Top of Form

**JavaScript Execution Context – How JS Works Behind the Scenes**

Let's look at an example so we can learn more:

var n = 5;

function square(n) {

var ans = n \* n;

return ans;

}

var square1 = square(n);

var square2 = square(8);

console.log(square1)

console.log(square2)

In the above code,

1. n is initialized and assigned a value of 5
2. We defined a function square() that accepts an argument n and returns the square of n.
3. We call the square() function and store the returned value in the square1 variable.
4. We call the square() function and store the returned value in the square2 variable.
5. Finally, it outputs both square1 and square2

Behind the scenes, JavaScript is doing so many things. Let's understand all of it.

**What is the Execution Context?**

**When the JavaScript engine scans a script file, it makes an environment called the Execution Context that handles the entire transformation and execution of the code**.

There are two types of execution contexts: **global** and **function**.

The global execution context is created when **a JavaScript script first starts to run, and it represents the global scope in JavaScript**.

A function execution **context is created whenever a function is called, representing the function's local scope**.

**Phases of the JavaScript Execution Context**

There are two phases of JavaScript execution context:

1. **Creation phase**

**Creates a global object that is window in the browser and global in NodeJs.**

**Sets up a memory for storing variables and functions.**

**Stores the variables with values as undefined and function references**.

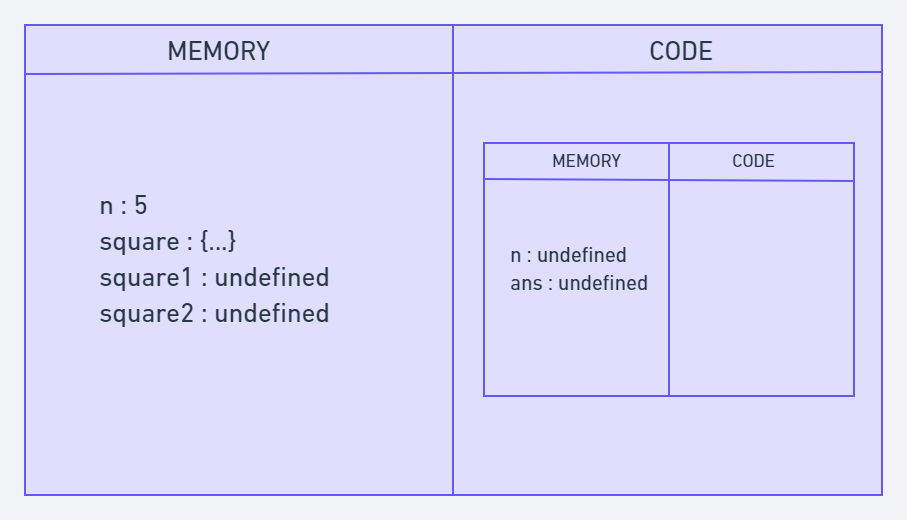
This is called the creation phase

1. **Execution phase**:

Code Execution Phase

Once the calculation is done, it assigns the value to the variable that was undefined before. The function will return the value, and the function execution context will be destroyed.





**Q2-What is the Call Stack?**

**1-To keep the track of all the contexts, including global and functional, the JavaScript engine uses a call stack.**

**2-**A call stack is also known as an 'Execution Context Stack', 'Runtime Stack', or 'Machine Stack'.

3-It **uses the LIFO principle (Last-In-First-Out).** When the engine first starts executing the script, it creates a global context and pushes it on the stack. Whenever a function is invoked, similarly, the JS engine creates a function stack context for the function and pushes it to the top of the call stack and starts executing it.

When execution of the current function is complete, then the JavaScript engine will automatically remove the context from the call stack and it goes back to its parent.

Let's see the following example:

function funcA(m,n) {

return m \* n;

}

function funcB(m,n) {

return funcA(m,n);

}

function getResult(num1, num2) {

return funcB(num1, num2)

}

var res = getResult(5,6);

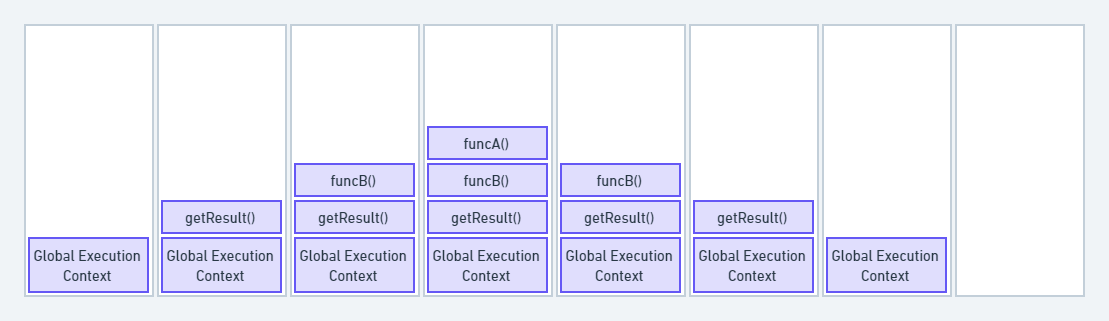
console.log(res); // 30

In this example, the JS engine creates a global execution context that enters the creation phase.

First it allocates memory for funcA, funcB, the getResult function, and the res variable. Then it invokes getResult(), which will be pushed on the call stack.

Then getResult() will call funcB(). At this point, funcB's context will be stored on the top of the stack. Then it will start executing and call another function funcA(). Similarly, funcA's context will be pushed.

Once execution of each function is done, it will be removed from the call stack. The following picture depicts the entire process of the execution:

Call Stack

The call stack has its own fixed size depending on the system or browser. If the number of contexts exceeds the limit, then a stack overflow error will occur. This happens with a recursive function that has no base condition.

function display() {

display();

}

display();

C:\Users\rwiteshbera\Desktop\Javascript\n.js:2

display();

^

RangeError: Maximum call stack size exceeded

Output

**Conclusion**

In conclusion, JavaScript execution context is a crucial part of understanding how JavaScript works behind the scenes. It determines the environment in which code is executed and what variables and functions are available to use.

Top of Form

Qsss**What is Hoisting in JavaScript | Hoisting Functions, Variables and Classes**

**---- Hoisting is a concept in JavaScript where the declaration of a function, variable, or class goes to the top of the scope they were defined in.**

**Function Hoisting**

Take a look at this code example:

function printHello() {

console.log("hello")

}

printHello()

// hello

Here, we declare printHello, and we execute the function just after the line it was declared. No errors; everything works!

