# 1. AJAX 🍎

## Getting things from the Internet

There are a lot of services and sites on the Internet that allow us to use some data or information that they hold such as the weather, sports scores, statistics about the environment, movies, music, etc. To get that information we have to send a request for the first to the site or service that offers the data in the first place. If we are allowed to make that request we get a response with a package of the data we requested in a format called JSON. If we are not allowed we get a response but with a status that describes why we didn't get our data. Through these requests, we communicate with these sites and services whether we want to get something or add something

## What is JSON

JSON is one of the most commonly used formats in which we send or get packages to the Internet. It's a simple text file that contains structured information. When you see the structure it is very similar to a JavaScript object. It consists of keys (in JS Objects, properties) and values. The values can be of any type such as numbers, strings boolean, array, or another object

### A JavaScript Object

let someObject = {  trainer: "Trainer name",

  assistant: "Assistant name",

  students: [    "Bob",    "Samantha",    "Chris",    "Jill",    "Greg"  ],

  academy: "Code" };

### A json file

```json {

  "trainer": "Trainer name",

  "assistant": "Assistant name",

  "students": [    "Bob",    "Samantha",    "Chris",    "Jill",    "Greg"  ],

  "academy": "Code" }

As we can see the two are very similar. But when we get JSON files like the one above from some request that we made, we can't use them right away. Like we said it is a simple text file, and thus in JavaScript, it is just a simple string. To get the information we need we first need to convert it to a JavaScript object. We do that with the \*\*JSON.parse()\*\* method

// Parse the json text file into JS object let jsObject = JSON.parse(myJsonResponse);

// We can use it like object now console.log(jsObject.trainer);

// We can create objects in to json strings as well let newJson = JSON.stringify(jsObject);

So when the JSON string is parsed we can treat it like a normal object. We can also convert it back if we need to send it to the service or site back. If you are not sure that your JSON is valid you can use some of the online sites that validate JSON objects such as https://jsonlint.com/

## What is AJAX

Making a request is always done through an HTTP protocol call. That is the standard call that we do when we request for any site to be opened. With this call, we can also request things from sites and services on the Internet. But for every call, we need to reload our page. To not reload the page every time we make a call, we use AJAX. AJAX is a technique with which we can send requests and get responses dynamically, without reloading everything. So how do we make a call with AJAX?

1. Find a service or site that allows requests for some data

2. Get the URL address from that site or service where we will make the calls

3. Write AJAX call and code to handle the response in the JavaScript script

4. Write code as if you already have the response in a success method

5. Write code as if you didn't get what you desired in an error method

We can make the AJAX call using plain JavaScript, JQyery, or with the new JavaScript feature called fetch

### Plain JavaScript

let xhr = new XMLHttpRequest(); xhr.onload = function () {   if (xhr.status >= 200 && xhr.status < 300) {    console.log('The request succeeded!');

    let resultParsed = JSON.parse(xhr.response);   console.log(resultParsed);  } else {    console.log('The request failed!');   console.log(xhr.responseText);  } };

xhr.open('GET', 'https://raw.githubusercontent.com/qa-codecademy/mkwd13-04-ajs/refs/heads/main/shared\_data/students.json'); xhr.send();

### JQuery

$.ajax({  url: "https://raw.githubusercontent.com/qa-codecademy/mkwd13-04-ajs/refs/heads/main/shared\_data/students.json",

  success: function (response) {    console.log('The request succeeded!');

    let resultParsed = JSON.parse(response);     console.log(resultParsed);  },

  error: function (response) {    console.log('The request failed!');     console.log(response.responseText);  } });

### Fetch

fetch('https://raw.githubusercontent.com/qa-codecademy/mkwd13-04-ajs/refs/heads/main/shared\_data/students.json')

    .then(function (response) {      return response.json();    })

    .then(function (myJson) {      console.log(myJson);    });

## APIs

API pages are pages that allow you to make requests and get some data from them. You send a request to an API and it returns the data that you need. You can request for the weather on some weather APIs, request for info about books, music, sports matches, etc. We request the information that we need with AJAX calls. Some APIs require a special key to be sent for them to return some data. A special key is usually acquired if you create an account on the page that is sometimes free and others paid. We can find out if the API requires a key or not through the documentation of the API. Some APIs don't require any key or account and payment. Here are some:

### SPACE INFO API

\* \*\*Link:\*\* http://open-notify.org/

\* \*\*How to use it:\*\*

    \* Write the base URL: http://api.open-notify.org/

    \* Write iss-now.json after the URL to see where the international space station is at right now

    \* Write astros.json after the URL to see how many astronauts are on the ISS right now

\* \*\*Example:\*\* http://api.open-notify.org/iss-now.json

### STAR WARS API

\* \*\*Link:\*\* https://swapi.dev/

\* \*\*How to use it:\*\*

    \* Write the base url: https://swapi.dev/api/

    \* Write a category: (Planets, Spaceships, Vehicles, People, Films and Species)

    \* Write an id ( number )

\* \*\*Example:\*\* https://swapi.dev/api/people/1/

### Bitcoin Price

\* \*\*Link:\*\* https://api.coindesk.com/v1/bpi/currentprice.json

\* \*\*How to use it:\*\*

    \* The link is by itself the endpoint. Just make a request and you will get data on Bitcoin price

\* \*\*Example:\*\* https://api.coindesk.com/v1/bpi/currentprice.json

### Cat API

\* \*\*Link:\*\* https://catfact.ninja/

\* \*\*How to use it:\*\*

    \* Write the base URL: https://catfact.ninja/

    \* Write next to its fact, facts, or breeds depending on what you need and make the request

\* \*\*Example:\*\* https://catfact.ninja/fact

### Dog API

\* \*\*Link:\*\* https://dog.ceo/dog-api/

\* \*\*How to use it:\*\*

    \* Write the base URL: https://dog.ceo/api/

    \* Write next to it breeds/list/all to list all breeds

    \* Write next to it breeds/image/random to get a random dog image

    \* Write next to it breed/ADD-BREED-NAME-HERE/images/random to get a random dog image from a certain breed

\* \*\*Example:\*\* https://dog.ceo/api/breeds/image/random

### Chuck Norris Jokes

\* \*\*Link:\*\* https://api.chucknorris.io/

\* \*\*How to use it:\*\*

    \* Write the base url: https://api.chucknorris.io/

    \* Write jokes/random next to it and make a request

\* \*\*Example:\*\* https://api.chucknorris.io/jokes/random

### What to do when bored

\* \*\*Link:\*\* https://www.boredapi.com/

\* \*\*How to use it:\*\*

    \* Write the base URL: https://www.boredapi.com/api/

    \* Write activity next to it and request to get a random activity to do

\* \*\*Example:\*\* https://www.boredapi.com/api/activity

## Bonus - Query parameters in URL

\* Some APIs require some parameters to be passed for them to work. To pass things through the URL we use Query  parameters. We write query parameters by first adding a ? after the link that we typed and then write the name of the  parameter = the value. All parameters are divided by an &

    \* \*\*Example:\*\* We have an API that returns jokes for specific language [cs, de, en, es, fr, pt]. The link

      is: https://v2.jokeapi.dev/

\* Our link should be: https://v2.jokeapi.dev/joke/Any?lang=en

## Extra Materials 📘

\* [Extra reading for making HTTP requests](https://www.freecodecamp.org/news/here-is-the-most-popular-ways-to-make-an-http-request-in-javascript-954ce8c95aaa/)

\* [DummyJson](https://dummyjson.com/)

\* [Huge list of public APIs](https://github.com/public-apis/public-apis)

# Functions pt1 🍋

## What are functions

Functions are named blocks of code that we can store in memory for later use. We always try and use functions and wrap pieces of logic in functions as much as we can. This is a good practice because functions make our code:

\* More organized

\* More readable

\* Reusable ( We can use one piece of code multiple times)

\* Scalable ( We can build upon it easy )

Functions have two phases. The declaration phase ( when we write the function ) and the execution phase ( when we call the function )

### Function declaration

When we write a function that is the moment where we declare it. When we declare a function we tell JavaScript that we have a block of code that we named and that it is ready to be executed when somebody calls it. Keep in mind that when we declare a function we just write it in memory. \*\*THE CODE IS NOT EXECUTED\*\*

// Declaration of a function function getFullName(firstName, lastName){    return result = `${firstName} ${lastName}`; }

### Parameters

Everything that a function needs should be passed( given ) to the function. We do that with parameters. Parameters are a point of entry of the function. They are variables that are acting as placeholders for values that we anticipate the caller to enter when calling the function. The parameters in the function above are firstName and lastName. When the person calls the function they will enter the first name in the first position and last name in the second position. That is how javascript knows which one is which, by the positions. Those names will then be replaced by the firstName and lastName placeholders in the whole function. If we call the function with more values than parameters the extra will be ignored. If we call a function with fewer values than parameters the ones that don't have a value will be undefined

### Return

Return is a keyword that means two things:

\* End of the execution of the function

\* Giving back value to the outside world ( the place that it was called )

The return keyword is the single point of the output of the function. If we want to get something from the function we must put that thing in return. Any value that is not in the return will be inaccessible to the outside world. Because it returns something outside of the function it means it is done with executing and that anything after the return keyword will not execute. If we write 10 lines of code after the return, they will never be executed

### Function execution

When we declare a function, none of the code is executed. If we want to execute the code we have to call the function. When we call a function on a certain line, the code execution goes to the line where the function is declared, it runs the code there and then gets back to the line in which it was executed with a result from the function. In simple terms when we call a function it will run all the code and give us a result that will be added in the same spot where we called the function. That means that if we call a function in a variable the function will first execute and whatever it returned will be stored in the variable. ( the whole function will not be stored in the variable after calling it )

#### Calling of a function ( the result will not be used, nothing will happen )

getFullName("Eric", "Praline");

#### Calling of a function in variable initialization

// The function will execute and the result then will be stored in the variable

let fullName = getFullName("Eric", "Praline");

#### Calling a function inside if statement ( if the full name is longer than 20 letters )

if(getFullName("Eric", "Praline The Third").length >= 20){    console.log("That's a long name!"); }

#### Calling a function inside another function

function printPersonDetails(firstName, lastName, age, address){    return `The person ${getFullName(firstName, lastName)} age ${age} lives on ${address}!`; }

#### How not to call function inside a function

// This will not work because you are declaring a function inside this function, not calling it.

function printPersonDetails(firstName, lastName, age, address){    let fullName;

    function getFullName(firstName, lastName){        fullName = `${firstName} ${lastName}`;    }

    return `The person ${fullName} age ${age} lives on ${address}!`;}

### Parameters vs Arguments

Parameters and Arguments are two different terms in programming. Like we said parameters are the variables which we anticipate values in a function ( Placeholders for the caller values ). The variables ( placeholders ) themselves are called parameters. But the values that the caller passes are called a different name. They are called arguments. So the variables that wait for a value are called parameters and the values given later are called arguments

// num1 and num2 are parameters function sum(num1, num2) {  return num1 + num2; }

// 2 and 3 are arguments console.log(sum(2, 3));

## Anonymous functions

Anonymous functions or more precisely function expressions are functions that are not declared. They are written on the spot where we want them to be executed and when the code runs and gets to the line where they are written, they immediately execute the code that they hold and return a value at the same spot. We use these functions every time we need a function to be executed at some point in time once

#### Anonymous function in a variable

let greeting = function (name) {  return `Hello ${name}`; }

// The variable becomes the function greeting("Bob");

#### Anonymous function in an event listener

let button = document.getElementsByTagName("button")[0];

button.addEventListener("click", function () {  console.log("Button is clicked!"); })

## Arrow Functions

Arrow functions are a somewhat new addition to the javascript language. They are a shorter version of the anonymous functions we discussed above. We can use them to simplify and shorten our code. Because they were added in the javascript standard in 2015 they may not be supported in very old browsers ( ex: internet explorer ). There are almost no differences between the two except that the one is shorter than the other. Arrow functions don't need \*\*{ }\*\* if they have only one expression or even a return. If you have more than one expression you will need { } and return. Parameters are put in \*\*( )\*\* only if there is more than one. One parameter doesn't need ( )

#### no parameter, one expression

// Anonymous function let logSomething = function () {  console.log("Hello there!");}

// Arrow function let logSomething = () => console.log("Hello there!");

#### one parameter, one expression

// Anonymous function let sumWith10 = function (num) {  return num + 10; }

// Arrow function let sumFunc = num => num + 10;

### multiple parameters, one expression

// Anonymous function let sumFunc = function (num1, num2) {  return num1 + num2; }

// Arrow function let sumFunc = (num1, num2) => num1 + num2;

#### multiple arguments, multiple expressions

// Anonymous function let sumAndLogFunc = function (num1, num2) {  let result = num1 + num2;

  console.log(`The result is: ${result}`)  return result; }

// Arrow function let sumFunc = (num1, num2) => {  let result = num1 + num2;

  console.log(`The result is: ${result}`)  return result; };

#### Arrow function in an event listener

let button = document.getElementsByTagName("button")[0];

button.addEventListener("click", () => {  console.log("Button is clicked!"); })

## Self invoked functions

Self-invoked functions or immediately-invoked function expressions (IIFE) are functions that are not just written on the spot but also called and executed at the same time. Basically what we usually do to create a function like this, is we create an anonymous function at some point in our code and if we want to call it and execute it immediately we put brackets around it and brackets after it just like calling a function. This makes our anonymous functions handy not only when we want to declare at the spot but call it on the spot as well. These functions help us organize and use variables in enclosed spaces where they will not affect or pollute the rest of our code. We can write code in our self-invoked functions that will run at the moment and we will be sure that it will be contained in that function at that moment only.

\*\*Note:\*\* We can name self invoked functions but there is no point in doing so most of the time

// Anonymous functions let sayHello = function (name) {  console.log(`Hello there ${name}!`)}

let sayHelloArrow = name => console.log(`Hello there ${name}!`);

sayHello("Bob"); // calling of anonymous function

// Self invoked functions (function (name) {  console.log(`Hello there ${name}!`) })("bob");

(name => console.log(`Hello there ${name}!`))("bob");

