
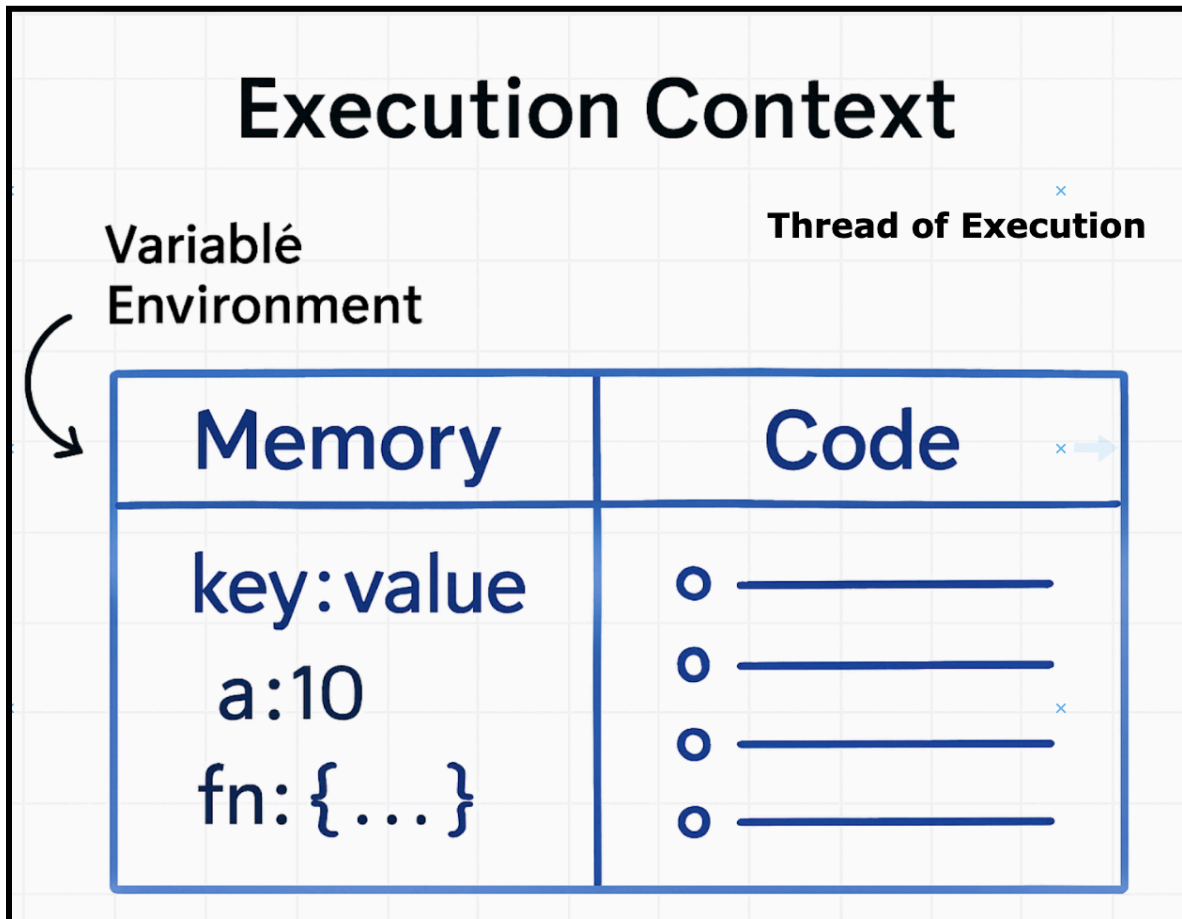


JS

Execution Context

 Everything in JS happens inside an execution context



An execution context is the environment in which JavaScript code runs.

There are three main types:

1. **Global Execution Context (GEC)** – Created when the JS file first runs. It sets up the global object (window in browsers) and this.
2. **Function Execution Context (FEC)** – Created every time a function is called. It has its own scope and variables.

3. **Eval Execution Context** – Rarely used, created inside eval().

Execution happens in **two** phases:

1. **Creation Phase** – Memory is allocated, and hoisting happens.
2. **Execution Phase** – Code runs line by line.

All contexts are managed in a **call stack** (LIFO). When a function is called, a new context is pushed to the stack; when it returns, it's popped off.

Example

For this Function,

```
var n = 2;
function square (num) {
    var ans = num * num;
    return ans;
}

var sq2 = square(n);
var sq4 = square(4);
```

1. **Global Execution Context (GEC)**

Created when the script runs.

Creation Phase:

- > n is declared → initialized to undefined
- > square is hoisted → function is stored
- > sq2 and sq4 declared → initialized to undefined

Execution Phase:

- > $n = 2$
- > `square(n)` is called → New Function Execution Context (FEC1)
- > `square(4)` is called → New Function Execution Context (FEC2)

Function Execution Context (FEC1) for `square(n)`

Creation Phase:

- > Arguments: $\text{num} = 2$
- > `ans` declared → undefined

Execution Phase:

- > $\text{ans} = 2 * 2 = 4$
- > `return 4` → returned to Global EC → $\text{sq2} = 4$

Function Execution Context (FEC2) for `square(4)`

Creation Phase:

- > Arguments: $\text{num} = 4$
- > `ans` declared → undefined

Execution Phase:

- > $\text{ans} = 4 * 4 = 16$
- > `return 16` → returned to Global EC → $\text{sq4} = 16$



Return will hand over the execution control to where the function was invoked.


 Once execution is completed, the function's execution context will be deleted.

Table for EC

Code Type	New Scope?	New Execution Context?
function myFunc() {}	Yes	Yes
if (true) {}	Yes	No
for (...) {}	Yes	No
while (...) {}	Yes	No
Global code	Yes	Yes

Recap:

What Happens During Execution Context Creation?

Every time you call a function or run a script, JS creates an Execution Context, and this happens in two phases:

1. Memory Creation Phase (aka Variable Environment Creation)

Here's what JS does:

What	Where
Reserves memory for variables, functions	Stack
Sets up Lexical Environment & Scope Chain	Heap (for closures and functions)
Hoists functions and variables	Initialized as undefined for var , and actual value for function

During this phase, variables are *declared*, not assigned their final values.

Where is memory allocated?

> Primitives (like numbers, strings) → stored in the Stack.

> Objects and functions → stored in the Heap, and the stack stores references to them.

Example:

```
function greet() {
  const name = "JavaScript";
  const obj = { lang: "JS" };
}
```


Memory Type	What gets stored
-------------	------------------

Stack	<code>name = "JavaScript"</code> (primitive), <code>obj =</code> reference to heap
Heap	<code>{ lang: "JS" }</code> object

2. Execution Phase (Code is actually run)

Now JS:

- > Assigns values
- > Runs code line by line
- > Resolves references

 We using the heap during memory creation phase only for functions and objects that are declared (including closures and function expressions).

