IMportant

============

<https://deepu.tech/concurrency-in-modern-languages-js/>

<https://levelup.gitconnected.com/javascript-call-stack-explained-a6e0ae1574be>

<https://felixgerschau.com/javascript-event-loop-call-stack/>

https://developer.mozilla.org/en-US/docs/Glossary/Call\_stack

===========

Javascript is a single threaded. Javascript does one single thing at a time.

## **Concurrency /The event loop in JavaScript & NodeJS**

*JavaScript has a concurrency model based on an event loop, which is responsible for executing the code, collecting and processing events, and executing queued sub-tasks. [জাভাস্ক্রিপ্টের একটি ইভেন্ট লুপের উপর ভিত্তি করে একটি কনকারেন্সি মডেল রয়েছে, যা কোড চালানো, ইভেন্ট সংগ্রহ ও প্রক্রিয়াকরণ এবং সারিবদ্ধ সাব-টাস্কগুলি চালানোর জন্য দায়ী।]*

\*\* The **call stack** keeps track of functions to be executed. When we call a function, it’s added or *pushed* to the top of the call stack. When the function returns, it’s removed, or *popped* from the call stack. Any asynchronous functions (fetch, setTimeout, async, etc.) are moved to the **event queue** (more on that later).(কল স্ট্যাক কার্যকর করা ফাংশন ট্র্যাক রাখে। যখন আমরা একটি ফাংশন কল করি, তখন এটি যোগ করা হয় বা কল স্ট্যাকের শীর্ষে ঠেলে দেওয়া হয়। যখন ফাংশনটি ফিরে আসে, এটি সরানো হয়, বা কল স্ট্যাক থেকে পপ করা হয়। যেকোনো অ্যাসিঙ্ক্রোনাস ফাংশন (ফেচ, সেটটাইমআউট, অ্যাসিঙ্ক, ইত্যাদি) ইভেন্ট সারিতে সরানো হয়।)

\*\***How does the Call Stack Work?**

You might have heard that JavaScript is single-threaded. This means it has only one call stack and it can only process one statement at a time. The call stack follows the LIFO (Last In, First Out) principle, which means it will always process the call on top of the stack first.

When a function is called it’s added to the stack. When a function calls *another* function, it’s added on top of the calling function.

The Call Stack is a mechanism for the JavaScript Engine to keep track of execution contexts, which to enter, which to exit, or which to return to.

\***How does the Event Loop work?**

When an async function is called it does not go to the call stack. Instead, it’s forwarded to the **event queue**. The connection between the call stack and the event queue is facilitated by the **event loop**. The event loop is constantly checking the call stack. If the call stack is empty it adds the first function in the event queue to the call stack for execution. Otherwise, it continues processing code in the call stack.(যখন একটি async ফাংশন কল করা হয় তখন এটি কল স্ট্যাকে যায় না। পরিবর্তে, এটি ইভেন্ট সারিতে পাঠানো হয়েছে। কল স্ট্যাক এবং ইভেন্ট সারির মধ্যে সংযোগ ইভেন্ট লুপ দ্বারা সহজতর হয়। ইভেন্ট লুপ ক্রমাগত কল স্ট্যাক চেক করছে। কল স্ট্যাক খালি থাকলে এটি কার্যকর করার জন্য কল স্ট্যাকে ইভেন্ট সারিতে প্রথম ফাংশন যোগ করে। অন্যথায়, এটি কল স্ট্যাকে কোড প্রক্রিয়াকরণ অব্যাহত রাখে।)

\*\***What is the Event Queue?**

The event queue is where the async code waits for execution. Unlike the call stack, the **event queue** follows the FIFO (First In, First Out) principle, which means it will always process calls in the same order they were added to the queue.