### Using self invoked functions

// Self invoked function result as variable let fullName = ((first, last) => `${first} ${last}`)("Cave", "Johnson");

// Self invoked function result as an argument function sum(num1, num2) {  return num1 + num2; }

let number1 = 25; let number2 = "15"; let result = sum(number1, (num => parseInt(num))(number2));

// Self invoked function result as a return of another function

function sayHello(inputName) {  return (name => {    if (name.length <= 0) {      return "No name"    } else if (name.length < 2) {      return "Too short"    } else {      return `Hello there ${name}`    }  })(inputName) }

console.log(sayHello("Bob"));

## Recursion

Functions can call other functions inside of them. But functions can also call themselves inside of their block of code. And because the function is calling itself and itself contains another call to the same function we can easily tell that recursion can have some unwanted effects right away. That is why when we are doing recursion ( calling the same function inside of itself) we always make a condition, so at some point, our function can stop calling itself.

// With function ( standard ) function sumTo(num) {  if (num === 0) {    return 0  }  return num + sumTo(num - 1)}

// With an arrow function ( shorter ) let sumTo = num => num === 0 ? 0 : num + sumTo(--num)

## Scope

The scope is nothing more than a barrier. This barrier doesn't let anyone access the code in the barrier from the

outside but allows the code that is inside it to access things outside of it. In JavaScript, there are three types of

scopes. Global, Function, and Block scope

### Function scope ( local scope )

In JavaScript, every function creates a scope around its code. So everything inside the function \*\*{ }\*\* and the

parameters are included in the scope of a function. Because everything is in the function scope, we can use variables that are declared inside of it only in the function. For the outside, those variables do not exist. When we declare a variable with the keyword \*\*var\*\* inside of a function, that variable is bound only to that function. Nobody can call it from the outside, because to the outside, it doesn't exist

function getFullName(firstName, lastName) {  var result = `${firstName} ${lastName}`;  console.log(result);}

getFullName("Cave", "Johnson"); // Will log Cave Johnson in the console

console.log(result); // Will throw an Uncaught Reference Error

// It throws this error because it can't find a variable named result

![Scopes Example](https://github.com/Drakso/AJS2019/blob/master/Class3/Img/scopeImage2.PNG?raw=true)

### Block scope

Block scope is a new barrier introduced in ES6. It is another type of scope that coexists with the function scope. So if the function scope created a barrier between the function and the outside world, the block scope creates a barrier between a block of code and the outside world. A block of code is code that is enclosed in \*\*{ }\*\*. When we declare a variable with the keyword \*\*let\*\* inside any block of \*\*{ }\*\* that variable is bound only to that block. Nobody can call it from outside of that block. To the outside, this variable does not exist

function getFullName(firstName, lastName) {  if (firstName.length > 1 && lastName.length > 1) {

    let blockResult = `${firstName} ${lastName}`;    var functionResult = `${firstName} ${lastName}`;

    console.log(blockResult); } // Will log full name in the console

  console.log(functionResult); // Will log full name in the console

  console.log(blockResult); } // Will throw Uncaught Reference Error

getFullName("Cave", "Johnson"); // Will log Cave Johnson in the console

console.log(blockResult); // Will throw an Uncaught Reference Error

console.log(functionResult); // Will throw an Uncaught Reference Error

functionResult can be accessed anywhere in the function because it is declared with \*\*var\*\* and is contained only in the function \*\*{ }\*\*. blockResult can be accessed only in the if block ( inside the if \*\*{ }\*\* ) because it is declared with \*\*let\*\*. It is always a good practice for our variables to be contained in the block that they are in so that is why let is always a smarter option to use

### Global scope

The global scope is the space of our script. Everything that is not inside a function or block scope, lives in the

global scope. That is why sometimes you might hear people calling the variables declared outside of all functions,

global variables

let first = "Cave"; let last = "Johnson";

function getFullName(firstName, lastName) {  if (firstName.length > 1 && lastName.length > 1) {

    let result = `${firstName} ${lastName}`;    console.log(result); // Will log full name in the console  }}

getFullName(first, last); // Will log Cave Johnson in the console

console.log(first); // Will log Cave in the console

console.log(last); // Will log Johnson in the console

console.log(result); // Will throw an Uncaught Reference Error

#### Multiple functions scopes example

let ten = 10; // Global scope function sumPlusOne(num1, num2) {  let one = 1; // Function sumPlusOne scope

  console.log(num1 + num2 + one);

  function add5(number) {    console.log(number + 5); // Function add5 scope  };   add5(one);

  function add10(number) {    console.log(number + ten); // Function add10 scope  };   add10(ten);}

sumPlusOne(ten, 7);

![Scopes Example Colored]

(https://github.com/Drakso/AJS2019/blob/master/Class3/Img/scopeImage1.PNG?raw=true)

### Hoisting

Declaring variables and functions in JavaScript are not treated as other pieces of code. There is a special event that

happens before the execution of the code. Before code is executed, all declarations of variables as well as all

declarations of functions, are moved at the top of the document. This of course is behind the scene and it ONLY happens for declarations. This means that if we declare and initialize, the initialization will not be moved to the top of the document. The initialization will stay on the line it was initially written and will execute when the code gets to that line. But the variable itself will be declared at the top, even if the initialization is on some other line. This is

called Hoisting. This process creates the possibility of calling functions for example, before they are declared. When you see the code, this would not make sense, but if you know what JavaScript does before code execution, then it makes perfect sense. We have to note that var, let and const all have different behavior when it comes to hoisting. Function declarations and function expressions also differ.

### Function declaration hoisting

// We call the function before we declare it in the code sayHello();

// This declaration is hoisted to the top before the code executes, so even if it seems like it is after the call, it is actually before function sayHello() {  console.log("Hello there user!");}

// This call will not work, since function expressions are not hoisted at the top. Only standard declarations.

sayHello(); let sayHello = () => console.log("Hello there user!");

### Variable declaration hoisting

// Hoisting with var

// This will print undefined, and not "hoisted is not defined"

// This is beause the declaration will be hoisted => var hoisted; console.log(hoisted);

// The hoisted = 5 initialization is not hoisted, so that is why we don't see 5 var hoisted = 5;

// Hoisting with let and const

// let and const do not allow for variables to be declared after they are used

console.log(hoisted); // This will show "hoisted is not defined" error let hoisted = 5;

// We can however do the declaration manually and this will give the same output as the var example

let hoisted; console.log(hoisted); // This will return undefined hoisted = 5;

## Extra materials 📘

\* [A good piece about functions](https://dev.to/howtocodejs/an-overview-of-javascript-functions-47id?utm\_source=digest\_mailer&utm\_medium=email&utm\_campaign=digest\_email)

\* [When to use arrow functions](https://medium.freecodecamp.org/when-and-why-you-should-use-es6-arrow-functions-and-when-you-shouldnt-3d851d7f0b26)

\* [Scopes explained](https://andy-carter.com/blog/variable-scope-in-modern-javascript)

\* [Self invoked functions and how to use them](https://idiallo.com/javascript/self-invoking-anonymous-function)

\* [Recursion in JS](https://medium.freecodecamp.org/recursion-in-javascript-1608032c7a1f)

# Functions pt2 🍊

## Functions as first class citizens

Functions are first-class citizens in the JavaScript language. But what does a first-class citizen mean? Well simply

put, if some entity in a programming language has the power to be used as any other entity in that language, it is

considered a first-class citizen. Why are functions in JavaScript considered first-class citizens? Because they can act and be used as any other entity such as:

\* Stored in a variable \* Stored in an array \* Used as an argument to another function

\* Used as a return value from another function \* Have properties like objects

\* Have methods like objects

If you don't believe it here are some crazy examples of what JavaScript functions are capable of:

### Storing function as a variable

let sayHello = function (name) {  console.log(`Hello there ${name}!`)};

// we call the function from the variable the same as we do with functions sayHello("Bob")

### Storing functions in an array and using them

let numberStatsFunctions = [  (num) => num > 0 ? "Positive" : "Negative",  (num) => num % 2 === 0 ? "Odd" : "Even",

  (num) => num >= 0 ? num.toString().length : num.toString().length – 1 ];

// We can call the functions from the array

// We can loop it with for as well

numberStatsFunctions[0](2); // Positive

numberStatsFunctions[1](2); // Even

numberStatsFunctions[2](2); // 1

### Using function as an argument

function calculator(calculateFunc, number1, number2) {  return calculateFunc(number1, number2); }

function sum(number1, number2) {  return number1 + number2 };

function difference(number1, number2) {  return number1 - number2 };

// We send functions sum and difference to the calculator function

// We can send anonymous functions as well

console.log(calculator(sum, 3, 6)); // 9 console.log(calculator(difference, 40, 12)); // 28

### Returning a function from another function

function calculator(operation) {  switch (operation) {    case  "+":      return function (number1, number2) {

        return number1 + number2      };      break;    case  "-":

      return function (number1, number2) {        return number1 - number2      };      break;    default:

      console.log("ERROR")      return null;  }}

// We call the calculator function and get back a function as a result let result = calculator("+");

// We can call the result function now console.log(result(2, 5)); // 7

// We can also call the function right away console.log(calculator("-")(7, 2)); // 5

### A function with properties and a method

function sayHello(name) {  return `Hello there ${name}!`;}

// Adding a properties to a function

sayHello.defaultName = "Bob"; sayHello.description = "This is a function that greets a person.";

// Getting property value from the function

sayHello(sayHello.defaultName); // Hello there Bob!

// Adding a method to a function sayHello.differentGreeting = function (name) {  return `Hi ${name}!`;}

// Calling the method from the function sayHello.differentGreeting("Bob"); // Hi Bob!

## Function arguments

Every functions has a collection of all the arguments sent to that function when it was called. Since the person calling the function can send as much arguments as he wants, it's important to know how to check how many arguments were passed before continuing with your execution

function longestString() {   console.log(arguments[0]) // first argument

  console.log(arguments[1]) // second argument   console.log(arguments[2]) // third argument

  let longest = '';  for (let i = 0; i < arguments.length; i++) {    if (arguments[i].length > longest.length) {

      longest = arguments[i];    }  }  return longest;}

## Higher-order functions

Higher-order functions are just functions that take other functions as arguments. With these functions, we can make our code more readable and more organized. We can combine higher-order functions to get results that would otherwise cost us 10+ lines of code, to be written and compacted into only one or two

### forEach

forEach is a function that accepts a function as an argument and it runs it for every element in an array. This function does not return anything. It just runs a code for every element without any resul in return

// Without higher order function

for (let i = 0; i < students.length; i++) {  console.log(`${students[i].firstName} ${students[i].lastName}`);}

// With higher order function using a function

function logFullNames(student) {  console.log(`${student.firstName} ${student.lastName}`);}

students.forEach(logFullNames);

// With higher order function using annonimous function students.forEach(function (student) {

  console.log(`${student.firstName} ${student.lastName}`)});

// With higher order function using arrow function

students.forEach(student => console.log(`${student.firstName} ${student.lastName}`));

### filter

Filter is a higher order function that accepts a function as an argument. That function has an expression that tests

every value from the collection and returns a new collection with values that are true for the expression or statement in the argument function

// Without higher order function let above18 = []; for (let i = 0; i < students.length; i++) {

  if (students[i].age >= 18) {    above18.push(students[i]);  }}

// With higher order function using a function function above18check(student) {  return student.age >= 18;}

let above18 = students.filter(above18check);

// With higher order function using annonimous function

let above18 = students.filter(function (student) {  return student.age >= 18 });

// With higher order function using arrow function let above18 = students.filter(student => student.age >= 18);

### map

Map is a higher order function that accepts a function as an argument as well. But this function will execute a code on every item of a collection and then return it. This means that we can modify or use every item in an array in a

particular way with only one line of code

// Without higher order function let fiveGradeStudentsNames = [];

for (let i = 0; i < students.length; i++) {  if (students[i].averageGrade === 5) {

    fiveGradeStudentsNames.push(`${students[i].firstName} ${students[i].lastName}`);  }}

// With higher order function using a function

function fiveGradeCheck(student) {  return student.averageGrade == 5;}

function fullName(student) {  return `${student.firstName} ${student.lastName}`;}

let fiveGradeStudentsNames = students.filter(fiveGradeCheck).map(fullName);

// With higher order function using annonimous function

let fiveGradeStudentsNames = students.filter(function (student) {      return student.averageGrade === 5    })

    .map(function (student) {      return `${student.firstName} ${student.lastName}`;    });

// With higher order function using arrow function

let fiveGradeStudentsNames = students

    .filter(student => student.averageGrade === 5)

    .map(student => `${student.firstName} ${student.lastName}`);

### reduce

Reduce is a function that accepts a function with two parameters as an argument and a starting value. This function aggregates multiple values from a collection into one place. It does this with the function passed in with two parameters. The first parameter is always an aggregate variable. This means that in that all calculations and results will be combined there. And after that, you have the second parameter which is changed after every cycle and represents a value from the collection

// Without higher order function

let allGradesExeptLowest = 0;

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

  if (students[i].averageGrade > 1) {

    allGradesExeptLowest += students[i].averageGrade;

  }

}

// With higher order function using a function

function lowestGradeCheck(student) {

  return student.averageGrade > 1;

}

function getGrades(student) {

  return student.averageGrade;

}

function agregateGrades(sum, grade) {

  return sum += grade;

}

let allGradesExeptLowest = students

    .filter(lowestGradeCheck)

    .map(getGrades)

    .reduce(agregateGrades, 0);

// With higher order function using annonimous function

let allGradesExeptLowest = students

    .filter(function (student) {

      return student.averageGrade > 1

    })

    .map(function (student) {

      return student.averageGrade

    })

    .reduce(function (sum, grade) {

      return sum += grade

    }, 0);

// With higher order function using arrow function

let allGradesExeptLowest = students

    .filter(student => student.averageGrade > 1)

    .map((student) => student.averageGrade)

    .reduce((sum, grade) => sum += grade, 0);

