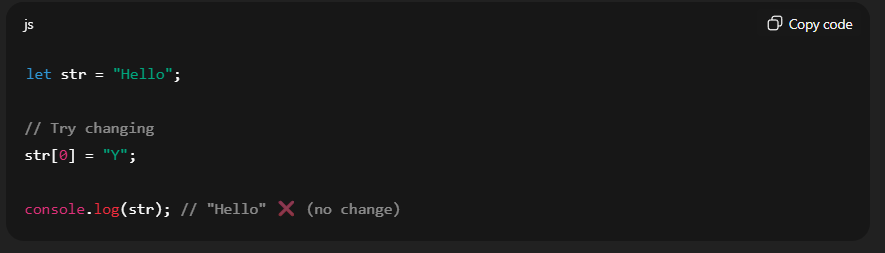
JS

**String methods:**

* length: A property that returns the length of the string (the number of characters).
* charAt(index): Returns the character at the specified index.
* charCodeAt(index): Returns the Unicode value of the character at the specified index.
* concat(string1, string2, ...): Joins two or more strings and returns a new combined string.
* includes(searchString, position): Checks if a string contains another specified string and returns true or false.
* indexOf(searchString, start): Returns the first occurrence of a specified value within the string, or -1 if not found.
* lastIndexOf(searchString, start): Returns the last occurrence of a specified value within the string, or -1 if not found.
* replace(searchValue, replaceValue): Replaces the first occurrence of a specified value (or a regular expression match) with another value.
* replaceAll(searchValue, replaceValue): Replaces all occurrences of a specified value (or a regular expression match) with another value.
* slice(startIndex, endIndex): Extracts a part of a string and returns it as a new string.
* split(separator, limit): Splits a string into an array of substrings based on a specified separator.limit -3 means first three will be inserted only.
* startsWith(searchString, position): Checks if a string begins with a specified string and returns true or false.
* endsWith(searchString, length): Checks if a string ends with a specified string and returns true or false.
* substring(startIndex, endIndex): Extracts a part of a string and returns it as a new string. Similar to slice(), but handles negative indices differently.
* toLowerCase(): Converts the string to lowercase.
* toUpperCase(): Converts the string to uppercase.
* trim(): Removes whitespace from both ends of a string.
* trimStart() / trimLeft(): Removes whitespace from the beginning of a string.
* trimEnd() / trimRight(): Removes whitespace from the end of a string.
* repeat(count): Returns a new string by repeating the original string a specified number of count times.
* match(regexp): Retrieves the result of matching a string against a regular expression.
* search(regexp): Searches a string for a specified value or regular expression and returns the position of the match.

**Q. Can we change the character of string?**

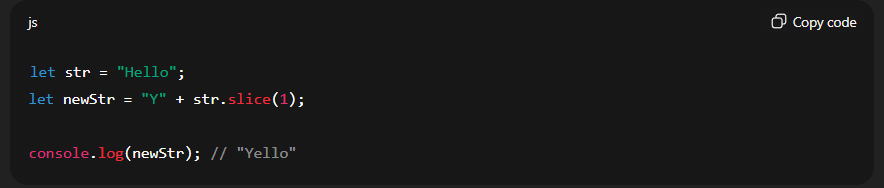
In **JavaScript**, **strings are immutable**, meaning you **cannot change a character directly** in a string (unlike arrays).



**✅ Ways to "change" a character in a string**

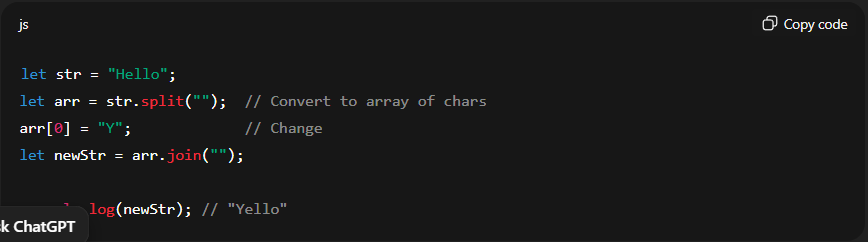
Since strings are immutable, you have to **create a new string**:

**1️⃣ Using substring / slice**



2️⃣**Using replace()**

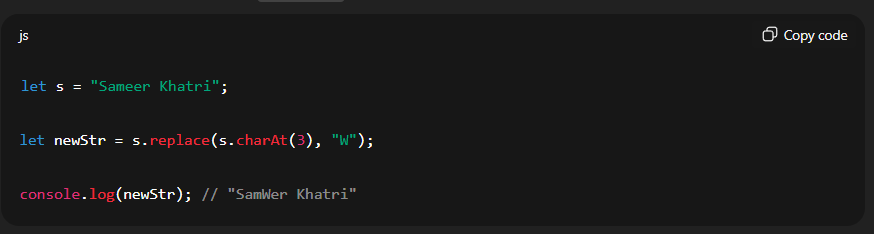
**3️⃣** Using **split + join**



NOTE : read carefully the **replace** method here

 replace() returns a **new string**.

 It does **not modify** the original string s.



Arrays:

**✅ Quick Comparison Table**

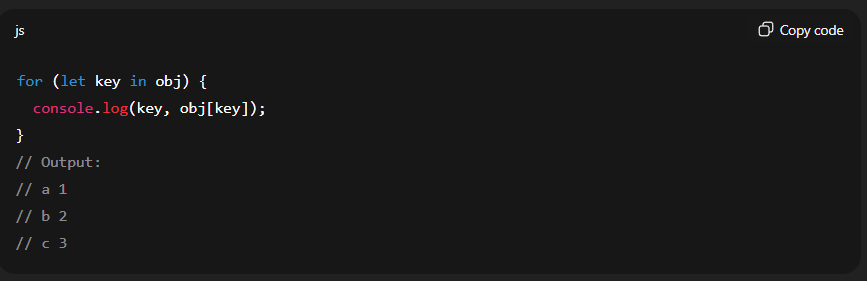
| **Method** | **Returns** | **Callback Params** |
| --- | --- | --- |
| map | New array (undefined) | currentValue, index, array |
| filter | New array (empty) | element, index, array |
| reduce | Single value (undefined) | accumulator, currentValue, index, array |
| forEach | Undefined, return nothing | currentValue, index, array |

Find return first mathc (undefned)

**For-in and for-of loop:**

**1. for in** – loops through the keys of an object

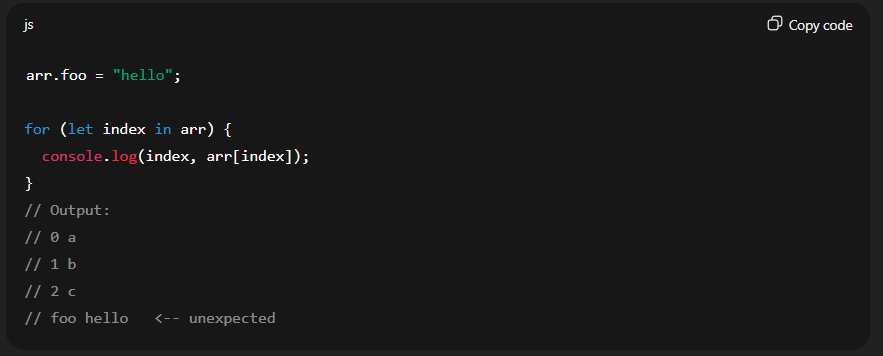
- works on array and object both





⚠️ Be careful:

* for…in will also iterate over **enumerable properties** added to the array object.

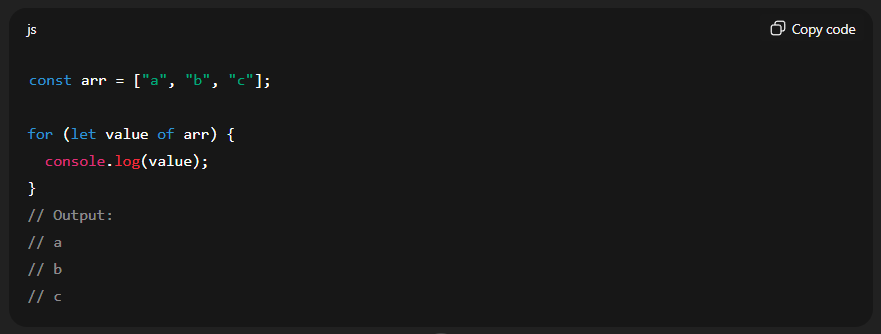


**2. for-of :** loops through the value of an object

- works on the on array

- work also on object but using the Obecy.keys() , Object.values() or Object.entries()

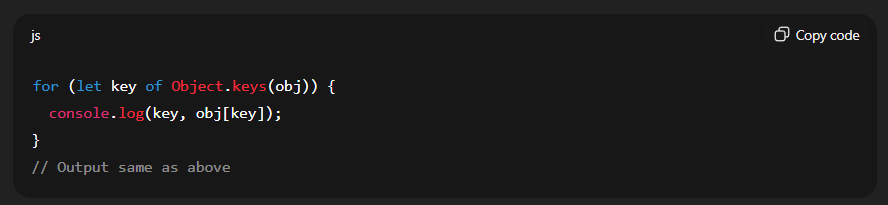
**for…of on Arrays**



**For on Objects:**

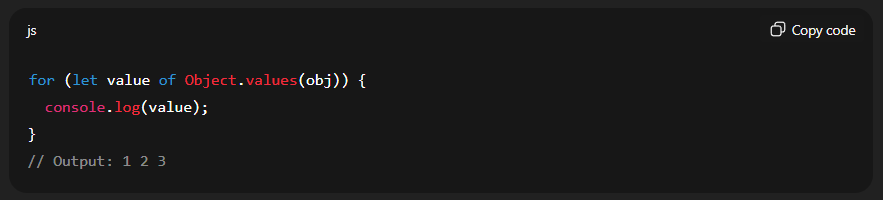
**1. Object.keys() + for…of**

You can convert keys into an iterable array.