It’s important to note that the event queue will only begin executing when the call stack is empty. If the call stack is already processing code, the event loop will not add any functions from the event queue. The event loop will not resume until the call stack has cleared.(ইভেন্ট সারি যেখানে অ্যাসিঙ্ক কোডটি কার্যকর হওয়ার জন্য অপেক্ষা করে। কল স্ট্যাকের বিপরীতে, ইভেন্ট সারিটি FIFO (ফার্স্ট ইন, ফার্স্ট আউট) নীতি অনুসরণ করে, যার অর্থ এটি সর্বদা কলগুলিকে একই ক্রমে প্রসেস করবে যেগুলি সারিতে যুক্ত করা হয়েছিল।

এটা মনে রাখা গুরুত্বপূর্ণ যে ইভেন্ট সারি শুধুমাত্র কল স্ট্যাক খালি হলেই কার্যকর করা শুরু করবে। যদি কল স্ট্যাকটি ইতিমধ্যেই কোড প্রসেসিং করে থাকে, তাহলে ইভেন্ট লুপ ইভেন্ট সারি থেকে কোনো ফাংশন যোগ করবে না। কল স্ট্যাক পরিষ্কার না হওয়া পর্যন্ত ইভেন্ট লুপ পুনরায় শুরু হবে না।

)

Var is a local and global variable. When using var inside in the function that is called a local variable and use outside in the function that is called a global variable.

Let is a local variable is not global. That is a block scoped.

* Use const when you do not want to reassign a variable.
* Use let when you want to reassign a variable.

Const variable declare all time uppercase

* JavaScript engine uses a call stack to manage execution contexts.
* The call stack uses the stack data structure that works based on the LIFO (last-in first-out) principle.

JavaScript is a single-threaded programming language. This means that JavaScript can do only one thing at a single point in time.

<https://www.atatus.com/blog/javascript-execution-context/>

<https://blog.bitsrc.io/understanding-execution-context-and-execution-stack-in-javascript-1c9ea8642dd0>

<https://medium.com/@happymishra66/execution-context-in-javascript-319dd72e8e2c>

<https://www.freecodecamp.org/news/javascript-execution-context-and-hoisting/>

What is Hoisting?

Hoisting is a mechanism where variables and function declarations are moved to the top of their scope before executing any code. In JavaScript, all declarations (functions, variables, or classes) are "hoisted" to the top of their scope. In other words, they are parsed or evaluated before any other code is executed. This is opposed to expressions, which are evaluated inline. All expressions in JavaScript are initially set to undefined.

Hoisting হল একটি মেকানিজম যেখানে ভেরিয়েবল এবং ফাংশন ডিক্লেয়ারেশন কোন কোড এক্সিকিউট করার আগে তাদের সুযোগের শীর্ষে নিয়ে যাওয়া হয়। জাভাস্ক্রিপ্টে, সমস্ত ঘোষণা (ফাংশন, ভেরিয়েবল, বা ক্লাস) তাদের সুযোগের শীর্ষে "উদ্ধার" করা হয়। অন্য কথায়, অন্য কোনো কোড কার্যকর করার আগে সেগুলি পার্স বা মূল্যায়ন করা হয়। এটি অভিব্যক্তির বিরোধী, যা ইনলাইনে মূল্যায়ন করা হয়। জাভাস্ক্রিপ্টের সমস্ত অভিব্যক্তি প্রাথমিকভাবে অনির্ধারিত সেট করা হয়েছে।

**Execution context** (**EC**)is defined as the environment in which the JavaScript code is executed. By environment, I mean the value of this, *variables*, *objects*, and *functions* JavaScript code has access to at a particular time.

Execution context in JavaScript is of three types as:

1. **Global execution context** (**GEC**): This is the default execution context in which JS code starts its execution when the file first loads in the browser. All of the global code i.e. code that is not inside any function or object is executed inside the global execution context. GEC cannot be more than one because only one global environment is possible for JS code execution as the JS engine is single-threaded.
2. **Functional execution context** (**FEC**): Functional execution context is defined as the context created by the JS engine whenever it finds any function call. Each function has its own execution context. It can be more than one. The functional execution context has access to all the code of the global execution context though vice versa is not applicable. While executing the global execution context code, if the JS engine finds a function call, it creates a new functional execution context for that function. In the browser context, if the code is executing in strict mode value of thisis undefinedelse it is windowobject in the function execution context.