```

### sort

Sort is a function that always returns an array of the same items, and the same length but the items are sorted by some

criteria. This function is different from the previous ones because not only does it return a result array, it also

changes the original array. So if we want to keep the original order of our array we have to create a new variable with

the copy of the original variable and sort that, so that our original array can stay the same. The sorting is done

through a simple system. We send a callback that takes two parameters. The two parameters are tested in some

expressions. If the expression returns less than 0, the first item is sent to a lower index than the second. If the

expression returns more than 0 then the second is put to a lower index than the first. If the result is 0 then nothing

happens

```javascript

// Without higher order function

function sortFunc(arr) {

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

    for (let j = i + 1; j < arr.length; j++) {

      if (arr[i].grade < arr[j].grade) {

        let temp = arr[i];

        arr[i] = arr[j];

        arr[j] = temp;

      }

    }

  }

  return arr;

}

let sorted = sortFunc(students);

// With higher order function using a function

function sortGradesDesc(student1, student2) {

  return student2.grade - student1.grade; // descending

}

function sortGradesAsc(student1, student2) {

  return student1.grade - student2.grade; // ascending

}

studentss.sort(sortGradesDesc);

// With higher order function using annonimous function

studentss.sort(function (student1, student2) {

  return student2.grade - student1.grade; // Descending

});

studentss.sort(function (student1, student2) {

  return student1.grade - student2.grade; // Ascending

});

// With higher order function using arrow function

studentss.sort((student1, student2) => student2.grade - student1.grade); // Descending

studentss.sort((student1, student2) => student1.grade - student2.grade); // Ascending

// Keeping the original array

function copyArray(array) {

  let copied = [];

  array.forEach(x => copied.push(x));

  return copied;

}

let sortedStudents = copyArray(students);

sortedStudents.sort((student1, student2) => student2.grade - student1.grade); // Descending

sortedStudents.sort((student1, student2) => student1.grade - student2.grade); // Ascending

```

#### Copy of an original array?

By just writing let newStudents = students; we do not make a copy of the students array in the newStudents. We just pass

a reference to the students array. This means that the two variables point to the same place in memory. When methods

like sort, change this array it changes in the memory, and with that in both variables.

```javascript

let sortedStudents = students;

sortedStudents.sort((student1, student2) => student2.grade - student1.grade);

console.log(sortedStudents); // Sorted

console.log(students); // Also sorted? :(

```

### Solution

```javascript

// We create a new array [] and inside the brackets with spread, we add all elements from the students array

function copyArray(array) {

  let copied = [];

  array.forEach(x => copied.push(x));

  return copied;

}

let sortedStudents = copyArray(students);

sortedStudents.sort((student1, student2) => student2.grade - student1.grade);

console.log(sortedStudents); // Sorted

console.log(students); // Original order

```

![Pass by reference ](https://raw.githubusercontent.com/sedc-codecademy/skwd9-04-ajs/main/Samples/passbyrefference.gif?raw=true)

## Other higher order functions (good to know)

### every

Every is a function that accepts a function as an argument. This function has an expression that tests every value from

the collection and returns true or false. If every value in the collection is true for the expression, the every

function returns true. If one value is false for the expression, "every" returns false.

```javascript

let ages = [18, 16, 30, 42, 9];

let areAllMature = ages.every(age => age >= 18); // false