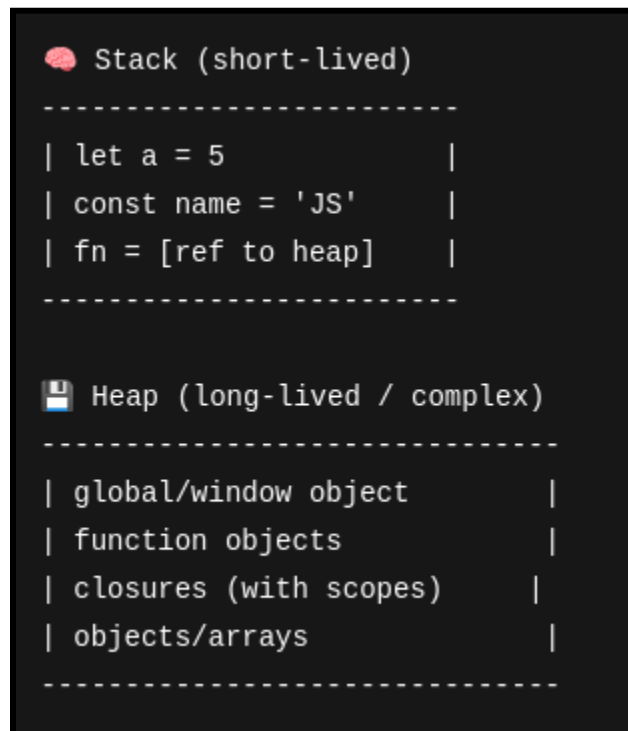
Variable initialization

Type	Stored In	Initialized When
<code>var x = 10;</code>	Stack (primitive)	<code>undefined</code> in memory phase, <code>10</code> in execution
<code>let y = 20;</code>	Stack (Temporal Dead Zone until initialized)	Only in execution phase

<code>const obj = {}</code>	Stack holds reference	Object in Heap, initialized in execution phase
<code>function hello() {}</code>	Stack with reference to heap	Fully hoisted in memory phase

In Short,

- > During the Memory Creation Phase, JS reserves stack space for variables.
- > Any objects/functions/closures go into the heap, and the stack just stores references.
- > Heap is used when something needs to persist or be referenceable, like closures or objects.



Call Stack

JS executes all the code by fetching it from the call stack, blocking main thread means, executing a long process that is pushed in the call stack.

JS creates a global execution context when the file runs.

This stays at the bottom of the stack.

Only synchronous code runs on the Call Stack

Async code (e.g., setTimeout, Promises) gets handled by the Web APIs + Task Queue.



Call stack runs the callback when it's empty

Hoisting

Hoisting happens in the Memory Creation Phase

Accessing variables (only undefined) and functions before initialization.

In the Memory Creation Phase, we can access variables, functions before initialization in the code. For variables, it is undefined, for functions it is the whole code.



Arrow and variable functions act as variables only, so their value is undefined.

Works for var, function declarations, but not for let, const, or arrow functions the same way.

let and const — Hoisting Behavior

1. During the Memory Creation Phase, they are hoisted but not initialized.
 2. They stay in the Temporal Dead Zone (TDZ) — a time between hoisting and actual declaration.
-

Window Object



Global space: anything not present in class/ function. Direct variables / functions.

Window object is the global object, created by js engine, at global space, it can access anywhere.

It acts as the top-level container for all global variables, functions, and browser APIs

At global space, “ this === window ” => true

Not defined: No variable/ function defined in the code.

Undefined: Variable defined, but not initialized yet. Js will allocate the undefined by default in the Memory creation phase.

Browser gives these WEB APIs to JS to use in code through the “window” global object.




Any var-declared variable or function at global scope becomes a property of window.

There is no window in Node.js. Instead, the global object is called global.

Lexical Environment & Scope chain

 Js will search for variable values from lower level to parent level.

 Lexical environment is nothing but the **local memory + lexical environment of the parent.**

```
function a() {  
  var x = 10;  
  function c() {  
    var y = 20;  
    console.log(x); // 🔍 Searches c → a → global  
  }  
  c();  
}  
a();
```

> Lexical Env of c:

Local: { y: 20 }

Outer: Lexical Env of a

> Lexical Env of a:

Local: { x: 10 }

Outer: Global

Here, the lexical environment of c is the local memory of the c + lexical environment of function “a”.

The lexical environment of “a” is the local memory of a + Global execution object.

Whenever the execution context is created, we will get the reference to the lexical environment of the parent.

Scope chain is searching the variable value from local scope to its parent scope to its grandparent and go on. If a variable is not found in any of the lexical environments, a variable not defined error will come.

Let & Const

Temporal Dead Zone

Both let and const are hoisted, but not initialized during the memory creation phase.

They exist in a Temporal Dead Zone (TDZ) — the time between hoisting and actual declaration.

Accessing them before initialization causes a **ReferenceError**.

Example:

```
console.log(a); // ❌ ReferenceError
```

```
let a = 10;
```

From the start of the block to the let a = 10; line, a is in TDZ.

Constant Reference, Not Constant Value

const means constant reference, not constant value!

Primitive values (number, string, boolean, etc.) are immutable by value.

const holds the value directly — so yes, it's fixed and cannot point elsewhere.

For Object, You can change the internal data, But you cannot reassign the object variable to a new object.

So const means: “You cannot change the reference, but you can mutate the content.”

Different types of Errors

Error Type	When it Occurs	Example
ReferenceError	Accessing a let/const in TDZ	console.log(a); let a = 1;
SyntaxError	Declaring same name with var + let	var x; let x;
SyntaxError	Declaring const without initializing	const y;
TypeError	Reassigning a const variable	const z = 1; z = 2;


Block Scope

We use blocks ({}) in places where JavaScript expects a single statement, like inside if, for, or while conditions.

By using a block, we can group multiple lines of code and treat them as one unit.

A block {} does not create a new execution context, only functions do that. But it does create a new scope, called a Lexical Environment.

When JS sees a new block, it creates a fresh lexical environment, which stores all let, const, and class declarations inside it. This environment is linked to the outer environment using the scope chain, so JS can still access variables from outer scopes if not found locally.

 **var is not block-scoped, it's function-scoped.** So if you declare a var inside a block, it still leaks out. But let, const, and class are safely scoped to the block.

Shadowing

Shadowing happens when a variable declared inside a block or function has the same name as one in an outer scope. The inner variable takes precedence, it “shadows” or hides the outer one within that scope only.

So, if JavaScript finds a variable name, it always picks the closest (local) version - and ignores outer ones during that time

Basic Example with let:

```
let message = "Hello from outer scope";

function showMessage() {
  let message = "Hello from inner scope";
```

```
    console.log(message); // "Hello from inner scope"
  }

  showMessage();
  console.log(message); // "Hello from outer scope"
```

Here, the message inside the function shadows the one declared outside. Both exist, but the inner one is used within the function.

Same with var:

```
var age = 30;

function test() {
  var age = 25; // Shadows the outer 'age'
  console.log(age); // 25
}

test();
console.log(age); // 30
```

Even though var is function-scoped (not block-scoped), the concept of shadowing still applies inside functions.

Block Shadowing (Let vs Var Difference)

```
let city = "Chennai";

{
  let city = "Bangalore"; // Shadows the outer `city` inside this block
}
```

```
console.log(city); // Bangalore
}

console.log(city); // Chennai
```

But if you do the same with `var`, it does **not** create a new block scope:

```
var city = "Chennai";

{
  var city = "Bangalore"; // Overrides the outer `city`, no block scope
  console.log(city); // Bangalore
}

console.log(city); // Bangalore (Changed!)
```

Closures

Closure is when a function remembers and continues to access variables from its parent scope even after the outer function has finished executing.

 Function + Its Lexical Environment (not just value)

Function carries the reference to its outer variables via the internal property `[[Environment]]`

```
function outer() {
  let counter = 0;

  return function inner() {
```



```
    counter++;  
    console.log(counter);  
  };  
}  
  
const fn = outer(); // returns inner  
fn(); // 1  
fn(); // 2
```

Here, the counter is remembered by the inner function.

Even after `outer()` is done, the counter variable lives because `inner()` (the closure) still uses it.

Lexical Environment in Closure

Each function has a Lexical Environment = local memory + reference to parent's Lexical Environment

Closure = function + reference to its outer Lexical Environment

This is how functions remember variables from outside scopes

Where is closure stored?

