**Promise**

The **Promise** object represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

A Promise is in one of these states:

* *pending*: initial state, neither fulfilled nor rejected.
* *fulfilled*: meaning that the operation was completed successfully.
* *rejected*: meaning that the operation failed.

const myPromise = new Promise((resolve, reject) => {

setTimeout(() => {

resolve('foo');

}, 300);

});

myPromise

.then(handleResolvedA, handleRejectedA)

.then(handleResolvedB, handleRejectedB)

.then(handleResolvedC, handleRejectedC);

myPromise

.then(handleResolvedA)

.then(handleResolvedB)

.then(handleResolvedC)

.catch(handleRejectedAny);

[**Promise.all(iterable)**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/all)

Wait for all promises to be resolved, or for any to be rejected.

If the returned promise resolves, it is resolved with an aggregating array of the values from the resolved promises, in the same order as defined in the iterable of multiple promises.

If it rejects, it is rejected with the reason from the first promise in the iterable that was rejected.

[**Promise.allSettled(iterable)**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/allSettled)

Wait until all promises have settled (each may resolve or reject).

Returns a Promise that resolves after all of the given promises is either fulfilled or rejected, with an array of objects that each describe the outcome of each promise.

[**Promise.any(iterable)**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/any)

Takes an iterable of Promise objects and, as soon as one of the promises in the iterable fulfills, returns a single promise that resolves with the value from that promise.

[**Promise.reject(reason)**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/reject)

Returns a new Promise object that is rejected with the given reason.

[**Promise.resolve(value)**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/resolve)

Returns a new Promise object that is resolved with the given value. If the value is a thenable (i.e. has a then method), the returned promise will "follow" that thenable, adopting its eventual state; otherwise, the returned promise will be fulfilled with the value.

Generally, if you don't know if a value is a promise or not, [Promise.resolve(value)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/resolve) it instead and work with the return value as a promise.

async and await in javascript

Async functions will **always return a value**. It makes sure that a promise is returned and if it is not returned then javascript automatically wraps it in a promise which is resolved with its value. Await: Await function is used to wait for the promise.

# Making asynchronous programming easier with async and await

### [**The async keyword**](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Async_await#the_async_keyword)

First of all we have the async keyword, which you put in front of a function declaration to turn it into an [async function](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function). An async function is a function that knows how to expect the possibility of the await keyword being used to invoke asynchronous code.

Try typing the following lines into your browser's JS console:

function hello() { return "Hello" };

hello();

Copy to Clipboard

The function returns "Hello" — nothing special, right?

But what if we turn this into an async function? Try the following:

async function hello() { return "Hello" };

hello();

Copy to Clipboard

Ah. Invoking the function now returns a promise. This is one of the traits of async functions — their return values are guaranteed to be converted to promises.

You can also create an [async function expression](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/async_function), like so:

let hello = async function() { return "Hello" };

hello();

Copy to Clipboard

And you can use [arrow functions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions):

let hello = async () => "Hello";

Copy to Clipboard

These all do basically the same thing.

To actually consume the value returned when the promise fulfills, since it is returning a promise, we could use a .then() block:

hello().then((value) => console.log(value))

Copy to Clipboard

or even just shorthand such as

hello().then(console.log)

### [**The await keyword**](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Async_await#the_await_keyword)

The advantage of an async function only becomes apparent when you combine it with the [await](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await) keyword. await only works inside async functions within regular JavaScript code, however it can be used on its own with [JavaScript modules.](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules)

await can be put in front of any async promise-based function to pause your code on that line until the promise fulfills, then return the resulting value.

You can use await when calling any function that returns a Promise, including web API functions.

Here is a trivial example:

async function hello() {

return await Promise.resolve("Hello");

};

hello().then(alert);

await can be put in front of any async promise-based function to pause your code on that line until the promise fulfills, then return the resulting value.