````

### some

Some is a function that accepts a function as an argument. This function has an expression that tests every value from

the collection and returns true or false. If one value in the collection is true for the expression, the some function

returns true. If all values are false for the expression, "some" returns false.

```javascript

let ages = [18, 16, 30, 42, 9];

let isSomeoneUnderage = ages.some(age => age < 18);  // true

````

### find

Find is a function that accepts a function as an argument. This function has an expression that tests every value from

the collection and returns true or false. If one value in the collection is true for the expression, the find function

returns that value. If all values are false for the expression, "find" returns undefined.

```javascript

let cities = ['Skopje', 'Barcelona', 'New York', 'Berlin', 'Paris'];

let barcelona = cities.find(city => city === 'Barcelona');  // 'Barcelona'

````

### findIndex

FindIndex is a function that accepts a function as an argument. This function has an expression that tests every value

from the collection and returns true or false. If one value in the collection is true for the expression, the findIndex

function returns the index of that value. If all values are false for the expression, "findIndex" returns -1.

```javascript

let cities = ['Skopje', 'Barcelona', 'New York', 'Berlin', 'Paris'];

let barcelonaIndex = cities.findIndex(city => city === 'Barcelona');  // 1

````

### includes

Includes is a function that accepts a value as an argument. This function checks if the value is in the collection. If

the value is in the collection, the includes function returns true. If the value is not in the collection, "includes"

returns false.

```javascript

let cities = ['Skopje', 'Barcelona', 'New York', 'Berlin', 'Paris'];

let isBarcelonaInArray = cities.includes('Barcelona');  // true

````

### flat

Flat is a function that accepts a number as an argument. This function flattens the array to the number of levels

specified in the argument. If no argument is passed, the flat function flattens the array to one level.

```javascript

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

let flattened = numbers.flat(2);  // [1, 2, 3, 4, 5, 6]

````

### flatMap

FlatMap is a function that accepts a function as an argument. This function has an expression that tests every value

from the collection and returns true or false. If one value in the collection is true for the expression, the flatMap

function returns that value. If all values are false for the expression, "flatMap" returns undefined.

```javascript

let words = ["apple", "banana", "cherry"];

let characters = words.flatMap(word => word.split('')); // ["a", "p", "p", "l", "e", "b", "a", "n", "a", "n", "a", "c", "h", "e", "r", "r", "y"]

````

### join

Join is a function that accepts a string as an argument. This function joins all elements of the array into a string

with the argument as a separator. If no argument is passed, the join function joins all elements of the array into a

string with a comma as a separator.

```javascript

let cities = ['Skopje', 'Barcelona', 'New York', 'Berlin', 'Paris'];

let joined = cities.join(' - '); // 'Skopje - Barcelona - New York - Berlin - Paris'

````

### slice

Slice is a function that accepts two numbers as arguments. This function returns a new array with the elements from the

first argument index to the second argument index. If no second argument is passed, the slice function returns a new

array with the elements from the first argument index to the end of the array.

```javascript

let cities = ['Skopje', 'Barcelona', 'New York', 'Berlin', 'Paris'];

let sliced = cities.slice(1, 3); // ['Barcelona', 'New York']

````

### splice

Splice is a function that accepts three arguments. The first argument is the index from which to start removing

elements. The second argument is the number of elements to remove. The third argument is the element to add at the

index. If no third argument is passed, the splice function only removes elements. Unlike Slice, Splice manipulates the

original array.

```javascript

let cities = ['Skopje', 'Barcelona', 'New York', 'Berlin', 'Paris'];

let spliced = cities.splice(1, 2, 'London'); // ['Skopje', 'London', 'Berlin', 'Paris']

````

### reverse

Reverse is a function that reverses the order of the elements in the array.

```javascript

let cities = ['Skopje', 'Barcelona', 'New York', 'Berlin', 'Paris'];

let reversed = cities.reverse(); // ['Paris', 'Berlin', 'New York', 'Barcelona', 'Skopje']

````

## Pure functions

Pure functions are functions that do not need or change anything in the outside world. There is no special syntax or

code for pure functions. You wrote some pure functions until this point for sure without even knowing that you did it. A

pure function is a state of a function that we always try to achieve. With pure function, our code is cleaner, more

organized, and decoupled (with few connections and ties). A code with pure functions is a code that can scale easily (

easier to continue working on and build on top of in the future). Of course, there are times when you need something

from the outside, or you need to change something outside the function, and it is not wrong to not write a function

that is not pure. But every time you write a function ask yourself:

\* Can I request the stuff that I need through parameters instead of accessing them directly and get the same results?

\* Can I return a value instead of changing a value from the outside directly and get the same results?

\* Do I get the same result for passing the same arguments every time?

```javascript

// Pure function

function increaseByOne(numbers) {

  let result = [];

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

    result.push(numbers[i] + 1)

  }

  return result;

}

// Impure function  ( using a variable from the outside )

let one = 1

function increaseByOne(numbers) {

  let result = [];

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

    result.push(numbers[i] + one)

  }

  return result;

}

// Impure function ( mutating data from the outside )

function increaseByOne(numbers) {

  let result = [];

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

    numbers[i] += 1

  }

  return numbers;

}

// Impure function ( changing the DOM outside of the function )

function increaseByOne(numbers) {

  let result = [];

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

    result.push(numbers[i] + 1)

    document.getElementById("result").innerHTML += numbers[i] + " ";

  }

  return result;

}

```

## What is strict typing in JavaScript?

Javascript lets us do all sorts of magic things. But some might say it is too flexible of a language. For this reason,

for securing our code from ourselves and our mistakes as developers, something that is called \*\*strict\*\* typing exists.

We tell at the start of our script or the start of a function \*\*'use strict';\*\*. With this, some extra rules are applied

to your javascript code. Some of those are:

\* Not allowed to pass value to an undeclared variable

\* Not allowing to delete variables, functions, or objects

\* Not allowing declaring two parameters of a function with the same name

\* Some extra names are not allowed to be used as variables ( are reserved for the language itself)

```javascript

'use strict';

// These lines of code will throw an error if we use strict

number = 15; // no declaration

delete number; // cant delete stuff

function sum(num1, num1) {

  return num1 + num1

}; // same parameter names

let eval = 5; // cant use keyword eval

let arguments = 2; // cant use ketworkd arguments

```

## Extra materials 📘

\* [First class functions](https://medium.freecodecamp.org/discover-the-power-of-first-class-functions-fd0d7b599b69)

\* [Filter,map,reduce] (https://www.freecodecamp.org/news/javascript-map-reduce-and-filter-explained-with-examples/)

\* [Pure functions and how to write them](https://blog.bitsrc.io/understanding-javascript-mutation-and-pure-functions-7231cc2180d3)

# Javascript under the hood 🥥

## Where and how is my JS code running?

Modern javascript runs most of the time in the browser. This means that it gets executed and lives in the browser, by

the browser engine and compiler. This is really important because javascript was not always compiled and refined like it

is today. Today we don't just execute javascript, but our scripts actually work with the browser hand in hand to load

modern web pages. Javascript works in one thread. This means that there is only one stack or queue for code or tasks to

be executed and they are being executed one by one. When we execute some functions they go to the top of the stack, they

get executed and go out of the stack one by one. But what if one of those functions is waiting for some response from

the internet? Basically, all functions after that should just wait for the function that waits for the call to finish

right? Well, this will freeze our web page, and we don't want our page frozen on every call we make. That is why

javascript delegates waiting for tasks to the browser

![javascript under the hood](https://raw.githubusercontent.com/sedc-codecademy/skwd9-04-ajs/main/Samples/javascriptandbrowser.png?raw=true)

1. \*\*JavaScript engine\*\*

   1.1 \*\*Memory Heap\*\* - A region in memory used for storing values and other types of data in an ordered fashion

   1.2 \*\*Call Stack\*\* - A structure where our functions are queued for running. It fills with tasks and it executes them

   in order LIFO ( Last In First Out )

2. \*\*Browser APIs\*\* - The browser APIs that help javascript run smoothly and without clogging.

3. \*\*Event Queue\*\* - A queue that holds the results of the browser APIs ( callbacks ) until they are added to the stack

   and executed. The order of giving out these callbacks is FIFO ( First In First Out )

4. \*\*Event Loop\*\* - A mechanism that checks if the stack is empty, and when it is pushing a task ( callback ) from the

   event queue onto the stack to be executed

## Working with the browser

In the browser, there are a lot of mechanisms for running our code smoothly. As we said there is a stack in the engine

for running our code. There are also some helper APIs or services that help our code. These are the DOM, that help us

map objects from our HTML, an HTTP client for making AJAX calls, services for waiting some amount of time or repeating

some code in some interval, etc. When a task that needs waiting comes to the stack like an AJAX call the stack throws it

out of the stack into the browser service. There it waits for a response and our stack can continue making calls. When

it is done it throws the code that needs to be executed from the AJAX call in a queue called task queue or callback

queue. A mechanism called the event loop stands between the stack where our code is executed and its job is to check if

the queue is empty and when it is, to put the code that is next in line from the callback queue into the stack so it can

be executed. After our stack is empty ( all tasks are executed in the stack ) our event loop gets the AJAX code and puts

it in the stack so it can be executed

#### setTimeout

Will execute a function passed as an argument when a given time passes. The second parameter of this function is how

many milliseconds do we actually want to wait, or delay some code

```javascript

    // with annon function

setTimeout(function () {

  console.log("This happens later!"

}, 2000);

// with arrow function

setTimeout(() => console.log("This happens later!"), 2000);

```

#### setInterval

Will execute a function passed as an argument when a given time passes. The second parameter of this function is how

many milliseconds do we actually want to wait or delay some code

```javascript

    setInterval(() => console.log("This happens every two seconds!"), 2000)

```

#### a simple piece of code

```javascript

let cb1 = () => console.log("cb1");

console.log("Hi");

setTimeout(cb1, 1000);

console.log("Bye");

```

\*\*Result:\*\*

Hi

Bye

cb1

#### behind the scenes

![behind the scenes](https://raw.githubusercontent.com/sedc-codecademy/skwd9-04-ajs/main/Samples/eventloop.gif?raw=true)

## Callback functions

Callback functions are functions that are executed inside of other functions ( usually after the execution of the other,

parent function ). Since javascript executes line by line and throws functions out of the stack when we have to wait for

them, it is easy to see that sometimes we would need a system that actually executes the code in our defined order. This

is where callbacks come into the picture. If we want to be sure that a function is executed after another, then we just

pass that function as an argument to the first one. That way when the first one completes inside of it we call it the

second one. This way we keep the order of execution

#### simple callback function

```javascript

function calculate(callback, num1, num2) {

  console.log("calculating...");

  return callback(num1, num2);

};

let result = calculate((x, y) => x + y, 2, 5);

console.log(result);

```

#### event ( callback ) queue

![event queue](https://raw.githubusercontent.com/sedc-codecademy/skwd9-04-ajs/main/Samples/callbackqueue.png?raw=true)

All callback functions from our code go to the corresponding browser API and then when it is done it goes to this queue.

