divide = a / b;

return [add, subtract, multiply, divide];

}

const [add, subtract, multiply, divide] = calculate(4, 7);

document.write("<p>Sum: " + add + "</p>");

document.write("<p>Difference " + subtract + "</p>");

document.write("<p>Product: " + multiply + "</p>");

document.write("<p>Quotient " + divide + "</p>");

</script>

</body>

</html>

Output:

Sum: 11

Difference -3

Product: 28

Quotient 0.5714285714285714

### **Spread Operator**

The JavaScript spread operator (...) allows us to quickly copy all or part of an existing array or object into another array or object.

Example:

const numbersOne = [1, 2, 3];

const numbersTwo = [4, 5, 6];

const numbersCombined = [...numbersOne, ...numbersTwo];

The spread operator is often used in combination with destructuring.

Example

Assign the first and second items from numbers to variables and put the rest in an array:

<!DOCTYPE html>

<html>

<body>

<script>

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

const [one, two, ...rest] = numbers;

document.write("<p>" + one + "</p>");

document.write("<p>" + two + "</p>");

document.write("<p>" + rest + "</p>");

</script>

</body>

</html>

Output:

1

2

3,4,5,6

## **Functions**

Functions are the core component in understanding JavaScript. The most basic function declaration looks like this:

function add(x, y) {

const total = x + y;

return total;

}

A JavaScript function can take 0 or more parameters. The function body can contain as many statements as you like and can declare its own variables which are local to that function. The return statement can be used to return a value at any time, terminating the function. If no return statement is used (or an empty return with no value), JavaScript returns undefined.

Functions can be called with more or fewer parameters than it specifies. If you call a function without passing the parameters it expects, they will be set to undefined. If you pass more parameters than it expects, the function will ignore the extra parameters.

add(); // NaN

// Equivalent to add(undefined, undefined)

add(2, 3, 4); // 5

// added the first two; 4 was ignored

There are a number of other parameter syntaxes available. For example, the rest parameter syntax allows collecting all the extra parameters passed by the caller into an array.

function avg(...args) {

let sum = 0;

for (const item of args) {

sum += item;

}

return sum / args.length;

}

avg(2, 3, 4, 5); // 3.5

In the above code, the variable args holds all the values that were passed into the function.

### **Anonymous functions**

JavaScript lets you create anonymous functions — that is, functions without names. In practice, anonymous functions are typically used as arguments to other functions, immediately assigned to a variable that can be used to invoke the function, or returned from another function.

// Note that there's no function name before the parentheses

const avg = function (...args) {

let sum = 0;

for (const item of args) {

sum += item;

}

return sum / args.length;

};

That makes the anonymous function invocable by calling avg() with some arguments — that is, it's semantically equivalent to declaring the function using the function avg() {} declaration syntax.

code:

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Functions</h2>

<p>After a function has been stored in a variable,

the variable can be used as a function:</p>

<p id="demo"></p>

<script>

const x = function (a, b) {return a \* b};

document.getElementById("demo").innerHTML = x(4, 3);

</script>

</body>

</html>

output

**JavaScript Anonymous Function**

After a function has been stored in a variable, the variable can be used as a function:

12

There's another way to define anonymous functions — using an arrow function expression. ES6 introduced arrow function expressions that provide a shorthand for declaring anonymous functions.

// Note that there's no function name before the parentheses

const avg = (...args) => {

let sum = 0;

for (const item of args) {

sum += item;

}

return sum / args.length;

};

// You can omit the `return` when simply returning an expression

const sum = (a, b, c) => a + b + c;

code:

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Functions</h1>

<h2>The Arrow Function</h2>

<p>This example shows an Arrow Function with a parameter.</p>

<p id="demo"></p>

<script>

let hello = "";

hello = (val) => "Hello " + val;

document.getElementById("demo").innerHTML = hello("Universe!");

</script>

</body>

</html>

Output

**JavaScript Functions**

The Arrow Function

This example shows an Arrow Function with a parameter.

Hello Universe!