Now look at this example:

printHello()

// hello

function printHello() {

console.log("hello")

}

Here, we execute printHello before the line the function was declared. And everything still works without errors. What happened here? **Hoisting**.

Before the interpreter executes the whole code, it first hoists (raises, or lifts) the declared function to the top of the scope it is defined in. In this case, printHello is defined in the global scope, so the function is hoisted to the top of the global scope. Through hoisting, the function (including the logic) becomes accessible even before the line it was declared in the code.

Let's see another example:

printHello()

// hello

printDillion()

// ReferenceError: printDillion is not defined

function printHello() {

console.log('hello')

function printDillion() {

console.log("dillion")

}

}

As you see here, we declare printHello. In this function, we first execute console.log('hello') then we declare another function called printDillion which executes console.log('dillion') when called.

Before printHello is declared in the code, we try to access it by executing printHello(). It's accessible (since it is hoisted to the top of the global scope), so we have "hello" printed on the console.

But then we try to access printDillion, and we get a reference error: **printDillion is not defined**. Does hoisting not occur on printDillion?

printDillion is hoisted, but it is only lifted to the top of the scope it was declared in. In this case, it is declared in a local scope--in printHello. Therefore, it would only be accessible in the function. Let's update our code:

printHello()

// hello

printHello()

// hello

function printHello() {

printDillion()

// dillion

console.log('hello')

function printDillion() {

console.log("dillion")

}

}

Now, we execute printDillion in printHello before the line where printDillion was actually declared. Since the function is hoisted to the top of the local scope, we're able to access it before the line where it was actually declared

**Variable Hoisting**.

**Hoisting var variables**

Take a look at this example:

console.log(name)

// undefined

var name = "Dillion"

Here, we declare a variable called name with a string value of "Dillion". But, we try to access the variable before the line it was declared. But we don't get any errors. **Hoisting happened**. **The name declaration is hoisted to the top, so the interpreter "knows" that there is a variable called name**. If the interpreter did not know, you would get **name is not defined**. Let's try it out:

console.log(name)

// ReferenceError: name is not defined

var myName = "Dillion"

We have a variable called myName but no name. We get the **name is not defined** error when we try to access name. The interpreter "does not know" about this variable.

Coming back to our example above:

console.log(name)

// undefined

var name = "Dillion"

Although hoisting happened here, the value of name is undefined when we access it before the line of declaration. With variables declared var, the variable declaration is hoisted but with a default value of undefined. The actual value is initialized when the declaration line is executed.

By accessing the variable after that line, we get the actual value:

console.log(name)

// undefined

var name = "Dillion"

console.log(name)

// Dillion

Let's say we declared name in a function:

print()

console.log(name)

// ReferenceError: name is not defined

function print() {

var name = "Dillion"

}

Here, we get a reference error: **name is not defined**. Remember, variables are hoisted but **only to the top of the scope they were declared in**. In this case, name is declared in print, so it will be hoisted to the top of that local scope. Let's try to access it in the function:

print()

function print() {

console.log(name)

// undefined

var name = "Dillion"

}

By trying to access name in the function, even though it's above the line of declaration, we do not get an error. That's because name is hoisted, but don't forget, **with a default value of undefined**.

**Hoisting let variables--**

Although variables declared with let are also hoisted, they have a different behavior. Let's see an example:

console.log(name)

// ReferenceError: Cannot access 'name' before initialization

let name = "Dillion"

Here, we get a reference error: **Cannot access 'name' before initialization**. Do you notice that the error does not say **name is not defined**? That's because the interpreter is "aware" of a name variable because the variable is hoisted.

"**Cannot access 'name' before initialization" occurs because variables declared with let do not have a default value when hoisted.** As we saw in var, the variables have a default value of undefined until the line where the declaration/initialization is executed. But with let, the variables are uninitialized.

The variables are hoisted to the top of the scope they are declared in (local, global, or block), but are not accessible because they have not been initialized. This concept is also referred to as the [Temporal Dead Zone](https://dillionmegida.com/p/temporal-dead-zone-in-javascript).

They can only be accessible after the initialization line has been executed:

let name = "Dillion"

console.log(name)

// Dillion

By accessing name after the line it was declared and initialized, we get no errors.

**Hoisting const variables**

Just like let, variables declared with const are hoisted, but not accessible. For example:

console.log(name)

// ReferenceError: Cannot access 'name' before initialization

const name = "Dillion"

The same concept of a **temporal dead zone** applies to const variables. Such variables are hoisted to the top of the scope they are defined in (local, global, or block), but they remain inaccessible until the variables are initialized with a value.

const name = "Dillion"

console.log(name)

// Dillion

By accessing the variable after it has been initialized with a value (as you see above), everything works fine.

Moving onto hoisting for classes.

**Class Hoisting**

Classes in JavaScript are also hoisted. Let's see an example:

const Dog = new Animal("Bingo")

// ReferenceError: Cannot access 'Animal' before initialization

class Animal {

constructor(name) {

this.name = name

}

}

Here, we declare a class called Animal. We try to access this class (instantiate a Dog object) before it was declared. We get a reference error: **Cannot access 'Animal' before initialization**. What does this error remind you of?

Just like let and const variables, classes are hoisted to the top of the scope they are defined in, but inaccessible until they are initialized. We do not get "Animal is not defined", which shows that the interpreter "knows" that there is an Animal class (due to hoisting). But we cannot access that class until the line of initialization is executed.

Let's update the code:

class Animal {

constructor(name) {

this.name = name

}

}

const Dog = new Animal("Bingo")

console.log(Dog)

// { name: 'Bingo' }

After Animal has been initialized, it becomes accessible, so we can instantiate the Dog object from the class without errors.

A **temporal dead zone (TDZ)** is the area of a block where a variable is inaccessible until the moment the computer completely initializes it with a value.

Here's a table summary showing the differences between these keywords:

| **KEYWORD** | **SCOPE** | **REDECLARATION & REASSIGNMENT** | **HOISTING** |
| --- | --- | --- | --- |
| var | Global, Local | yes & yes | yes, with default value(undefined) |
| let | Global, Local, Block | no & yes | yes, without default value(**Cannot access 'name' before initialization)** |
| const | Global, Local, Block | no & no | yes, without default value  (**Cannot access 'name' before initialization)** |

These factors I've explained, play a role in determining how you declare variables in JavaScript.

If you never want a variable to change, const is the keyword to use.