Then it waits its turn to get on the stack and get executed. This includes callbacks from event handlers such as click

events ( when clicking a button the callback we attached to the handler comes here in the callback queue ), waiting on

calls from an AJAX request, waiting on a setTimeout, etc.

## Synchronous and asynchronous executing

So as you can tell, javascript does not want to wait on code. Every piece of code that it knows has to wait, it just

delegates it to the browser and continues its execution. This means that even tho we can write code in a certain order,

javascript will not guarantee us that it will execute it by the order that we wrote it in. This for the most part is

good, our page doesn't freeze when it waits for something and we can get the data we asked for later anyways. This is

called: executing our code asynchronously. But what happens when we need our code to be running in a certain order?

#### two function calls in order

```javascript

function first() {

  console.log("Frst thing!");

}

function second() {

  console.log("Second thing");

}

first();

second();

```

\*\*Result:\*\*

First thing

Second thing

#### two function calls but the first is delayed

```javascript

function first() {

  setTimeout(() => console.log("First thing"), 1000);

}

function second() {

  console.log("Second thing!");

}

first();

second();

```

\*\*Result:\*\*

Second thing

First thing

#### two function calls but the first is delayed ( solved with a callback )

```javascript

function first(callback) {

  setTimeout(() => {

    console.log("First thing");

    callback();

  }, 1000);

}

function second() {

  console.log("Second thing!");

}

first(second);

```

\*\*Result:\*\*

First thing

Second thing

#### making an ajax call

```javascript

function makeCall(url) {

  $.ajax({

    url: url,

    success: function (response) {

      console.log('The request succeeded!');

      return response;

    },

    error: function (response) {

      console.log('The request failed!');

      return response.responseText;

    }

  });

}

function print(results) {

  console.log(results);

}

print(makeCall("https://swapi.py4e.com/api/people/1/"));

```

\*\*Result:\*\*

undefined

The request succeeded!

#### making an ajax call with a callback

```javascript

function makeCall(url, callback) {

  $.ajax({

    url: url,

    success: function (response) {

      console.log('The request succeeded!');

      callback(response)

    },

    error: function (response) {

      console.log('The request failed!');

      callback(response.responseText);

    }

  });

}

function print(results) {

  console.log(results);

}

makeCall("https://swapi.py4e.com/api/people/1/", print);

```

\*\*Result:\*\*

The request succeeded!

{our response}

## Extra materials 📘

\* [Very useful tool](http://latentflip.com/loupe/)

\* [How does the event loop work](https://blog.sessionstack.com/how-javascript-works-event-loop-and-the-rise-of-async-programming-5-ways-to-better-coding-with-2f077c4438b5)

\* [Callbacks explained](https://www.sitepoint.com/callbacks-javascript/)

\* [Amazing video about the event loop](https://www.youtube.com/watch?v=cCOL7MC4Pl0)

\* [Event Loop code Visualized with animations](https://github.com/Drakso/DarkSecretsOfTheEventLoop)

\* [JS Visualizer](https://www.jsv9000.app/)

# Javascript under the hood 🥥

## Where and how is my JS code running?

Modern javascript runs most of the time in the browser. This means that it gets executed and lives in the browser, by

the browser engine and compiler. This is really important because javascript was not always compiled and refined like it

is today. Today we don't just execute javascript, but our scripts actually work with the browser hand in hand to load

modern web pages. Javascript works in one thread. This means that there is only one stack or queue for code or tasks to

be executed and they are being executed one by one. When we execute some functions they go to the top of the stack, they

get executed and go out of the stack one by one. But what if one of those functions is waiting for some response from

the internet? Basically, all functions after that should just wait for the function that waits for the call to finish

right? Well, this will freeze our web page, and we don't want our page frozen on every call we make. That is why

javascript delegates waiting for tasks to the browser

![javascript under the hood](https://raw.githubusercontent.com/sedc-codecademy/skwd9-04-ajs/main/Samples/javascriptandbrowser.png?raw=true)

1. \*\*JavaScript engine\*\*

   1.1 \*\*Memory Heap\*\* - A region in memory used for storing values and other types of data in an ordered fashion

   1.2 \*\*Call Stack\*\* - A structure where our functions are queued for running. It fills with tasks and it executes them

   in order LIFO ( Last In First Out )

2. \*\*Browser APIs\*\* - The browser APIs that help javascript run smoothly and without clogging.

3. \*\*Event Queue\*\* - A queue that holds the results of the browser APIs ( callbacks ) until they are added to the stack

   and executed. The order of giving out these callbacks is FIFO ( First In First Out )

4. \*\*Event Loop\*\* - A mechanism that checks if the stack is empty, and when it is pushing a task ( callback ) from the

   event queue onto the stack to be executed

## Working with the browser

In the browser, there are a lot of mechanisms for running our code smoothly. As we said there is a stack in the engine

for running our code. There are also some helper APIs or services that help our code. These are the DOM, that help us

map objects from our HTML, an HTTP client for making AJAX calls, services for waiting some amount of time or repeating

some code in some interval, etc. When a task that needs waiting comes to the stack like an AJAX call the stack throws it

out of the stack into the browser service. There it waits for a response and our stack can continue making calls. When

it is done it throws the code that needs to be executed from the AJAX call in a queue called task queue or callback

queue. A mechanism called the event loop stands between the stack where our code is executed and its job is to check if

the queue is empty and when it is, to put the code that is next in line from the callback queue into the stack so it can

be executed. After our stack is empty ( all tasks are executed in the stack ) our event loop gets the AJAX code and puts

it in the stack so it can be executed

#### setTimeout

Will execute a function passed as an argument when a given time passes. The second parameter of this function is how

many milliseconds do we actually want to wait, or delay some code

```javascript

    // with annon function

setTimeout(function () {

  console.log("This happens later!"

}, 2000);

// with arrow function

setTimeout(() => console.log("This happens later!"), 2000);

```

#### setInterval

Will execute a function passed as an argument when a given time passes. The second parameter of this function is how

many milliseconds do we actually want to wait or delay some code

```javascript

    setInterval(() => console.log("This happens every two seconds!"), 2000)

```

#### a simple piece of code

```javascript

let cb1 = () => console.log("cb1");

console.log("Hi");

setTimeout(cb1, 1000);

console.log("Bye");

```

\*\*Result:\*\*

Hi

Bye

cb1

#### behind the scenes

![behind the scenes](https://raw.githubusercontent.com/sedc-codecademy/skwd9-04-ajs/main/Samples/eventloop.gif?raw=true)

## Callback functions

Callback functions are functions that are executed inside of other functions ( usually after the execution of the other,

parent function ). Since javascript executes line by line and throws functions out of the stack when we have to wait for

them, it is easy to see that sometimes we would need a system that actually executes the code in our defined order. This

is where callbacks come into the picture. If we want to be sure that a function is executed after another, then we just

pass that function as an argument to the first one. That way when the first one completes inside of it we call it the

second one. This way we keep the order of execution

#### simple callback function

```javascript

function calculate(callback, num1, num2) {

  console.log("calculating...");

  return callback(num1, num2);

};

let result = calculate((x, y) => x + y, 2, 5);

console.log(result);

```

#### event ( callback ) queue

![event queue](https://raw.githubusercontent.com/sedc-codecademy/skwd9-04-ajs/main/Samples/callbackqueue.png?raw=true)

All callback functions from our code go to the corresponding browser API and then when it is done it goes to this queue.

Then it waits its turn to get on the stack and get executed. This includes callbacks from event handlers such as click

events ( when clicking a button the callback we attached to the handler comes here in the callback queue ), waiting on

calls from an AJAX request, waiting on a setTimeout, etc.

## Synchronous and asynchronous executing

So as you can tell, javascript does not want to wait on code. Every piece of code that it knows has to wait, it just

delegates it to the browser and continues its execution. This means that even tho we can write code in a certain order,

javascript will not guarantee us that it will execute it by the order that we wrote it in. This for the most part is

good, our page doesn't freeze when it waits for something and we can get the data we asked for later anyways. This is

called: executing our code asynchronously. But what happens when we need our code to be running in a certain order?

#### two function calls in order

```javascript

function first() {

  console.log("Frst thing!");

}

function second() {

  console.log("Second thing");

}

first();

second();

```

\*\*Result:\*\*

First thing

Second thing

#### two function calls but the first is delayed

```javascript

function first() {

  setTimeout(() => console.log("First thing"), 1000);

}

function second() {

  console.log("Second thing!");

}

first();

second();

```

\*\*Result:\*\*

Second thing

First thing

#### two function calls but the first is delayed ( solved with a callback )

```javascript

function first(callback) {

  setTimeout(() => {

    console.log("First thing");

    callback();

  }, 1000);

}

function second() {

  console.log("Second thing!");

}

first(second);

```

\*\*Result:\*\*

First thing

Second thing

#### making an ajax call

```javascript

function makeCall(url) {

  $.ajax({

    url: url,

    success: function (response) {

      console.log('The request succeeded!');

      return response;

    },

    error: function (response) {

      console.log('The request failed!');

      return response.responseText;

    }

  });

}

function print(results) {

  console.log(results);

}

print(makeCall("https://swapi.py4e.com/api/people/1/"));

```

\*\*Result:\*\*

undefined

The request succeeded!

#### making an ajax call with a callback

```javascript

function makeCall(url, callback) {

  $.ajax({

    url: url,

    success: function (response) {

      console.log('The request succeeded!');

      callback(response)

    },

    error: function (response) {

      console.log('The request failed!');

      callback(response.responseText);

    }

  });

}

function print(results) {

  console.log(results);

}

makeCall("https://swapi.py4e.com/api/people/1/", print);

```

\*\*Result:\*\*

The request succeeded!

