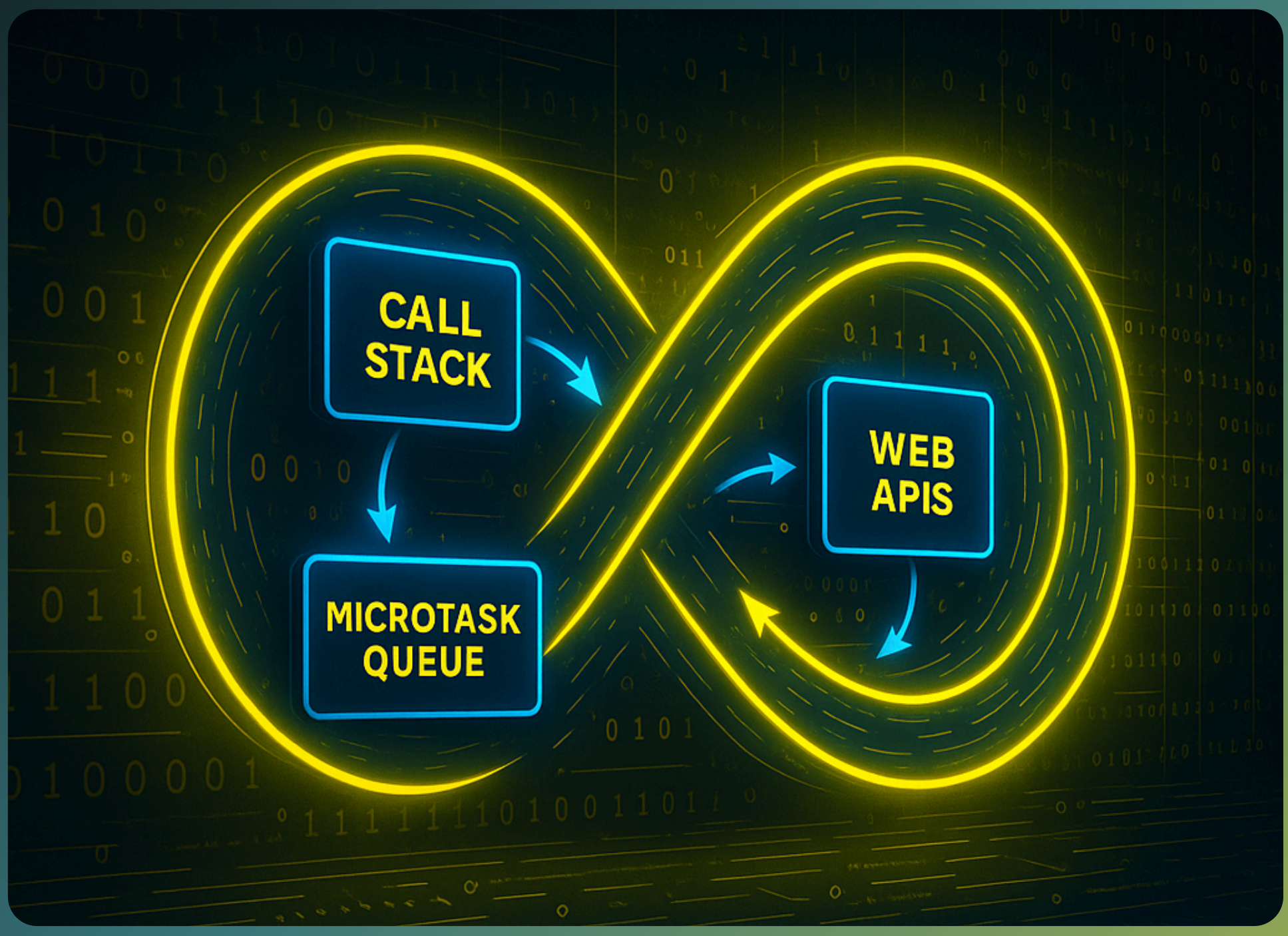
JS

JavaScript

Deep Dive into Event Loop



Master JavaScript's Async Magic

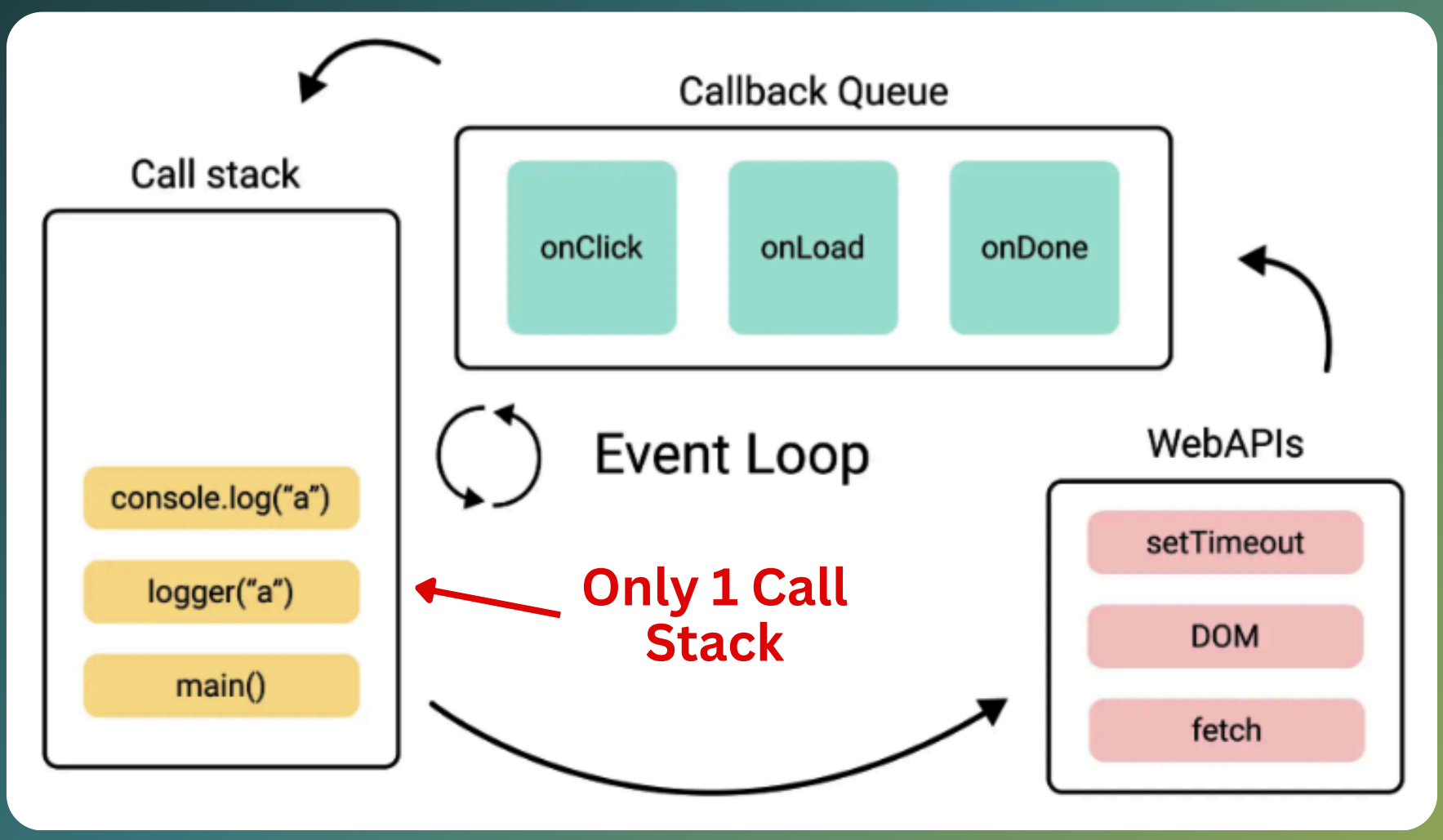
Swipe

Shivendra Dwivedi
@shivendra-dwivedi

JavaScript’s Single-Threaded Nature

JS has only one call stack and can execute one task at a time.

There’s only one "thread of execution", so JS reads, executes, and completes one piece of code before moving to the next.



Why single-threaded:

No Race Conditions: Only one operation can modify the DOM at a time.

Predictable Order: Events (clicks, network responses,

@Shivendra Dwivedi

etc.) are queued and first come first serve rule is followed in the same queue.

Problem If JavaScript Were Multi-Threaded

Half-Loaded DOM Clicks:

- Thread A is still loading/rendering the DOM.
- Thread B detects a click on a partially loaded button, executes an event handler.