# Getting Started with Modern JavaScript — Template Literals

Introduced in 2015 with ECMAScript6, **template literals** let us dynamically construct strings of text and **embedded expressions**, even **multi-line** strings of text, in a more elegant way than concatenation.

var a = 4;

var b = 2;

console.log('a + b = ' + a + b);

// -> a + b = 42

backtick

const name = 'Jack';

const str = `My name is ${name}.`;

console.log(str);

// -> My name is Jack.

const a = 4;

const b = 2;

console.log(`a + b = ${a + b}`);

// -> a + b = 6

# Multiline

# Getting Started with Modern JavaScript — Arrow Functions

One of the most popular new features in ECMAScript 2015 is arrow functions. Is it because of the new cleaner syntax or the sharing of this with the parent scope?

var sum = function(a, b) {

return a + b;

}

sum(1, 2) // -> 3

And here is the same function as an arrow function:

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

sum(1, 2) // -> 3

no need of function keyword , single line expression then no need of curly braces and no return statement required here.

If we have multiple lines in the function we always have to use the curly braces and return:

Parentheses are optional when only one parameter is present:

const square = n => n \* n;

square(3); // -> 9

Arrow functions are sometimes called **fat arrow functions**, because of the token => that resembles a fat arrow.

<>Copy

const players = [

{ name:'Michael', age:44},

{ name:'Karl', age:33},

{ name:'Lisa', age:37}

];

const ages = players.map(player => player.age);

// [44, 33, 37]

# This

We have seen how nice the new syntax can be but there is a another better reason to use arrow functions. And that is the sometimes confusing behavior of the this keyword in JavaScript. It doesn’t depend on where it’s declared but on how and where the function is called, the **execution context**.

In the global scope, value of this is always the window object.

Arrow functions however share the **lexical scope** with their parent. This means that it uses this from the code that contains the arrow function.

## **Constructor**

An arrow function cannot be used as a constructor. It will throw an error when used with new:

# Getting Started with Modern JavaScript — Destructuring

The two most used data structures in JavaScript are Object and Array. The **destructuring assignment** introduced in ECMAScript 2015 is shorthand syntax that allows us to extract array values or object properties into variables. In this article, we go through the syntax of both data structures and give examples of when you can use them.

Note that the destructuring assignment pattern works for any iterable.

With ES6 destructuring, the code becomes simpler:

<>

const [apple, banana, kiwi] = fruit;

Sometimes you might want to skip over items in the array being destructured:

<>

const [,,kiwi] = ['apple', 'banana', 'kiwi'];

Or you can save the other items in an array with the **rest** pattern:

<>

const [apple, …rest] = ['apple', 'banana', 'kiwi'];

console.log(rest); // -> ['banana', 'kiwi']

Note that the destructuring assignment pattern works for any iterable

# Destructuring Objects

**Object destructuring** lets you extract properties from objects and bind them to variables. Instead of accessing width through rectangle.width we can access the property and assign it to a variable:

# Default Values

When you destructure on properties that are not defined, you get undefined:

<>Copy

const { password } = {};

console.log(password); // -> undefined

And now let’s swap back with ES6 destructuring:

<>Copy

[me, you] = [you, me];

Destructuring is a JavaScript expression that allows us to extract data from arrays and objects. In this article, we have explored the syntax and various ways we can use it in our code. Destructuring arrays and objects is something that will make your code a bit shorter and cleaner all over the place.

# Getting Started with Modern JavaScript — Spread vs Rest

In short we could say that:

* **spread operator** unpacks elements.
* **rest parameter** packs elements.

## **Definitions**

* \*\*argument\*\*— An argument is a value passed to a function.
* \*\*parameter\*\* — A Parameter is a variable used in the declaration of a function.
* \*\*iterable\*\*— A collection of something that we can iterate (loop) over. For example array, list, set, and string.

# Spread Operator

The spread operator unpacks iterables into individual elements. Let’s look into different scenarios when this is useful.