{our response}

## Extra materials 📘

\* [Very useful tool](http://latentflip.com/loupe/)

\* [How does the event loop work](https://blog.sessionstack.com/how-javascript-works-event-loop-and-the-rise-of-async-programming-5-ways-to-better-coding-with-2f077c4438b5)

\* [Callbacks explained](https://www.sitepoint.com/callbacks-javascript/)

\* [Amazing video about the event loop](https://www.youtube.com/watch?v=cCOL7MC4Pl0)

\* [Event Loop code Visualized with animations](https://github.com/Drakso/DarkSecretsOfTheEventLoop)

# Asynchronous 🍒

## Solving problems with callback functions

With callbacks, we can control the flow of our code and be sure that our code will execute in some particular order. This is helping us write reliable code and handle different kinds of problems regarding dependency ( entities depending on each other completion ). But with this method of solving these issues, we run into a different kind of problem. If we have a lot of dependencies and we write a huge chain of callbacks we get unorganized and hard to read code

#### Simple example

In simple examples we don't really see the problem. In this case we have two things that need to happen in a certain order

```javascript

function simple() {

  setTimeout(function() {

    console.log('1. First thing, preparing for the second');

    setTimeout(function() {

      console.log('2. Second thing');

    }, 2000);

  }, 2000);

};

```

#### Complex example

But in more complex examples we can see what happens. A huge wave of code that is hard to examine and read

```javascript

function complex() {

  setTimeout(function() {

    console.log('1. First thing, preparing for the second');

    setTimeout(function() {

      console.log('2. Second thing, preparing for the third');

      setTimeout(function() {

      console.log('3. Third thing, preparing for the forth');

      setTimeout(function() {

      console.log('4. Forth thing, preparing for the fifth');

      setTimeout(function() {

        console.log('5. Fifth thing, preparing for the Sixth');

        setTimeout(function() {

          console.log('6. Sixth and last thing');

        }, 2000);

        }, 2000);

      }, 2000);

    }, 2000);

    }, 2000);

  }, 2000);

};

```

This phenomenon has many names. Callback Hell, Hadouken programming, Pyramid of doom among the many

## Better solution: Promises

In 2015 javascript developers finally got the highly anticipated feature, promise. Promises are basically a more elegant and sophisticated way of waiting on things. When we know that we have to wait for some kind of data we create a Promise object and put it as a placeholder for the data that we need. The promise then goes through its states that represent the status of the data that we need. A promise can be in one of 3 states at any point in time

1. Pending

2. Fulfilled

3. Rejected

When we wait for our data, the promise is in the pending state. The moment we get a result of our request for the data, the state will change either to Fulfilled or Rejected depending on the data that we get. If we successfully get the data, the promise state will turn in to Fulfilled and the promise will execute the methods that we have written to handle the requested data. If we don't get the data that we need or there were some problems in the process of acquiring the data, the state will change to Rejected and promise the code that we have written when expecting problems and errors

### Handling promises

The cool thing about promises is that we don't have to write code inside the promise to have something executed in a particular order ( Synchronously ). We can write the function that returns the promise and then calls methods that will be executed after the promise is resolved or rejected.

\* myFunction()\*\*.then\*\*( data => ...handling the data ) - a function that is executed when a promise is resolved and accepts a callback with a parameter in which the data from the resolved promise is stored.

\* myFunction().then(...)\*\*.catch\*\*( error => ...handling error ) - a function that is executed when a promise is rejected and accepts a callback with a parameter in which the error message from the rejected promise is stored. This can also be executed if there is an exception or error while executing the code in the .then() function.

\* myFunction().then(...).catch(...).\*\*finally\*\*(()=>... some code) - a function that is executed at the end of the whole resolve chain. This function is executed always no matter if the promise is resolved or rejected and has no parameters

#### A simple promise

A simple promise that waits for the first function to finish so it can execute second.

```javascript

function first(workTime){

  return new Promise((resolve, reject)=>{

    if(workTime <= 0){

      reject("It's too short of a work time. Please try again!");

    }

    setTimeout(() => {

      resolve("First thing, preparing for the second");

    }, workTime);

  })

}

function second(){

  console.log("second thing!");

}

first(1000)

.then(success => {

  console.log(success);

  second();

  })

.catch(error => console.log(`ERROR: ${error}`))

.finally(()=> console.log(`Everything is done at: ${new Date()}`))

```

#### Handling AJAX call with a promise

An ajax call to some documents online that returns a promise. When the promise is resolved ( the ajax call is over ) we call the show documents function with the documents

```javascript

function  getDocuments(){

    return new Promise((resolve, reject)=>{

        $.ajax({

            url:"https://raw.githubusercontent.com/sedc-codecademy/skwd9-04-ajs/main/Samples/documents.json",

            success: (response)=> {

                resolve(JSON.parse(response));

            },

            error: (error)=> {

                reject(error);

            }

        })

    })

}

function showDocuments(documents){

    if(!documents && typeof(documents) != "object"){

        throw new Error("Problem with documents!");

    }

    if(documents.length === 0){

        throw new Error("There is no documents!")

    }

    documents.forEach(doc => {

        console.log(`${doc.name}.${doc.type} (${doc.size}mb)`);

    });

}

getDocuments()

.then(data => {

    console.log("We got the documents!");

    showDocuments(data);

})

.catch(error => console.log(error.message));

```

#### Chaining promises

Promises can be chained. This means that we can ask for something, get a promise as a result and then add another \*\*then\*\* and wait for that promise next. This way we can wait for multiple things in a certain order

```javascript

function  getDocuments(){

    return new Promise((resolve, reject)=>{

        $.ajax({

            url:"https://raw.githubusercontent.com/sedc-codecademy/skwd9-04-ajs/main/Samples/documents.json",

            success: (response)=> {

                resolve(JSON.parse(response));

            },

            error: (error)=> {

                reject(error);

            }

        })

    })

}

function getImportantDocuments(documents){

    let importantDocuments = documents.filter(doc => doc.important);

    return new Promise((resolve, reject)=>{

        if(importantDocuments.length === 0){

          reject("There are no important documents!");

        }

        setTimeout(() => {

          resolve(importantDocuments);

        }, 3000);

    })

}

function checkDocuments(documents){

    if(!documents || typeof(documents) != "object"){

        throw new Error("Problem with documents!");

    }

    if(documents.length === 0){

        throw new Error("There is no documents!")

    }

}

function showDocuments(documents){

    documents.forEach(doc => {

        console.log(`${doc.name}.${doc.type} (${doc.size}mb)`);

    });

}

getDocuments()

.then(data => {

    console.log("We got the documents!");

    checkDocuments(data);

    return getImportantDocuments(data);

})

.then(data =>{

   showDocuments(data);

})

.catch(error => console.log(error.message));

```

### Using fetch

fetch is arguably one of the easiest ways to get data from the outside world and send requests. Basically, we write fetch and write the address where we want to make a request. The fetch API makes a call for us and returns a promise. The promise contains the request that we sent. In order to extract our data from the request object, we must first call the \*\*.json()\*\* method on the response and return it. Then we can get the data that we requested

```javascript

fetch("https://raw.githubusercontent.com/sedc-codecademy/skwd9-04-ajs/main/Samples/documents.json")

.then(response => response.json())

.then(data => showDocuments(data))

.catch(error => console.log(error.message))

.finally(()=> console.log("Everything is done at: " + new Date()));

```

## Async/await

Looking now at promises we can safely say that callbacks are not that fun anymore. We can do the same but quicker and more organized with promises. But in 2016 along with ES7 came another even nicer-looking feature in javascript that help us solve the problem with waiting for data and executing code in a particular order. This is the async/await feature and it works on top of the feature we discussed previously promises. Basically, we create a \*\*function\*\* and we write \*\*async\*\* before it. This makes the function asynchronous, meaning that some code inside of it can wait until it is ready without blocking the whole execution stack. If we want to wait on some function ( that returns promise, because it works with promises ) we only write \*\*await\*\* before the function call that has a result promise. Then the function waits on that result and then continues with the code inside that function. The code outside of the function is executed as normal ( doesn't wait for the await inside the async function )

#### A simple fetch call with async/await

```javascript

async function getDataFromFetch(){

    let response = await fetch("https://raw.githubusercontent.com/sedc-codecademy/skwd9-04-ajs/main/Samples/documents.json");

    let data = await response.json();

    console.log(data);

}

getDataFromFetch();

```

#### An example of the first-second exercise with async/await

```javascript

async function runFunctions(){

    console.log(await first(2000)); // 2

    second(); // 4

    console.log(`Everything is done at: ${new Date()}`); // 5

}

runFunctions(); // 1

console.log("This does not wait for the async function to finish!"); // 3

```

#### An example of the documents example with async/await

```javascript

async function showImportantDocuments(){

    let startTime = new Date().getTime(); // 2

    let documents = await getDocuments(); // 3

    checkDocuments(documents); // 5

    let importantDocs = await getImportantDocuments(documents); // 6

    showDocuments(importantDocs); // 7

    console.log(`Done in: ${( new Date().getTime() - startTime) / 1000}s`); // 8

}

showImportantDocuments(); // 1

console.log("This does not wait for the async function to finish!"); // 4

```

### Bonus: Error handling in javascript

When handling errors, in promises we used the catch function to catch any errors that might happen in the then method or if the promise changed its state to rejected. But we can also handle errors outside of the promises. We do that with the try/catch block. The try/catch block is an error-handling method used not only in javascript but in other languages as well. It works pretty simply. There are two blocks where we need to write code. The first is the \*\*try\*\* block. In it, we write our code and logic. That code will be observed and when error strikes, it is delegated in the other block called \*\*catch\*\*. The catch block accepts an error parameter and in it, the error that we get is stored. We can then handle the error in any way we see fit

```javascript

try{

  showImportantDocuments();

  console.log("This does not wait for the async function to finish!");

}

catch(error){

  console.log(error);

}

```

## Extra materials 📘

\* [From callback hell to async/await](https://blog.hellojs.org/asynchronous-javascript-from-callback-hell-to-async-and-await-9b9ceb63c8e8)

\* [Async/Await by example](https://codeburst.io/javascript-es-2017-learn-async-await-by-example-48acc58bad65)

\* [Understanding Javascript Promises](https://flaviocopes.com/javascript-promises/)

\* [Another good piece about promises](https://hackernoon.com/understanding-promises-in-javascript-13d99df067c1)

# Asynchronous 🍒

## Solving problems with callback functions

With callbacks, we can control the flow of our code and be sure that our code will execute in some particular order. This is helping us write reliable code and handle different kinds of problems regarding dependency ( entities depending on each other completion ). But with this method of solving these issues, we run into a different kind of problem. If we have a lot of dependencies and we write a huge chain of callbacks we get unorganized and hard to read code