- The button's functionality might rely on unloaded resources (JS/CSS), causing crashes or undefined behavior.

Note: There are many more problems like Inconsistent State, Locking Overhead, etc.

How JavaScript Executes Code

JavaScript code execution happens in two phases:

1: Memory Creation Phase (Creation Phase)

- Memory is allocated for variables and functions.
- Hoisting happens:
  - var declarations are hoisted and initialized with undefined.

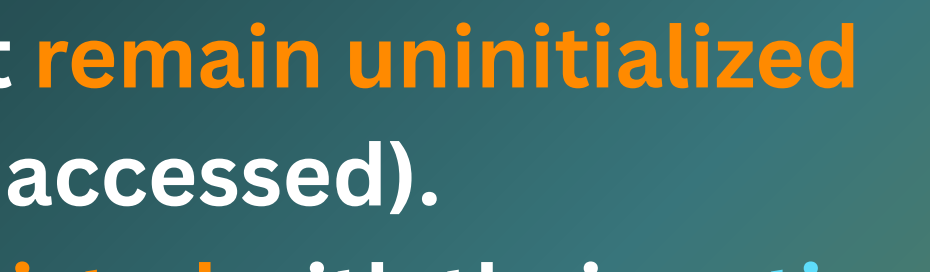
@Shivendra Dwivedi

- let and const are hoisted but remain uninitialized (show <value unavailable> if accessed).
- Function declarations are hoisted with their entire function body.

- Scope Setup:
  - Top-level var and functions → Global Scope
  - Top-level let and const → Script Scope (part of Global).

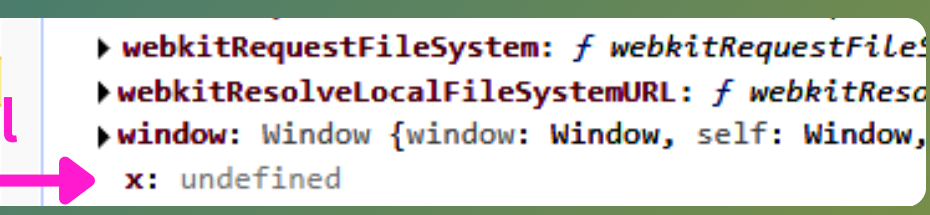
```
4145
4146    let y = "Shivendra";
4147    var x = "John";
4148
4149    function greet() {
4150      var z = "INDIA";
```