### **Function Execution Context (FEC)**

Whenever a function is called, the JavaScript engine creates a different type of Execution Context known as a Function Execution Context (FEC) within the GEC to evaluate and execute the code within that function.

1. **Eval**: Execution context inside eval function.

# **Execution Context**

Execution context is something that you need to know to understand how JavaScript code runs. First of all, it is an ***abstract*** *concept* that represents the *environment* in which JavaScript runs. What happens inside the execution context is two things basically. The first is parsing the code line by line and the second is storing the variables and functions in the memory. In simple terms, it has two types:

* Global Execution Context
* Local Execution Context

**Global Execution Context** is the first thing that is created when you write JavaScript code. It is the default context.

**Local Execution Context** is created when you call a function (not defining a function).

**Execution context stack (ECS):** Execution context stack is a stack data structure, i.e. last in first out data structure, to store all the execution stacks created during the life cycle of the script. Global execution context is present by default in execution context stack and it is at the bottom of the stack. While executing the global execution context code, if JS engines find a function call, it creates a functional execution context for that function and pushes it on top of the execution context stack. JS engine executes the function whose execution context is at the top of the execution context stack. Once all the code of the function is executed, JS engines pop out that function’s execution context and start executing the function which is below it.

When the JavaScript engine executes the JavaScript code, it creates execution contexts.

**What is an Execution Context?**

Simply put, an execution context is an abstract concept of an environment where the Javascript code is evaluated and executed. Whenever any code is run in JavaScript, it’s run inside an execution context.

**What is Execution Context?**

The concept of an execution context is used to describe how a code works inside. JavaScript execution context refers to the environment that allows JavaScript code to be executed.

As a result, the execution context creates an environment where the given code can run.

## **Global Execution Context (GEC) in JavaScript**

Whenever we execute JavaScript code, it creates a Global Execution Context (also known as Base Execution Context). The global execution context has two phases.

**Creation Phase**

In the creation phase, two unique things get created:

* A global object is called a window (for the client-side JavaScript).
* A global variable is called this.

## **Function Execution Context(FEC)**

A Function Execution Context gets created when a function is invoked.

Anonymous functions are functions without names

**A function that calls itself is called a recursive function**.

# **Default parameters**

Default function parameters allow named parameters to be initialized with default values if no value or undefined is passed.

# **Arrow functions vs regular functions in JavaScript**

#### **Following are the main differences:**

* Syntax
* Arguments binding
* Use of this keyword
* Using a new keyword
* No duplicate named parameters

# **“arguments” binding:**

Arguments objects are not available in arrow functions but are available in regular functions.

#### **4) Using a new keyword**

Regular functions created using function declarations or expressions are constructible and callable. Regular functions are constructible; they can be called using the new keyword.

However, the arrow functions are only callable and not constructible, i.e., arrow functions can never be used as constructor functions.

#### **5) No duplicate named parameters**

Arrow functions can never have duplicate named parameters, whether in strict or non-strict mode.

However, We can use duplicate named parameters for regular function in non-strict mode.

# **JavaScript Promise and Promise Chaining**

In JavaScript, a promise is a good way to handle asynchronous operations. It is used to find out if the asynchronous operation is successfully completed or not.

A promise may have one of three states.

* Pending
* Fulfilled
* Rejected

A promise starts in a pending state. That means the process is not complete. If the operation is successful, the process ends in a fulfilled state. And, if an error occurs, the process ends in a rejected state.

For example, when you request data from the server by using a promise, it will be in a pending state. When the data arrives successfully, it will be in a fulfilled state. If an error occurs, then it will be in a rejected state.

## **JavaScript Promise Chaining**

Promises are useful when you have to handle more than one asynchronous task, one after another.

# **await**

The await operator is used to wait for a [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise). It can only be used inside an [async function](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function) 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)

## the async keyword

The async keyword allows you to define a function that handles asynchronous operations.