## **Arrays**

The spread operator unpacks an array into separate arguments:

<>Copy

const array = [1, 2, 3];

console.log(…array);

// -> 1 2 3</span>

<>Copy

const array = [4, 2, 9, 5];

Math.max(…array); // -> 9</span>

const arrayA = [1, 2, 3];

const arrayB = [0, ...arrayA, 4];

console.log(arrayB);

// -> [0, 1, 2, 3, 4]

This is useful when we want to add elements into an existing array. We can even **merge** two arrays:

<>Copy

const first = [1, 2];

const other = [3, 4];

const combo = [...first, ...other];

// [1, 2, 3, 4]

const word = 'test';

console.log([...word]);

// -> ["t", "e", "s", "t"]

<>Copy

const arrayA = [1, 2, 3];

const arrayB = [...arrayA];

console.log(arrayB);

// -> [1, 2, 3]

# Rest Parameter

Where the *spread operator* unpacks the contents of an iterable into single elements, the *rest parameter* instead collects all the **remaining** elements into an **array**.

In JavaScript, it’s possible to call a function with any number of arguments.  
We can use the *rest parameter* when we don’t know how many arguments will be used or just want to collect some of them into an array.

function sum(...args) {

let result = 0;

for (let arg of args) {

result += arg;

}

return result

}

sum(4, 2) // -> 6

sum(3, 4, 5, 6) // -> 18

const player = {

name: 'Max Best',

age: 42,

game: 'Football'

}

We can also use the rest syntax with objects together with **destructuring**:

const { name, ...rest } = player;

console.log(name); // -> Max Best

console.log(rest);

// -> {age: 42, game: 'Football'}

Conclusion

* **Rest Parameter** collects all remaining elements into an array.
* **Spread Operator** expands collected elements such as arrays into single elements.
* The spread syntax only does a **shallow copy** one level deep.
* Only the **last parameter** can be a *rest parameter*.

# Getting Started with Modern JavaScript — Classes

We often need to create many objects of the same kind, like users or cars. In object-oriented languages, a class is often used for this purpose.JavaScript, being a prototype-based language, has something called constructor functions

# Defining a Class

class Car {

// Body of the class

}

<>Copy

const myCar = new Car();

<>Copy

console.log(typeof Car); // -> function

## **Hoisting**

An important difference between function declarations and class declarations is that function declarations are **hoisted** and class declarations are not. Therefore you need to first declare the class before you can initialize objects with the new keyword.

As you might remember variables declared with var are also hoisted. So, keeping with the “modern” way it makes then sense that classes just like variables declared with let or const are not hoisted.

class Car {

constructor(brand) {

this.brand = brand;

}

}

const car = new Car('Volvo');

console.log(car.brand); // -> Volvo

class Car {

constructor(brand) {

this.brand = brand;

}

getBrand() {

return this.brand;

}

}

const car = new Car('Tesla');

console.log(car.getBrand()); // -> Tesla

class Double {

static double(n) {

return n \* 2;

}

}

Double.double(2); // -> 4

We don’t have to create an instance of the class but can call the method directly on the class itself. Static methods are often used in utility classes for making computations.

## **Getters and Setters**

JavaScript classes can have getter and setter functions.

<>Copy

class Car {

constructor(brand) {

this.\_brand = brand;

}

get brand() {

return this.\_brand;

}

set brand(value) {

this.\_brand = value;

}

}

const car = new Car('');

car.brand = 'Tesla;

console.log(car.brand); // -> Tesla

Inheritance , super same as Java

Conclusion

* Use classes when you need to create **multiple objects** of the same kind.
* Classes are a special kind of functions that run in **strict mode** and are **not hoisted**.
* Classes can have fields and methods that we can access from a class instance.
* We can inherit functionality from other classes by **extending** them.

# Object.freeze()

const obj = {

prop: 42

};

Object.freeze(obj);

obj.prop = 33;

// Throws an error in strict mode

console.log(obj.prop);

// expected output: 42

# [Why is typeof null "object"?](https://stackoverflow.com/questions/18808226/why-is-typeof-null-object)

# LocalStorage vs. Cookies: All You Need to Know About Storing JWT Tokens Securely in the Front-End

# Getting Started with Modern JavaScript  —  Variables and Scope

Conclusion

* Variables declared on the **global scope** are accessible everywhere in your code.
* Variables declared in a **local scope** can not be accessed from outside that scope.
* const and let use block scope which exists between two curly braces.
* Mostly use const for variables whose values will never change.
* For everything else use let.
* Don’t use var to avoid confusion.