### **IIFE**

There's another way that anonymous functions can be useful: it can be simultaneously declared and invoked in a single expression, called an Immediately invoked function expression (**IIFE**):

(function () {

// …

})();

(function(value){

var greet = 'Hello';

console.log(greet+ ' ' + value);

})('IIFEs');

In above example when javascript engine execute above code it will create global execution context when it sees code and create function object in memory for IIFE. And when it reaches on function line, the function is Invoked and a new execution context is created on the fly and so greet variable goes into that function execution context not into the global this is what makes it unique. This ensures that code inside IIFE does not interfere with other code or be interfered by another code and so code is safe.

Note: Whenever the JavaScript engine receives a script file, it first creates a default Execution Context known as the Global Execution Context (GEC) . The GEC is the base/default Execution Context where all JavaScript code that is not inside of a function gets executed. For every JavaScript file, there can only be one GEC.

### **Functions are first-class objects**

JavaScript functions are first-class objects. This means that they can be assigned to variables, passed as arguments to other functions, and returned from other functions. In addition, JavaScript supports closures out-of-the-box without explicit capturing, allowing you to conveniently apply functional programming styles.

## **Classes**

JavaScript is a prototype-based language — an object's behaviors are specified by its own properties and its prototype's properties. However, with the addition of classes, the creation of hierarchies of objects and the inheritance of properties and their values are much more in line with other object-oriented languages such as Java. In this section, we will demonstrate how objects can be created from classes.

Declaring a class

Classes are usually created with class declarations.

class MyClass {

// class body...

}

Constructing a class

After a class has been declared, you can create instances of it using the new operator.

const myInstance = new MyClass();

console.log(myInstance.myField); // 'foo'

myInstance.myMethod();

### **Constructor**

Perhaps the most important job of a class is to act as a "factory" for objects.

As an example, we would create a class called Color, which represents a specific color. Users create colors through passing in an RGB triplet. Open your browser's devtools, paste the below code into the console, and then create an instance:

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Object Constructors</h2>

<p id="demo"></p>

<script>

// Constructor function for Person objects

function Person(first, last, age, eye) {

this.firstName = first;

this.lastName = last;

this.age = age;

this.eyeColor = eye;

}

// Create a Person object

const myFather = new Person("John", "Doe", 50, "blue");

// Display age

document.getElementById("demo").innerHTML =

"My father is " + myFather.age + ".";

</script>

</body>

</html>

Output:

JavaScript Functions

The Arrow Function

This example shows an Arrow Function with a parameter.

My father is 50.

JavaScript Object Constructors

Please refer topics on Instance methods, Private fields, Accessor fields, Public fields for indepth coverage about classes

### **Extends and inheritance**

A key feature that classes bring about (in addition to ergonomic encapsulation with private fields) is inheritance, which means one object can "borrow" a large part of another object's behaviors, while overriding or enhancing certain parts with its own logic.

For example, suppose our Color class now needs to support transparency. We may be tempted to add a new field that indicates its transparency:

class Color {

#values;

constructor(r, g, b, a = 1) {

this.#values = [r, g, b, a];

}

get alpha() {

return this.#values[3];

}

set alpha(value) {

if (value < 0 || value > 1) {

throw new RangeError("Alpha value must be between 0 and 1");

}

this.#values[3] = value;

}

}

However, this means every instance — even the vast majority which aren't transparent (those with an alpha value of 1) — will have to have the extra alpha value, which is not very elegant. Plus, if the features keep growing, our Color class will become very bloated and hard to maintain.

Instead, in object-oriented programming, we would create a derived class. The derived class has access to all public properties of the parent class. In JavaScript, derived classes are declared with an extends clause, which indicates the class it extends from.

class ColorWithAlpha extends Color {

#alpha;

constructor(r, g, b, a) {

super(r, g, b);

this.#alpha = a;

}

get alpha() {

return this.#alpha;

}

set alpha(value) {

if (value < 0 || value > 1) {

throw new RangeError("Alpha value must be between 0 and 1");

}

this.#alpha = value;

}

}

There are a few things that have immediately come to attention. First is that in the constructor, we are calling super(r, g, b). It is a language requirement to call super() before accessing this. The super() call calls the parent class's constructor to initialize this — here it's roughly equivalent to this = new Color(r, g, b). You can have code before super(), but you cannot access this before super() — the language prevents you from accessing the uninitialized this.