To define an async function, you place the async keyword in front of the function keyword as follows:

async function sayHi() {

return 'Hi';

}

Asynchronous functions execute asynchronously via the [event loop](https://www.javascripttutorial.net/javascript-event-loop/). It always returns a Promise.

## The await keyword

You use the await keyword to wait for a Promise to settle either in resolved or rejected state. And you can use the await keyword only inside an async function:

async function display() {

let result = await sayHi();

console.log(result);

}

Code language: JavaScript (javascript)

In this example, the await keyword instructs JavaScript engine to wait for the sayHi() function to complete before displaying the message.

Note that if you use the await operator outside of an async function, you will get an error.

A Generator

A generator is a function that **can stop midway** and then continue *from where it stopped.* **In short, a generator *appears* to be a function but it *behaves* like an iterator.**

**Using a generator,**

* **you can stop the execution of a function from anywhere inside the function**
* **and continue executing code from a halted position**

## Introduction to the JavaScript yield keyword

**The yield keyword allows you to pause and resume a** [**generator**](https://www.javascripttutorial.net/es6/javascript-generators/) **function (function\*).**

#### **What is a Javascript Set Object ?**

**In Mathematics, a set is a collection of distinct elements. The elements may be similar or dissimilar types.**

**Set that stores a collection of unique values of any type.**

# **JavaScript Map**

**In this tutorial, you will learn about JavaScript Maps and WeakMaps with the help of examples.**

**Map is similar to objects in JavaScript that allows us to store elements in a key/value pair.**

## **JavaScript try...catch Statement**

**The try...catch statement is used to handle the exceptions. Its syntax is:**

**try {**

**// body of try**

**}**

**catch(error) {**

**// body of catch**

**}**

**The main code is inside the try block. While executing the try block, if any error occurs, it goes to the catch block. The catch block handles the errors as per the catch statements.**

## **JavaScript try...catch...finally Statement**

**You can also use the try...catch...finally statement to handle exceptions. The finally block executes both when the code runs successfully or if an error occurs.**

**The syntax of try...catch...finally block is:+**

# **JavaScript throw Statement**

# **In JavaScript, the throw statement handles user-defined exceptions.**

## What is reflection

**In computer programming, reflection is the ability of a program to manipulate** [**variables**](https://www.javascripttutorial.net/javascript-variables/)**, properties, and methods of** [**objects**](https://www.javascripttutorial.net/javascript-objects/) **at runtime.**

What is a JavaScript Proxy object

**A JavaScript Proxy is an** [**object**](https://www.javascripttutorial.net/javascript-objects/) **that wraps another object (target) and intercepts the fundamental operations of the target object.**

## What is implicit and explicit binding? Manual processes

## **Arguments**

**arguments** is an object that is accessible inside functions that contain the values of the arguments passed to that function.

## **The Arguments Object**

## The **Arguments object** contains the values of the arguments in an Array-like object. Note that it is not an Array but an Array-like object.

**arguments** is an object that is accessible inside functions that contain the values of the arguments passed to that function.

যে ফাংশনে পাস করা আর্গুমেন্টের মান রয়েছে।

**The answer is short and simple:** Scope pertains to the visibility of variables, and context refers to the object to which a function belongs.

## **Context in JavaScript**

Context is related to objects. It refers to the object to which a function belongs. When you use the JavaScript “this” keyword, it refers to the object to which function belongs.

<https://www.freecodecamp.org/news/javascript-lexical-scope-tutorial/>

<https://www.educative.io/answers/lexical-scope-in-javascript>

*Lexical scope* means that in a nested group of functions, **the inner functions have access to the variables and other resources of their parent scope**.

**Lexical scope** is the ability for a function scope to access variables from the parent scope.

var func = function() {

var b =20;

console.log("a and b is accessible outer:" , a, b);

var innerFunc = function(){

var c = 30;

console.log("a and b and c is accessible inner:", a,b,c);

}

innerFunc();

return;

}

undefined

func();

\_app-97fa6b4822a15531e07d.js:1 a and b is accessible outer: 10 20

\_app-97fa6b4822a15531e07d.js:1 a and b and c is accessible inner: 10 20 30

undefined

console.log("only a is accessible global", a);

\_app-97fa6b4822a15531e07d.js:1 only a is accessible global 10

Undefined

## Introduction to JavaScript prototype

In JavaScript, objects can inherit features from one another via prototypes. Every object has its own property called prototype.