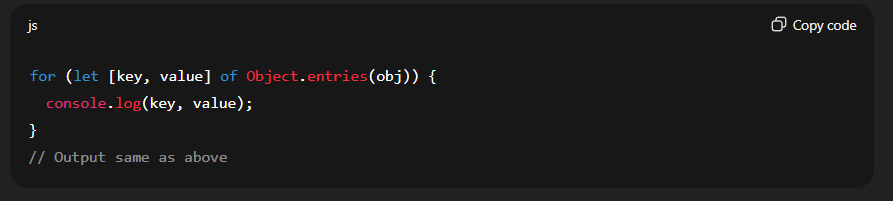
**2. Object.values() + for…of**

Iterate over values directly:



3. **Object.entries() + for…of**

Iterate over [key, value] pairs:



**🔹 Quick Comparison**

| **Feature** | **for…in** | **for…of** |
| --- | --- | --- |
| Iterates over | Index/key | Value |
| Type of variable | String | Actual element |
| Works on array? | Yes, but risky | Yes, safe |
| Works on object? | Yes, keys | No (objects not iterable) (can work by converting values or keys in array using Object) |

**Console methods:**

* console.log(): The most frequently used method, log() outputs general messages, variables, and objects to the console. It's versatile for displaying any type of data during execution.
* Console.error(): Used to log error messages, typically displayed in red in the console, often including a stack trace to help pinpoint the source of the error.
* Console.warn(): Logs warning messages, usually displayed in yellow, indicating potential issues or non-critical problems.
* Console.info(): Logs informational messages, similar to console.log() but sometimes styled differently in some environments to denote informational content.
* Console.table(): Displays tabular data, such as arrays of objects or single objects, in a structured, readable table format within the console.
* Console.assert(): Logs an error message to the console only if the provided assertion (first argument) evaluates to false.
* Console.trace(): Outputs a stack trace of the current execution point, showing the sequence of function calls that led to the console.trace() call.
* Console.clear() : clear the terminal

**Array:**

**Note :**  array is an object in the JS.

Mutating Methods (Modify the original array):

* push(): Adds one or more elements to the end of an array and returns the new length.
* pop(): Removes the last element from an array and returns that element.
* shift(): Removes the first element from an array and returns that element.
* unshift(): Adds one or more elements to the beginning of an array and returns the new length.
* splice(): Changes the contents of an array by removing or replacing existing elements and/or adding new elements.
* sort(): Sorts the elements of an array in place and returns the sorted array.
* reverse(): Reverses the order of the elements in an array in place.
* fill(): Fills all the elements of an array from a start index to an end index with a static value.

2. Non-Mutating Methods (Return a new array or value, leaving the original array unchanged):

* concat(): Joins two or more arrays and returns a new array.
* slice(): Returns a shallow copy of a portion of an array into a new array.
* map(): Creates a new array populated with the results of calling a provided function on every element in the calling array.
* filter(): Creates a new array with all elements that pass the test implemented by the provided function.
  + Example:

const numbers = [1, 2, 3, 4, 5, 6];

const evenNumbers = numbers.filter((number) => {

return number % 2 === 0;

});

* reduce(): Executes a reducer function on each element of the array, resulting in a single output value.

Example:

const numbers = [1, 2, 3, 4, 5];

const sum = numbers.reduce((accumulator, currentValue) => {

return accumulator + currentValue;

}, 0);

* forEach(): Executes a provided function once for each array element.
* find(): Returns the value of the first element in the array that satisfies the provided testing function.
  + Example:

const numbers = [1, 3, 7, 4, 9, 6];

const firstEven = numbers.find((num) => num % 2 === 0);

console.log(firstEven); // Output: 4

* findIndex(): Returns the index of the first element in the array that satisfies the provided testing function.
* includes(): Determines whether an array includes a certain value among its entries, returning true or false.
* indexOf(): Returns the first index at which a given element can be found in the array, or -1 if it is not present.
* lastIndexOf(): Returns the last index at which a given element can be found in the array, or -1 if it is not present.
* some(): Tests whether at least one element in the array passes the test implemented by the provided function.
  + Example:

const users = [

{ name: 'Alice', age: 20 },

{ name: 'Bob', age: 17 },

{ name: 'Charlie', age: 22 }

];

const hasMinor = users.some((user) => user.age < 18);

console.log(hasMinor); // Output: true

* every(): Tests whether all elements in the array pass the test implemented by the provided function.
  + Example:

const mixed = [2, 4, 7];

const allEven2 = mixed.every((num) => num % 2 === 0);

console.log(allEven2); // Output: false

* join(): Joins all elements of an array into a string.
* toString(): Returns a string representing the specified array and its elements.
* at(): Returns the element at a specified index, supporting negative indexing to access elements from the end of the array.
* flat(): Creates a new array with all sub-array elements concatenated into it recursively up to the specified depth.
  + Example: (also removes the empty slots)

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

const flatOnce = nested.flat(); // Default depth = 1

console.log(flatOnce); // [1, 2, [3, [4]]]

const flatTwice = nested.flat(2);

console.log(flatTwice); // [1, 2, 3, [4]]

const fullyFlat = nested.flat(Infinity);

console.log(fullyFlat); // [1, 2, 3, 4]

* flatMap(): Maps each element using a mapping function, then flattens the result into a new array.

Example:

**Example: Removing Null/Empty Results While Mapping**

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

// Double even numbers, ignore odd

const result = arr.flatMap(num => (num % 2 === 0 ? [num \* 2] : []));

console.log(result); // Output: [4, 8]

**map() + flat() vs flatMap()**

arr.map(x => [x \* 2]).flat(); // ✅ works

arr.flatMap(x => [x \* 2]); // ✅ same result, better performance

**🧱 1. DOM (Document Object Model)**

* **What it is**: A tree-like structure that represents the **HTML content** of a web page.
* **Used by**: JavaScript to **read, modify, and update** the page.
* **Example**:

document.getElementById("title").innerText = "Hello!";

* **Performance**: Real DOM manipulations are **slow** when done repeatedly because they cause reflow and repaint in the browser.

**🌐 2. BOM (Browser Object Model)**

* **What it is**: Represents the **browser** features outside the HTML page (not part of the DOM).
* **Includes**:
  + window
  + navigator (browser info)
  + screen
  + location (URL info)
  + history
  + alert, prompt, etc.
* **Example**:

alert("Hello!");

console.log(window.location.href);

**🧠 3. Virtual DOM (used in React, Vue, etc.)**

* **What it is**: A **lightweight JavaScript copy** of the real DOM.
* **Used by**: Modern frameworks like **React** to optimize performance.
* **How it works**:
  + Updates happen in the virtual DOM first.
  + Then it compares (diffs) the old vs. new virtual DOM.
  + It applies only the **minimal changes** to the real DOM (called *reconciliation*).
* **Advantage**: **Much faster** UI updates.

**🆚 Quick Comparison Table**

| **Feature** | **DOM** | **BOM** | **Virtual DOM** |
| --- | --- | --- | --- |
| Represents | HTML structure | Browser environment | Abstract version of DOM |
| Controlled by | Browser & JavaScript | Browser APIs | JavaScript frameworks |
| Performance | Slower for many updates | Not related to UI performance | Fast and optimized |
| Example | document.body | window.location | React’s internal tree |
| Use Case | Modify HTML/CSS | Browser control (alerts, etc) | Efficient UI rendering |

firstChild – can be some text, comments and whitespaces

firstElementChild – should be an html tag

**contains, closest and matches**

**1. Contains (works on the single node)**

**🧠 Syntax:**

parent.contains(child)

**✅ Example:**

<div id="parent">

<p id="child">Hello</p>

</div>

**js**

const parent = document.getElementById("parent");

const child = document.getElementById("child");

console.log(parent.contains(child)); // true

console.log(child.contains(parent)); // false

**Note:**  getElementsByClassName() return the html collection whereas to get the class element by query selector returns the single element.



**2. Mathches**

**Element.matches(selector)**

* Checks if the element **itself** matches the given **CSS selector**.
* Returns true or false

**👇 Example HTML:**

<div class="card selected" id="myCard">Card Content</div>

**✅ JavaScript Example:**

const card = document.getElementById("myCard");

if (card.matches(".selected")) {

console.log("This element has the 'selected' class."); //output

} else {

console.log("Not matched.");

}

**🧠 More Examples:**

card.matches("#myCard"); // true

card.matches("div.card"); // true

card.matches(".not-there"); // false

card.matches("section"); // false

**What is closest()?**

* **element.closest(selector)** looks **up the DOM tree** (including the element itself) to find the **nearest ancestor** that matches the given **CSS selector**.
* Returns the **element** if found, or null if not.

**👇 HTML Example:**

<div class="container">

<div class="card">

<button id="myBtn">Click Me</button>

</div>

</div>

**✅ JavaScript:**

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

// Find the nearest parent with class "card"

const card = btn.closest(".card");

console.log(card); // <div class="card">...</div>

**📘 Explanation:**

* btn.closest(".card") checks:
  1. Is btn itself .card? ❌
  2. Is its parent .card? ✅ → returns that <div class="card">

**innerHTML and outerHTML**

**innerHTML**

* Returns or sets the HTML inside an element (between its opening and closing tags).
* Does not include the element itself**.**

**🔍 Example:**

<div id="box">

<p>Hello</p>

</div>

**js**

const box = document.getElementById("box");

console.log(box.innerHTML); // "<p>Hello</p>"

**outerHTML**

* Returns or sets the **HTML of the entire element**, **including** the element itself.
* When set, it **replaces** the whole element.

**🔍 Example:**

**js**

console.log(box.outerHTML); // "<div id=\"box\"><p>Hello</p></div>"

**🔧 Common Attribute Methods**

| **Method** | **Description** |
| --- | --- |
| getAttribute() | Gets the value of an attribute |
| setAttribute() | Sets/updates an attribute |
| removeAttribute() | Removes an attribute |
| hasAttribute() | Checks if an attribute exists |
| getAttributeNames() | Returns an array of all attribute names (🔹 ES6+) |

**✅ Example HTML:**

<a id="myLink" href="https://example.com" target="\_blank">Visit</a>

**✅ JavaScript Examples:**

const link = document.getElementById("myLink")

// 🔍 has attribute

console.log(link.hasAttribute("href")); // true

console.log(link.hasAttribute("class")); // false

// 🔍 Get attribute

console.log(link.getAttribute("href")); // "https://example.com"

console.log(link.getAttribute("target")); // "\_blank"

// ✏️ Set attribute

link.setAttribute("title", "Go to example");

link.setAttribute("target", "\_self");

// ❌ Remove attribute

link.removeAttribute("title");

// ❓ Check if attribute exists

console.log(link.hasAttribute("href")); // true

console.log(link.hasAttribute("title")); // false

// 📋 Get all attribute names

console.log(link.getAttributeNames()); // ["id", "href", "target"]

**Another Example:**



As in above example:

**data- \* :** like this we can make the custom attribute like in above example decleared as

data-info

**#methods**

| **Method** | **Inserts where?** | **Inside or Outside?** |
| --- | --- | --- |
| append() | At the **end** of target | ✅ Inside |
| prepend() | At the **start** of target | ✅ Inside |
| before() | **Before** the target element | ❌ Outside |
| after() | **After** the target element | ❌ Outside |
| replaceWith() | Replaces the element itself | ❌ Outside (replaces) |