## **Asynchronous programming**

JavaScript is single-threaded by nature. There's no paralleling; only concurrency. Asynchronous programming is powered by an event loop, which allows a set of tasks to be queued and polled for completion.

There are three idiomatic ways to write asynchronous code in JavaScript:

Callback-based (such as setTimeout())

Promise-based

async/await, which is a syntactic sugar for Promises

For example, here's how a file-read operation may look like in JavaScript:

// Callback-based

fs.readFile(filename, (err, content) => {

// This callback is invoked when the file is read, which could be after a while

if (err) {

throw err;

}

console.log(content);

});

// Code here will be executed while the file is waiting to be read

// Promise-based

fs.readFile(filename)

.then((content) => {

// What to do when the file is read

console.log(content);

}).catch((err) => {

throw err;

});

// Code here will be executed while the file is waiting to be read

// Async/await

async function readFile(filename) {

const content = await fs.readFile(filename);

console.log(content);

}

The core language doesn't specify any asynchronous programming features, but it's crucial when interacting with the external environment — from asking user permissions, to fetching data, to reading files. Keeping the potentially long-running operations async ensures that other processes can still run while this one waits — for example, the browser will not freeze while waiting for the user to click a button to grant permission.

### **Asynchronous Example**

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript async / await</h2>

<h1 id="demo"></h1>

<script>

async function myDisplay() {

let myPromise = new Promise(function(resolve, reject) {

resolve("Hello world !!");

});

document.getElementById("demo").innerHTML = await myPromise;

}

myDisplay();

</script>

</body>

</html>

Output

**JavaScript async / await**

**Hello world!!**

The two arguments (resolve and reject) are pre-defined by JavaScript. We will not create them, but call one of them when the executor function is ready. Very often we will not need a reject function.

### **without reject**

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript async / await</h2>

<h1 id="demo"></h1>

<script>

async function myDisplay() {

let myPromise = new Promise(function(resolve) {

resolve("Hello world !!");

});

document.getElementById("demo").innerHTML = await myPromise;

}

myDisplay();

</script>

</body>

</html>

Output

**JavaScript async / await without**

**Hello world !!**

### **Waiting for a Timeout**

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript async / await</h2>

<p>Wait 3 seconds (3000 milliseconds) for this page to change.</p>

<h1 id="demo"></h1>

<script>

async function myDisplay() {

let myPromise = new Promise(function(resolve) {

setTimeout(function() {resolve("Hello world !!");}, 3000);

});

document.getElementById("demo").innerHTML = await myPromise;

}

myDisplay();

</script>

</body>

</html>

Output

**JavaScript async / await**

Wait 3 seconds (3000 milliseconds) for this page to change.

Hello world !!

### **Callback function**

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Callbacks</h2>

<p id="demo"></p>

<script>

function filter(numbers, callback) {

let results = [];

for (const number of numbers) {

if (callback(number)) {

results.push(number);

}

}

return results;

}

let numbers = [1, 2, 4, 7, 3, 5, 6];

//anonymous function to the filter() function

let oddNumbers = filter(numbers, function (number) {

return number % 2 != 0;

});

//ES6 arrow function

//let oddNumbers = filter(numbers, (number) => number % 2 != 0);

document.getElementById("demo").innerHTML=oddNumbers;

</script>

</body>

</html>

Output

**JavaScript Callbacks**

1,7,3,5

## **Further Exploration**

This course offers a basic insight into various JavaScript features. If you want to learn more about the language itself and the nuances with each feature, you can read the JavaScript guide from [here](https://javascript.info/).

There are some essential parts of the language that we have omitted due to time, focus on React, but you can explore on your own:

Inheritance and the prototype chain

Closures

Regular expressions

Iteration

Modules