🧠 JavaScript is Synchronous & Single-Threaded

➡️ JavaScript runs code **line by line**, with only **one main call stack**.

```
console.log("Start");
setTimeout(() => console.log("Timeout"), 0);
console.log("End");
```

🧾 Output:

```
Start
End
Timeout
```

📌 This happens because JS uses a **Call Stack**, **Web APIs**, and an **Event Loop** to manage asynchronous code.

🔁 **Callback Functions in JavaScript**

➡️ In JavaScript, **functions are first-class citizens** — this means we can:

- Assign them to variables
- Pass them as arguments
- Return them from other functions

```
setTimeout(function () {
  console.log("⏱️ Timer");
}, 1000);
```

🔹 `setTimeout` takes two arguments:

1. A **callback function** (which gets called later)
2. A **timer** (delay in milliseconds)

📌 **Definition**: A *callback function* is a function passed into another function as an argument and executed later.

📌 A **callback function** is:

- Passed as an argument to another function
- Executed **later**, either asynchronously or after a condition

```
function greet(name, callback) {
  console.log("Hello", name);
  callback();
```

```
}

greet("Darshan", () => console.log("👋 Welcome!"));
```

💡 JavaScript treats functions as **first-class citizens**, allowing:

- Passing functions as arguments
- Returning functions
- Assigning functions to variables

⏱️ setTimeout Example – Delayed Execution

```
setTimeout(() => {
  console.log("🕐 Delayed by 1 second");
}, 1000);
```

🧠 The callback is queued by **Web APIs**, and only pushed to the **Call Stack** after 1 second.

🛑 Blocking the Main Thread

```
function blockFor30Sec() {
  let start = Date.now();
  while (Date.now() - start < 30000) {} // blocks for 30 seconds
  console.log("⌛ Done!");
}
```

🚫 This blocks the only thread — **user can't scroll, click, or interact!**

☑️ Use async patterns to avoid blocking (e.g., `setTimeout`, `fetch`, `Promises`, `async/await`).

🔍 JS: Synchronous but Capable of Async

🧠 JavaScript is a **synchronous**, **single-threaded** language.

But thanks to **callback functions**, we can handle **asynchronous operations** like timers, event listeners, and network calls.

```
setTimeout(function () {
  console.log("🕐 timer");
}, 5000);

function x(y) {
  console.log("x");
  y();
```

```
}

x(function y() {
  console.log("y");
});
```

🧾 **Output Order**:

```
x
y
(timer after 5 seconds)
```

📦 Call Stack + Callback Flow

```
function x(y) {
  console.log("x");
  y();
}

x(function y() {
  console.log("y");
});

setTimeout(() => {
  console.log("Timer");
}, 5000);
```

🧾 Output:

```
x
y
(after 5 sec) Timer
```

🧵 Call Stack Explained

📌 What happens in the call stack?

1. `x()` is invoked and pushed onto the stack.
2. `y()` (callback) is invoked from within `x()` and also added to the stack.
3. After both execute, they're popped off — stack is empty.
4. After 5 seconds, `setTimeout`'s callback is pushed and executed.

⚠️ **Blocking Alert**: If any function (e.g., `x()`) takes too long (say 30 seconds), it **blocks the call stack** ⛔ — JS can't proceed!

✅ **Pro Tip**: Always use **asynchronous techniques** (e.g., `setTimeout`, `Promises`, `async/await`) for long tasks to avoid freezing the main thread.

🖱️ Event Listeners & Callbacks

📄 **HTML**:

```
<button id="clickMe">Click Me!</button>
```

📜 **JS**:

```
document.getElementById("clickMe")
  .addEventListener("click", function xyz() {
    console.log("🖱️ Button clicked");
  });
```

🎯 The function `xyz()` is the **callback**, executed when the event occurs and pushed into the **call stack**.

🧮 Counter Example (Callback + Closure)

❌ Using a global variable (bad practice):

```
let count = 0;
document.getElementById("clickMe")
  .addEventListener("click", function xyz() {
    console.log("Button clicked", ++count);
  });
```

✅ **Using Closure for Encapsulation**:

```
function attachEventList() {
  let count = 0;
  document.getElementById("clickMe")
    .addEventListener("click", function xyz() {
      console.log("Button clicked", ++count);
    });
}
attachEventList();
```

📦 Here, `xyz()` forms a **closure** with `count`, keeping it private and persistent between clicks.

🔍 Scope + Closure with Event Listener

```
function createClickHandler() {
  let secret = "🎁 Secret Value";
  return function () {
    console.log("Accessed:", secret);
  };
}

document.getElementById("clickMe")
  .addEventListener("click", createClickHandler());
```

🔓 Even after `createClickHandler()` finishes, the inner function still has access to `secret`.

🧹 Garbage Collection & removeEventListener

🧠 Event listeners:

- Create closures
- Keep references to DOM nodes
- Can cause **memory leaks** if not cleaned up

```
function setup() {
  const btn = document.getElementById("clickMe");

  function handler() {
    console.log("✅ Clicked");
  }

  btn.addEventListener("click", handler);

  // Remove listener when no longer needed
  btn.removeEventListener("click", handler);
}
```

🧼 **Best Practices**:

- Always `removeEventListener` when the listener is no longer needed
- Helps in **garbage collection** and memory optimization

⚠️ Callback Hell (aka Pyramid of Doom)

```
function printStr(str, cb) {
  setTimeout(() => {
    console.log(str);
```

```
    cb();
  }, Math.floor(Math.random() * 100));
}

function printAll() {
  printStr("A", () => {
    printStr("B", () => {
      printStr("C", () => {});
    });
  });
}

printAll(); // Output: A B C (in order)
```

🌀 This is known as **callback hell** — a common issue in deeply nested asynchronous code.

✅ Later solved using:

- **Promises**
- **async/await**
- Libraries like **RxJS**

✅ Key Takeaways

Concept	Key Idea
🔁 Callback	A function passed to another function and executed later
🧵 Single-threaded	JavaScript has only one call stack (main thread)
⏱️ setTimeout	Built-in async function for delayed execution
🖱️ Event Listener	Uses callback functions to respond to user actions
🎯 Closure	Callback retains access to lexical scope (like `count`)
🧹 Memory Management	Remove unused listeners to avoid memory leaks
⚠️ Callback Hell	Nested callbacks make code hard to read and maintain

💡 Final Thoughts

- ✅ Callback functions enable async behavior in a sync language
- 🧵 JavaScript has a single-threaded call stack
- ❌ Avoid blocking the main thread with long-running sync code
- 🧠 Closures help encapsulate state (like `count`)
- 🧹 Clean up event listeners to optimize memory and performance

---