‘y’ is created in script scope →



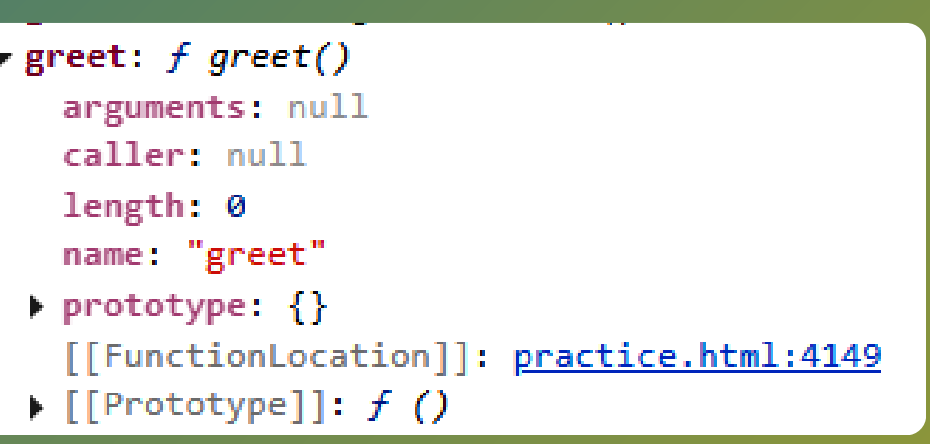

```
let y = "Shivendra";
var x = "John";
```

‘x’ is created in Global scope →




```
4146    let y = "Shivendra";
4147    var x = "John";
4148
4149    function greet() {
4150      var z = "INDIA";
4151
4152      console.log("Hi, I'm", name, "from", country);
4153    }
```

‘greet’ is created in Global scope →



2: Code Execution Phase (Execution Phase)

- Code is executed line-by-line top to bottom.

@Shivendra Dwivedi

- Global Execution Context is created first (named 'anonymous').
- When a function is invoked:
  - A new Function Execution Context is created.
  - It has its own Local Scope + a reference to its Lexical Environment.

These execution contexts are pushed to the Call Stack

```
let y = "Shivendra";
var x = "John";

function greet() {
  var z = "INDIA";

  console.log("Hi, I'm", name, "from", country);
}

greet();
```

'x' got its value on execution

```
▸ webkitRequestFileSystem: ƒ webkitRequestFileSyste
▸ webkitResolveLocalFileSystemURL: ƒ webkitResolveLo
▸ window: Window {window: Window, self: Window, docu
  x: "John"
▸ __REDUX_DEVTOOLS_EXTENSION_COMPOSE__: ƒ $r(...e)
▸ __REDUX_DEVTOOLS_EXTENSION__: ƒ Re(e)
  Infinity: Infinity
▸ AICreateMonitor: ƒ AICreateMonitor()
▸ AbortController: ƒ AbortController()
▸ AbortSignal: ƒ AbortSignal()
▸ AbsoluteOrientationSensor: ƒ AbsoluteOrientationS
```

```
4142  //     // Infinite blocking inside microtask queue
4143  //   }
4144  // });
4145
4146  let y = "Shivendra";
4147  var x = "John";
4148
4149  function greet() {
4150    var z = "INDIA";
4151
4152    console.log("Hi, I'm", name, "from", country);
4153  }
4154
4155  greet();
```

hoisted 'z' of greet function in its local scope

'y' got its value on execution

Call stack

global execution context

greet function execution context

```
▼ Scope
▼ Local
  ▸ this: Window
    z: undefined
▼ Script
    y: "Shivendra"
▸ Global                    Window
▼ Call Stack
  greet                     practice.html:4150
  (anonymous)               practice.html:4155
▸ XHR/fetch Breakpoints
▸ DOM Breakpoints
```

@Shivendra Dwivedi

The Call Stack: JavaScript's Task Manager

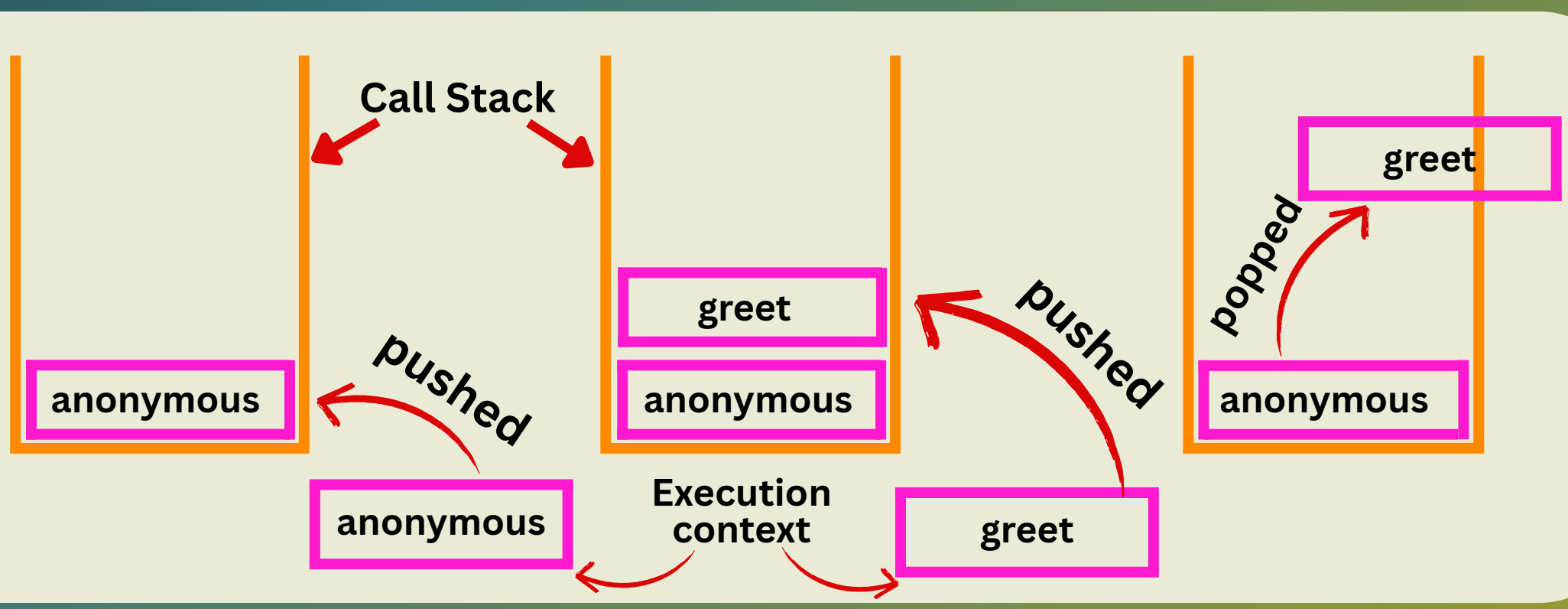
The Call Stack is a stack structure that keeps track of execution contexts.