If you want to reassign values:

* and you want the hoisting behavior, var is the keyword to use
* if you don't want it, let is the keyword for you

# Q-What is Variable Shadowing in JavaScript?

shadowing occurs when a variable declared in a certain scope (e.g. a **local variable**) has the same name as a variable in an outer scope (e.g. a **global variable**). When this happens, the outer variable is said to be shadowed by the inner variable.

## Variable Shadowing in the Global Scope

In the **global scope**, shadowing occurs when a variable declared with the var keyword has the same name as a variable declared with the let or const keyword. When this happens, the global variable is said to be shadowed by the function-scoped variable.

In this code, the global variable x is shadowed by the function-scoped variable x declared inside the foo() function. As a result, the value of x is different inside and outside the foo() function.

## Variable Shadowing in the Function Scope

In the **function scope**, shadowing occurs when a variable declared with the let or const keyword has the same name as a variable declared with the var keyword. When this happens, the function-scoped variable is said to be shadowed by the block-scoped variable.

Top of Form

# JavaScript Map, Reduce, and Filter - JS Array Functions Explained with Code Examples

## Map

The map() method is used for creating a new array from an existing one, applying a function to each one of the elements of the first array.

### Syntax

var new\_array = arr.map(function callback(element, index, array) {

// Return value for new\_array

}[, thisArg])

In the callback, only the array element is required. Usually some action is performed on the value and then a new value is returned.

### Example

In the following example, each number in an array is doubled.

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

const doubled = numbers.map(item => item \* 2);

console.log(doubled); // [2, 4, 6, 8]

## Filter

The filter() method takes each element in an array and it applies a conditional statement against it. If this conditional returns true, the element gets pushed to the output array. If the condition returns false, the element does not get pushed to the output array.

### Syntax

var new\_array = arr.filter(function callback(element, index, array) {

// Return true or false

}[, thisArg])

The syntax for filter is similar to map, except the callback function should return true to keep the element, or false otherwise. In the callback, only the element is required.

### Examples

In the following example, odd numbers are "filtered" out, leaving only even numbers.

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

const evens = numbers.filter(item => item % 2 === 0);

console.log(evens); // [2, 4]

In the next example, filter() is used to get all the students whose grades are greater than or equal to 90.

const students = [

{ name: 'Quincy', grade: 96 },

{ name: 'Jason', grade: 84 },

{ name: 'Alexis', grade: 100 },

{ name: 'Sam', grade: 65 },

{ name: 'Katie', grade: 90 }

];

const studentGrades = students.filter(student => student.grade >= 90);

return studentGrades; // [ { name: 'Quincy', grade: 96 }, { name: 'Alexis', grade: 100 }, { name: 'Katie', grade: 90 } ]

## Reduce

The reduce() method reduces an array of values down to just one value. To get the output value, it runs a reducer function on each element of the array.

### ****Syntax****

arr.reduce(callback[, initialValue])

The callback argument is a function that will be called once for every item in the array. This function takes four arguments, but often only the first two are used.

* accumulator - the returned value of the previous iteration
* currentValue - the current item in the array
* index - the index of the current item
* array - the original array on which reduce was called
* The initialValue argument is optional. If provided, it will be used as the initial accumulator value in the first call to the callback function.

### Examples

The following example adds every number together in an array of numbers.

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

const sum = numbers.reduce(function (result, item) {

return result + item;

}, 0);

console.log(sum); // 10

In the next example, reduce() is used to transform an array of strings into a single object that shows how many times each string appears in the array. Notice this call to reduce passes an empty object {} as the initialValue parameter. This will be used as the initial value of the accumulator (the first argument) passed to the callback function.

var pets = ['dog', 'chicken', 'cat', 'dog', 'chicken', 'chicken', 'rabbit'];

var petCounts = pets.reduce(function(obj, pet){

if (!obj[pet]) {

obj[pet] = 1;

} else {

obj[pet]++;

}

return obj;

}, {});

console.log(petCounts);

/\*

Output:

{

dog: 2,

chicken: 3,

cat: 1,

rabbit: 1

}

\*/

**Definitions(map vs foreach)**

The map method receives a function as a parameter. Then it applies it on each element and returns an entirely new array populated with the results of calling the provided function.

This means that it returns a new array that contains an image of each element of the array. It will always return the same number of items.

const myAwesomeArray = [5, 4, 3, 2, 1]

myAwesomeArray.map(x => x \* x)

// >>>>>>>>>>>>>>>>> Output: [25, 16, 9, 4, 1]

Like map , the forEach() method receives a function as an argument and executes it once for each array element. However, instead of returning a new array like map, it returns undefined.

const myAwesomeArray = [

{ id: 1, name: "john" },

{ id: 2, name: "Ali" },

{ id: 3, name: "Mass" },

]

myAwesomeArray.forEach(element => console.log(element.name))

// >>>>>>>>> Output : john

// Ali

// Mass

**1. The returning value**

The first difference between map() and forEach() is the returning value. The forEach() method returns undefined and map() returns a new array with the transformed elements. Even if they do the same job, the returning value remains different.

const myAwesomeArray = [1, 2, 3, 4, 5]

myAwesomeArray.forEach(x => x \* x)

//>>>>>>>>>>>>>return value: undefined

myAwesomeArray.map(x => x \* x)

//>>>>>>>>>>>>>return value: [1, 4, 9, 16, 25]

**2. Ability to chain other methods**

The second difference between these array methods is the fact that map() is chainable. This means that you can attach reduce(), sort(), filter() and so on after performing a map() method on an array.

That's something you can't do with forEach() because, as you might guess, it returns undefined.

const myAwesomeArray = [1, 2, 3, 4, 5]

myAwesomeArray.forEach(x => x \* x).reduce((total, value) => total + value)

//>>>>>>>>>>>>> Uncaught TypeError: Cannot read property 'reduce' of undefined

myAwesomeArray.map(x => x \* x).reduce((total, value) => total + value)

//>>>>>>>>>>>>>return value: 55

**Q-** -

# Functions

## [Defining functions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions#defining_functions)

### [Function declarations](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions#function_declarations)

A **function definition** (also called a **function declaration**, or **function statement**) consists of the [function](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function) keyword, followed by:

* The name of the function.
* A list of parameters to the function, enclosed in parentheses and separated by commas.
* The JavaScript statements that define the function, enclosed in curly brackets, { /\* … \*/ }.

For example, the following code defines a simple function named square:

### [Function expressions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions#function_expressions)

While the function declaration above is syntactically a statement, functions can also be created by a [function expression](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/function).

