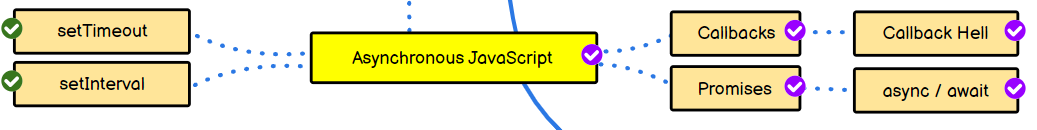
**JavaScript:**

**JavaScript Cheat Sheet:**

[**https://www.codecademy.com/learn/introduction-to-javascript/modules/learn-javascript-introduction/cheatsheet**](https://www.codecademy.com/learn/introduction-to-javascript/modules/learn-javascript-introduction/cheatsheet)

**Asynchronous JavaScript : ( CallBacks, Promises and Async Await):**

****

If you've been learning JavaScript for a while now, then you've probably heard the term "asynchronous" before.

This is because JavaScript is an asynchronous language...but what does that really mean? In this article, I hope to show you that the concept is not as difficult as it sounds.

**Synchronous vs Asynchronous**

Before we hop into the real deal, let's look at these two words – synchronous and asynchronous.

By default, JavaScript is a synchronous, single threaded programming language. This means that instructions can only run one after another, and not in parallel. Consider the little code snippet below:

let a = 1;

let b = 2;

let sum = a + b;

console.log(sum);

The above code is pretty simple – it sums two numbers and then logs the sum to the browser console. The interpreter executes these instructions one after another in that order until it is done.

But this method comes along with disadvantages. Say we wanted to fetch some large amount of data from a database and then display it on our interface. When the interpreter reaches the instruction that fetches this data, the rest of the code is blocked from executing until the data has been fetched and returned.

Now you might say that the data to be fetched isn't that large and it won't take any noticeable time. Imagine that you have to fetch data at multiple different points. This delay compounded doesn't sound like something users would want to come across.

Luckily for us, the problems with synchronous JavaScript were addressed by introducing asynchronous JavaScript.

Think of asynchronous code as code that can start now, and finish its execution later. When JavaScript is running asynchronously, the instructions are not necessarily executed one after the other as we saw before.

In order to properly implement this asynchronous behavior, there are a few different solutions developers has used over the years. Each solution improves upon the previous one, which makes the code more optimized and easier to understand in case it gets complex.

To further understand the asynchronous nature of JavaScript, we will go through callback functions, promises, and async and await.

**What are Callbacks in JavaScript?**

A callback is a function that is passed inside another function, and then called in that function to perform a task.

Confusing? Let's break it down by practically implementing it.

console.log('fired first');

console.log('fired second');

setTimeout(()=>{

console.log('fired third');

},2000);

console.log('fired last');

The snippet above is a small program that logs stuff to the console. But there is something new here. The interpreter will execute the first instruction, then the second, but it will skip over the third and execute the last.

The setTimeout is a JavaScript function that takes two parameters. The first parameter is another function, and the second is the time after which that function should be executed in milliseconds. Now you see the definition of callbacks coming into play.

The function inside setTimeout in this case is required to run after two seconds (2000 milliseconds). Imagine it being carried off to be executed in some separate part of the browser, while the other instructions continue executing. After two seconds, the results of the function are then returned.

That is why if we run the above snippet in our program, we will get this:

fired first

fired second

fired last

fired third

You see that the last instruction is logged before the function in the setTimeout returns its result. Say we used this method to fetch data from a database. While the user is waiting for the database call to return results, the flow in execution will not be interrupted.

This method was very efficient, but only to a certain point. Sometimes, developers have to make multiple calls to different sources in their code. In order to make these calls, callbacks are being nested until they become very hard to read or maintain. This is referred to as **Callback Hell**

To fix this problem, promises were introduced.

**What are Promises in JavaScript?**

We hear people make promises all the time. That cousin of yours who promised to send you free money, a kid promising to not touch the cookie jar again without permission...but promises in JavaScript are slightly different.