It pushes (adds) new execution contexts at the top and keeps the oldest at the bottom when functions are called, and pops (removes) them when the function execution is complete.

```
let y = "Shivendra";
var x = "John";

function greet() {
  var z = "INDIA";

  console.log("Hi, I'm", name, "from", country);
}

greet();
```

'x', 'y' will be created in Global execution context

'z' will be created in greet function's execution context



@Shivendra Dwivedi

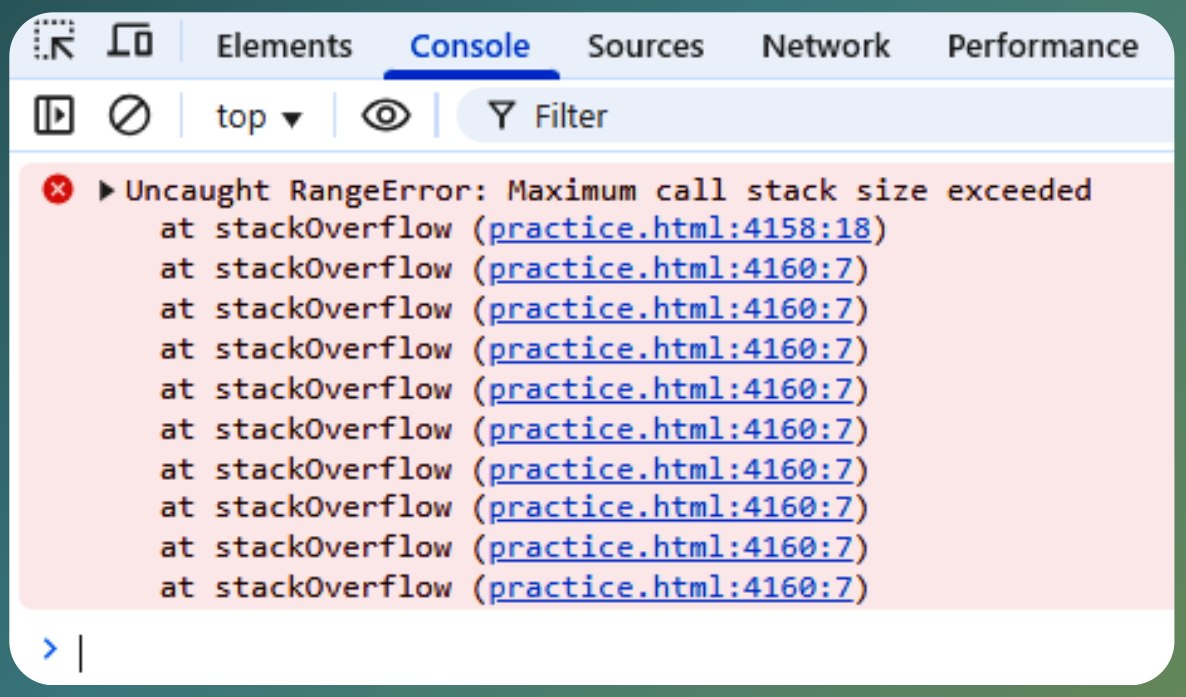
Stack Overflow Error (in JavaScript)