Such a function can be **anonymous**; it does not have to have a name. For example, the function square could have been defined as:

const square = function (number) {

return number \* number;

};

console.log(square(4)); // 16

Copy to Clipboard

However, a name can be provided with a function expression. Providing a name allows the function to refer to itself, and also makes it easier to identify the function in a debugger's stack traces:

const factorial = function fac(n) {

return n < 2 ? 1 : n \* fac(n - 1);

};

console.log(factorial(3)); // 6

Copy to Clipboard

## [Calling functions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions#calling_functions)

Defining a function does not execute it. Defining it names the function and specifies what to do when the function is called.

**Calling** the function actually performs the specified actions with the indicated parameters. For example, if you define the function square, you could call it as follows:

square(5);

# Q-What are first-class vs higher-order functions in JavaScript?

First-class functions are JavaScript functions that can behave like variables.

They can also be parsed as arguments to higher-order functions.

Higher-order functions are functions that return a function or take in a function as an argument

Higher-order functions include:

* map
* sort
* filter

# Q-JavaScript Immediately Invoked Function Expression

A JavaScript immediately **invoked function expression is a**[**function**](https://www.javascripttutorial.net/javascript-function/)**defined as an expression and executed immediately after creation. The** following shows the syntax of defining an immediately invoked function expression:

(function(){

*//...*

})();

Code language: JavaScript (javascript)

## Why IIFEs

When you define a [function](https://www.javascripttutorial.net/javascript-function/), the JavaScript engine adds the function to the global object.

If you have many global variables and functions, the JavaScript engine will only release the memory allocated for them until the global object loses its scopes.

# QJavaScript Rest vs Spread Operator

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

-rest operator is last one in function

# Callback function

A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.

Here is a quick example:

function greeting(name) {

alert(`Hello, ${name}`);

}

function processUserInput(callback) {

const name = prompt("Please enter your name.");

callback(name);

}

processUserInput(greeting);

Copy to Clipboard

The above example is a [synchronous](https://developer.mozilla.org/en-US/docs/Glossary/Synchronous) callback, as it is executed immediately.

Note, however, that callbacks are often used to continue code execution after an [asynchronous](https://developer.mozilla.org/en-US/docs/Glossary/Asynchronous) operation has completed — these are called asynchronous callbacks. A good example is the callback functions executed inside a [.then()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then) block chained onto the end of a promise after that promise fulfills or rejects. This structure is used in many modern web APIs, such as [fetch()](https://developer.mozilla.org/en-US/docs/Web/API/fetch).

# Arrow Functions vs Regular Functions in JavaScript – What's the Difference?

In JavaScript, there are two types of functions. You have normal functions and arrow functions. Let's explore the difference between them in this article.

Arrow functions was introduced in ES6. And it introduced a simple and shorter way to create functions.

Here's how to create a normal function, with arguments, which returns something:

function multiply(num1, num2) {

const result = num1 \* num2

return result

}

If you want to transform this into an arrow function, here's what you'll have:

const multiply = (num1, num2) => {

const result = num1 \* num2

return result

}

If the return statement is the only statement in the function, you can even have a shorter function expression. For example:

const multiply = (num1, num2) => {

return num1 \* num2

}

This function only contains the return statement. With arrow functions, we can have something shorter like this:

const multiply = (num1, num2) => num1 \* num2

We skip the curly braces and the return keyword. Shorter; one-liner.

But the syntax of writing both types of functions is not the only difference. There's more, so let's look at them.

## 1. No arguments object in arrow functions

A normal function has an arguments object which you can access in the function:

function print() {

console.log(arguments)

}

The arguments object is a local variable that contains the arguments passed to the function when called. Let's try it out:

print("hello", 400, false)

// {

// '0': 'hello',

// '1': 400,

// '2': false

// }

As you can see here, the three arguments passed when calling print() are contained in the arguments object which we log to the console. We can access the first argument with arguments[0], the second with arguments[1] and the third with arguments[2]

But this object does not exist in arrow functions. Let's try it out. Say we have print using an arrow function:

const print = () => {

console.log(arguments)

}

print("hello", 400, false)

// Uncaught ReferenceError: arguments is not defined

Now we have a reference error: **arguments is not defined**. That's because the arguments variable does not exist in arrow functions.

## 2. Arrow functions do not create their own this binding

In normal functions, a this variable is created which references the objects that call them. For example:

const obj = {

name: 'deeecode',

age: 200,

print: function() {

console.log(this)

}

}

obj.print()

// {

// name: 'deeecode',

// age: 200,

// print: [Function: print]

// }

As you can see here, the this in the print method points to obj, which is the object that calls the method.

Here's another example:

const obj = {

name: 'deeecode',

age: 200,

print: function() {

function print2() {

console.log(this)

}

print2()

}

}

obj.print()

// Window

Here, we have two functions. The first one is print which is a method of the obj object. The second is print2 which is a function declared inside print. print2() is also called directly.

In this case, print is called by obj (obj.print()) but no object calls print2 (print2()). So the this in print2 would reference the window object by default.

Now let's see what happens with an arrow function.

const obj = {

name: 'deeecode',

age: 200,

print: () => {

console.log(this)

}

}

obj.print()

// Window

By using an arrow function for print, this function does not automatically create a this variable. As a result, any reference to this would point to what this was before the function was created.

As you see in the result, **this was pointing to the Window object before print was created.**

Let's see another example:

const obj = {

name: 'deeecode',

age: 200,

print: function() {

const print2 = () => {

console.log(this)

}

print2()

}

}

obj.print()

// {

// name: 'deeecode',

// age: 200,

// print: [Function: print]

// }

Here, we have print as a normal function which means a this variable is automatically created in it. Then we have print2 which is an arrow function.

Because obj is calling print (as in obj.print()), the this in print would point to obj.

Since print2 is an arrow function, it doesn't create its own this variable. Therefore, any reference to this would point to what the value of this was before the function was created. In this case where obj calls print, this was pointing to obj before print2 was created. As you can see in the results, by logging this from print2, obj is the result.

You can learn more about [this in my article here](https://dillionmegida.com/p/this-demystified/)

## 3. Arrow functions cannot be used as constructors

With normal functions, you can create constructors which serve as a special function for instantiating an object from a class.