<https://indepth.dev/posts/1357/getting-started-with-modern-javascript-variables-and-scope>

Closures ->

**Disadvantages of closures:**

* Variables used by closure will not be garbage collected.
* Memory snapshot of the application will be increased if closures are not used properly.

how to deallocate memory in javascript

Hence there is no explicit way to allocate or free up memory in JavaScript. **Just initializing objects allocates memory for them**. When the variable goes out of scope, it is automatically garbage collected(frees up memory taken by that object.)

# Memory Management

Low-level languages like C, have manual memory management primitives such as [malloc()](https://pubs.opengroup.org/onlinepubs/009695399/functions/malloc.html) and [free()](https://en.wikipedia.org/wiki/C_dynamic_memory_allocation#Overview_of_functions). In contrast, JavaScript automatically allocates memory when objects are created and frees it when they are not used anymore (garbage collection). This automaticity is a potential source of confusion: it can give developers the false impression that they don't need to worry about memory management.

A Memory leak can be defined as a piece of memory that is no longer being used or required by an application but for some reason is not returned back to the OS and is still being occupied needlessly. ... A Javascript memory leak occurs **when you may no longer need an object but the JS runtime still thinks you do**.

Function Expression allows us **to create an anonymous function** which doesn't have any function name which is the main difference between Function Expression and Function Declaration. A function expression can be used as an IIFE (Immediately Invoked Function Expression) which runs as soon as it is defined.

Techniques, strategies and recipes for building a **modern web app** with **multiple teams** that can **ship features independently**.

## **What are Micro Frontends?**

The term **Micro Frontends** first came up in [ThoughtWorks Technology Radar](https://www.thoughtworks.com/radar/techniques/micro-frontends) at the end of 2016. It extends the concepts of micro services to the frontend world. The current trend is to build a feature-rich and powerful browser application, aka single page app, which sits on top of a micro service architecture. Over time the frontend layer, often developed by a separate team, grows and gets more difficult to maintain. That’s what we call a [Frontend Monolith](https://www.youtube.com/watch?v=pU1gXA0rfwc).

The idea behind Micro Frontends is to think about a website or web app as **a composition of features** which are owned by **independent teams**. Each team has a **distinct area of business** or **mission** it cares about and specialises in. A team is **cross functional** and develops its features **end-to-end**, from database to user interface.

However, this idea is not new. It has a lot in common with the [Self-contained Systems](http://scs-architecture.org/) concept. In the past approaches like this went by the name of [Frontend Integration for Verticalised Systems](https://dev.otto.de/2014/07/29/scaling-with-microservices-and-vertical-decomposition/). But Micro Frontends is clearly a more friendly and less bulky term.

## **Core Ideas behind Micro Frontends**

* **Be Technology Agnostic**  
  Each team should be able to choose and upgrade their stack without having to coordinate with other teams. [Custom Elements](https://micro-frontends.org/#the-dom-is-the-api) are a great way to hide implementation details while providing a neutral interface to others.
* **Isolate Team Code**  
  Don’t share a runtime, even if all teams use the same framework. Build independent apps that are self contained. Don’t rely on shared state or global variables.
* **Establish Team Prefixes**  
  Agree on naming conventions where isolation is not possible yet. Namespace CSS, Events, Local Storage and Cookies to avoid collisions and clarify ownership.
* **Favor Native Browser Features over Custom APIs**  
  Use [Browser Events for communication](https://micro-frontends.org/#parent-child-communication--dom-modification) instead of building a global PubSub system. If you really have to build a cross team API, try keeping it as simple as possible.
* **Build a Resilient Site**  
  Your feature should be useful, even if JavaScript failed or hasn’t executed yet. Use [Universal Rendering](https://micro-frontends.org/#serverside-rendering--universal-rendering) and Progressive Enhancement to improve perceived performance.