Closure variables live in Heap Memory (not Stack).

JS keeps closure variables in heap until no one refers to them.

Once the reference is gone → Garbage Collector will clean up.

What's inside Closure?

FunctionObject = {

code: "...",

[[Environment]]: reference to outer scope

}

the function carries its scope chain with it.

Currying (Closures in action)

```
function multiply(x) {  
  return function (y) {  
    return x * y;  
  };  
}  
  
const double = multiply(2);  
console.log(double(5)); // 10
```

Example

```
let globalVar = 'I am global';  
let not_included_in_closure = '✗ Not used in inner()'; // Not include in  
closure  
  
function outer() {  
  let outerVar = 'I am outer';  
  
  return function inner() {  
    console.log(globalVar); // Used → Included in closure  
    console.log(outerVar); // Used → Included in closure  
  };  
}
```

```
}  
  
const fn = outer();  
fn();
```

Output:

```
I am global  
I am outer
```

Update a closure-included variable (globalVar):

```
globalVar = 'I am global2';  
fn();
```

Output:

```
I am global2  
I am outer
```

- > Closure captures only what is used inside inner().
- > not_included_in_closure is not used, so it's not included in the closure.
- > Changing globalVar affects the closure, since it's just a reference.

Call Vs apply vs bind

All three methods are used to set the **this** value of a function explicitly.

call()

1. Call the function immediately.
2. First argument: the object to use as this
3. Remaining arguments: passed individually (comma-separated).

```
function greet(greeting, punctuation) {  
  console.log(`${greeting}, ${this.name}${punctuation}`);  
}  
const user = { name: "Alice" };  
greet.call(user, "Hello", "!"); // Hello, Alice!
```

apply()

1. Calls the function immediately.
2. First argument: the object to use as this.
3. Second argument: an array of arguments.

```
greet.apply(user, ["Hi", "!"]); // Hi, Alice!!
```

bind()

1. Does not call the function immediately.
2. Returns a new function with this set permanently.
3. You can call it later.

```
const greetAlice = greet.bind(user, "Hey", "!!!");  
greetAlice(); // Hey, Alice!!!
```

this

this refers to the object that is executing the current function, determined dynamically based on how the function is called - except in arrow functions, where it lexically inherits this from the surrounding scope

1. this in Global Space

```
console.log(this); // In browser: window, In Node: {}
```

2. this Inside a Function

```
function show() {  
  console.log(this);  
}  
show(); // In browser: window (non-strict mode)
```

3. this in Strict Mode

```
"use strict";  
function strictShow() {  
  console.log(this);  
}  
strictShow(); // undefined
```

4. this Depends on How the Function Is Called

```
function print() {  
  console.log(this);  
}  
print(); // window (in browser)  
  
const obj = { print };  
obj.print(); // obj
```

5. this Inside an Object's Method

```
const person = {  
  name: 'Alice',  
  greet() {  
    console.log(this.name);  
  }  
};  
person.greet(); // Alice
```

6. call, apply, bind – Method Borrowing

```
const user = { name: "John" };  
function sayHi() {  
  console.log(this.name);  
}  
sayHi.call(user); // John  
sayHi.apply(user); // John
```

```
const bound = sayHi.bind(user);  
bound();      // John
```

7. this Inside Arrow Function

```
const obj = {  
  name: 'CaratLane',  
  greet: () => {  
    console.log(this.name);  
  }  
};  
obj.greet(); // undefined (arrow gets `this` from global)
```

8. this Inside Nested Arrow Function

```
const obj = {  
  name: 'CaratLane',  
  greet() {  
    const inner = () => {  
      console.log(this.name);  
    };  
    inner(); // CaratLane (gets from outer 'greet' context)  
  }  
};  
obj.greet();
```

9. this in the DOM (Event Handlers)

```
<button onclick="console.log(this)">Click me</button>
```

Here, this refers to the DOM element (<button>).

```
document.querySelector("button").addEventListener("click", function () {  
  console.log(this); // button  
});
```

// But with arrow function

```
document.querySelector("button").addEventListener("click", () => {  
  console.log(this); // window (arrow doesn't bind this)  
});
```

Autoboxing

Autoboxing is the process where JavaScript automatically wraps a primitive value (like a string, number, or boolean) in its corresponding object wrapper (String, Number, Boolean) so you can use object methods on them.

```
const greeting = "hello";  
console.log(greeting.toUpperCase()); // "HELLO"
```

// JavaScript behind the scenes does this:

```
const greeting = new String("hello"); // temporarily  
console.log(greeting.toUpperCase());
```

Prototype

In JavaScript, every object has a hidden internal property called `[[Prototype]]`. This property is essentially a link to another object, from which it can inherit properties and methods. This is how JavaScript achieves inheritance, through a mechanism known as prototypal inheritance.

```
const person = {  
  greet() {  
    console.log("Hello!");  
  }  
};  
  
const user = {  
  name: "Alice"  
};  
  
user.__proto__ = person;  
  
user.greet(); // Output: Hello! (inherited from person)
```

Here, the user does not have a greet method. When we call `user.greet()`, JavaScript looks up the prototype chain and finds `greet` in `person`.

Extending Built-In Prototypes:

You can also add methods to built-in prototypes like `Array.prototype`, `String.prototype`, etc

```
Array.prototype.last = function () {  
  return this[this.length - 1];  
};  
  
const nums = [10, 20, 30];  
console.log(nums.last()); // 30
```

Here, we've added a custom `last()` method to all arrays. Now every array has access to this method through its prototype.

Prototypal Inheritance

It's the mechanism where an object inherits properties/methods from another object via the prototype chain.

```
console.log(user.name); // Alice    (own property)  
console.log(user.greet()); // Hello! (inherited from person)
```

If the property isn't found in the user, JavaScript looks up the chain via `__proto__`.

Prototype Chain:

```
let obj = {};  
// obj → Object.prototype → null
```

```
let arr = [];  
// arr → Array.prototype → Object.prototype → null
```

```
let fn = function () {};
```

```
// fn → Function.prototype → Object.prototype → null
```

prototype and __proto__:

> prototype:


1. A property of constructor functions (like Array, Function, Object)
2. Defines methods to share across instances
3. Not every new instance gets a new copy of the method.
4. All instances share the same method through the prototype.

```
function User(name) {  
  this.name = name;  
}  
  
// Method defined on the prototype  
User.prototype.sayHello = function () {  
  console.log("Hi, I'm " + this.name);  
};  
  
const u1 = new User("Alice");  
const u2 = new User("Bob");  
  
console.log(u1.sayHello === u2.sayHello); // true (same method)
```

> sayHello exists once on User.prototype

> u1 and u2 don't have their own copies

> When you call u1.sayHello(), JavaScript checks:

1. Does u1 have sayHello? No
2. Then it looks up the prototype chain → User.prototype  and uses that method

> __proto__

1. If you want to manually link two objects (for inheritance or fallback).

```
const base = {  
  greet() {  
    console.log("Hi!");  
  }  
};  
  
const child = {  
  name: "John"  
};  
  
// Set prototype manually  
child.__proto__ = base;  
  
child.greet(); // Hi!
```

child doesn't have greet, so JS checks __proto__ → finds it in base.

Use __proto__ when manually creating a prototype chain between plain objects.

```
function Animal() {}  
const dog = new Animal();
```

Internally:

1. `dog.__proto__ === Animal.prototype`
2. `Animal.prototype.__proto__ === Object.prototype`

In Short,

```
function Animal() {}  
const dog = new Animal();
```

- > `Animal` is a function
- > `dog` is an object created from `Animal`

`Animal.prototype` is the object that will be shared by all instances (like `dog`) created using `new Animal()`

`Animal.__proto__` is about the function `Animal` itself.