Here is an example of an Animal class which we instantiate two objects from:

class Animal {

constructor(name, numOfLegs) {

this.name = name

this.numOfLegs = numOfLegs

}

sayName() {

console.log(`My name is ${this.name}`)

}

}

const Dog = new Animal("Bingo", 4)

const Bird = new Animal("Steer", 2)

Dog.sayName()

// My name is Bingo

Bird.sayName()

// My name is Bird

Here, we have the Animal constructor which can be instantiated with different parameters. In the constructor, two arguments are required: name and noOfLegs.

In the case of Dog, we create a new instance of Animal using "Bingo" as the name and **4** as the noOfLegs.

In the case of Bird, we create a new instance of Animal using "Steer" as the name and **2** as the noOfLegs.

By calling sayName on Dog and Bird, you can see how the method works currently with each object. The this variable points to the objects and the name field is gotten from each of them.

The this variable is very important for classes and constructors. In point 2, we saw that arrow functions cannot create their own this. For this reason, arrow functions also cannot be used as constructors.

Let's attempt it and see what happens:

class Animal {

constructor = (name, numOfLegs) => {

this.name = name

this.numOfLegs = numOfLegs

}

sayName() {

console.log(`My name is ${this.name}`)

}

}

// Uncaught SyntaxError: Classes may not have a field named 'constructor'

Here, we have an arrow function used for the constructor. But, we get a SyntaxError: **Classes may not have a field named 'constructor'**.

Because arrow functions involve expressions that are assigned variables, JavaScript now sees constructor as a field. And in classes, you cannot have a field named constructor as that is a reserved name.

But, we can use arrow functions for the methods in the class without getting errors. For example:

class Animal {

constructor (name, numOfLegs){

this.name = name

this.numOfLegs = numOfLegs

}

sayName = () => {

console.log(`My name is ${this.name}`)

}

}

const Dog = new Animal("Bingo", 4)

Dog.sayName()

// My name is Bingo

Here, we have a normal function for the constructor, and an arrow function for the sayName method. sayName is a field. And we do not get errors.

By calling sayName() on Dog, we still get "My name is Bingo". Though sayName as an arrow function does not create its own this, remember that any reference to this would point to the value of it before the arrow function was created. In this case, the value of this pointed to Dog before sayName was created.

## 4. Arrow functions cannot be declared

When it comes to functions, you need to understand **function declaration** and **function expression**.

Function declarations involve the function keyword and a name for the function. For example:

function printHello() {

console.log("hello")

}

printHello is a **declared function**. But, check out this example:

const printHello = function() {

console.log("hello")

}

In this case, printHello is not a declared function. We have an anonymous function (not named) on the right side of the assignment operator. This function is a **function expression**, which is assigned to the printHello variable.

Though the function keyword is used, there is no name assigned, which makes it an expression and not a declaration. To prove that it is not a declaration, try the following:

function() {

console.log("hello")

}

Because this expression is not assigned to a variable, you get an error: **SyntaxError: Function statements require a function name**

Back to arrow functions. Normal functions can be declared when you use the function keyword and a name, but arrow functions cannot be declared. They can only be expressed because they are anonymous:

const printHello = () => {

console.log("hello")

}

As you see here, we have an anonymous function (starting from () => ...) which is assigned to the printHello variable. printHello is not a declared function here. It is a variable that holds the evaluated value from the function expression.

## 5 Arrow functions cannot be accessed before initialization

Hoisting is a concept where a variable or function is lifted to the top of its global or local scope before the whole code is executed. This makes it possible for such a variable/function to be accessed before initialization. Here's a function example:

printName()

console.log("hello")

function printName() {

console.log("i am dillion")

}

// i am dillion

// hello

As you can see here, we called printName before it was actually declared in the code. But we don't get any errors. printName() is executed (logging "i am dillion" to the console) before console.log("hello").

What happens here is hoisting.

The printName function is raised to the top of the global scope (the scope it is declared in) before the whole code is executed, thereby making it possible to execute the function earlier.

But not all kinds of functions can be accessed before initialization. All functions and variables in JavaScript are hoisted, but **only declared functions can be accessed before initialization**.

Here's an example with an arrow function:

printName()

console.log("hello")

const printName = () => {

console.log("i am dillion")

}

// ReferenceError: Cannot access 'printName' before initialization

Running this code, you get an error: **ReferenceError: Cannot access 'printName' before initialization**.

As we saw in point 4, printName is not a declared function. It is a variable, declared with const which is assigned a function expression. Variables declared with let and const are hoisted, but they cannot be accessed before the line they are initialized.

Let's say we use var for our arrow function:

printName()

console.log("hello")

var printName = () => {

console.log("i am dillion")

}

// TypeError: printName is not a function

Here, we have declared the printName variable with var. The error we get now is **TypeError: printName is not a function**. The reason for this is that variables declared with var are hoisted and accessible, but they have a default value of undefined. So attempting to access printName before the line it was initialized with the function expression is interpreted as undefined(), and as you know, "undefined is not a function".

You can learn more about [the hoisting differences between var, let, and const here](https://www.freecodecamp.org/news/javascript-let-and-const-hoisting/)

What is lexical scope and block scope?

Lexical Scope allows inner functions to access the scope of their outer function

A **closure** is a function having access to the parent scope, even after the parent function has closed.

Let's note the first part of the sentence before the comma:

...**a function having access to the parent scope**

**That's describing lexical scope!**

**But we need the second part of the definition to give an example of a closure...**

**...even af ter the parent function has closed**.

const myFunction = () => {

    let myValue = 2;

    console.log(myValue);

    const childFunction = () => {

         console.log(myValue += 1);

    }

    return childFunction;

}

const result = myFunction();

console.log(result);

result();

result();

// result(

Output-

2

[Function: childFunction]

3

4

**What is Module Pattern**

The Module Pattern is one of the important**patterns in JavaScript**.It is a commonly used Design Pattern which is used to wrap a set of variables and functions together in a single scope.

* It is used to define objects and specify the variables and the functions that can be accessed from outside the scope of the function.
* We expose certain properties and function as public and can also restrict the scope of properties and functions within the object itself, making them private.
* This means that those variables cannot be accessed outside the scope of the function.
* We can achieve data hiding an abstraction using this pattern

Let's look for the simple implementation of Module Pattern with only public fields:

The code contains the function which defines a certain variable and returns those variables back from the function. The values that are returned on function call is saved as an object literal in the variable newEmployee**.**We are specifying the key-value pair that need to be returned. This is one of the simple implementations of Module Pattern. We can access the properties “name”, “age”, “designation” from the object returned. In the above code, we can access all the properties defined by the object since they are exposed from the function call.