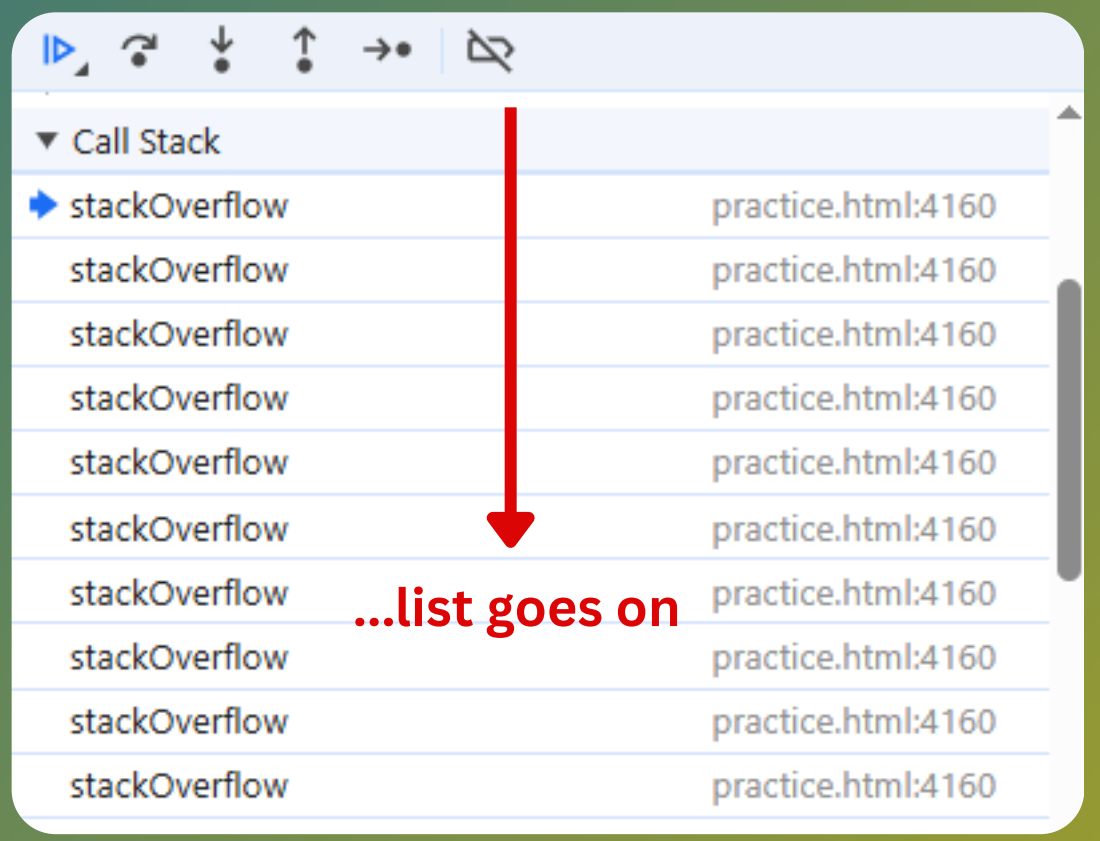
When too many execution contexts pile up without completing, the call stack overflows, causing a stack overflow error.

This is usually due to infinite recursion or extremely deep function calls.

```
function stackOverflow() {
  var name = "Recursion";

  stackOverflow();
}

stackOverflow();
```

self call





@Shivendra Dwivedi

Note: The Call Stack is the **main thread of JavaScript**. All code execution **happens on this single thread.**

Blocking VS Non-Blocking code (the real fight)

1: Blocking Code:

Blocking code **stops the execution** of further code **until the current task finishes.**

```
let count = 0;
for (i = 1; i <= 1e9; i++) {
  count += i;
}

console.log(count); // 500000000067109000
```

After 9 seconds

This for loop **blocked** the JavaScript **main thread for 9 seconds,** delaying the DOM rendering. Until the loop finished, the **browser couldn't update or render anything.**

2: Non-Blocking Code:

Non-blocking code **allows the execution** of other code

@Shivendra Dwivedi

without waiting for the current task to finish.

```
function nonBlocking() {
  return new Promise((resolve,reject) => {
    setTimeout(() => {
      resolve("Done")
    }, 3000)
  })
}

nonBlocking().then((res) => {
  console.log(res); // Done
})

console.log("hi there"); // Hi there
```

after 3 seconds from execution

just after executing the code

JavaScript continues executing without waiting for the 3-second timer.
The Promise resolves later, allowing the main thread to stay free and responsive.

Note: Asynchronous code (callbacks, Promises, async/await) is non-blocking.

Web APIs (Browser's Superpowers)

Web APIs are features provided by the browser (like setTimeout, DOM APIs, fetch, etc.) which JavaScript

@Shivendra Dwivedi

can use to perform tasks outside the main thread without blocking it.

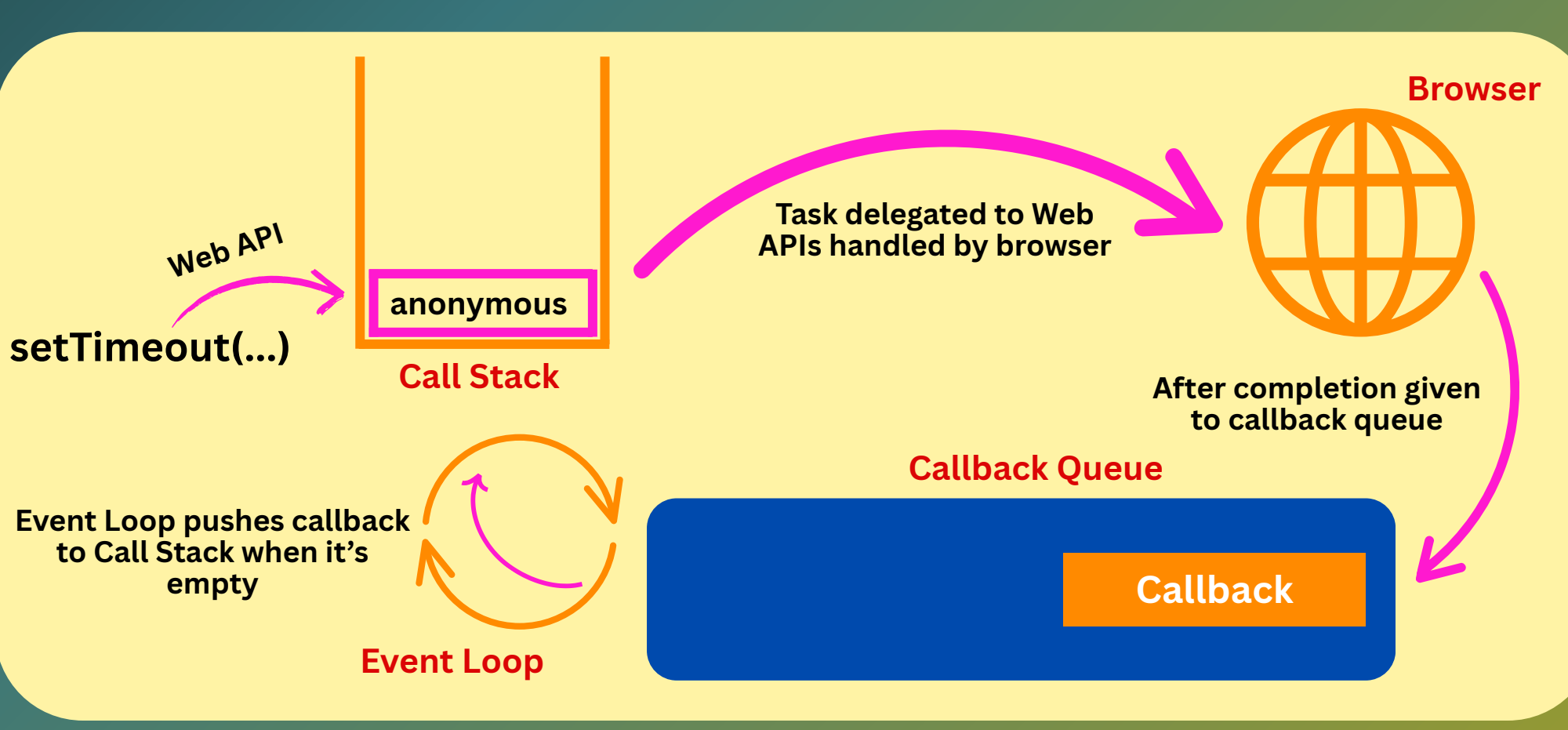
Web APIs are not part of JavaScript.