A promise, in our context, is something which will take some time to do. There are two possible outcomes of a promise:

* We either run and resolve the promise, or
* Some error occurs along the line and the promise is rejected

Promises came along to solve the problems of callback functions. A promise takes in two functions as parameters. That is, resolve and reject. Remember that resolve is success, and reject is for when an error occurs.

Let's take a look at promises at work:

const getData = (dataEndpoint) => {

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

//some request to the endpoint;

if(request is successful){

//do something;

resolve();

}

else if(there is an error){

reject();

}

});

};

The code above is a promise, enclosed by a request to some endpoint. The promise takes in resolve and reject like I mentioned before.

After making a call to the endpoint for example, if the request is successful, we would resolve the promise and go on to do whatever we want with the response. But if there is an error, the promise will get rejected.

Promises are a neat way to fix problems brought about by callback hell, in a method known as **promise chaining**. You can use this method to sequentially get data from multiple endpoints, but with less code and easier methods.

But there is an even better way! You might be familiar with the following method, as it's a preferred way of handling data and API calls in JavaScript.

**What is Async and Await in JavaScript?**

The thing is, chaining promises together just like callbacks can get pretty bulky and confusing. That's why Async and Await was brought about.

To define an async function, you do this:

const asyncFunc = async() => {

}

Note that calling an async function will always return a Promise. Take a look at this:

const test = asyncFunc();

console.log(test);

Running the above in the browser console, we see that the asyncFunc returns a promise.

Let's really break down some code now. Consider the little snippet below:

const asyncFunc = async () => {

const response = await fetch(resource);

const data = await response.json();

}

The async keyword is what we use to define async functions as I mentioned above. But how about await ? Well, it stalls JavaScript from assigning fetch to the response variable until the promise has been resolved. Once the promise has been resolved, the results from the fetch method can now be assigned to the response variable.

The same thing happens on line 3. The .json method returns a promise, and we can use await still to delay the assigning until the promise is resolved.

**To Block Code or Not to Block Code**

When I say 'stalling', you must think that implementing Async and Await somehow blocks code execution. Because what if our request takes too long, right?

Fact is, it doesn't. Code that is inside the async function is blocking, but that doesn't affect program execution in any way. The execution of our code is just as asynchronous as ever. To show this,

const asyncFunc = async () => {

const response = await fetch(resource);

const data = await response.json();

}

console.log(1);

cosole.log(2);

asyncFunc().then(data => console.log(data));

console.log(3);

console.log(4);

In our browser console, the output of the above would look something like this:

1

2

3

4

data returned by asyncFunc

You see that as we called asyncFunc, our code continued running until it was time for the function to return results.

**Spread Vs Rest Operator:**

JavaScript uses three dots (...) for both the rest and spread operators. But these two operators are not the same.

The main difference between rest and spread is that the rest operator puts the rest of some specific user-supplied values into a JavaScript array. But the spread syntax expands iterables into individual elements.

For instance, consider this code that uses rest to enclose some values into an array:

// Use rest to enclose the rest of specific user-supplied values into an array:

function myBio(firstName, lastName, ...otherInfo) {

return otherInfo;

}

// Invoke myBio function while passing five arguments to its parameters:

myBio("Oluwatobi", "Sofela", "CodeSweetly", "Web Developer", "Male");

// The invocation above will return:

["CodeSweetly", "Web Developer", "Male"]

In the snippet above, we used the ...otherInfo rest parameter to put "CodeSweetly", "Web Developer", and "Male" into an array.

Now, consider this example of a spread operator:

// Define a function with three parameters:

function myBio(firstName, lastName, company) {

return `${firstName} ${lastName} runs ${company}`;

}

// Use spread to expand an array’s items into individual arguments:

myBio(...["Oluwatobi", "Sofela", "CodeSweetly"]);

// The invocation above will return:

“Oluwatobi Sofela runs CodeSweetly”

In the snippet above, we used the spread operator (...) to spread ["Oluwatobi", "Sofela", "CodeSweetly"]’s content across myBio()’s parameters.

## What Exactly Is the Rest Operator?

The **rest operator** is used to put the rest of some specific user-supplied values into a JavaScript array.