**EXAMPLE:**

const box = document.getElementById("box");

box.append(" 👇"); // inside at the end

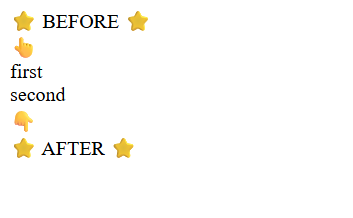
box.prepend("👆 "); // inside at the start

box.before("⭐ BEFORE ⭐ "); // outside before

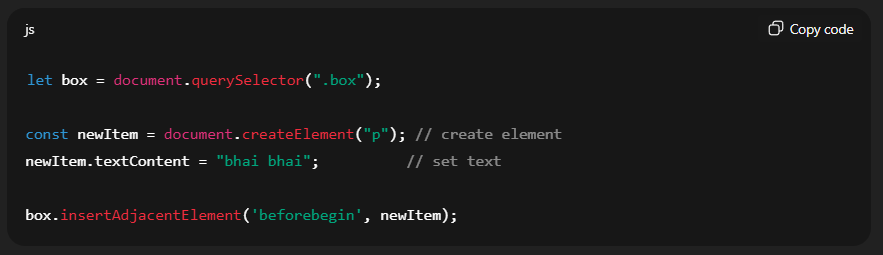
box.after("⭐ AFTER ⭐ "); // outside after

box.replaceWith("❌ Replaced ❌"); // replace it entirely (all the target element content with html)

**Output**

****

**insertAdjacentElement**

****

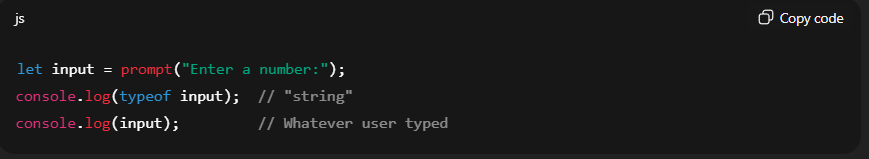
**⚡ Comparison**

| **Method** | **Works Inside / Outside** | **Position** |
| --- | --- | --- |
| append() | Inside | End of element |
| prepend() | Inside | Start of element |
| before() | Outside | Before element itself |
| after() | Outside | After element itself |
| insertAdjacent... | Inside + Outside | All 4 positions (beforebegin, afterbegin, beforeend, afterend) |
| **work with class**   * element.classList.add() * element.classList.remove() * element.classList.toggle() * element.classList : list add classes * element.classList.contains() |  |  |

**// Mine**

**Prompt**

In **JavaScript**, the prompt() function **always returns a string**, no matter what the user types.



Even if the user types 123, prompt() returns "123" (string), not a number.

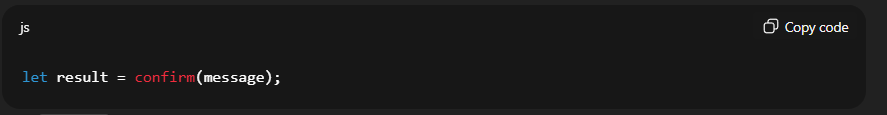
To convert into number:

let num = Number(input);

let num = parseInt(input, 10

**confirm**

confirm() function **always returns a boolean value.**

****

 message → string displayed in the dialog box

 Returns:

* true → if the user clicks **OK**
* false → if the user clicks **Cancel**

**DOM = Document Object Model**

**Definition:**  
The **DOM** is a **programmatic representation of the HTML (or XML) document**. It allows JavaScript to **access, manipulate, and update** the content, structure, and style of a web page dynamically.

* Think of it as a **tree-like structure** where every HTML element is a **node**.
* Nodes can be **elements, attributes, text, comments**, etc.

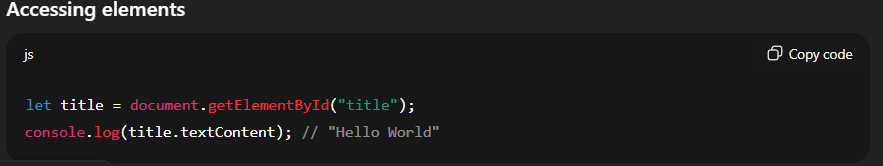




**Why DOM is Important**

* Lets JavaScript **read and modify HTML elements**.
* Example tasks:
  + Change text: <h1> → “Welcome”
  + Update styles: change color, font, visibility
  + Add/remove elements dynamically
  + Respond to user events (clicks, typing, scrolling)
  + Example objects: document, document.getElementById(), document.body.

**Accessing elements**



✅ **Summary**

* **DOM = bridge between HTML and JS**
* **HTML elements → nodes in a tree**
* **JS can manipulate nodes**: text, attributes, style, structure
* Core concept for **interactive web pages**

**2. BOM (Browser Object Model)**

**Definition:**  
The BOM provides **objects to interact with the browser itself**, **outside the document**. It allows JavaScript to **manipulate the browser window, history, location, and navigator**.

**Key Points:**

* Not part of the web page content.
* Tree includes objects like: window, navigator, screen, history, location.
* Tasks: open new windows, get screen size, redirect, detect browser info.



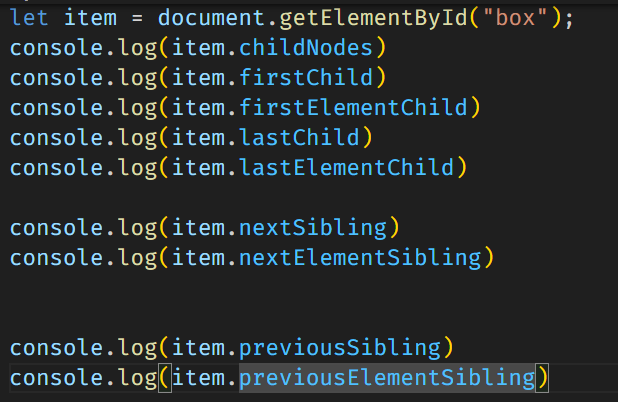
✅ **Summary**

* **DOM** → for the page content (HTML)
* **BOM** → for browser control (windows, location, navigator)
* Both are accessible via **JavaScript**.

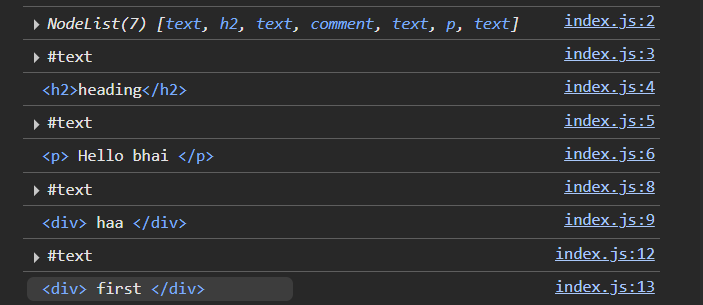
**Nodes:**

* There are main 3 types of nodes.
  + Element nodes (div, p, h1)
  + Text node (simple text)
  + Comment node (//commented text)

**Accessing nodes (child and elementNodes)**

** **

**Output:**

****

**Searching the DOM**

list out the all functions like:

* getElementById()
* getElementByClassName() : return the collection of nodes so cannot perform the matches, closests like operation
* querySelector() : return one node element whether searched fir the class
* querySelectorAll()
* getElementByTagName()

**textContent vs innerText**

**textContent**

* Returns **all the text** inside an element, including hidden text (CSS display: none or visibility: hidden).
* Faster than innerText (doesn’t trigger reflow).
* Ignores CSS styling and layout.
* Example:



**innerText**

* Returns **only visible text**, respects CSS styles like display: none, visibility: hidden, etc.
* Causes **reflow** (browser computes styles/layout before returning).
* Slower compared to textContent.
* Example:



✅ **Rule of Thumb**:

 Use **textContent** when you just need text (faster).

 Use **innerText** when you care about what the **user actually sees** on the page.

**⚡ Key Differences**

| **Feature** | **textContent** | **innerText** |
| --- | --- | --- |
| Includes hidden text? | ✅ Yes | ❌ No |
| Performance | Faster | Slower (triggers reflow) |
| CSS awareness | ❌ Ignores CSS | ✅ Respects CSS |
| Use case | When you just want raw text (for scripts, data processing) | When you want text **as seen by user** |

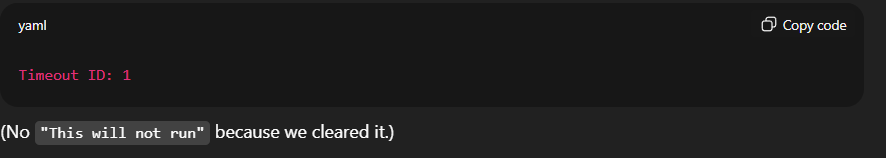
**SetTimeout vs SetInterval**

Both setTimeout and setInterval return a numeric ID (in browsers) or an object/ID (in Node.js).  
This ID can be used to cancel the scheduled execution.

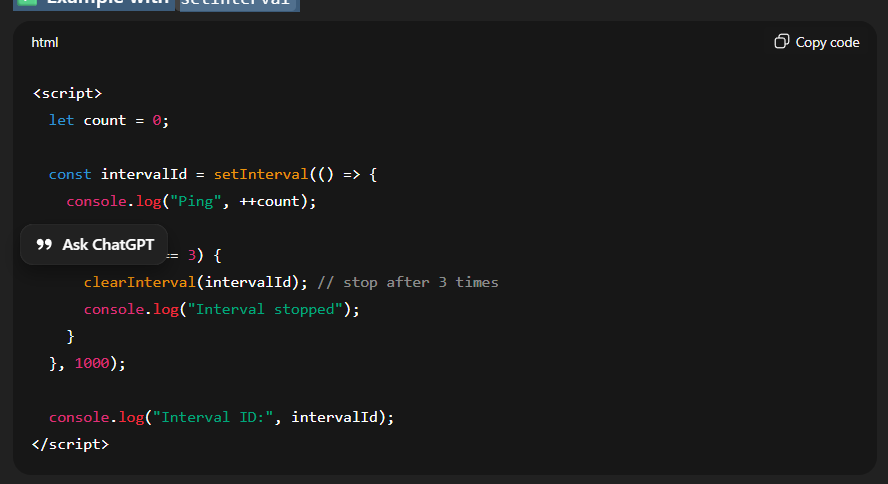
**✅ Example with setTimeout**



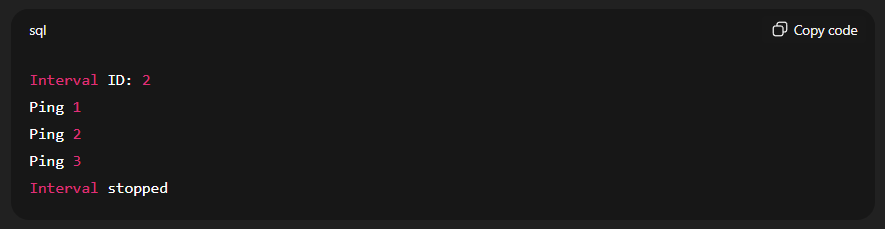
✅ Output:



**✅ Example with setInterval**



✅ Output:

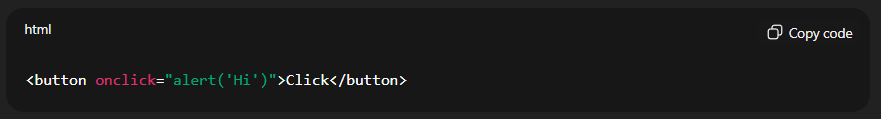


**🔹 What are Browser Events?**

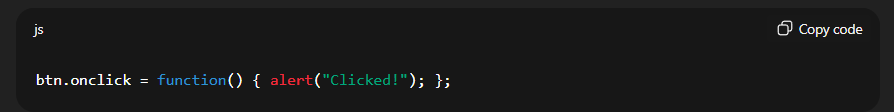
An **event** is a signal that something has happened in the browser, like:

**⚡ Ways to Attach Events**

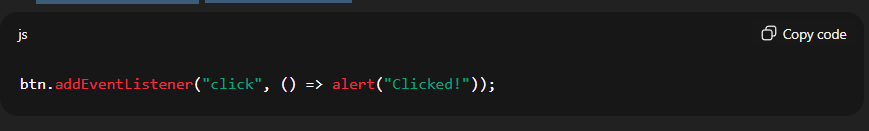
1. **Inline HTML attribute** (not recommended)



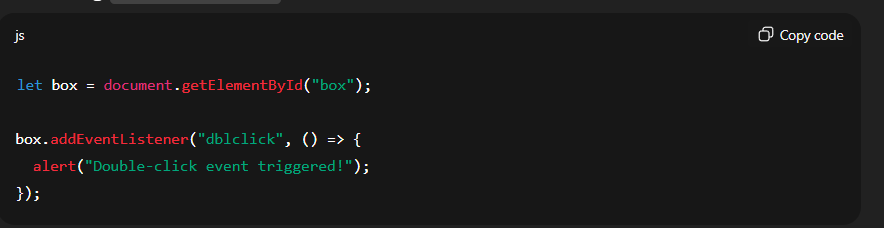
2. Assigning to element property



**3. Modern way (best): addEventListener**



**Double click:**

****

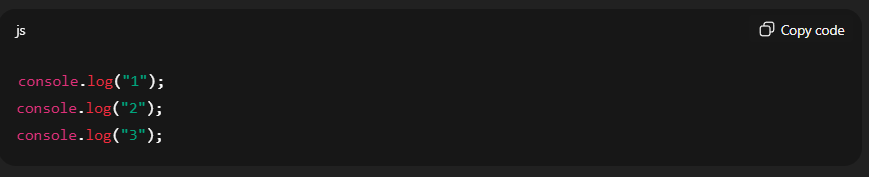
**🔹 Synchronous vs Asynchronous in JavaScript**

**🟢 Synchronous**

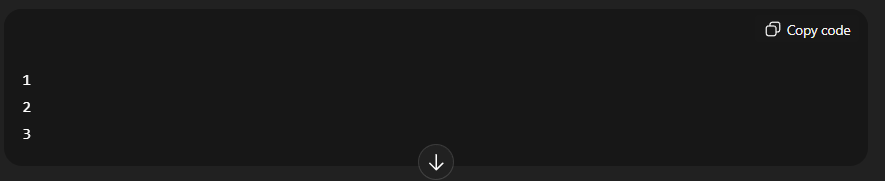
* Code executes **line by line**, **one after the other**.
* A line must finish **before the next one starts**.
* If something takes a long time (e.g., a heavy loop), everything else **waits** (blocking).

Eg: if a loop is running then the code after that loop will wait of loop execution.

✅ Example:



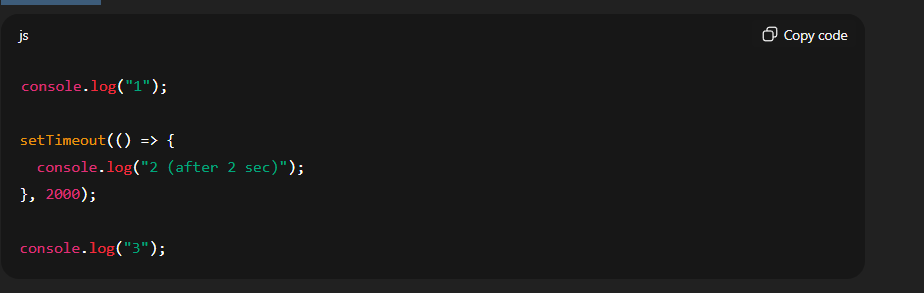
📌 Output (always in order):



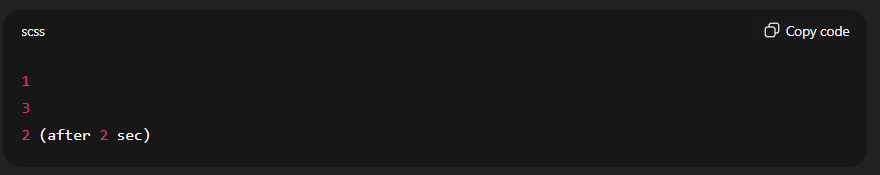
**🔵 Asynchronous**

* Code **does not block** the execution of the next line.
* Tasks can be **delegated** (to browser APIs like timers, fetch, etc.).
* The JS engine continues execution while those tasks are handled in the background, and their result comes later via the **callback queue + event loop**.

✅ Example:



📌 Output:



**🛠️ Real Life Analogy**

* **Synchronous** → Standing in a queue at a single billing counter. You must wait for the person ahead to finish.
* **Asynchronous** → Ordering food in a restaurant. You order, continue chatting, and food is delivered later when ready.

**Note:** JavaScript also provides mechanisms to handle asynchronous tasks, allowing for non-blocking operations. This is achieved through features like:

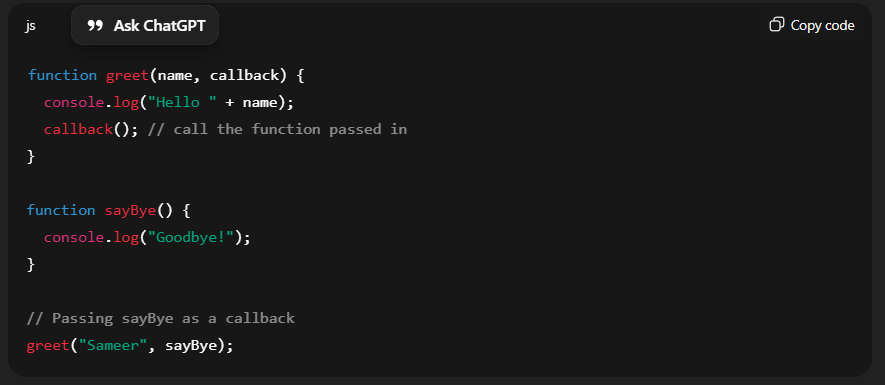
Callback, promise and Async/Await

✅ So the truth:

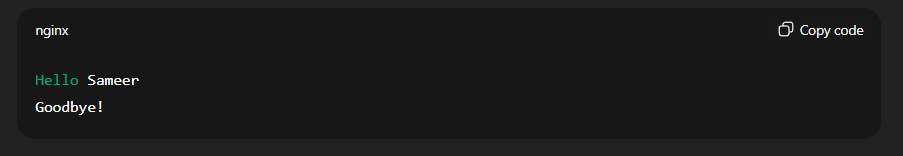
* **JS engine** (V8, SpiderMonkey, etc.) runs code synchronously.
* **Async behavior** (like setTimeout, fetch, promises) is managed by **browser APIs** + **event loop**, not by JS core itself.

**1.** **What is a Callback Function?**

A **callback function** is simply a function that you **pass as an argument to another function**, and that function will "call it back" later.

✅ Example 1 – Basic Callback

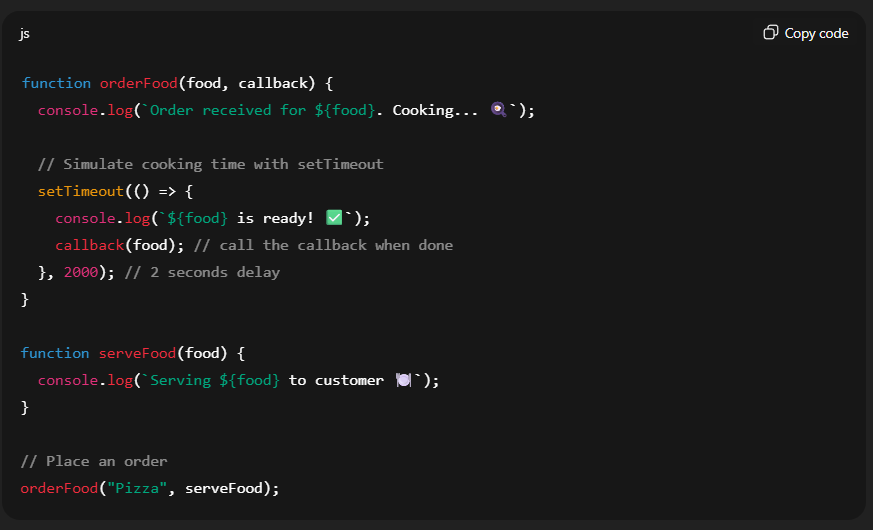
**Output:**



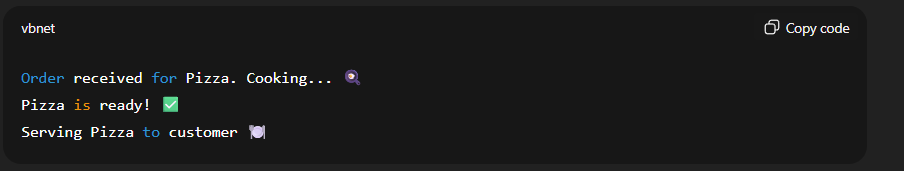
✅ Example 2 – Callback in Asynchronous Code

**Simulating a Food Order System**

Imagine you order food online. The system needs time to "cook" (asynchronous task). Once it’s ready, the system calls you back.