When JavaScript control encounters a statement like setTimeout(),

➡ Call Stack executes that statement once — meaning, it initiates it.

➡ Then delegates the actual work (like starting the timer) to the Web API (Browser).

➡ After completion, callback enters Callback Queue and Event Loop pushes it back to Call Stack for execution.

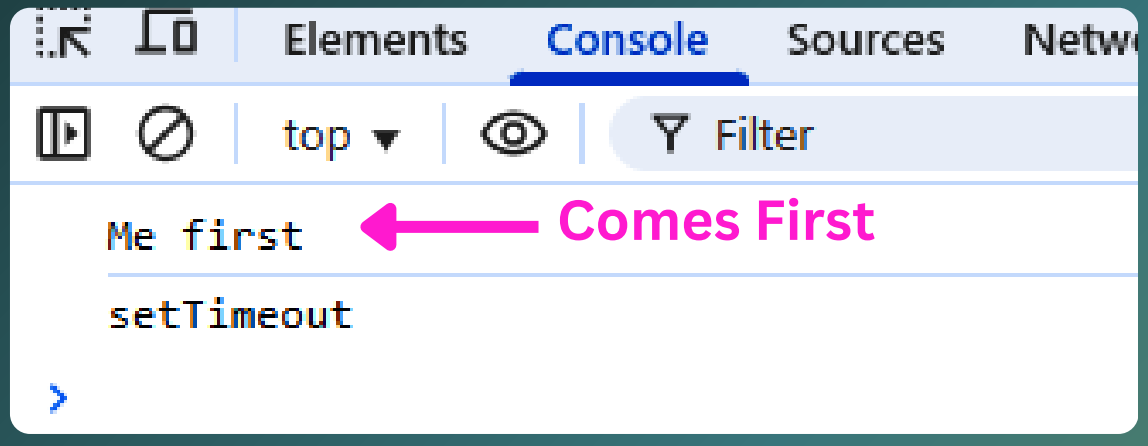


@Shivendra Dwivedi

```
setTimeout(() => {
  console.log("setTimeout"); // setTimeout
}, 0);

console.log("Me first"); // Me first
```

Asynchronous code (Web API)

Synchronous code



Note: The Call Stack always prioritizes synchronous code first. Asynchronous tasks are handled separately and executed later after all synchronous code is done.

Queues In Javascript

Queues are waiting lines where tasks are stored when they are ready to be executed but the Call Stack is busy.

1: Microtask Queue (Priority Workers):

After an asynchronous task (like Promise) completes inside Web APIs, its callback is pushed into the Microtask Queue.

@Shivendra Dwivedi

Tasks like Promise resolutions, queueMicrotask() tasks, MutationObserver goes into microtask queue.

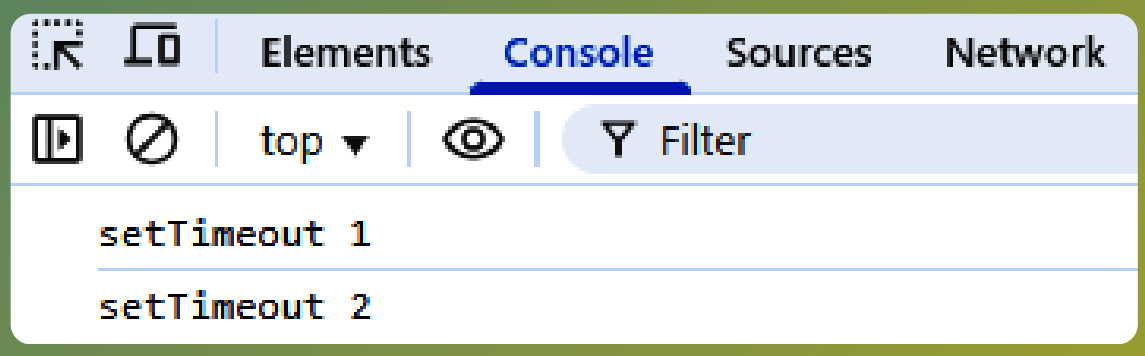
2: Macrotask Queue (Slow and Steady):

After asynchronous tasks (like setTimeout) complete in the Web API section, their callbacks are pushed into the Macrotask Queue.