`Animal.__proto__ === Function.prototype`

- > All functions inherit from `Function.prototype`
 - > That's why `Animal.__proto__` points to `Function.prototype`.
 - > When you do `'new Animal()'`, the new object's `__proto__` will point to `Animal.prototype`. [`dog.__proto__ === Animal.prototype`].
-

First Order Function

In JavaScript, functions are first-class citizens.

This means:

- > You can store them in variables
- > You can pass them as arguments to other functions

- > You can return them from functions
- > You can assign properties to them

1. Assigned to a variable

```
const greet = function(name) {  
  return `Hello, ${name}`;  
};
```

2. Passed as argument

```
function callFn(fn, value) {  
  return fn(value);  
}  
console.log(callFn(greet, "Alice")); // Hello, Alice
```

3. Returned from another function

```
function outer() {  
  return function () {  
    console.log("Inner function!");  
  };  
}  
const innerFn = outer();  
innerFn(); // Inner function!
```

Types of Functions in JavaScript

Function Type	Syntax	Hoisted	Has this	Use Case / Notes
Function Declaration	<code>function foo() {}</code>	Yes	Yes	Named and hoisted. Good for general-purpose reusable functions.
Function Expression	<code>const foo = function() {}</code>	No	Yes	Assigned to variables. Not hoisted. Useful when functions are defined conditionally.
Arrow Function	<code>const foo = () => {}</code>	No	No (lexical)	Shorter syntax, inherits this from parent. Cannot be used as a constructor.
Named Function Expression	<code>const foo = function bar() {}</code>	No	Yes	Function has an internal name (bar) used for recursion/debugging.

Immediately Invoked Function (IIFE)	<code>(function() {})() or (() => {})()</code>	No		Runs immediately. Useful for private scopes or polyfills (Add features safely without polluting or breaking existing code).
Constructor Function	<code>function Person(name) { this.name = name; }</code>	Yes	Yes	Called with new. Used to instantiate objects.
Generator Function	<code>function* gen() { yield 1; }</code>	Yes	Yes	Can pause execution using yield. Returns an iterator.
Async Function	<code>async function foo() {}</code>	Yes	Yes	Always returns a Promise. Can use await inside.
Async Arrow Function	<code>const foo = async () => {}</code>	No	No	Async + arrow → inherits this, can't use arguments, always returns Promise.

Method Definition (in Object)	<code>const obj = { greet() {} }</code>	Yes	Yes	this refers to the object.
Class Method	<code>class A { method() {} }</code>	Yes	Yes	Defined inside a class body.
Static Method	<code>class A { static method() {} }</code>	Yes	Yes	Called on the class itself, not instances.

CallBack functions

A callback function is a function that is passed as an argument to another function, and is invoked later (either immediately or after an event/asynchronously).

Passed through arguments, can be called anywhere and its hold lexical environment of caller method (Closure). When it needs to execute, JS pushes this function to call stack and execute it.

A callback function is a type of first-class function in JavaScript.

Example:

```
console.log("1: Start");

setTimeout(() => {
  console.log("2: In Timer");
}, 2000);

console.log("3: End");
```

Output:

```
1: Start
3: End

// After 2 seconds
2: In Timer
```

What Happens Internally?

1. Execution Starts (Call Stack)

JavaScript starts running line by line.

- `console.log("1: Start")` is executed → printed.
- Next, it sees `setTimeout(...)`.

2. Callback Passed to Web API

> The `setTimeout` function is not part of JS itself — it's provided by the browser Web APIs.

> JavaScript registers the callback function `() => { console.log(...) }` along with the timer (2 sec) with the **Web API environment**.

> JavaScript doesn't block or wait. It offloads the task and continues executing.

3. Callback Stored Outside Call Stack

> The callback function is stored in the Web API environment (outside JS engine) along with a timer.

> JavaScript's execution context moves on — it doesn't wait.

> `console.log("3: End")` is printed next

4. Timer Ends → Task Queued

> After 2 seconds, the browser/Web API signals the JS engine that the timer is done.

> It moves the callback function to the Task Queue (Callback Queue).

> The Event Loop continuously checks. When it is empty, it pushes the callback from the task queue onto the call stack.

> Now, `console.log("2: In Timer")` runs.

Event Loop

 JS is a synchronous single threaded language.

Runtime Components

When JS runs inside a browser (or Node.js), it gets access to several runtime components that together handle async tasks:

1. Call Stack

- > Where JS executes functions.
- > Follows Last-In-First-Out (LIFO).
- > Runs synchronous code

2. Web APIs (Provided by Browser/Node)

- > Timer APIs (setTimeout, setInterval)
- > DOM Events (click, scroll)
- > Network APIs (fetch, XHR)
- > Storage, Geolocation, etc

3. Callback Queue (Task Queue / Macrotask Queue)

Queues callbacks from:

- > setTimeout, setInterval
- > DOM events, etc.

4. Microtask Queue

Higher priority than Callback Queue.

Includes:

- Promise.then(), catch, finally
- async/await
- queueMicrotask()
- MutationObserver
- Runs after the call stack is empty, but before the callback

queue.

5. Event Loop

- > Watch the Call Stack and Queues.
- > If Call Stack is empty, it:

- Flushes all Microtasks
- Then runs 1 item from Callback Queue

How It All Works (Event Loop Flow)

1. JS code starts running → functions go into Call Stack.
2. When async function (like `setTimeout`, `fetch`) is called:
 - > JS hands it over to Web APIs.
 - > Web API handles it outside the Call Stack.
3. After the async task completes:
 - > Web API puts the callback into:
 - > Microtask Queue (e.g., `Promise`)
 - > Callback Queue (e.g., `setTimeout`)
4. The Event Loop checks:
 - > Is Call Stack empty?
 - > Yes → Runs all Microtasks
 - > Then → Takes next item from Callback Queue

Where Things Are Stored?

Phase	Stored In	Notes
While waiting	Web APIs	e.g., timers running, network requests pending
Ready for execution	Callback/Microtask Queue	Depends on type of async task

Executing code	Call Stack	When JS is actively running that function
----------------	------------	---

Example

```
console.log("Start");

fetch("https://api.com/data")
  .then((res) => res.json())
  .then((data) => console.log(data));

console.log("End");
```

Internally:

- > "Start" → Logged
- > fetch() → Sent to Web API → Browser handles request
- > "End" → Logged
- > Response comes back:
 - > .then() handlers → Pushed into Microtask Queue
- > Event Loop:
 - > Finishes current script
 - > Runs Microtasks → Logs data

 fetch response uses Microtask Queue.