Output:



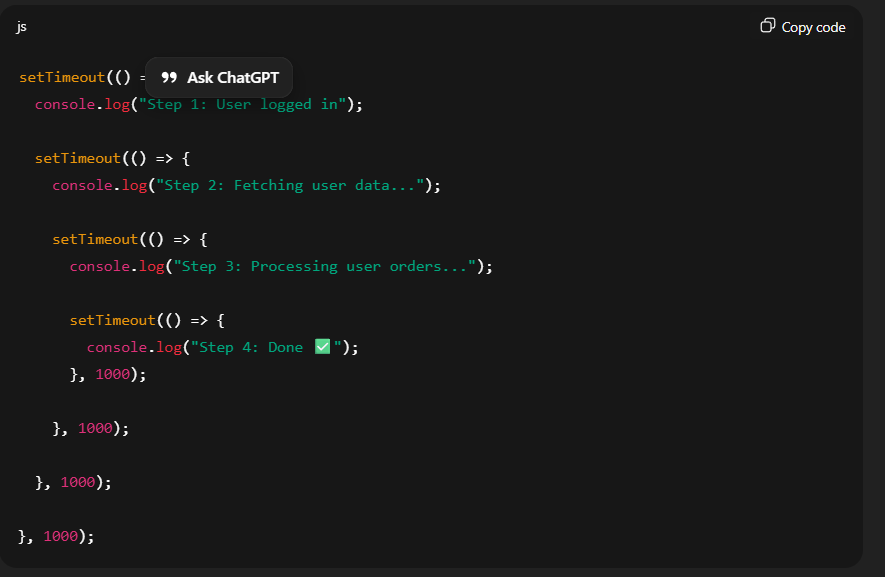
Here:

* orderFood takes a **callback** (serveFood).
* The callback is called **only when the async task (cooking) is done**.

**🔹 Callback Hell (Pyramid of Doom)**

The term **Callback Hell** (also called the **Pyramid of Doom**) happens in JavaScript when you have **too many nested callbacks**, usually for async operations like fetching data, reading files, or handling timers.

Example of Callback Hell (Pyramid of Doom)



👉 Notice how the code **keeps indenting to the right** like a pyramid ⛰️.  
👉 This makes code **hard to read, maintain, and debug**.

**🔹 Why does Callback Hell happen?**

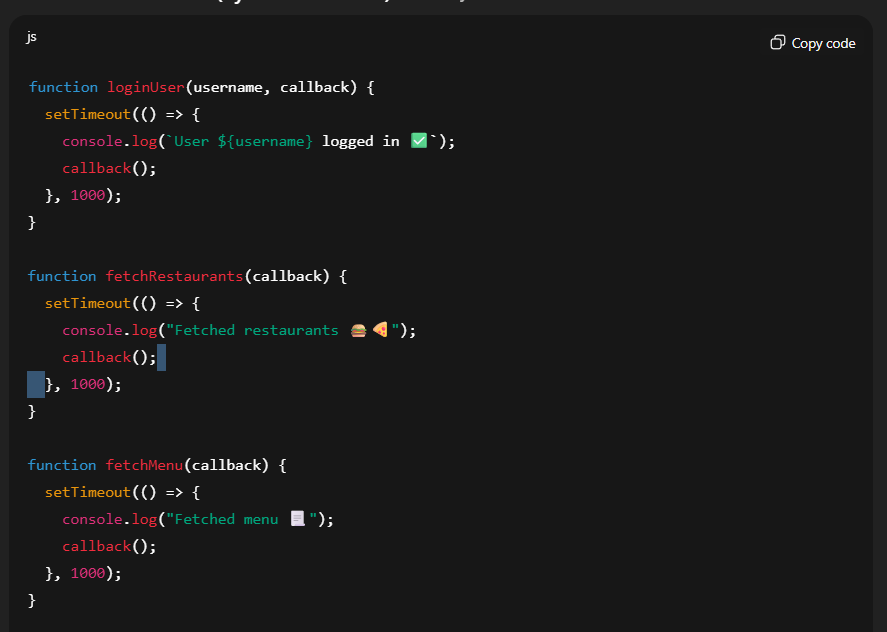
Because async functions depend on the result of previous async functions, so we keep nesting callbacks inside callbacks.

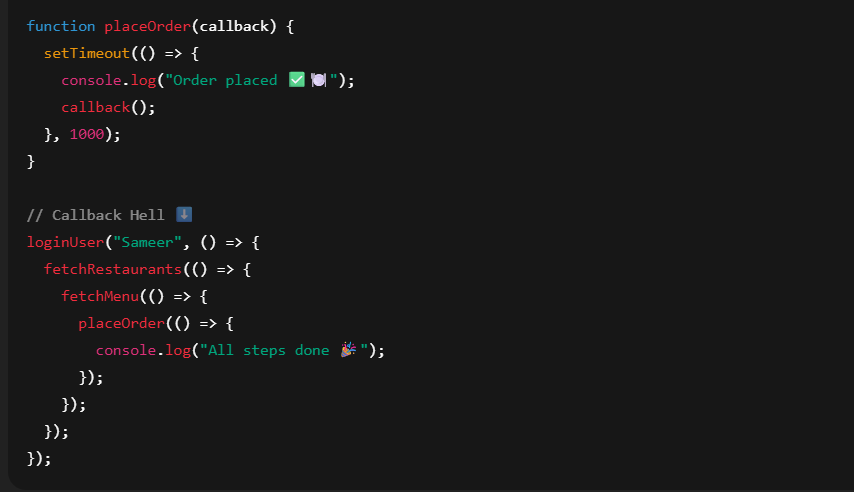
**Example:**

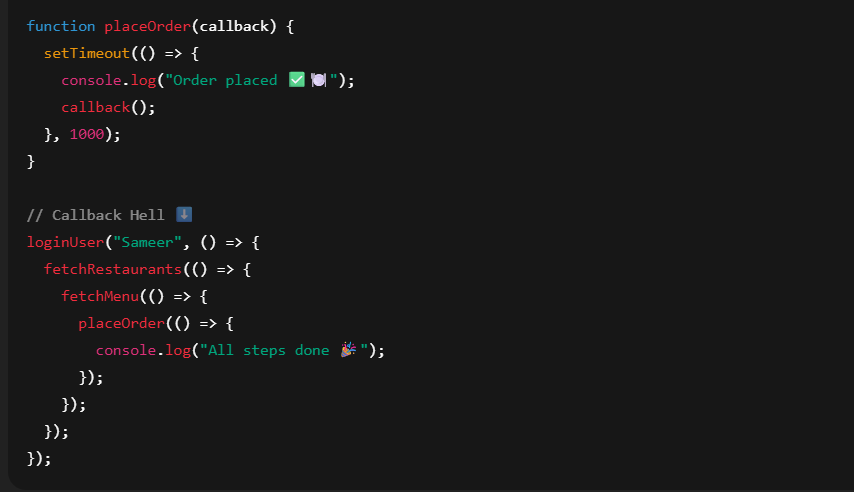
**real-life async flow** that you can relate to.

Imagine a food ordering app (like Zomato/Swiggy).  
Steps:

1. Login user
2. Fetch restaurants
3. Select a restaurant → fetch menu
4. Place order

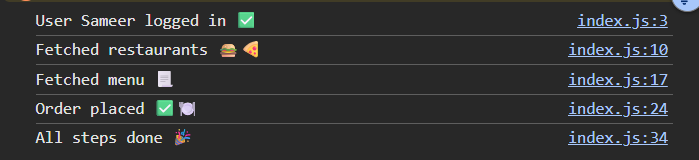






👉 This works, but look how **nested** it is! That’s the **Pyramid of Doom**.

Output:



**2.**  **What is a Promise?**

A **Promise** in JavaScript is an object that represents the **eventual result** of an asynchronous operation.  
It can be in one of three states:

1. **Pending** → initial state (operation not finished yet).
2. **Fulfilled** → operation completed successfully (resolved).
3. **Rejected** → operation failed (error occurred).

Think of it like ordering food online 🍔:

* While waiting = **Pending**
* Food delivered = **Fulfilled**
* Restaurant says "not available" = **Rejected**

**Basic Promise Example**



**🔹 Key Methods**

* **resolve(value)** → marks the promise as successful, passing a result.
* **reject(error)** → marks the promise as failed, passing an error.
* **then()** → handles success.
* **catch()** → handles failure.
* **finally()** → runs always (success or fail).

**What is Promise Chaining?**

* When one .then() returns a promise, the **next .then() waits for it to finish**.  
  This allows you to run asynchronous tasks in sequence **without nesting callbacks**.

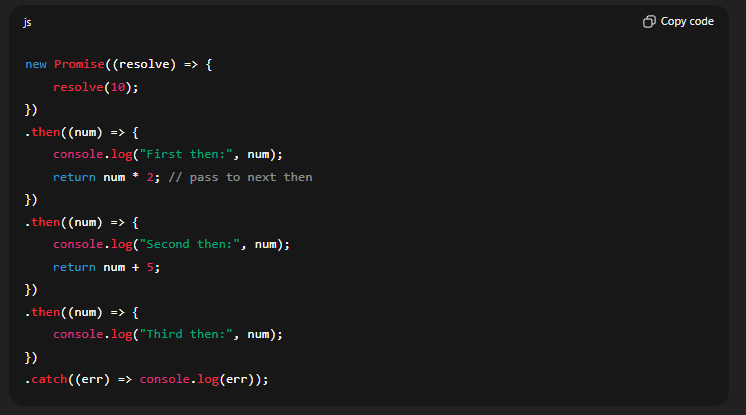


**✅ Why this solves callback hell:**

1. **Flat structure** → No pyramid shape, easy to read.
2. **Error handling in one place** → .catch() handles all errors instead of checking inside each callback.
3. **Sequential flow** → Each .then() waits for the previous one to finish.
4. **Reusability** → Functions like demo, add, print return promises and can be reused anywhere.

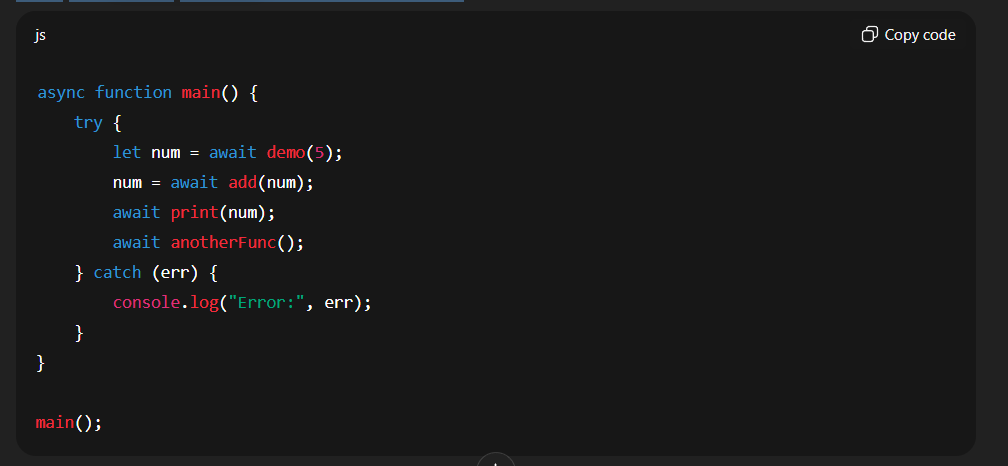
**🔹 Example 2: Returning Values**

You can also **pass data along the chain**.