#### Simple example

In simple examples we don't really see the problem. In this case we have two things that need to happen in a certain order

```javascript

function simple() {

  setTimeout(function() {

    console.log('1. First thing, preparing for the second');

    setTimeout(function() {

      console.log('2. Second thing');

    }, 2000);

  }, 2000);

};

```

#### Complex example

But in more complex examples we can see what happens. A huge wave of code that is hard to examine and read

```javascript

function complex() {

  setTimeout(function() {

    console.log('1. First thing, preparing for the second');

    setTimeout(function() {

      console.log('2. Second thing, preparing for the third');

      setTimeout(function() {

      console.log('3. Third thing, preparing for the forth');

      setTimeout(function() {

      console.log('4. Forth thing, preparing for the fifth');

      setTimeout(function() {

        console.log('5. Fifth thing, preparing for the Sixth');

        setTimeout(function() {

          console.log('6. Sixth and last thing');

        }, 2000);

        }, 2000);

      }, 2000);

    }, 2000);

    }, 2000);

  }, 2000);

};

```

This phenomenon has many names. Callback Hell, Hadouken programming, Pyramid of doom among the many

## Better solution: Promises

In 2015 javascript developers finally got the highly anticipated feature, promise. Promises are basically a more elegant and sophisticated way of waiting on things. When we know that we have to wait for some kind of data we create a Promise object and put it as a placeholder for the data that we need. The promise then goes through its states that represent the status of the data that we need. A promise can be in one of 3 states at any point in time

1. Pending

2. Fulfilled

3. Rejected

When we wait for our data, the promise is in the pending state. The moment we get a result of our request for the data, the state will change either to Fulfilled or Rejected depending on the data that we get. If we successfully get the data, the promise state will turn in to Fulfilled and the promise will execute the methods that we have written to handle the requested data. If we don't get the data that we need or there were some problems in the process of acquiring the data, the state will change to Rejected and promise the code that we have written when expecting problems and errors

### Handling promises

The cool thing about promises is that we don't have to write code inside the promise to have something executed in a particular order ( Synchronously ). We can write the function that returns the promise and then calls methods that will be executed after the promise is resolved or rejected.

\* myFunction()\*\*.then\*\*( data => ...handling the data ) - a function that is executed when a promise is resolved and accepts a callback with a parameter in which the data from the resolved promise is stored.

\* myFunction().then(...)\*\*.catch\*\*( error => ...handling error ) - a function that is executed when a promise is rejected and accepts a callback with a parameter in which the error message from the rejected promise is stored. This can also be executed if there is an exception or error while executing the code in the .then() function.

\* myFunction().then(...).catch(...).\*\*finally\*\*(()=>... some code) - a function that is executed at the end of the whole resolve chain. This function is executed always no matter if the promise is resolved or rejected and has no parameters

#### A simple promise

A simple promise that waits for the first function to finish so it can execute second.

```javascript

function first(workTime){

  return new Promise((resolve, reject)=>{

    if(workTime <= 0){

      reject("It's too short of a work time. Please try again!");

    }

    setTimeout(() => {

      resolve("First thing, preparing for the second");

    }, workTime);

  })

}

function second(){

  console.log("second thing!");

}

first(1000)

.then(success => {

  console.log(success);

  second();

  })

.catch(error => console.log(`ERROR: ${error}`))

.finally(()=> console.log(`Everything is done at: ${new Date()}`))

```

#### Handling AJAX call with a promise

An ajax call to some documents online that returns a promise. When the promise is resolved ( the ajax call is over ) we call the show documents function with the documents

```javascript

function  getDocuments(){

    return new Promise((resolve, reject)=>{

        $.ajax({

            url:"https://raw.githubusercontent.com/sedc-codecademy/skwd9-04-ajs/main/Samples/documents.json",

            success: (response)=> {

                resolve(JSON.parse(response));

            },

            error: (error)=> {

                reject(error);

            }

        })

    })

}

function showDocuments(documents){

    if(!documents && typeof(documents) != "object"){

        throw new Error("Problem with documents!");

    }

    if(documents.length === 0){

        throw new Error("There is no documents!")

    }

    documents.forEach(doc => {

        console.log(`${doc.name}.${doc.type} (${doc.size}mb)`);

    });

}

getDocuments()

.then(data => {

    console.log("We got the documents!");

    showDocuments(data);

})

.catch(error => console.log(error.message));

```

#### Chaining promises

Promises can be chained. This means that we can ask for something, get a promise as a result and then add another \*\*then\*\* and wait for that promise next. This way we can wait for multiple things in a certain order

```javascript

function  getDocuments(){

    return new Promise((resolve, reject)=>{

        $.ajax({

            url:"https://raw.githubusercontent.com/sedc-codecademy/skwd9-04-ajs/main/Samples/documents.json",

            success: (response)=> {

                resolve(JSON.parse(response));

            },

            error: (error)=> {

                reject(error);

            }

        })

    })

}

function getImportantDocuments(documents){

    let importantDocuments = documents.filter(doc => doc.important);

    return new Promise((resolve, reject)=>{

        if(importantDocuments.length === 0){

          reject("There are no important documents!");

        }

        setTimeout(() => {

          resolve(importantDocuments);

        }, 3000);

    })

}

function checkDocuments(documents){

    if(!documents || typeof(documents) != "object"){

        throw new Error("Problem with documents!");

    }

    if(documents.length === 0){

        throw new Error("There is no documents!")

    }

}

function showDocuments(documents){

    documents.forEach(doc => {

        console.log(`${doc.name}.${doc.type} (${doc.size}mb)`);

    });

}

getDocuments()

.then(data => {

    console.log("We got the documents!");

    checkDocuments(data);

    return getImportantDocuments(data);

})

.then(data =>{

   showDocuments(data);

})

.catch(error => console.log(error.message));

```

### Using fetch

fetch is arguably one of the easiest ways to get data from the outside world and send requests. Basically, we write fetch and write the address where we want to make a request. The fetch API makes a call for us and returns a promise. The promise contains the request that we sent. In order to extract our data from the request object, we must first call the \*\*.json()\*\* method on the response and return it. Then we can get the data that we requested

```javascript

fetch("https://raw.githubusercontent.com/sedc-codecademy/skwd9-04-ajs/main/Samples/documents.json")

.then(response => response.json())

.then(data => showDocuments(data))

.catch(error => console.log(error.message))

.finally(()=> console.log("Everything is done at: " + new Date()));

```

## Async/await

Looking now at promises we can safely say that callbacks are not that fun anymore. We can do the same but quicker and more organized with promises. But in 2016 along with ES7 came another even nicer-looking feature in javascript that help us solve the problem with waiting for data and executing code in a particular order. This is the async/await feature and it works on top of the feature we discussed previously promises. Basically, we create a \*\*function\*\* and we write \*\*async\*\* before it. This makes the function asynchronous, meaning that some code inside of it can wait until it is ready without blocking the whole execution stack. If we want to wait on some function ( that returns promise, because it works with promises ) we only write \*\*await\*\* before the function call that has a result promise. Then the function waits on that result and then continues with the code inside that function. The code outside of the function is executed as normal ( doesn't wait for the await inside the async function )

#### A simple fetch call with async/await

```javascript

async function getDataFromFetch(){

    let response = await fetch("https://raw.githubusercontent.com/sedc-codecademy/skwd9-04-ajs/main/Samples/documents.json");

    let data = await response.json();

    console.log(data);

}

getDataFromFetch();

```

#### An example of the first-second exercise with async/await

```javascript

async function runFunctions(){

    console.log(await first(2000)); // 2

    second(); // 4

    console.log(`Everything is done at: ${new Date()}`); // 5

}

runFunctions(); // 1

console.log("This does not wait for the async function to finish!"); // 3

```

#### An example of the documents example with async/await

```javascript

async function showImportantDocuments(){

    let startTime = new Date().getTime(); // 2

    let documents = await getDocuments(); // 3

    checkDocuments(documents); // 5

    let importantDocs = await getImportantDocuments(documents); // 6

    showDocuments(importantDocs); // 7

    console.log(`Done in: ${( new Date().getTime() - startTime) / 1000}s`); // 8

}

showImportantDocuments(); // 1

console.log("This does not wait for the async function to finish!"); // 4

```

### Bonus: Error handling in javascript

When handling errors, in promises we used the catch function to catch any errors that might happen in the then method or if the promise changed its state to rejected. But we can also handle errors outside of the promises. We do that with the try/catch block. The try/catch block is an error-handling method used not only in javascript but in other languages as well. It works pretty simply. There are two blocks where we need to write code. The first is the \*\*try\*\* block. In it, we write our code and logic. That code will be observed and when error strikes, it is delegated in the other block called \*\*catch\*\*. The catch block accepts an error parameter and in it, the error that we get is stored. We can then handle the error in any way we see fit

```javascript

try{

  showImportantDocuments();

  console.log("This does not wait for the async function to finish!");

}

catch(error){

  console.log(error);

}

```

## Extra materials 📘

\* [From callback hell to async/await](https://blog.hellojs.org/asynchronous-javascript-from-callback-hell-to-async-and-await-9b9ceb63c8e8)

\* [Async/Await by example](https://codeburst.io/javascript-es-2017-learn-async-await-by-example-48acc58bad65)

\* [Understanding Javascript Promises](https://flaviocopes.com/javascript-promises/)

\* [Another good piece about promises](https://hackernoon.com/understanding-promises-in-javascript-13d99df067c1)

# Asynchronous 🍒

## Solving problems with callback functions

With callbacks, we can control the flow of our code and be sure that our code will execute in some particular order. This is helping us write reliable code and handle different kinds of problems regarding dependency ( entities depending on each other completion ). But with this method of solving these issues, we run into a different kind of problem. If we have a lot of dependencies and we write a huge chain of callbacks we get unorganized and hard to read code

#### Simple example

In simple examples we don't really see the problem. In this case we have two things that need to happen in a certain order

```javascript

function simple() {

  setTimeout(function() {

    console.log('1. First thing, preparing for the second');

    setTimeout(function() {

      console.log('2. Second thing');

    }, 2000);

  }, 2000);

};

```

#### Complex example

But in more complex examples we can see what happens. A huge wave of code that is hard to examine and read

```javascript

function complex() {

  setTimeout(function() {

    console.log('1. First thing, preparing for the second');

    setTimeout(function() {

      console.log('2. Second thing, preparing for the third');

      setTimeout(function() {

      console.log('3. Third thing, preparing for the forth');

      setTimeout(function() {

      console.log('4. Forth thing, preparing for the fifth');

      setTimeout(function() {

        console.log('5. Fifth thing, preparing for the Sixth');

        setTimeout(function() {

          console.log('6. Sixth and last thing');

        }, 2000);

        }, 2000);

      }, 2000);

    }, 2000);

    }, 2000);

  }, 2000);

};

```

This phenomenon has many names. Callback Hell, Hadouken programming, Pyramid of doom among the many

## Better solution: Promises

In 2015 javascript developers finally got the highly anticipated feature, promise. Promises are basically a more elegant and sophisticated way of waiting on things. When we know that we have to wait for some kind of data we create a Promise object and put it as a placeholder for the data that we need. The promise then goes through its states that represent the status of the data that we need. A promise can be in one of 3 states at any point in time

1. Pending

2. Fulfilled

3. Rejected

When we wait for our data, the promise is in the pending state. The moment we get a result of our request for the data, the state will change either to Fulfilled or Rejected depending on the data that we get. If we successfully get the data, the promise state will turn in to Fulfilled and the promise will execute the methods that we have written to handle the requested data. If we don't get the data that we need or there were some problems in the process of acquiring the data, the state will change to Rejected and promise the code that we have written when expecting problems and errors

### Handling promises

The cool thing about promises is that we don't have to write code inside the promise to have something executed in a particular order ( Synchronously ). We can write the function that returns the promise and then calls methods that will be executed after the promise is resolved or rejected.