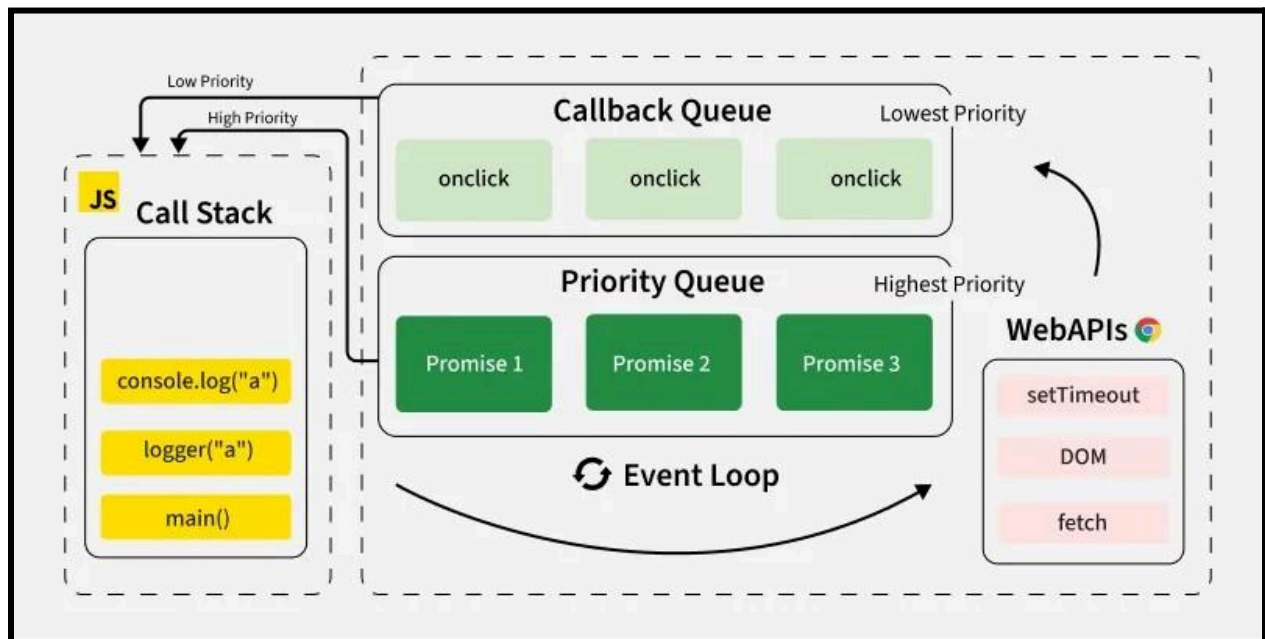
What is a Web API?

Web APIs are browser-provided or Node.js-provided environments for handling tasks that JavaScript itself cannot do alone, like:

- > Timers (setTimeout, setInterval)
- > DOM Events (click, input)
- > Network requests (fetch, XMLHttpRequest)
- > Geolocation, Storage, etc.

How JavaScript Handles Web APIs Internally

- > JS Code calls Web API
- > Web API runs task outside JS engine
- > Once complete, it pushes callback → callback queue / microtask queue
- > Event Loop monitors & pushes the callback to the Call Stack
- > JS executes the callback



Web APIs

1. setTimeout / setInterval

Category: Callback

Queue: Callback Queue

```
setTimeout(() => console.log("Timer done"), 1000);
```

Flow:

setTimeout() sent to Web API timer thread.

Waits for 1000ms → Callback ready.

Callback placed in Callback Queue.

When Call Stack is empty & microtasks done → callback goes to Call Stack.

2. fetch

Category: Combo (Web API + Promise)

Queue: Microtask Queue

```
fetch("api/data")  
  .then((res) => res.json())  
  .then((data) => console.log(data));
```

Flow:

JS engine sends fetch() to Web API (browser or Node's lib).

Browser handles requests (non-blocking).

Response arrives → .then() registered as microtask.

Microtask runs after the current call stack is empty.

3. DOM Events (click, input, etc.)

Category: Callback

Queue: Callback Queue (event listeners)

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



```
console.log("Clicked!");  
});
```

Flow:

Browser registers callback in the Event Table.

When event (click) occurs:

- > Browser pushes callback to Callback Queue
- > Event Loop sends it to Call Stack when ready

4. Promises (then, catch, finally, async/await)

Category: Microtask

Queue: Microtask Queue

```
Promise.resolve().then(() => console.log("Promise resolved"));
```

Flow:

Promise callback sent to Microtask Queue.

Event Loop prioritizes and pushes to Call Stack before macrotasks.

JS Engine

Just in Time Compilation

JavaScript is not a compiled language like C/C++, and not purely interpreted either. Instead, modern JavaScript engines like V8 (used in Chrome and Node.js) use JIT compilation to achieve better performance.

How it works:

1. **Parsing:** The JS source code is parsed and converted into an Abstract Syntax Tree (AST).
2. **Bytecode Generation:** Instead of running directly from source code, it's converted into bytecode (intermediate form).
3. **Baseline Interpreter:** Initially, a lightweight interpreter (e.g., Ignition in V8) runs this bytecode quickly.
4. **Profiler Watches Execution:** If some code is run frequently (e.g. a loop), it's marked as "hot".
5. **Optimization via JIT Compiler:** The JIT compiler (e.g., TurboFan in V8) compiles the hot code to highly optimized machine code. This compiled machine code runs faster than interpreted bytecode.
6. **De-optimization:** If assumptions made during optimization are wrong (e.g. changing types), the engine de-optimizes and falls back to a slower version.

JIT Example

```
function multiply(a, b) {  
  return a * b;  
}  
  
function computeSum(array) {  
  let sum = 0;  
  for (let i = 0; i < array.length; i++) {  
    sum += multiply(array[i], 2);  
  }  
}
```

```

    }
    return sum;
}

function run() {
    const nums = [];

    // Fill array with numbers
    for (let i = 0; i < 1_000_000; i++) {
        nums.push(i);
    }

    console.time('optimized');
    console.log(computeSum(nums)); // JIT optimizes multiply and
computeSum
    console.timeEnd('optimized');

    // Introduce a de-optimization
    nums[500_000] = 'oops'; // Now there's a string in the array

    console.time('deoptimized');
    console.log(computeSum(nums)); // Causes de-optimization!
    console.timeEnd('deoptimized');
}

run();

```

Breakdown of What Happens:

1. Initial Run:

> Hot & Optimized: computeSum() and multiply() are called 1 million times with numeric input.

> JIT:

1. Inlines multiply(a, b) inside computeSum()
2. Optimizes for Number * Number
3. Loop is compiled into native machine code

2. De-optimization Triggered:

> We mutate one element of the array: nums[500_000] = 'oops'.

> Now JS engine:

1. De-optimizes the loop and multiply()
2. Falls back to the interpreter
3. Introduces type checks in every iteration

You can run the script using this

> node --trace-opt --trace-deopt complex.js

You will get output like this:

```

> node --trace-opt --trace-deopt jit.js
[marking 0x258b990a3c21 <JSFunction run (sfi = 0x193012d94d31)> for optimization to TURBOFAN, ConcurrencyMode:kConcurrent, reason: hot and stable]
[compiling method 0x258b990a3c21 <JSFunction run (sfi = 0x193012d94d31)> (target TURBOFAN) OSR, mode: ConcurrencyMode:kConcurrent]
[completed compiling 0x193012d96c71 <JSFunction run (sfi = 0x193012d94d31)> (target TURBOFAN) OSR - took 0.708, 12.750, 0.000 ms]
[completed optimizing 0x193012d96c71 <JSFunction run (sfi = 0x193012d94d31)> (target TURBOFAN) OSR]
[bailout (kind: deopt-eager, reason: Insufficient type feedback for generic named access): begin. deoptimizing 0x193012d96c71 <JSFunction run (sfi = 0x193012d94d31)>, 0x0c5fe0f82311 <Code TURBOFAN>, opt id 0, bytecode offset 39, deopt exit 5, FP to SP delta 128, caller SP 0x00016f1cd268, pc 0x00010778bf40]
[marking 0x320a05905081 <JSFunction multiply (sfi = 0x193012d94c81)> for optimization to TURBOFAN, ConcurrencyMode:kConcurrent, reason: hot and stable]
[compiling method 0x320a05905081 <JSFunction multiply (sfi = 0x193012d94c81)> (target TURBOFAN), mode: ConcurrencyMode:kConcurrent]
[marking 0x320a059050c1 <JSFunction computeSum (sfi = 0x193012d94cd9)> for optimization to TURBOFAN, ConcurrencyMode:kConcurrent, reason: hot and stable]
[compiling method 0x320a059050c1 <JSFunction computeSum (sfi = 0x193012d94cd9)> (target TURBOFAN) OSR, mode: ConcurrencyMode:kConcurrent]
[completed compiling 0x320a05905081 <JSFunction multiply (sfi = 0x193012d94c81)> (target TURBOFAN) - took 0.000, 0.584, 0.000 ms]
[completed optimizing 0x320a05905081 <JSFunction multiply (sfi = 0x193012d94c81)> (target TURBOFAN)]
[completed compiling 0x320a059050c1 <JSFunction computeSum (sfi = 0x193012d94cd9)> (target TURBOFAN) OSR - took 0.000, 0.750, 0.000 ms]
[completed optimizing 0x320a059050c1 <JSFunction computeSum (sfi = 0x193012d94cd9)> (target TURBOFAN) OSR]
[bailout (kind: deopt-eager, reason: overflow): begin. deoptimizing 0x320a059050c1 <JSFunction computeSum (sfi = 0x193012d94cd9)>, 0x0c5fe0f82929 <Code TURBOFAN>, opt id 2, bytecode offset 30, deopt exit 11, FP to SP delta 144, caller SP 0x00016f1cd1f0, pc 0x00010778c668]
[marking 0x320a059050c1 <JSFunction computeSum (sfi = 0x193012d94cd9)> for optimization to TURBOFAN, ConcurrencyMode:kConcurrent, reason: hot and stable]
[compiling method 0x320a059050c1 <JSFunction computeSum (sfi = 0x193012d94cd9)> (target TURBOFAN) OSR, mode: ConcurrencyMode:kConcurrent]
[completed compiling 0x320a059050c1 <JSFunction computeSum (sfi = 0x193012d94cd9)> (target TURBOFAN) OSR - took 0.000, 0.625, 0.000 ms]
[completed optimizing 0x320a059050c1 <JSFunction computeSum (sfi = 0x193012d94cd9)> (target TURBOFAN) OSR]
999999000000
optimized: 10.25ms
[bailout (kind: deopt-eager, reason: wrong map): begin. deoptimizing 0x320a059050c1 <JSFunction computeSum (sfi = 0x193012d94cd9)>, 0x0c5fe0f82ba9 <Code TURBOFAN>, opt id 3, bytecode offset 4, deopt exit 1, FP to SP delta 144, caller SP 0x00016f1cd1f0, pc 0x00010778c9c0]
[marking 0x320a059050c1 <JSFunction computeSum (sfi = 0x193012d94cd9)> for optimization to TURBOFAN, ConcurrencyMode:kConcurrent, reason: hot and stable]
[compiling method 0x320a059050c1 <JSFunction computeSum (sfi = 0x193012d94cd9)> (target TURBOFAN) OSR, mode: ConcurrencyMode:kConcurrent]
[completed compiling 0x320a059050c1 <JSFunction computeSum (sfi = 0x193012d94cd9)> (target TURBOFAN) OSR - took 0.000, 0.667, 0.042 ms]
[completed optimizing 0x320a059050c1 <JSFunction computeSum (sfi = 0x193012d94cd9)> (target TURBOFAN) OSR]
[bailout (kind: deopt-eager, reason: not a Smi): begin. deoptimizing 0x320a059050c1 <JSFunction computeSum (sfi = 0x193012d94cd9)>, 0x0c5fe0f92be1 <Code TURBOFAN>, opt id 4, bytecode offset 2, deopt exit 10, FP to SP delta 144, caller SP 0x00016f1cd1f0, pc 0x00010778e534]
[marking 0x320a059050c1 <JSFunction computeSum (sfi = 0x193012d94cd9)> for optimization to TURBOFAN, ConcurrencyMode:kConcurrent, reason: hot and stable]
[compiling method 0x320a059050c1 <JSFunction computeSum (sfi = 0x193012d94cd9)> (target TURBOFAN) OSR, mode: ConcurrencyMode:kConcurrent]
[completed compiling 0x320a059050c1 <JSFunction computeSum (sfi = 0x193012d94cd9)> (target TURBOFAN) OSR - took 0.000, 0.625, 0.000 ms]
[completed optimizing 0x320a059050c1 <JSFunction computeSum (sfi = 0x193012d94cd9)> (target TURBOFAN) OSR]
NaN
deoptimized: 4.64ms

```

What's Happening Step-by-Step

1. JS engine Marks run, multiply, and computeSum Hot

[marking ... <JSFunction run> for optimization to TURBOFAN, reason: hot and stable]

> JS functions are initially run in interpreted mode.

> If a function is called multiple times with stable usage patterns, it is marked “hot” and sent to the optimizing compiler (Turbofan)

2. JIT Optimization Happens (Turbofan)

[marking ... <JSFunction run> for optimization to TURBOFAN, reason: hot and stable]

> Turbofan generates optimized machine code based on assumptions (e.g. inputs are always integers).

3. Deoptimization Begins (Assumptions Break!)

[bailout (kind: deopt-eager, reason: wrong map)]

[bailout (kind: deopt-eager, reason: not a Smi)]

> These bailouts mean: “Assumption was wrong. Fall back to the interpreter.”

> wrong map: object shape changed.

> not a Smi: expected small integer (Smi), but got something else (e.g., string or object).

JS Engine saw multiply and computeSum only using numbers, so it compiled optimized machine code, when a string or a large number or a non-integer sneaked in → deopt.

Why JavaScript Doesn't Optimize Immediately

JavaScript is dynamically typed and interpreted.

That means:

- > At the start, JS doesn't know what types your variables will be.
- > multiply(a, b) could receive:
 - Numbers (a = 3, b = 5)
 - Strings (a = 'hi', b = {})
 - Objects, Arrays, undefined, etc.

So initially, JavaScript engines take the safe route: Interpret line-by-line without assuming types.

Once the engine notices:

- > A function is called many times (hot code)
- > And it's always called with same types (e.g., only numbers)

Then the JIT Compiler steps in.

JIT Does

- > Assumes types are fixed (e.g., a and b are numbers)
- > Removes function call overhead
- > Skips type checks and coercions
- > Translates JS to fast native CPU instructions

Example:

```
function multiply(a, b) {  
  return a * b;  
}
```

```
}
```

Initially, the engine does `typeof a`, `typeof b`, converts if needed.

After JIT, Machine code: just do `MUL a, b`.

In Short,

1. Baseline execution (Ignition) – V8 runs code via the interpreter first.
 2. Profiling starts – After a few executions, it starts collecting type feedback.
 3. Optimization is queued – If the code becomes hot (above the threshold), V8 sends it to TurboFan for optimizing compilation.
 4. Optimized code runs – Now your function runs faster.
 5. Deoptimization may occur – If assumptions break (e.g., data type changes), V8 falls back to unoptimized code.
-

Memory Heap

The Memory Heap is not the same as the heap data structure. It's just a general area of memory used for dynamic memory allocation, where objects, closures, and functions are stored.

1. All objects and functions are stored in the Memory Heap.
 2. It is an unstructured region of memory managed by the JavaScript engine.
 3. Managed via garbage collection, unused memory is freed automatically.
-

Higher-Order Function

A function that either takes one or more functions as arguments, returns a function, or both.

Example:

```
function greet(name) {  
  return `Hello, ${name}`;  
}  
  
function processUserInput(callback) {  
  const name = "Alice";  
  console.log(callback(name));  
}  
  
processUserInput(greet); // Hello, Alice
```

processUserInput is a higher-order function because it takes greet (a function) as an argument.

First-Class Functions → Ability: What JavaScript allows functions to be used as.