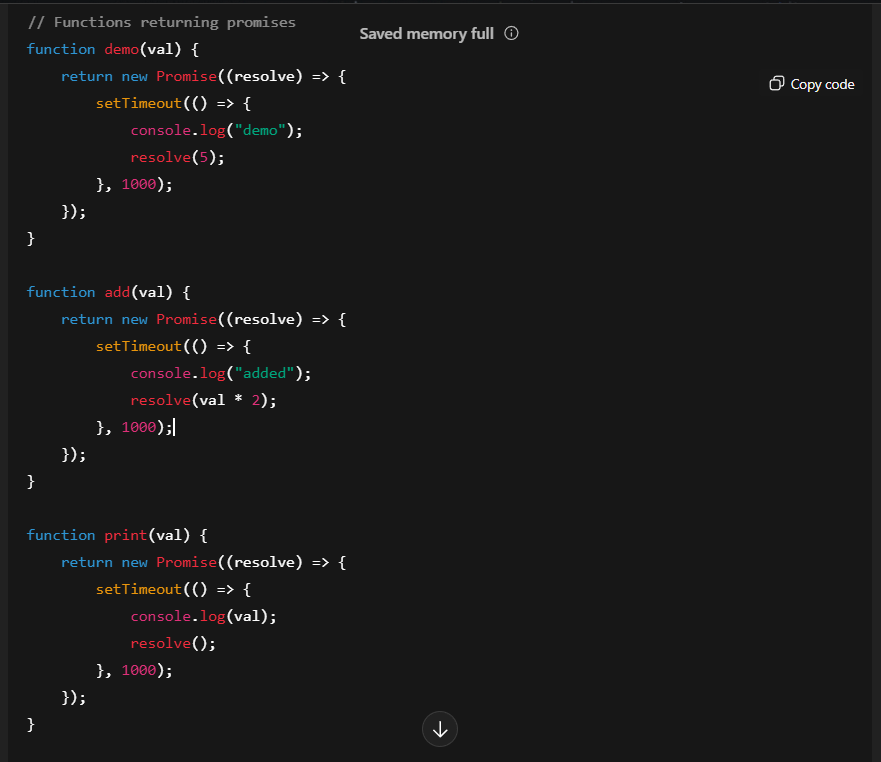
**🔹 Even Cleaner: Async/Await**

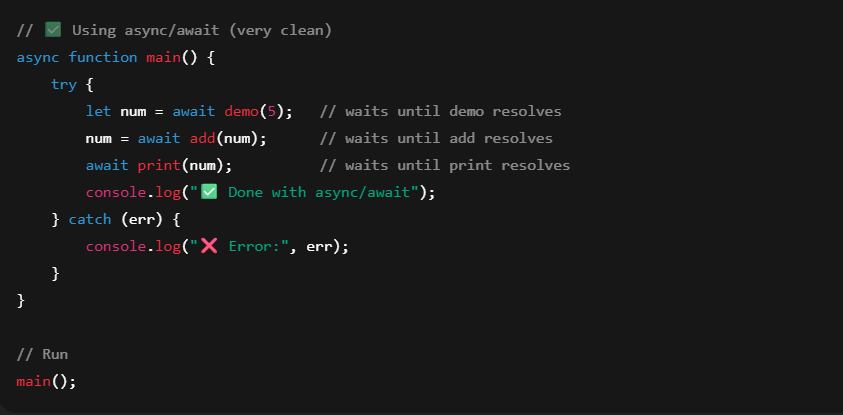
With async/await, it looks like **synchronous code**:



**3. Using Async/Await**

Async/await makes it look like synchronous code:





**Output:**

* After 1s → demo
* After 2s → added
* After 3s → 10
* After 3s → ✅ Done with async/await

**Note :**  Async/await is basically **Promises under the hood**, but it makes code **look clean and simple**.

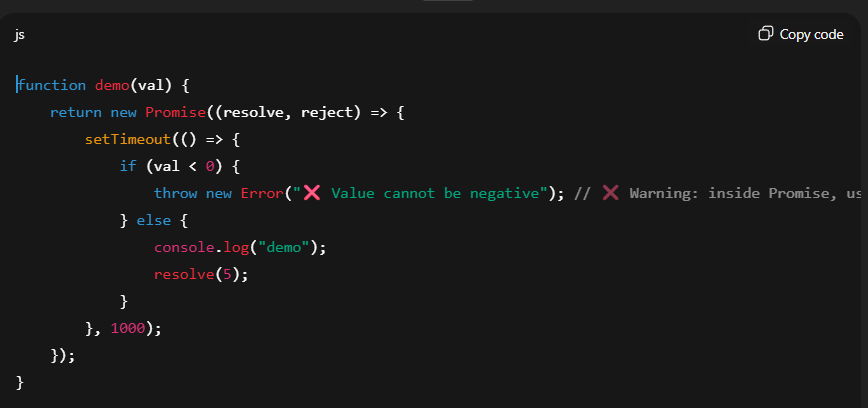
**🔹 Comparison Summary**

| **Feature** | **Callbacks 😵** | **Promises 🙂** | **Async/Await 😍** |
| --- | --- | --- | --- |
| Code Structure | Nested (pyramid of doom) | Flat (.then() chaining) | Synchronous-looking |
| Error Handling | Each callback needs handling | .catch() in one place | try/catch like normal code |
| Readability | Poor | Better | Best |
| Debugging | Hard | Easier | Very easy |
| Reusability | Low | High | High |

**Throw , try and catch**

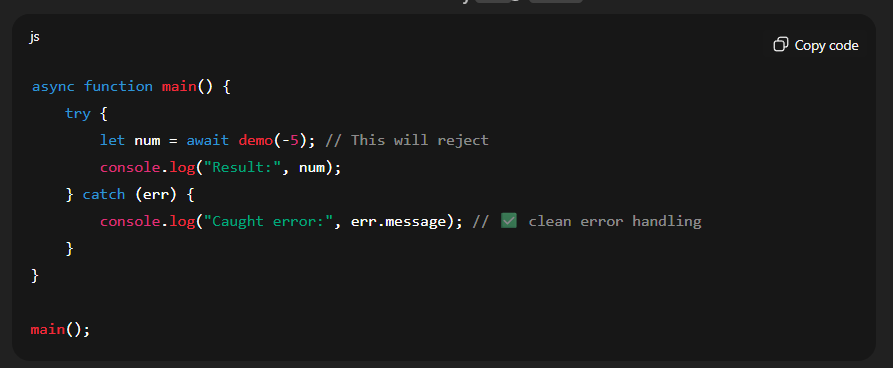
**🔹 1. Throwing an Error**

In JavaScript, you can **manually throw an error** using throw:

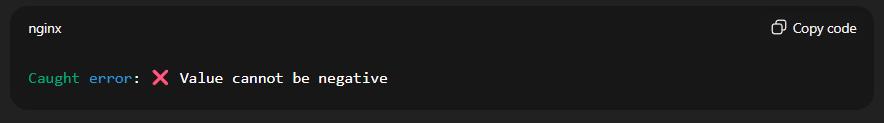


✅ **Important:** Inside a **Promise**, you should use reject() instead of throw, because throw inside a Promise executor will crash the program.

Correct way:



Output:



**Destructuring**

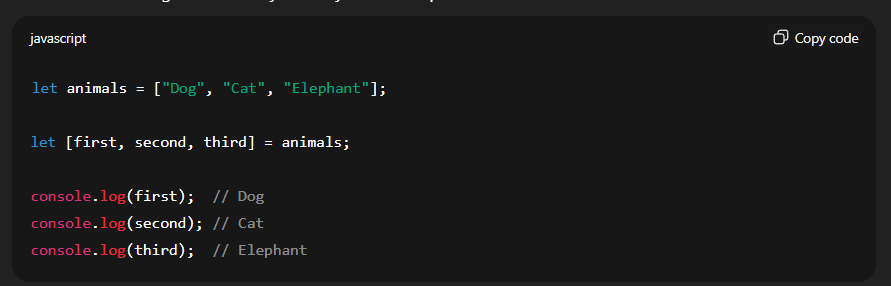
Array Destructuring:

🔹 1. Access items directly from an array

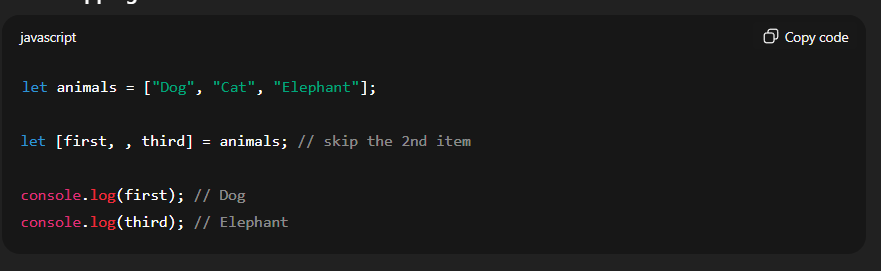


**🔹 2. Array Destructuring**

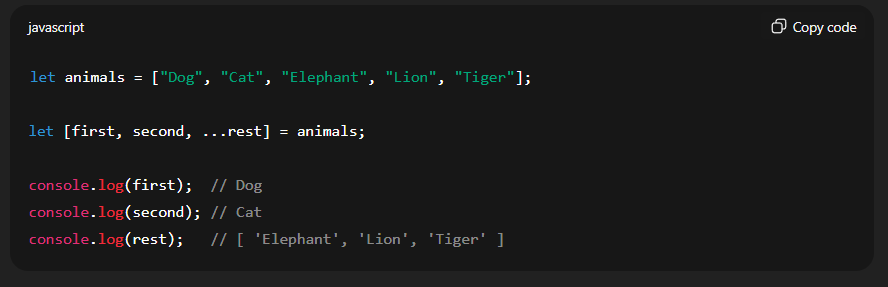
Instead of accessing each item by index, you can "unpack" values into variables:



🔹 3. Skipping Items



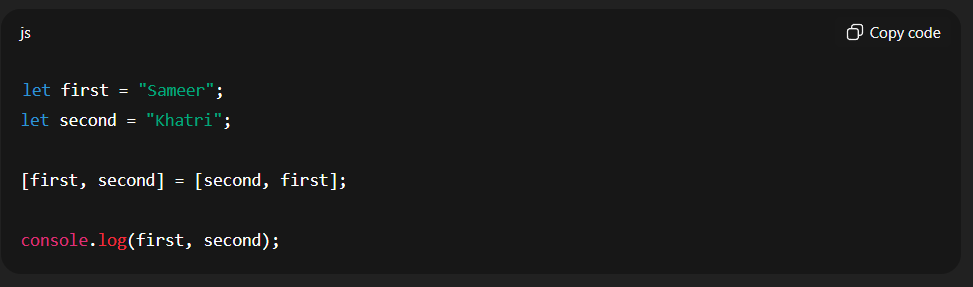
**🔹 4. With Rest Operator (...)**



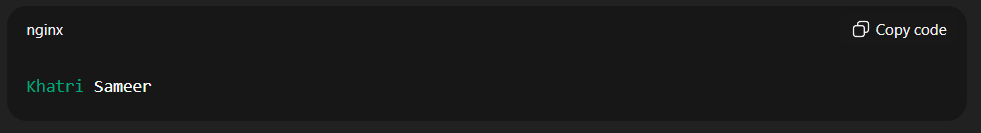
**🧠 3️⃣ Difference in Purpose**

| **Expression** | **Meaning** | **Example** | **Output** |
| --- | --- | --- | --- |
| ...arr | Spreads elements individually | console.log(...[1,2,3]) | 1 2 3 |
| [...arr] | Creates a *new array copy* | console.log([...[1,2,3]]) | [1,2,3] |

**Swapping Two Variables Using Destructuring**



🧾 **Output:**

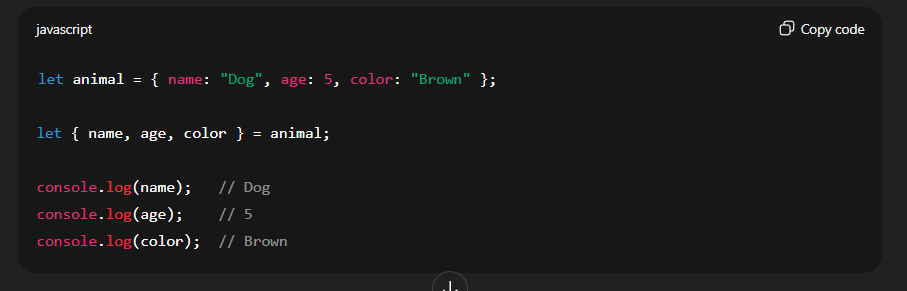


Object Destructuring:

🔹 1. Object Access (Normal Way)



🔹 2. Object Destructuring



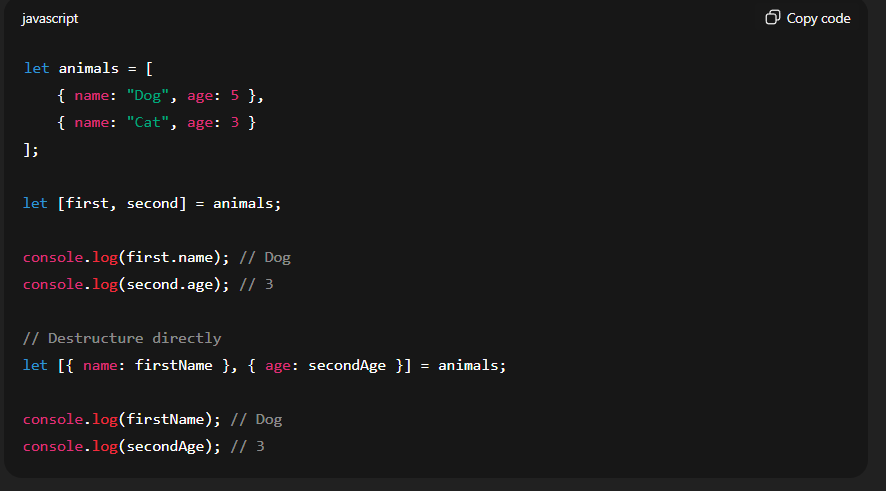
**🔹 3. Destructuring with Different Variable Names**



🔹 4. Nested Object Destructuring



**🔹 5. Mixing Array + Object Destructuring**



**Other Example (…rest operator)**

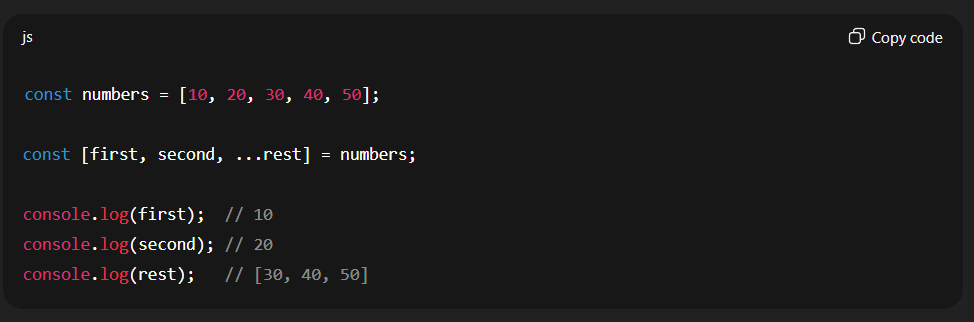
The **rest operator** allows you to collect the **remaining items** of an array or object into a new variable.

It’s mainly used in:

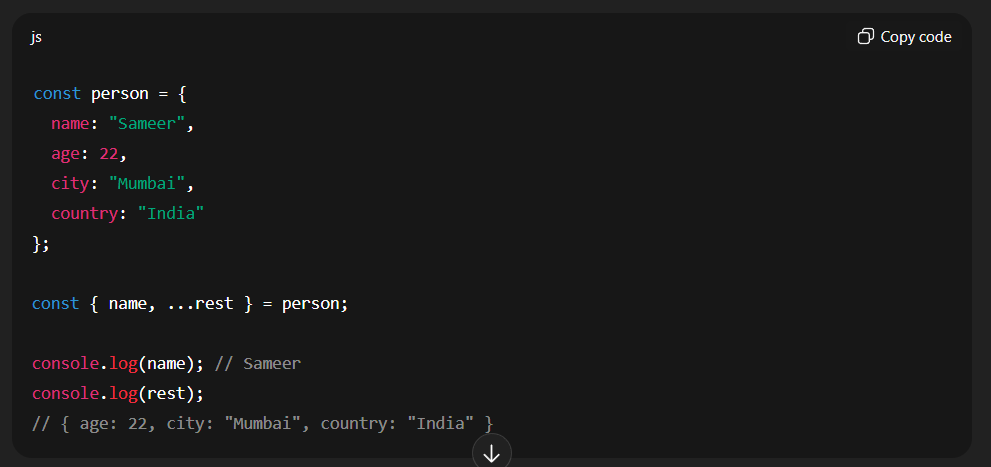
* **Destructuring (arrays/objects)**
* **Function parameters**

🧠 Syntax:  
const [a, ...rest] = array  
const {x, ...rest} = object

📘 1. Rest Operator with **Arrays**



📗 2. Rest Operator with **Objects**

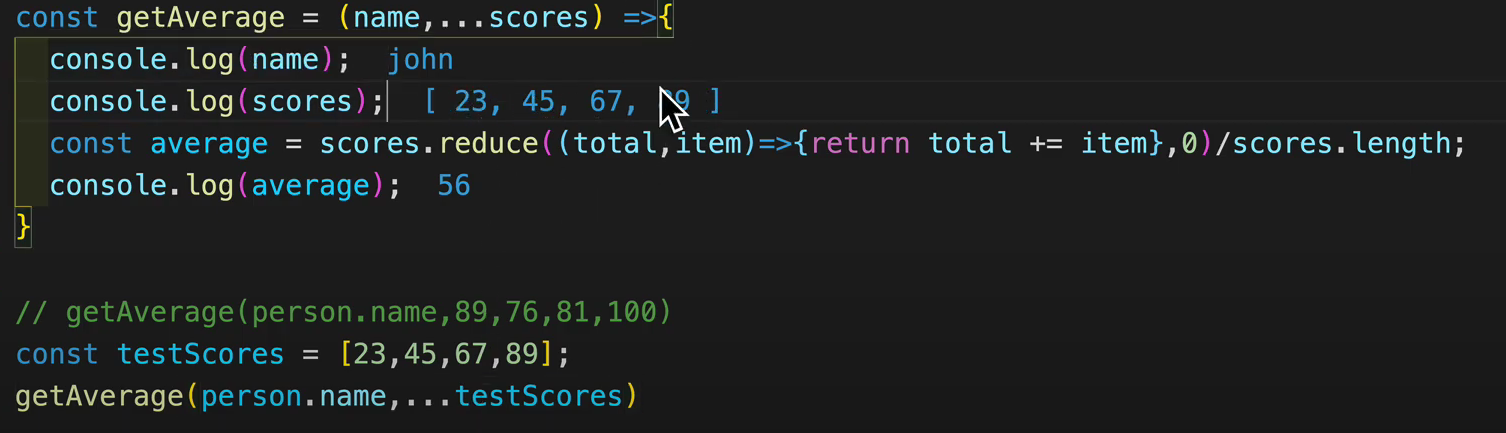


Note:

1. spread operator : …

2. rest operator : …rest

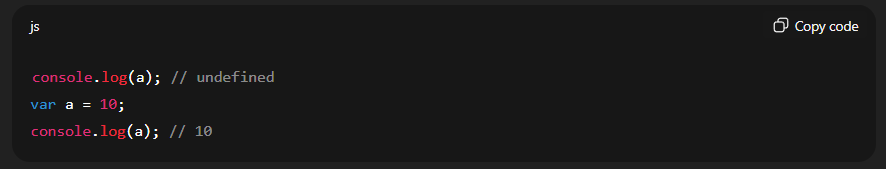
**Like var-arg** methods to take the argument number parameters



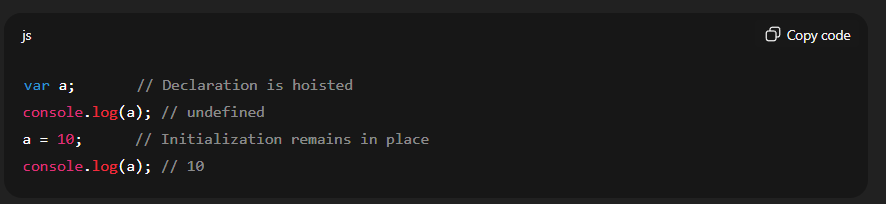
**🔹 What is Hoisting?**

Hoisting is **JavaScript’s default behavior of moving declarations to the top** of the current scope (either global or function scope) **before code execution**

🔹 Example 1 — Variable Hoisting with var

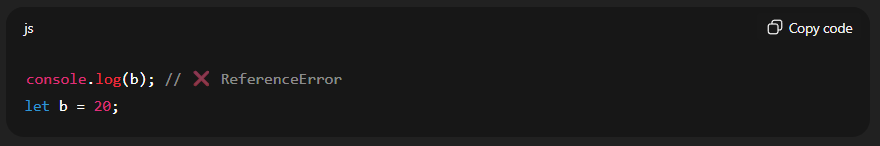


✅ **What happens behind the scenes:**  
JavaScript interpreter reads your code like this:



🧠 Only **declarations** are hoisted, not **initializations**.