### ****What is Currying?****

Currying is a function that takes one argument at a time and returns a new function expecting the next argument. It is a conversion of functions from callable as f(a,b,c)into callable as f(a)(b)(c).

Basically Currying doesn’t call a function. It just transforms a function. They are constructed by chaining closures by immediately returning their inner functions simultaneously.

If you don't know what closures are [Watch previous video for closures](https://www.youtube.com/watch?v=sZjlEKbaykc)

Let's see currying in an example. 👇🏼

### ****Convert f(a, b) into f(a)(b).****

/\*f(a,b) implementation \*/

function f(a,b) {

return "Works"

}

COPY

COPY

/\*f(a)(b) implementation \*/

function f(a) {

return (b) => {

"Works"

}

}

console.log(f(1)(2)) // works

console.log(f(1)); /\* (b) => {return "Works" } \*/

### ****Why should currying be used?****

Following are the reasons why currying is good :

✅ It makes a function pure which makes it expose to less errors and side effects.

✅ It helps in avoiding the same variable again and again.

✅ It divides one function into multiple functions so that one handles one set of responsibility.

unction sum(a) {

return (b) => {

return (c) => {

return a + b + c

}

}

}

/\* you can call it in two ways\*/

1️⃣ console.log(sum(1)(2)(3)); //6

2️⃣ const sum1 = sum(1);

const sum2 = sum1(2);

const result = sum2(3);

console.log(result); //

<https://www.youtube.com/redirect?event=video_description&redir_token=QUFFLUhqa1NheFRUSlI3QkVaMFdMQ1ctVHpGNTFUbjdjZ3xBQ3Jtc0trRmxaVmdKV3U5NkN0dk5xX1ZQamk5aW96Xzd2WXkyUEcta2FxaGY5Z2VweGpwMkNyVjlaZlRDRUZsMERaY2VxVFZINDh0c2hWVldIYXJQZGdEaVNJUXhnN2FxS1pFSGVrZG5LY3FZSEhMcFc4OGVNbw&q=https%3A%2F%2Froadsidecoder.hashnode.dev%2Fjavascript-interview-questions-currying-output-based-questions-partial-application-and-more&v=k5TC9i5HonI>

//link of currying

Q-javascript object

//https://www.freecodecamp.org/news/objects-in-javascript/

Q-copying

#### Primitive data types

Primitive data types include the following:

* Number — e.g. 1
* String — e.g. 'Hello'
* Boolean — e.g. true
* undefined
* null

When you create these values, they are tightly coupled with the variable they are assigned to. They only exist once. That means you do not really have to worry about copying primitive data types in JavaScript. When you make a copy, it will be a real copy. Let’s see an example:

const a = 5

let b = a // this is the copy

b = 6

console.log(b) // 6

console.log(a) // 5

By executing b = a , you make the copy. Now, when you reassign a new value to b, the value of b changes, but not of a.

#### Composite data types — Objects and Arrays

Technically, arrays are also objects, so they behave in the same way. I will go through both of them in detail later.

Here it gets more interesting. These values are actually stored just once when instantiated, and assigning a variable just creates **a pointer (reference) to that value**.

Now, if we make a copy b = a , and change some nested value in b, it actually changes a’s nested value as well, since a and b actually point to the same thing. Example:

When you have a nested object (or array) and you copy it, nested objects inside that object will not be copied, since they are only pointers / references. Therefore, if you change the nested object, you will change it for both instances, meaning you would end up doing a **shallow copy again**. Example:// BAD EXAMPLE

const a = {

foods: {

dinner: 'Pasta'

}

}

let b = {...a}

b.foods.dinner = 'Soup' // changes for both objects

console.log(b.foods.dinner) // Soup

console.log(a.foods.dinner) // Soup

To make a **deep copy of nested objects**, you would have to consider that. One way to prevent that is manually copying all nested objects:

const a = {

foods: {

dinner: 'Pasta'

}

}

let b = {foods: {...a.foods}}

b.foods.dinner = 'Soup'

console.log(b.foods.dinner) // Soup

console.log(a.foods.dinner) // Pasta

In case you were wondering what to do when the object has more keys than only foods , you can use the full potential of the spread operator. When passing more properties after the ...spread , they overwrite the original values, for example const b = {...a, foods: {...a.foods}}

*What is the difference between event.currentTarget and event.target in JS*

Now every event goes through three phases of event propagation:  
1. From window to the target element phase.

2. The event target phase and

3. From the event target back to the window phase.

event.currentTarget tells us on which element the event was attached or the element whose eventListener triggered the event.

event.target tells where the event started

**Call** is a function that helps you change the context of the invoking function. In layperson's terms, it helps you replace the value of this inside a function with whatever value you want.

func.call(thisObj, args1, args2, ...)

* **func** is a function that needs to be invoked with a different this object
* **thisObj** is an object or a value that needs to be replaced with the this keyword present inside the function func
* **args1, args2** are arguments that are passed to the invoking function with the changed this object.

**Apply** is very similar to the call function. The only difference is that in apply you can pass an array as an argument list.

**Bind** is a function that helps you create another function that you can execute later with the new context of this that is provided.

The main differences between bind() and call() is that the call() method:

1. Accepts additional parameters as well
2. Executes the function it was called upon right away.
3. The call() method does not make a copy of the function it is being called on.

call() and apply() serve the **exact same purpose.** The ***only difference between how they work is that*** call() expects all parameters to be passed in individually, whereas apply() expects an array of all of our parameters.

Q-debouncing in javascript

**What is a debounce function?**

Debounce function limits the execution of a function call and waits for a certain amount of time before running it again.

Let’s take a real world scenario where we use this the most.

Most of the websites have search bar which helps the user to search with a specific keyword to get what they want. If we take e-commerce use-case, when the user tries to search a specific product, one can see multiple suggestions popping up even before he enters the entire keyword.

So what’s happening here is once the user starts typing the keyword we are making API call for each of the character. If the user types “Apple MacBook Pro”, then 17 times (including spaces) we have made API calls to get the results.

**This approach is not ideal even though we have the best intention in mind to display suggestions to user while he still typing**.

We can optimise and reduce the count of API calls by **debounce logic.** We **can wait for certain amount of time before making the next API call.**

Here we monitor the delay user gives between two key presses. If this delay matches our threshold limit, then we make another API call.

Let’s keep 500ms as threshold limit for this delay, it means if the user takes 500ms or more to type the next character, only then we make our next API call. Ideally when the user is typing “Apple MacBook Pro”, he takes 500ms or more delay between each key press is only 3 to 5 times. This way we can reduce the count of API calls from 17 to just 3-5.