Higher-Order Functions → Behavior / Usage Pattern: How we use functions because of that ability.



Because functions are first-class, we can write higher-order functions

Callback Hell



Each function depends on the result of the previous one, and indentation keeps growing

```
doTask1(function(result1) {  
  doTask2(result1, function(result2) {  
    doTask3(result2, function(result3) {  
      doTask4(result3, function(finalResult) {  
        console.log("All done!");  
      });  
    });  
  });  
});
```

It is a result of deeply nested async functions using callbacks.

This structure not only makes the code difficult to maintain and debug, but also causes inversion of control, where the flow of execution is handed off to external APIs or services via callbacks. Because the logic depends on external responses.

Promises

A Promise is a **JavaScript object** that represents the **eventual completion (or failure)** of an asynchronous operation and its resulting value.
(Container for a future value).

Lifecycle of a Promise

Pending – Initial state, neither fulfilled nor rejected.

Fulfilled – The operation completed successfully (resolve() was called).

Rejected – The operation failed (reject() was called).

Object structure:

```
{  
  [[PromiseState]]: "pending" | "fulfilled" | "rejected",  
  [[PromiseResult]]: undefined | value | error,  
  then: function(onFulfilled, onRejected),  
  catch: function(onRejected),  
  finally: function(onFinally)  
}
```



Once a Promise is fulfilled or rejected, it becomes settled and cannot transition again.

Example:

```
console.log("Start");  
  
fetch("https://jsonplaceholder.typicode.com/posts/1")  
  .then(response => response.json())  
  .then(data => {  
    console.log("Post title:", data.title);  
  })  
  .catch(error => {  
    console.error("Fetch error:", error);  
  });  
  
console.log("End");
```

- > fetch() is a Web API (browser-provided function, not part of JS engine).
- > It initiates an HTTP request outside the main thread.
- > Immediately returns a Promise in pending state.

The fetch() finishes in the background via Web API.

It resolves the Promise, pushing the .then() callback to the Microtask Queue.

Once the callstack is empty, event Loop sees the Microtask Queue has entries, and pushes .then(response => response.json()) onto the Call Stack.

```
Call Stack
  |
  v
[ console.log("Start") ]
  |
  v
[ fetch() -----> Web API (async) ]
  |
  v
[ console.log("End") ]
  |
  v
Stack Empty --> Event Loop checks Microtask Queue
  |
  v
Microtask Queue:
  -> .then(response => response.json())
  -> .then(data => console.log(...))
```

Creating Promise

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

```
// async operation here
if (/* success */) {
  resolve(result); // fulfilled
} else {
  reject(error); // rejected
}
});

myPromise.then(// do something).catch(err) {// handle errors}
```

resolve() - Call this when the operation succeeds.

reject() - Call this when the operation fails.

then() - Used to handle success.

catch() - Used to handle failure.

Understanding .then()

.then() is a method used to register callbacks to be executed after a Promise is fulfilled (resolved).

```
const p = Promise.resolve("Hello");

p.then(result => {
  console.log(result); // "Hello"
});
```

> .then() runs when the promise is fulfilled.

> It returns a new Promise, allowing chaining.

What if you add multiple independent .then() calls?

```
const p = Promise.resolve("Done");  
  
p.then(val => console.log("First:", val));  
p.then(val => console.log("Second:", val));
```

Output:

First: Done

Second: Done

> Because you're attaching both `.then()`s to the same original resolved promise.

> Once a promise is resolved, all registered `.then()` handlers (subscribers) are called.

> This is like an event emitter, all listeners are notified.

Understanding `.catch()`

`.catch()` is used to handle rejected Promises (errors)

```
Promise.reject("Error!")  
  .catch(err => {  
    console.log("Caught:", err); // Caught: Error!  
  });
```

Two Catch Scenarios:

Chained `.catch()`:

```
Promise.reject("Oops")
```

```
.catch(err => {  
  console.log("First Catch");  
  // no rethrow  
})  
.catch(err => {  
  console.log("Second Catch"); // ❌ won't run  
});
```

Only the first `.catch()` runs, because the error was already handled and not rethrown.

Independent `.catch()`s:

```
const promise = Promise.reject("Boom");  
  
promise.catch(err => console.log("Catch A"));  
promise.catch(err => console.log("Catch B"));
```

Both Catch A and Catch B will run.

Understanding `.finally()`

`.finally()` is a method used to execute code regardless of whether the promise was fulfilled or rejected.

It's mainly used for cleanup logic.

```
Promise.resolve("done")  
  .finally(() => console.log("Cleanup"));
```

`.finally()` never passes data to the next `.then()` or `.catch()`.

Order matters in a Promise chain

`then` → `catch` → `finally`

```
Promise.reject("Error")
  .then(() => {
    console.log("then");
  })
  .catch((err) => {
    console.log("catch:", err);
  })
  .finally(() => {
    console.log("finally");
  });
```

Output:

catch: Error

finally

`catch` → `then` → `finally`

```
Promise.reject("Error")
  .catch((err) => {
    console.log("catch:", err);
  })
  .then(() => {
    console.log("then");
  })
  .finally(() => {
```

```
console.log("finally");  
});
```

Output:

catch: Error

then

finally

> .then() executes after the catch, because the .catch() handled the rejection and passed a resolved state to the next .then().

Promise APIs

1. Promise.all()

Promise.all() takes an iterable (usually an array) of Promises and returns a single Promise that:

> Resolves when all Promises resolve.

> Rejects immediately when any one of the Promises rejects.

```
Promise.all([promise1, promise2, promise3]);
```

1. Execution flow:

1. All Promises passed to Promise.all() start executing in parallel.
2. The returned Promise waits for every input Promise to be fulfilled.
3. If all succeed, it resolves with an array of results in the same order (Order is preserved).
4. If any one fails, it immediately rejects that error.

2. Are Promises Dispatched Simultaneously?

Promises are called one-by-one as you write them,

```
const p1 = fetch('/a');
```

```
const p2 = fetch('/b');
```

```
const p3 = fetch('/c');
```

- > Each `fetch()` returns a Promise immediately and starts its request.
- > Dispatched sequentially in JS.
- > But the actual asynchronous work is handed off to the browser's Web API layer.
- > These requests execute in parallel outside of the JS thread.

3. How Does `Promise.all()` Subscribe to Each Promise?

When you do:

```
Promise.all([p1, p2, p3]);
```

The method internally adds `.then()` and `.catch()` handlers to each promise.

```
function promiseAll(promises) {  
  return new Promise((resolve, reject) => {  
    const results = [];  
    let completed = 0;  
  
    if (promises.length === 0) {  
      return resolve([]);  
    }  
  
    promises.forEach((promise, index) => {  
      promise.then((value) => {  
        results[index] = value;  
        completed++;  
      })  
    })  
  
    if (completed === promises.length) {  
      resolve(results);  
    }  
  })  
}
```

```
    completed++;

    if (completed === promises.length) {
        resolve(results); // final resolve
    }
}).catch((error) => {
    reject(error); // fail-fast
});
});
});
}
```

It subscribes to each promise using `.then()` and `.catch()`.

If a `.then()` is attached to a pending promise, it gets stored in the microtask queue when resolved and updates the results array.

4. What if one Promise fails?

```
Promise.all([
    Promise.resolve('a'),
    Promise.reject('b'), // fails here
    Promise.resolve('c'),
])
.then(res => console.log(res))
.catch(err => console.error("Error:", err));
```

As soon as one rejects (b), the entire `Promise.all()` rejects.