Tasks like setTimeout, setInterval, UI events like click, scroll, MessageChannel comes into the macrotask queue.

Note: Queues follow the FIFO (First In, First Out) rule: The task that is queued first will be executed first, followed by the second, and so on.

```
setTimeout(() => {   <-- Queued 1st
  console.log("setTimeout 1"); // setTimeout 1
}, 0);

setTimeout(() => {   <-- Queued 2nd
  console.log("setTimeout 2"); // setTimeout 2
}, 0);
```



@Shivendra Dwivedi

Priority & Execution Order in the Event Loop

1: Synchronous Code (Highest Priority)

- Runs directly on the Call Stack. No waiting.
- Must finish before anything else starts.

2: Microtasks (like Promise.then, queueMicrotask)

- Comes right after synchronous code.
- Handled before rendering and macrotasks.

3: Rendering / Paint

- The browser paints UI changes after microtasks (generally), before macrotasks.

Example: reflow, repaint, layout changes.

4: Macrotasks (like setTimeout, setInterval, DOM events)

- Comes after rendering.
- New event loop tick starts with a macrotask.

Real Execution Order

Sync code → Microtasks → Rendering → Macrotasks

@Shivendra Dwivedi

Event Loop in JavaScript

The Event Loop is a mechanism that constantly checks:

- Is the Call Stack empty?
- If yes, then push the next task from the queue (microtask/macrotask) to the Call Stack.

It is called as loop because it keeps looping infinitely, checking again and again.

Manages execution of asynchronous code

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

setTimeout(() => {
  console.log("Timeout callback");
}, 0);

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

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

Sync code

Async code

After Promise, setTimeout callback will be pushed in call stack

After sync code, event loop will push resolved promise in call stack 1st

Sync code

```
Start
End
Promise resolved
Timeout callback
```

@Shivendra Dwivedi

Event Loop Checkpoint After Each Macrotask

After every single macrotask is executed, the event loop always checks the microtask queue.

- If there are any pending microtasks, it executes all of them first, one by one, before going back to the next macrotask.

All Together (Code Example)

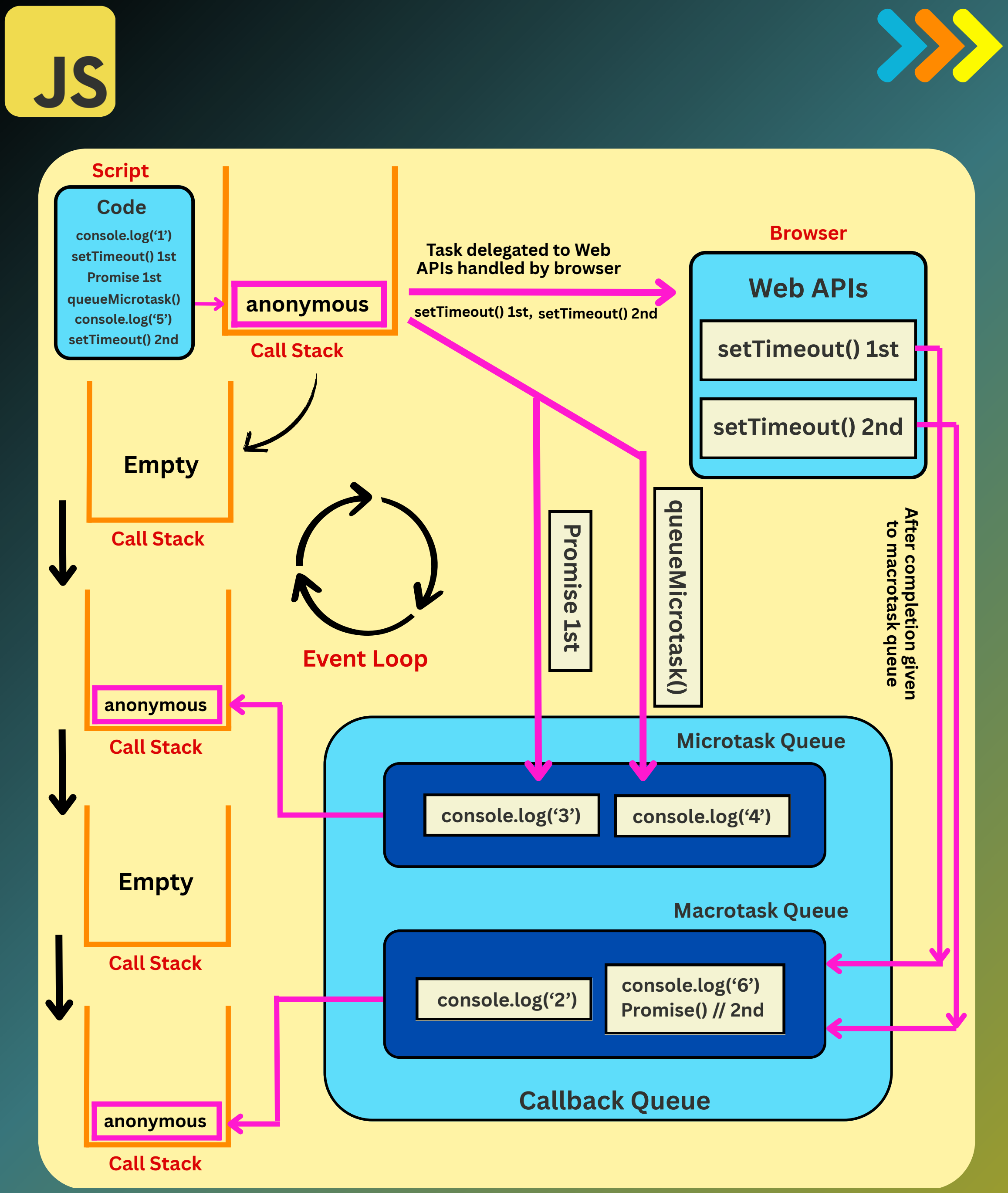
```
console.log('1'); // Sync                    ←— 1

setTimeout(() => { // 1st                    ←— 2
  console.log('2'); // Macrotask
}, 0);

Promise.resolve().then(() => {               ←— 3
  console.log('3'); // Microtask
}); // 1st

queueMicrotask(() => {                       ←— 4
  console.log('4'); // Microtask
});

console.log('5'); // Sync                    ←— 5

setTimeout(() => { // 2nd                    ←— 6
  console.log('6'); // Macrotask
  Promise.resolve().then(() => console.log('7')); // 2nd
  // Microtask inside Macrotask
}, 0);
```