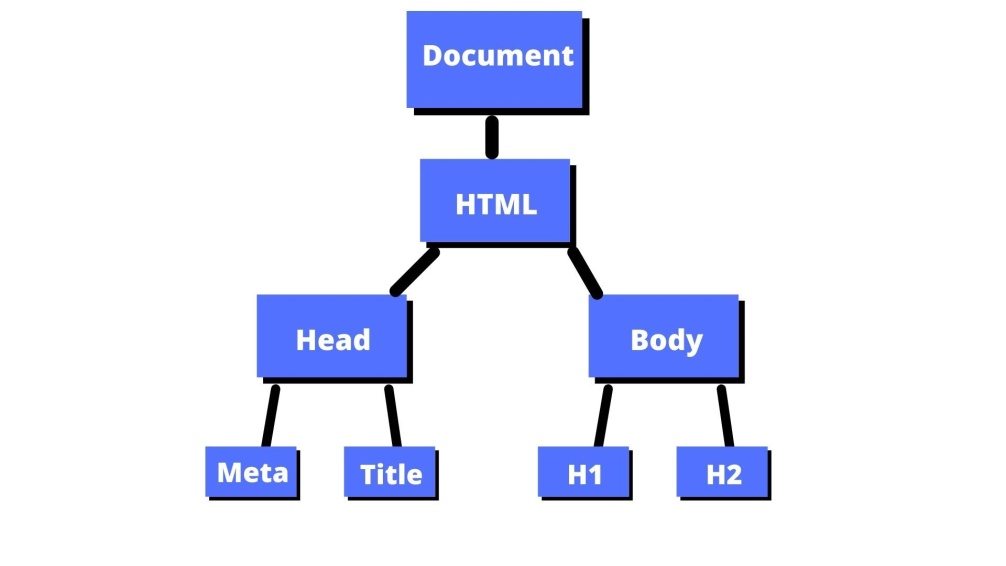
**What is throttle function?**

Throttling is a technique, to limit the execution of an event handler function, even when this event triggers continuously due to user actions. (ex: browser resizing)

## What is the DOM?

DOM stands for Document Object Model. It is a programming interface that allows us to create, change, or remove elements from the document. We can also add events to these elements to make our page more dynamic.

The DOM views an HTML document as a tree of nodes. A node represents an HTML element.



## How to Select Elements in the Document

There are a few different methods for selecting an element in the HTML document.

In this article, we will focus on three of those methods:

* getElementById()
* querySelector()
* querySelectorAll()

### getElementById()

In HTML, ids are used as unique identifiers for the HTML elements. This means you cannot have the same id name for two different elements.

This would be incorrect:

<p id="para">This is my first paragraph.</p>

<p id="para">This is my second paragraph.</p>

You would have to make sure those ids are unique like this:

<p id="para1">This is my first paragraph.</p>

<p id="para2">This is my second paragraph.</p>

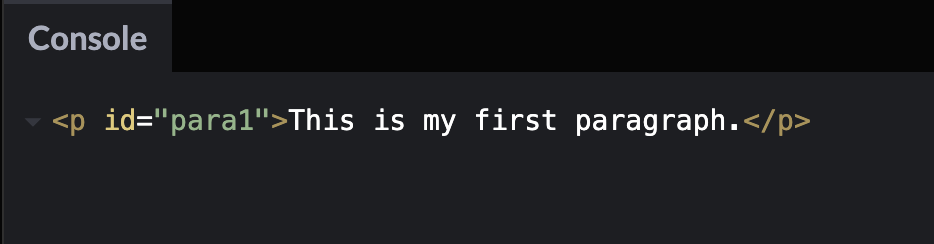
In JavaScript, we can grab an HTML tag by referencing the id name.

document.getElementById("id name goes here")

This code tells the computer to get the <p> element with the id of para1 and print the element to the console.

const paragraph1 = document.getElementById("para1");

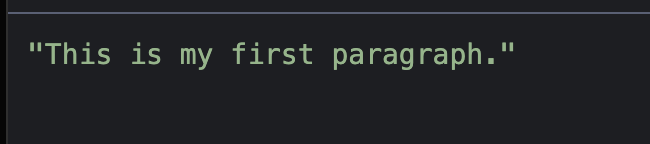
console.log(paragraph1);



If we wanted to just read the content of the paragraph, then we can use the textContent property inside the console.log().

const paragraph1 = document.getElementById("para1");

console.log(paragraph1.textContent);



### querySelector()

You can use this method to find elements with one or more CSS selectors.

I have created this HTML example of my favorite tv shows:

<h1>Favorite TV shows</h1>

<ul class="list">

<li>Golden Girls</li>

<li>Archer</li>

<li>Rick and Morty</li>

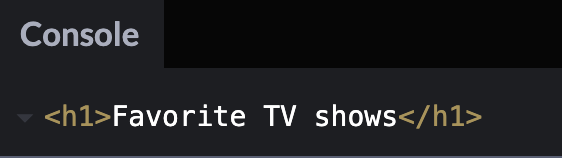
<li>The Crown</li>

</ul>

If I wanted to find and print to the console the h1 element, then I could use that tag name inside the querySelector().

const h1Element = document.querySelector("h1");

console.log(h1Element);

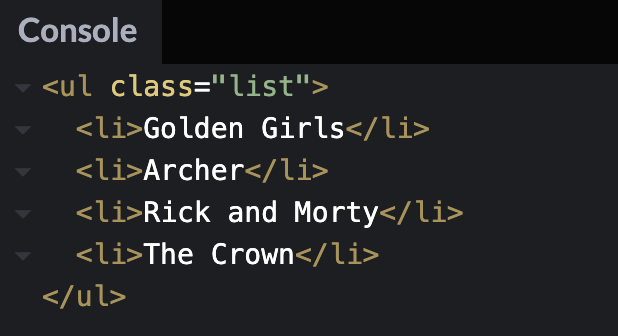


If I wanted to target the class="list" to print out the unordered list to the console, then I would use .list inside the  querySelector().

The . before list tells the computer to target a class name. If you wanted to target an id then you would use a # symbol before the name.

const list = document.querySelector(".list");

console.log(list);



### querySelectorAll()

This method finds all of the elements that match the CSS selector and returns a list of all of those nodes.