Prototypes are the mechanism by which JavaScript objects inherit features from one another.

In JavaScript, every function and object has a property named prototype by default.

**Aggregation**

Aggregation is when an object is formed from an enumerable collection of subobjects. An aggregate is an object which *contains* other objects. Each subobject in an aggregation retains its own reference identity and could be losslessly destructured from the aggregate.

The ability to combine several objects into a new one is known as aggregation or composition.

**Aggregation**

The ability to combine several objects into a new one is known as aggregation or composition.

Multiple small objects are easier to manage than one large object.

There are several ways to do with JavaScript.

We can use the spread operator or Object.assign method to combine multiple objects into one.

To use the Object.assign method, we can write:

const obj1 = {

foo: 1

}

const obj2 = {

bar: 2

}

const obj3 = {

bax: 3

}

const merged = Object.assign({}, obj1, obj2, obj3);

We have 3 objects, and we passed them all into Object.assign so we can merge them together and returns a new object with the properties of all of them.

Also, we can use the spread operator by writing:

const obj1 = {

foo: 1

}

const obj2 = {

bar: 2

}

const obj3 = {

bax: 3

}

const merged = {

...obj1,

...obj2,

...obj3

};

This will also combine the 3 objects together.

Another way to aggregate objects is having child objects by having as properties of another object.

For instance, a Book object can have multiple Author objects, Publisher objects, Chapter objects, and so on.

### **let and const**

let and const create block-scoped variables. When declared within a block, they are only accessible within that block. This behavior was demonstrated in our previous examples:

## Block Scope

Before ES6 (2015), JavaScript had only Global Scope and Function Scope.

ES6 introduced two important new JavaScript keywords: let and const.

These two keywords provide Block Scope in JavaScript.

Variables declared inside a { } block cannot be accessed from outside the block

## **Shallow copy**

Shallow copy is a bit-wise copy of an object. A new object is created that has an exact copy of the values in the original object. If any of the fields of the object are references to other objects, just the reference addresses are copied i.e., only the memory address is copied.

## **Deep copy**

A deep copy copies all fields, and makes copies of dynamically allocated memory pointed to by the fields. A deep copy occurs when an object is copied along with the objects to which it refers.

## **Accessor Property**

In JavaScript, accessor properties are methods that get or set the value of an object. For that, we use these two keywords:

* get - to define a getter method to get the property value
* set - to define a setter method to set the property value

### **JavaScript Getter**

* In JavaScript, getter methods are used to access the properties of an object.

<https://www.programiz.com/javascript/getter-setter#:~:text=In%20JavaScript%2C%20accessor%20properties%20are,to%20set%20the%20property%20value>

const student = {

// data property

firstName: 'Monica',

// accessor property(getter)

get getName() {

return this.firstName;

}

};

## **JavaScript Setter**

In JavaScript, setter methods are used to change the values of an object.

const student = {

firstName: 'Monica',

//accessor property(setter)

set changeName(newName) {

this.firstName = newName;

}

};

console.log(student.firstName); // Monica

// change(set) object property using a setter

student.changeName = 'Sarah';

console.log(student.firstName); // Sarah

// accessing data property

console.log(student.firstName); // Monica

// accessing getter methods

console.log(student.getName); // Monica

// trying to access as a method

console.log(student.getName()); // error

Heap: This is where memory allocation for all your variables take place.

Stack: This is where all the javascript code is pushed and gets executed line by line and is popped out after its execution. Javascript maintain the order of execution contexts using this stack only. It is also known as program stack or execution context stack.

Callback Queue: This is the place where all the asynchronous code goes. The callback functions from setTimeout, setInterval, HTML events and ajax requests are pushed into this queue when they are ready for execution.