// Define a function with two regular parameters and one rest parameter:

function myBio(firstName, lastName, ...otherInfo) {

return otherInfo;

}

The rest operator (...) instructs the computer to add whatever otherInfo (arguments) supplied by the user into an array. Then, assign that array to the otherInfo parameter.

As such, we call ...otherInfo a rest parameter.

**Note:**Arguments are optional values you may pass to a function’s parameter through an invocator.

**Here’s another example:**

// Define a function with two regular parameters and one rest parameter:

function myBio(firstName, lastName, ...otherInfo) {

return otherInfo;

}

// Invoke myBio function while passing five arguments to its parameters:

myBio("Oluwatobi", "Sofela", "CodeSweetly", "Web Developer", "Male");

// The invocation above will return:

["CodeSweetly", "Web Developer", "Male"]

In the snippet above, notice that myBio’s invocation passed five arguments to the function.

In other words, "Oluwatobi" and "Sofela" got assigned to the firstName and lastName parameters.

At the same time, the rest operator added the remaining arguments ( "CodeSweetly", "Web Developer", and "Male") into an array and assigned that array to the otherInfo parameter.

Therefore, myBio() function correctly returned ["CodeSweetly", "Web Developer", "Male"] as the content of the otherInfo rest parameter.

**JSON:**

# JavaScript and JSON

In this tutorial, you will learn about JSON and how JavaScript is used with JSON with the help of examples.

JSON stands for Javascript Object Notation. JSON is a text-based data format that is used to store and transfer data. For example,

// JSON syntax

{

"name": "John",

"age": 22,

"gender": "male",

}

In JSON, the data are in **key/value** pairs separated by a comma ,.

JSON was derived from JavaScript. So, the JSON syntax resembles JavaScript object literal syntax. However, the JSON format can be accessed and be created by other programming languages too.

**Note**: JavaScript Objects and JSON are not the same. You will learn about their differences later in this tutorial.

## **JSON Data**

JSON data consists of **key/value** pairs similar to JavaScript object properties. The key and values are written in double quotes separated by a colon :. For example,

// JSON data

"name": "John"

**Note**: JSON data requires double quotes for the key.

## **JSON Object**

The JSON object is written inside curly braces { }. JSON objects can contain multiple **key/value** pairs. For example,

// JSON object

{ "name": "John", "age": 22 }

## **JSON Array**

JSON array is written inside square brackets [ ]. For example,

// JSON array

[ "apple", "mango", "banana"]

// JSON array containing objects

[

{ "name": "John", "age": 22 },

{ "name": "Peter", "age": 20 }.

{ "name": "Mark", "age": 23 }

]

**Note**: JSON data can contain objects and arrays. However, unlike JavaScript objects, JSON data cannot contain functions as values.

## **Accessing JSON Data**

You can access JSON data using the dot notation. For example,

// JSON object

const data = {

"name": "John",

"age": 22,

"hobby": {

"reading" : true,

"gaming" : false,

"sport" : "football"

},

"class" : ["JavaScript", "HTML", "CSS"]

}

// accessing JSON object

console.log(data.name); // John

console.log(data.hobby); // { gaming: false, reading: true, sport: "football"}

console.log(data.hobby.sport); // football

console.log(data.class[1]); // HTML

We use the . notation to access JSON data. Its syntax is: variableName.key

You can also use square bracket syntax [] to access JSON data. For example,

// JSON object

const data = {

"name": "John",

"age": 22

}

// accessing JSON object

console.log(data["name"]); // John

## **Converting JSON to JavaScript Object**

You can convert JSON data to a JavaScript object using the built-in JSON.parse() function. For example,

// json object

const jsonData = '{ "name": "John", "age": 22 }';

// converting to JavaScript object

const obj = JSON.parse(jsonData);

// accessing the data

console.log(obj.name); // John

## **Converting JavaScript Object to JSON**

You can also convert JavaScript objects to JSON format using the JavaScript built-in JSON.stringify() function. For example,

// JavaScript object

const jsonData = { "name": "John", "age": 22 };

// converting to JSON

const obj = JSON.stringify(jsonData);

// accessing the data

console.log(obj); // "{"name":"John","age":22}"

## Use of JSON

# Iterators

### Functions Assigned to Variables

In JavaScript, functions are a data type just as strings, numbers, and arrays are data types. Therefore, functions can be assigned as values to variables, but are different from all other data types because they can be invoked.