If I wanted to find all of the <li> items in our example, I could use the > child combinator to find all of the children of the <ul>.

const listItems = document.querySelectorAll("ul > li");

console.log(listItems);



If we wanted to print out the actual <li> items with the tv shows, we can use a forEach() to loop over the NodeList and print out each item.

const listItems = document.querySelectorAll("ul > li");

listItems.forEach((item) => {

console.log(item);

});



## How to Add New Elements to the Document

This is what the code looks like all together.

let unorderedList = document.createElement("ul");

document.body.appendChild(unorderedList);

let listItem1 = document.createElement("li");

listItem1.textContent = "It's free";

unorderedList.appendChild(listItem1);

let listItem2 = document.createElement("li");

listItem2.textContent = "It's awesome";

unorderedList.appendChild(listItem2);

This is what the output looks like on the page:

## How to Use the Style Property to Change Inline CSS Styles

The style property gives you the ability to change the CSS in your HTML document.

In this example, we are going to change the h1 text from black to blue using the style property.

Here is our HTML.

<h1>I was changed to blue using JavaScript</h1>

We first need to grab the h1 tag using the querySelector() method.

const h1 = document.querySelector("h1");

We then use h1.style.color to change the h1 text from black to blue.

const h1 = document.querySelector("h1");

h1.style.color = "blue";

This is what the result looks like in the browser:



You can use this style property to change a number of CSS inline styles including background-color, border-style, font-size and more.

## How to Use addEventListener() to Listen for Events on the Page

This method allows you to attach an event to an HTML element like a button.

In this example, when the user clicks on the button, an alert message will pop up.

In our HTML, we have a button element with the id of btn.

<button id="btn">Show alert</button>

We can target that element in our JavaScript using the getElementById() method and assigning that to the variable called button.

const button = document.getElementById("btn");

The addEventListener() takes in an event type and a function. The event type will be a click event and the function will trigger the alert message.

This is the code to add the event listener to the button variable.

button.addEventListener("click", () => {

alert("Thank you for clicking me");

});

This is the complete code where you can click the button and the alert message will pop up:

# Understanding Event Loop, Call Stack, Event & Job Queue in Javascript

In this article we will dig into how javascript works under the hood, how it executes our asynchronous javascript code, and in what order (Promise vs setTimeout), how it generates stack trace and much more..

As most developers know, **that Javascript is single threaded, means, two statement in javascript can not be executed in parallel**. Execution happens line by line, which means each javascript statements are synchronous and blocking. But there is a way to run your code asynchronously, if you use setTimeout() function, a Web API given by browser, which makes sure that your code executes after specified time (in millisecond). Example code:

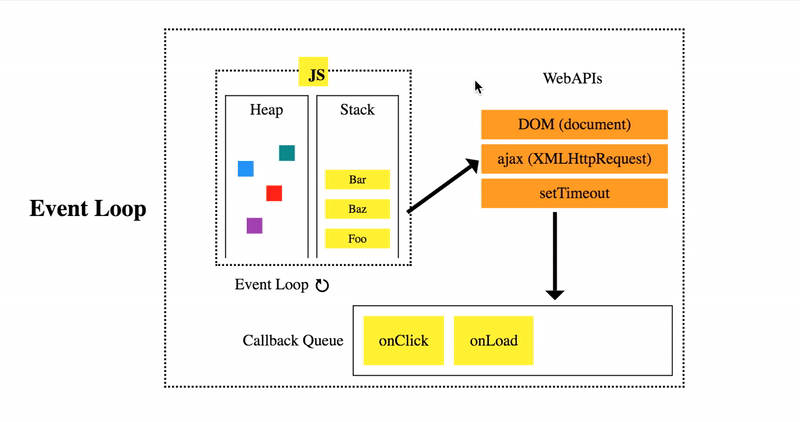
console.log('Message 1');// Print message after 100 millisecond  
setTimeout(function() {  
 console.log('Message 2');  
}, 100);console.log('Message 3');

setTimeout takes a callback function as first parameter, and time in millisecond as second parameter.

After executing above statements, browser will print “Message 1” & “Message 3” first, then it will print “Message 2”. This is where event loop comes in, which makes sure your asynchronous code runs after all the synchronous code is done executing.

# ****Event Loop Visualisation****

I have created the structure of Event loop using HTML and CSS. You can check it out on [Codepen](https://codepen.io/rahul-sagore/pen/jpaxGN" \t "_blank) :



**Stack:**This is where all your javascript code gets pushed and executed one by one as the interpreter reads your program, and gets popped out once the execution is done. If your statement is asynchronous: setTimeout, ajax(), promise, or click event, then that code gets forwarded to Event table, this table is responsible for moving your asynchronous code to callback/event queue after specified time.

**Heap:**This is where all the memory allocation happens for your variables, that you have defined in your program.

**Callback Queue:**This is where your asynchronous code gets pushed to, and waits for the execution.

**Event Loop:**Then comes the Event Loop, which keeps running continuously and checks the Main stack, if it has any frames to execute, if not then it checks Callback queue, if Callback queue has codes to execute then it pops the message from it to the Main Stack for the execution.

**Job Queue:**Apart from Callback Queue, browsers have introduced one more queue which is “Job Queue”, reserved only for new Promise() functionality. So when you use promises in your code, you add .then() method, which is a callback method. These `thenable` methods are added to Job Queue once the promise has returned/resolved, and then gets executed.

**Quick Question now:** Check these statements for example, can you predict the sequence of output?:

console.log('Message no. 1: Sync');setTimeout(function() {  
 console.log('Message no. 2: setTimeout');  
}, 0);var promise = new Promise(function(resolve, reject) {  
 resolve();  
});promise.then(function(resolve) {  
 console.log('Message no. 3: 1st Promise');  
})  
.then(function(resolve) {  
 console.log('Message no. 4: 2nd Promise');  
});console.log('Message no. 5: Sync');

Some of you might answer this:

// Message no. 1: Sync  
// Message no. 5: Sync  
**// Message no. 2: setTimeout**  
// Message no. 3: 1st Promise  
// Message no. 4: 2nd Promise

because setTimeout was pushed to Callback Queue first, then promise was pushed. But this is not the case, the output will be:

// Message no. 1: Sync  
// Message no. 5: Sync  
// Message no. 3: 1st Promise  
// Message no. 4: 2nd Promise  
**// Message no. 2: setTimeout**

All `thenable` callbacks of the promise are called first, then the setTimeout callback is called.

**Why?:** Job Queue has high priority in executing callbacks, if event loop tick comes to Job Queue, it will execute all the jobs in job queue first until it gets empty, then will move to callback queue.