Event Loop: The main function of event loop is to check the program stack and find if it is empty. If the stack is empty, then it checks the callback queue. If the callback queue is not empty then it pushes the code block from callback queue to main stack and executes it sequentially.

**Prototype**

In JavaScript, every function and object has a property named prototype by default. Whenever we create a JavaScript function, JavaScript adds a prototype property to that function. A prototype is an object, where it can add new variables and methods to the existing object. I.e. Prototype is a base class for all the objects, and it helps us to achieve the inheritance.

(

জাভাস্ক্রিপ্টে, প্রতিটি ফাংশন এবং অবজেক্ট ডিফল্টভাবে প্রোটোটাইপ নামে একটি বৈশিষ্ট্য থাকে। যখনই আমরা একটি জাভাস্ক্রিপ্ট ফাংশন তৈরি করি, জাভাস্ক্রিপ্ট সেই ফাংশনে একটি প্রোটোটাইপ বৈশিষ্ট্য যোগ করে। একটি প্রোটোটাইপ একটি বস্তু, যেখানে এটি বিদ্যমান বস্তুতে নতুন ভেরিয়েবল এবং পদ্ধতি যোগ করতে পারে। অর্থাৎ, প্রোটোটাইপ হল সমস্ত বস্তুর জন্য একটি বেস ক্লাস, এবং এটি আমাদের উত্তরাধিকার অর্জন করতে সাহায্য করে।

)

In JavaScript, every function and object has a property named prototype by default.

In JavaScript, a prototype can be used to add properties and methods to a constructor function. And objects inherit properties and methods from a prototype.

Prototypes are the mechanism by which JavaScript objects inherit features from one another.

JavaScript is a prototype based language (not class based like Java, C++, etc. even though from ES6 the class feature has been added behind the scene it also uses prototypes). Everything in JavaScript are objects. All objects in JavaScript are instances of Object.

When an object (i.e: Array, Function, Object, Number, etc ) is created that inherits properties and methods from Object.prototype. For example, whenever we create a function in JavaScript, the JavaScript engine adds a prototype property to the function. The prototype property is basically an object where we can attach methods and properties which will be shareable across all the instances of that constructor function.

**main difference between regular function and arrow function**

[**https://blog.alexdevero.com/category/javascript/page/2/**](https://blog.alexdevero.com/category/javascript/page/2/)

[**https://blog.alexdevero.com/regular-arrow-functions-differences/**](https://blog.alexdevero.com/regular-arrow-functions-differences/)

### **The *this* and arrow functions**

**When it comes to this, arrow function is much simpler and always behaves in the same way. The value of this is always the value from the parent, or outer, function. This is because arrow function doesn’t have its own this. It “gets” its this lexically, from its lexical scope, outer scope.**

**If you try to change this of an arrow function with call() or apply(), arrow function will ignore it. It will still get its this from its lexical scope.**

**Arrow functions do not have an arguments binding.**

**, the arrow functions are only callable and not constructible**

~~Data types~~

~~Dynamic typing~~

~~\*\*\*Concurrency Model and Event Loop~~

~~\*\*What is implicit and explicit binding?~~

~~\*\*\*The Difference Between Regular Functions and Arrow Functions~~

~~\*\*\*Blocking & Non-blocking~~

~~\*\*\*What is hoisting?~~

~~What is context?~~

~~What is execution context?~~

~~\*\*\*What is lexical scope in JavaScript?~~

~~\*\*\*What is closure in JavaScript?~~

Operators:

Loops:

~~Dynamically adding properties to objects:~~

~~Variable scope:~~

Constructors:

~~\*\*Prototypes~~:

~~\*\*Composition Vs Inheritance~~

~~What is aggregation in javascript?~~

~~\*\*\*`this` and `new` keywords:~~

~~\*\*\*Promises in JavaScript:~~

~~Shallow and deep copy in JavaScript~~

~~Getters & Setters in JavaScript~~

Array functions

Static methods

JavaScript Higher Order Functions

ES6, ES7, ES8, ES9, ES10