execution will go top to bottom, line by line.

Sync code will get executed directly

Async code is delegated to the Web APIs (in the browser) or scheduled for later execution.

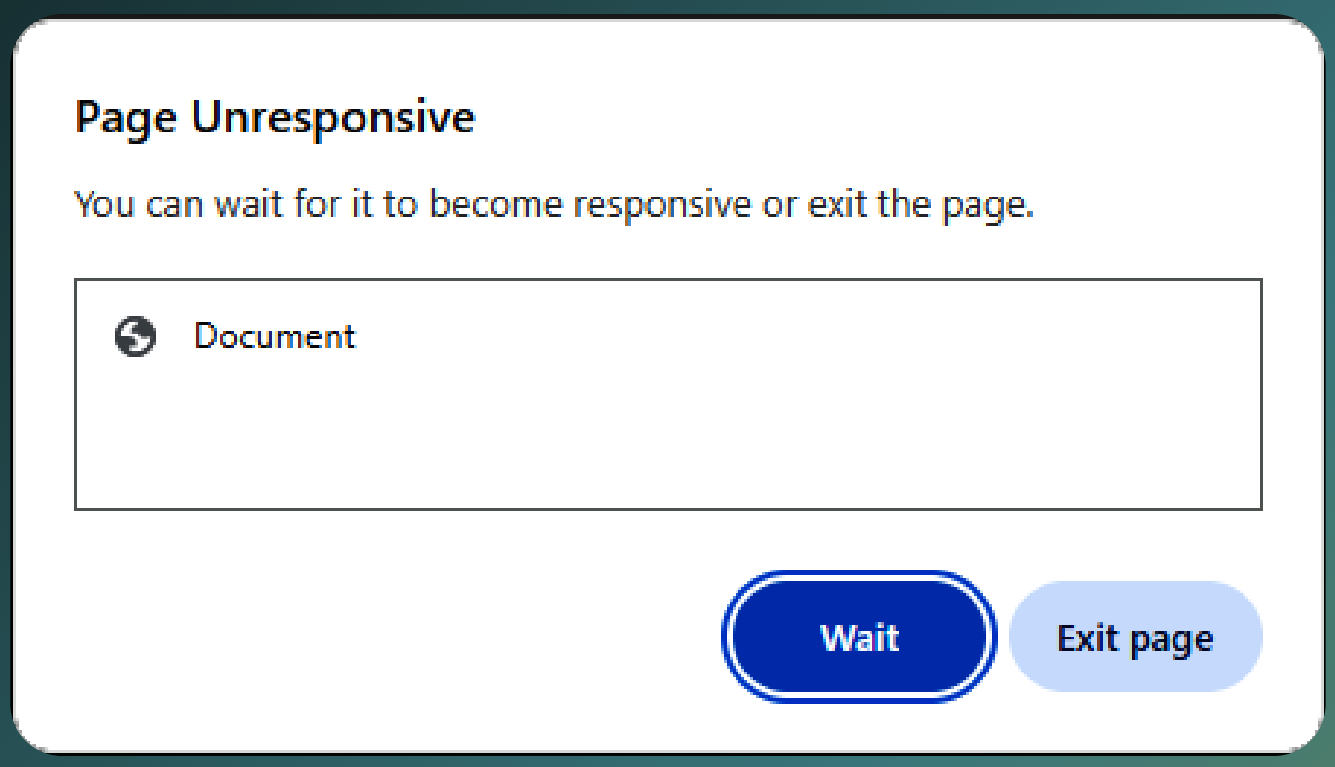
@Shivendra Dwivedi



@Shivendra Dwivedi

Pro Tip

Window Unresponsive Error



The "Window is not responding" or “Page Unresponsive” error occurs when the main thread (call stack) in JavaScript is blocked for too long, especially by synchronous or heavy computations, so the browser can't do other tasks.

```
Promise.resolve().then(() => {
  while (true) {
    // Infinite blocking inside microtask queue
  }
});
```

will run infinitely and block the thread

Note: Use Web Workers to offload heavy or blocking tasks to a separate thread, keeping the main thread responsive.

@Shivendra Dwivedi

JavaScript

Was this helpful?

Like, Comment & Follow

for more JavaScript insights!

Shivendra Dwivedi
@Shivendra-Dwivedi

Connect

Repost