Other promises still continue to run (they're not cancelled), but:

- > Their results are ignored

> Their .then() handlers (inside Promise.all) are never used

Deep dive into Promise.all

1. Promise.all iterates the array. Each promise begins executing immediately. If any promise involves external work (like fetch, setTimeout), it interacts with Web APIs.
2. When each individual promise resolves, its .then() callback is added to the microtask queue.
3. When that microtask runs, it updates the results array.
4. If all promises are resolved (completed === length), master promise resolves and its .then() is queued.
5. In failure case,
 - > If any one fails, master promise immediately rejects and .catch() is queued.
 - > But remaining pending promises still resolve later and their .then() gets added to the microtask queue.
 - > Each .then() has closure over results[] and memory is retained.
 - > GC (Garbage Collector) won't clean it until all callbacks complete.

Example:

```
const p1 = new Promise(res => setTimeout(() => res("A"), 100));
const p2 = new Promise((_, rej) => setTimeout(() => rej("FAIL"), 50));
```

```
Promise.all([p1, p2])
  .then(res => console.log("ALL:", res))
  .catch(err => console.log("ERROR:", err));

p1.then(val => {
  console.log("p1 still runs:", val); // This will run!
});
```

Output:

ERROR: FAIL

p1 still runs: A

- > Promise.all rejected due to p2
- > p1 still runs its .then()
- > So it's not cancelled, even if the master failed.
- > It schedules its .then() into the microtask queue independently.

2. Promise.allSettled()

Promise.allSettled() takes an array of Promises and returns a new Promise that always resolves with an array of outcomes, regardless of whether each individual promise was fulfilled or rejected.

```
Promise.allSettled([promise1, promise2, promise3]);
```

Each result object will have one of the following forms:

```
{ status: "fulfilled", value: ... }
```

```
{ status: "rejected", reason: ... }
```

Simplified Internal Logic:

```
function promiseAllSettled(promises) {  
  return new Promise((resolve) => {  
    const results = [];  
    let completed = 0;  
  
    promises.forEach((promise, index) => {  
      Promise.resolve(promise)  
        .then(value => {  
          results[index] = { status: 'fulfilled', value };  
        })  
        .catch(reason => {  
          results[index] = { status: 'rejected', reason };  
        })  
        .finally(() => {  
          completed++;  
          if (completed === promises.length) {  
            resolve(results);  
          }  
        });  
    });  
  });  
}
```

We usually check the result's status before accessing its value:

```
function handleAllTasks(promises) {  
  return Promise.allSettled(promises).then(results => {
```

```
const successes = [];  
const failures = [];  
  
results.forEach((result, index) => {  
  if (result.status === "fulfilled") {  
    successes.push({  
      index,  
      value: result.value  
    });  
  } else {  
    failures.push({  
      index,  
      reason: result.reason  
    });  
  }  
});  
  
return {  
  allResults: results,  
  successes,  
  failures  
};  
});  
}
```

3. Promise.race()

Promise.race(iterable) returns a new Promise that:

1. Settles (resolves or rejects) as soon as the first promise in the iterable settles.

2. It doesn't wait for all promises, only the fastest one to finish decides the result

```
const p1 = new Promise((resolve) => setTimeout(() => resolve("P1 done"), 1000));
const p2 = new Promise((resolve) => setTimeout(() => resolve("P2 done"), 500));

Promise.race([p1, p2]).then(console.log);
```

Output:

P2 done

Simplified Internal Logic:

```
function promiseRace(promises) {
  return new Promise((resolve, reject) => {
    let settled = false;

    for (let p of promises) {
      Promise.resolve(p).then(
        (val) => {
          if (!settled) {
            settled = true;
            resolve(val);
          }
        },
        (err) => {
          if (!settled) {
```

```
        settled = true;
        reject(err);
      }
    }
  );
}
});
}
```

4. Promise.any()

Promise.any() takes an iterable of Promises and returns a single promise that:

- > Resolves as soon as any one of the input promises resolves.

- > Rejects only when all input promises are rejected, and throws an AggregateError.

Promise.any([p1, p2, p3])

.then(result => console.log("Resolved with:", result))

.catch(err => console.log("All failed:", err));

Simplified Internal Logic:

```
function promiseAny(promises) {
  return new Promise((resolve, reject) => {
    let rejections = [];
    let completed = 0;
```



```
if (promises.length === 0) {
  return reject(new AggregateError([], "All promises were rejected"));
}

promises.forEach((promise, index) => {
  Promise.resolve(promise)
    .then(value => {
      resolve(value); // Succeed fast
    })
    .catch(err => {
      rejections[index] = err;
      completed++;
      if (completed === promises.length) {
        reject(new AggregateError(rejections, "All promises were
rejected"));
      }
    });
});
});
}
```

- > Resolves fast as soon as the first successful promise resolves.
 - > Keeps collecting rejections in an array.
-

Async Await

Adding async before a function declaration marks it asynchronous.

It always returns a Promise.

- > If the function returns a value → it is wrapped with `Promise.resolve(value)`.
- > If it returns a Promise → it is returned as-is.

```
async function foo() {  
  return 42; // returns Promise.resolve(42)  
}  
foo().then(console.log); // Prints: 42
```

await is used inside async functions only and it pauses execution of that function until the Promise resolves/rejects.

When to Use async and await?

1. To handle Promises more cleanly: Instead of chaining `.then()` and `.catch()`, use `await` to pause for the result, making the flow more readable.
2. When you want code that “looks synchronous”: Async/await allows asynchronous operations to appear in a top-down, linear manner - easier to write, read, and debug.
3. To avoid callback hell or promise chaining and better error handling using `try/catch`.

It does not block the entire JavaScript thread - only the async function is paused



The `await` pauses only that function, not the entire thread.

```
function syncLog(msg) {
```

```
    console.log(`[SYNC] ${msg}`);
  }

  async function asyncFunction() {
    syncLog("Async function started");

    await new Promise((res) => {
      setTimeout(() => {
        syncLog("Promise resolved after 2s");
        res();
      }, 2000);
    });

    syncLog("Async function resumed");
  }

  syncLog("Script started");
  asyncFunction();
  syncLog("Script still running...");
```

Output:

```
[SYNC] Script started
[SYNC] Async function started
[SYNC] Script still running...
(wait 2 seconds...)
[SYNC] Promise resolved after 2s
[SYNC] Async function resumed
```

Here,

> Script started and Script still running... are printed immediately.

> Even though `await` pauses `asyncFunction`, the rest of the script continues. `await` suspends the execution of the current `async` function.

> The event loop doesn't stop - it's just that `asyncFunction` is "paused" and resumes after the `Promise` resolves. The main thread continues executing the next line after calling `asyncFunction()`.

> When the `setTimeout` finishes (after 2s), the `Promise` resolves, and `asyncFunction` resumes - it re-enters from the `await` point.

When does the `await` resume?

After the `Promise` resolves, the awaited function:

1. The continuation is pushed into the microtask queue.
2. JS waits until the call stack is empty.
3. Then, it restores the `async` function's context and resumes from the `await` line.

How is context preserved?

> The `async` function's execution context is not discarded.

> The context is preserved in the heap.

> This saved state is stored in the heap. (as a "closure" or `async` continuation record (The data structure the JS engine uses to store the paused execution of an `async` function)).


> When the awaited `Promise` resolves, the saved context is reloaded, and execution resumes from where it paused.

Rejection Handling with `await`:

If the awaited `Promise` rejects:

1. An error is thrown at the await line.
2. You should handle it with a try...catch.

```
async function safe() {  
  try {  
    const res = await fetch("bad-url");  
  } catch (err) {  
    console.log("Caught error:", err);  
  }  
}
```

 await also works with non-promises - it just wraps them.

await 42; // same as await Promise.resolve(42)

The End
Still Scratching the Surface ...