\* myFunction()\*\*.then\*\*( data => ...handling the data ) - a function that is executed when a promise is resolved and accepts a callback with a parameter in which the data from the resolved promise is stored.

\* myFunction().then(...)\*\*.catch\*\*( error => ...handling error ) - a function that is executed when a promise is rejected and accepts a callback with a parameter in which the error message from the rejected promise is stored. This can also be executed if there is an exception or error while executing the code in the .then() function.

\* myFunction().then(...).catch(...).\*\*finally\*\*(()=>... some code) - a function that is executed at the end of the whole resolve chain. This function is executed always no matter if the promise is resolved or rejected and has no parameters

#### A simple promise

A simple promise that waits for the first function to finish so it can execute second.

```javascript

function first(workTime){

  return new Promise((resolve, reject)=>{

    if(workTime <= 0){

      reject("It's too short of a work time. Please try again!");

    }

    setTimeout(() => {

      resolve("First thing, preparing for the second");

    }, workTime);

  })

}

function second(){

  console.log("second thing!");

}

first(1000)

.then(success => {

  console.log(success);

  second();

  })

.catch(error => console.log(`ERROR: ${error}`))

.finally(()=> console.log(`Everything is done at: ${new Date()}`))

```

#### Handling AJAX call with a promise

An ajax call to some documents online that returns a promise. When the promise is resolved ( the ajax call is over ) we call the show documents function with the documents

```javascript

function  getDocuments(){

    return new Promise((resolve, reject)=>{

        $.ajax({

            url:"https://raw.githubusercontent.com/sedc-codecademy/skwd9-04-ajs/main/Samples/documents.json",

            success: (response)=> {

                resolve(JSON.parse(response));

            },

            error: (error)=> {

                reject(error);

            }

        })

    })

}

function showDocuments(documents){

    if(!documents && typeof(documents) != "object"){

        throw new Error("Problem with documents!");

    }

    if(documents.length === 0){

        throw new Error("There is no documents!")

    }

    documents.forEach(doc => {

        console.log(`${doc.name}.${doc.type} (${doc.size}mb)`);

    });

}

getDocuments()

.then(data => {

    console.log("We got the documents!");

    showDocuments(data);

})

.catch(error => console.log(error.message));

```

#### Chaining promises

Promises can be chained. This means that we can ask for something, get a promise as a result and then add another \*\*then\*\* and wait for that promise next. This way we can wait for multiple things in a certain order

```javascript

function  getDocuments(){

    return new Promise((resolve, reject)=>{

        $.ajax({

            url:"https://raw.githubusercontent.com/sedc-codecademy/skwd9-04-ajs/main/Samples/documents.json",

            success: (response)=> {

                resolve(JSON.parse(response));

            },

            error: (error)=> {

                reject(error);

            }

        })

    })

}

function getImportantDocuments(documents){

    let importantDocuments = documents.filter(doc => doc.important);

    return new Promise((resolve, reject)=>{

        if(importantDocuments.length === 0){

          reject("There are no important documents!");

        }

        setTimeout(() => {

          resolve(importantDocuments);

        }, 3000);

    })

}

function checkDocuments(documents){

    if(!documents || typeof(documents) != "object"){

        throw new Error("Problem with documents!");

    }

    if(documents.length === 0){

        throw new Error("There is no documents!")

    }

}

function showDocuments(documents){

    documents.forEach(doc => {

        console.log(`${doc.name}.${doc.type} (${doc.size}mb)`);

    });

}

getDocuments()

.then(data => {

    console.log("We got the documents!");

    checkDocuments(data);

    return getImportantDocuments(data);

})

.then(data =>{

   showDocuments(data);

})

.catch(error => console.log(error.message));

```

### Using fetch

fetch is arguably one of the easiest ways to get data from the outside world and send requests. Basically, we write fetch and write the address where we want to make a request. The fetch API makes a call for us and returns a promise. The promise contains the request that we sent. In order to extract our data from the request object, we must first call the \*\*.json()\*\* method on the response and return it. Then we can get the data that we requested

```javascript

fetch("https://raw.githubusercontent.com/sedc-codecademy/skwd9-04-ajs/main/Samples/documents.json")

.then(response => response.json())

.then(data => showDocuments(data))

.catch(error => console.log(error.message))

.finally(()=> console.log("Everything is done at: " + new Date()));

```

## Async/await

Looking now at promises we can safely say that callbacks are not that fun anymore. We can do the same but quicker and more organized with promises. But in 2016 along with ES7 came another even nicer-looking feature in javascript that help us solve the problem with waiting for data and executing code in a particular order. This is the async/await feature and it works on top of the feature we discussed previously promises. Basically, we create a \*\*function\*\* and we write \*\*async\*\* before it. This makes the function asynchronous, meaning that some code inside of it can wait until it is ready without blocking the whole execution stack. If we want to wait on some function ( that returns promise, because it works with promises ) we only write \*\*await\*\* before the function call that has a result promise. Then the function waits on that result and then continues with the code inside that function. The code outside of the function is executed as normal ( doesn't wait for the await inside the async function )

#### A simple fetch call with async/await

```javascript

async function getDataFromFetch(){

    let response = await fetch("https://raw.githubusercontent.com/sedc-codecademy/skwd9-04-ajs/main/Samples/documents.json");

    let data = await response.json();

    console.log(data);

}

getDataFromFetch();

```

#### An example of the first-second exercise with async/await

```javascript

async function runFunctions(){

    console.log(await first(2000)); // 2

    second(); // 4

    console.log(`Everything is done at: ${new Date()}`); // 5

}

runFunctions(); // 1

console.log("This does not wait for the async function to finish!"); // 3

```

#### An example of the documents example with async/await

```javascript

async function showImportantDocuments(){

    let startTime = new Date().getTime(); // 2

    let documents = await getDocuments(); // 3

    checkDocuments(documents); // 5

    let importantDocs = await getImportantDocuments(documents); // 6

    showDocuments(importantDocs); // 7

    console.log(`Done in: ${( new Date().getTime() - startTime) / 1000}s`); // 8

}

showImportantDocuments(); // 1

console.log("This does not wait for the async function to finish!"); // 4

```

### Bonus: Error handling in javascript

When handling errors, in promises we used the catch function to catch any errors that might happen in the then method or if the promise changed its state to rejected. But we can also handle errors outside of the promises. We do that with the try/catch block. The try/catch block is an error-handling method used not only in javascript but in other languages as well. It works pretty simply. There are two blocks where we need to write code. The first is the \*\*try\*\* block. In it, we write our code and logic. That code will be observed and when error strikes, it is delegated in the other block called \*\*catch\*\*. The catch block accepts an error parameter and in it, the error that we get is stored. We can then handle the error in any way we see fit

```javascript

try{

  showImportantDocuments();

  console.log("This does not wait for the async function to finish!");

}

catch(error){

  console.log(error);

}

```

## Extra materials 📘

\* [From callback hell to async/await](https://blog.hellojs.org/asynchronous-javascript-from-callback-hell-to-async-and-await-9b9ceb63c8e8)

\* [Async/Await by example](https://codeburst.io/javascript-es-2017-learn-async-await-by-example-48acc58bad65)

\* [Understanding Javascript Promises](https://flaviocopes.com/javascript-promises/)

\* [Another good piece about promises](https://hackernoon.com/understanding-promises-in-javascript-13d99df067c1)

# JediArchives 📚

## Idea 💡

The Jedi Archives has long been forgotten, but that should not stop us from rebuilding this masterpiece of knowledge.

The JediArchives app is the first step to rebuilding the collective knowledge about the Star Wars universe. The app

should help any user that is interested in some Star Wars data to go through the archive and find what they are looking

for. A small step for the internet, but a huge step for Star Wars fans.

## Requirements 📃

The web page should be:

\* Single page application

\* There should be only one page

\* The logo should show at the front of the page

\* On the page there should be an image of \*\*person\*\* and \*\*space-ship\*\* that get data for the corresponding image and

  display a table

\* Tables:

    \* Person

        \* Name

        \* Height

        \* Mass

        \* Gender

        \* Birth Year

        \* Appearances ( Count of movies they appeared in )

    \* Ship

        \* Name

        \* Model

        \* Manufacturer

        \* Cost

        \* People Capacity ( Max people on board )

        \* Class

\* There should be 10 records per page of a table

\* There should be next/previous buttons to change the pages

## Flow 🌈

1. Person opens the web app

2. Right away the StarWars logo is shown and images of a person and spaceship

3. The person clicks on the person image

4. A table is generated with 10 people from the StarWars universe and below, a next button

5. The person clicks on the Next button

6. Immediately the view changes to a new table with new people and a previous button appears

7. A table is generated with 10 ships from the StarWars universe and below, a next button

8. The person clicks on the Next button

9. Immediately the view changes to a new table with new ships and a previous button appears

10. The person clicks on the previous button

11. Immediately the view changes to a new table with the first ships and the previous button disappears

## Extra features 🎈

\* Sorting in the tables

\* Loading animation while the application gets the data

\* Nice error message when a request has been denied ( Ex: unavailable, request limit, no page like that, access denied )

\* Add planets table

\* Improved pagination. Add the current page that the user is on from the total available pages (example: Page 3 of 13)

\* Add search functionality for people table