**let plusFive = (number) => {**

**return number + 5;**

**};**

**// f is assigned the value of plusFive**

**let f = plusFive;**

**plusFive(3); // 8**

**// Since f has a function value, it can be invoked.**

**f(9); // 14**

### Callback Functions

In JavaScript, a callback function is a function that is passed into another function as an argument. This function can then be invoked during the execution of that higher order function (that it is an argument of).

Since, in JavaScript, functions are objects, functions can be passed as arguments.

**const isEven = (n) => {**

**return n % 2 == 0;**

**}**

**let printMsg = (evenFunc, num) => {**

**const isNumEven = evenFunc(num);**

**console.log(`The number ${num} is an even number: ${isNumEven}.`)**

**}**

**// Pass in isEven as the callback function**

**printMsg(isEven, 4);**

**// Prints: The number 4 is an even number: True.**

### Higher-Order Functions

In Javascript, functions can be assigned to variables in the same way that strings or arrays can. They can be passed into other functions as parameters or returned from them as well.

A “higher-order function” is a function that accepts functions as parameters and/or returns a function.

### JavaScript Functions: First-Class Objects

JavaScript functions are first-class objects. Therefore:

* They have built-in properties and methods, such as the name property and the .toString() method.
* Properties and methods can be added to them.
* They can be passed as arguments and returned from other functions.
* They can be assigned to variables, array elements, and other objects.

**//Assign a function to a variable originalFunc**

**const originalFunc = (num) => { return num + 2 };**

**//Re-assign the function to a new variable newFunc**

**const newFunc = originalFunc;**

**//Access the function's name property**

**newFunc.name; //'originalFunc'**

**//Return the function's body as a string**

**newFunc.toString(); //'(num) => { return num + 2 }'**

**//Add our own isMathFunction property to the function**

**newFunc.isMathFunction = true;**

**//Pass the function as an argument**

**const functionNameLength = (func) => { return func.name.length };**

**functionNameLength(originalFunc); //12**

**//Return the function**

**const returnFunc = () => { return newFunc };**

**returnFunc(); //[Function: originalFunc]**

### The .reduce() Method

The .reduce() method iterates through an array and returns a single value.

In the above code example, the .reduce() method will sum up all the elements of the array. It takes a callback function with two parameters (accumulator, currentValue) as arguments. On each iteration, accumulator is the value returned by the last iteration, and the currentValue is the current element. Optionally, a second argument can be passed which acts as the initial value of the accumulator.

**const arrayOfNumbers = [1, 2, 3, 4];**

**const sum = arrayOfNumbers.reduce((accumulator, currentValue) => {**

**return accumulator + currentValue;**

**});**

**console.log(sum); // 10**

### The .forEach() Method

The .forEach() method executes a callback function on each of the elements in an array in order.

In the above example code, the callback function containing a console.log() method will be executed 5 times, once for each element.

co**nst numbers = [28, 77, 45, 99, 27];**

**numbers.forEach(number => {**

**console.log(number);**

**});**

### The .filter() Method

The .filter() method executes a callback function on each element in an array. The callback function for each of the elements must return either true or false. The returned array is a new array with any elements for which the callback function returns true.

In the above code example, the array filteredArray will contain all the elements of randomNumbers but 4.

**const randomNumbers = [4, 11, 42, 14, 39];**

**const filteredArray = randomNumbers.filter(n => {**

**return n > 5;**

**});**

### The .map() Method

The .map() method executes a callback function on each element in an array. It returns a new array made up of the return values from the callback function.

The original array does not get altered, and the returned array may contain different elements than the original array.

In the example code above, the .map() method is used to add ' joined the contest.' string at the end of each element in the finalParticipants array.

**const finalParticipants = ['Taylor', 'Donald', 'Don', 'Natasha', 'Bobby'];**

**// add string after each final participant**

**const announcements = finalParticipants.map(member => {**

**return member + ' joined the contest.';**

**})**

**console.log(announcements);**