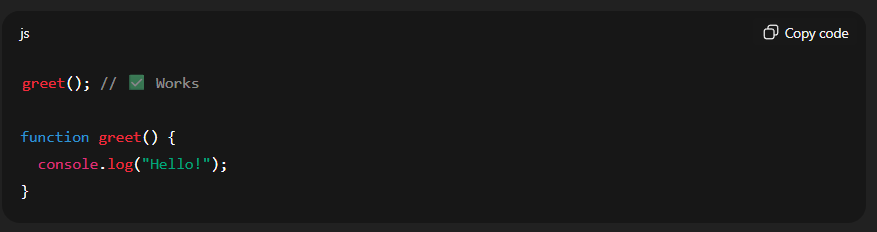
🔹 Example 2 — let and const



Variables declared with **let** and **const** are also hoisted,  
👉 but they stay in the **Temporal Dead Zone (TDZ)** until the line where they are initialized.

So they **cannot be accessed** before declaration.

🔹 Example 3 — Function Hoisting



✅ Function declarations are **fully hoisted**, meaning both their **name and body** are moved to the top.

**🔹 In Short (Summary Table)**

| **Type** | **Hoisted?** | **Initialized?** | **Accessible before declaration?** |
| --- | --- | --- | --- |
| var | ✅ Yes | ❌ No | ⚠️ Yes, gives undefined |
| let | ✅ Yes | ❌ No | ❌ No (TDZ error) |
| const | ✅ Yes | ❌ No | ❌ No (TDZ error) |
| Function Declaration | ✅ Yes | ✅ Yes | ✅ Yes |
|  |  |  |  |

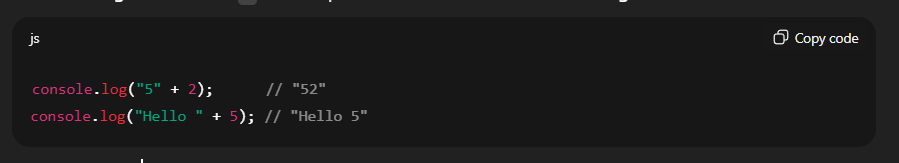
**⚖️ Summary Table**

| **Type** | **Examples** | **Stored In** | **Copied As** | **Affects Original?** |
| --- | --- | --- | --- | --- |
| **Primitive** | string, number, boolean, null, undefined, symbol, bigint | Stack | Value | ❌ No |
| **Reference** | object, array, function | Heap | Reference (pointer) | ✅ Yes |

**Type Conversion**

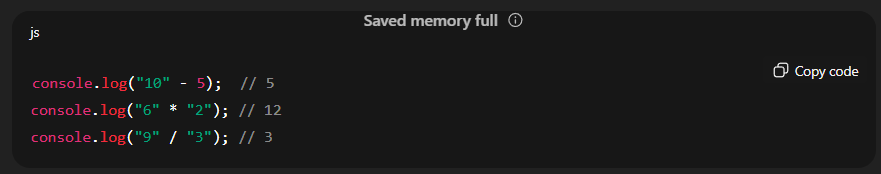
**🔹 1. String Conversion**

When a **string** is involved in +, JavaScript converts the other value to a **string**.



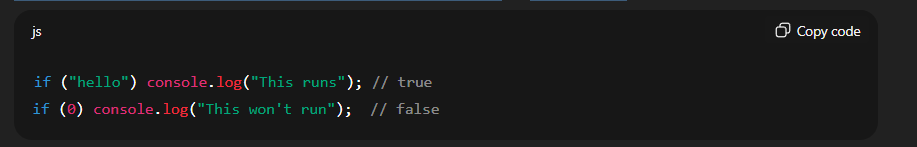
**🔹 2. Number Conversion**

When a mathematical operator (-, \*, /, %) is used,  
JavaScript converts operands to **numbers**.



**🔹 3. Boolean Conversion**

JavaScript converts values to **boolean** in logical contexts (like if conditions).



🧠 **Truthy values:** all values except 0, "", null, undefined, and NaN.  
🧠 **Falsy values:** 0, "", null, undefined, NaN, false.

**Function with default values**

const square = (n=6) =>

{

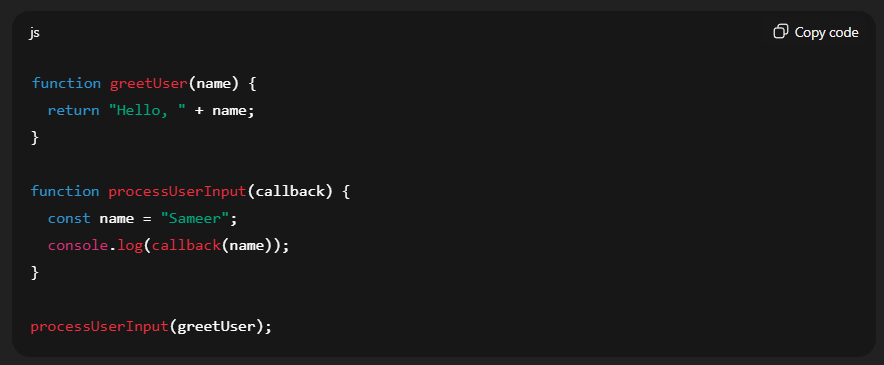
    return n \* n;

}

console.log(square()) // 36

**Higher Order Fuction:**

A Higher-Order Function (HOF) is a function that **takes another function as an argument**, **returns a function**, or **does both**.



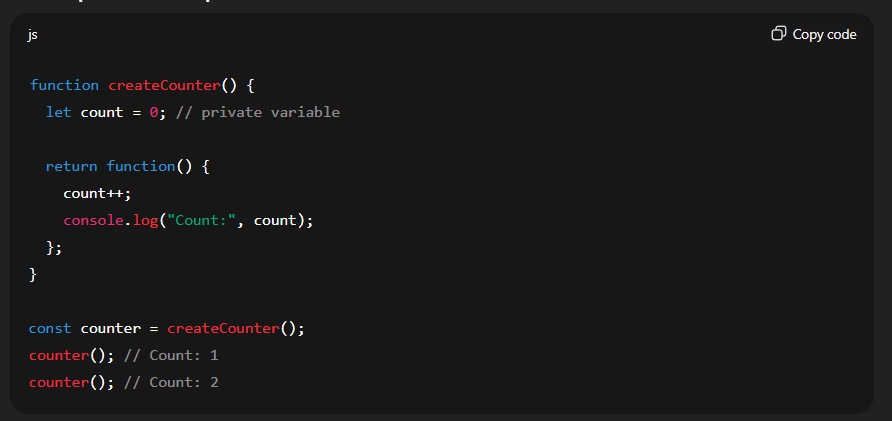
**🧩 Definition of Closure**

A **closure** in JavaScript is **a function that remembers variables from its outer scope**, even after that outer function has finished executing.

In other words —

A closure gives you **access to an outer function’s variables** from **inside an inner function**, even after the outer function has returned.

**Simple Real Example:**

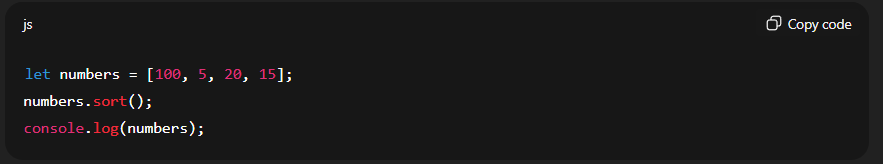


Here,

* count remains private inside createCounter()
* Every time you call counter(), it **remembers** the previous value of count.

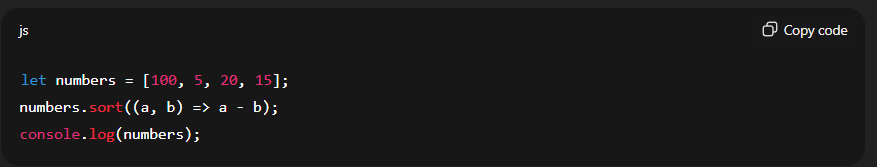
**Sort**

Example 1: Default Sorting (as strings)

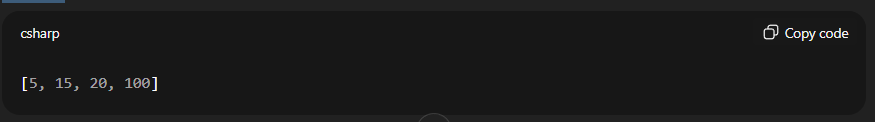
****

**🧩 Example 2: Numeric Sorting (Ascending)**

To sort numbers correctly, use a compare function:

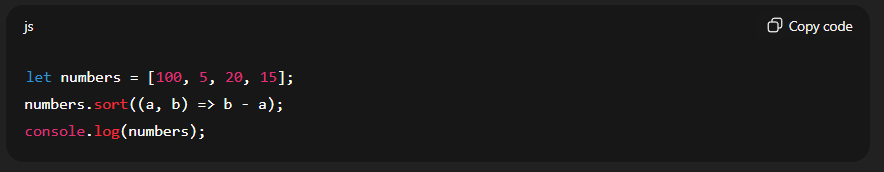
****

**Output:**

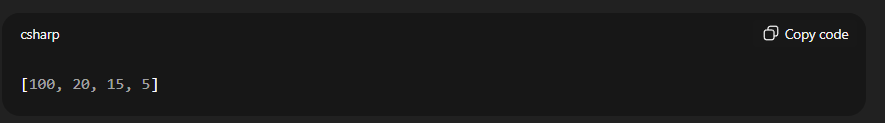
****

**Output:**

**🧩 Example 3: Numeric Sorting (Descending)**

****

**Output:**

****

**🔍 Step-by-Step Execution Example**

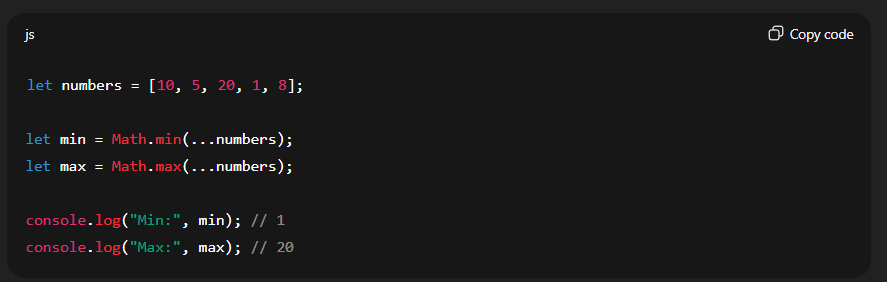
Array: [100, 5, 20, 15]

Compare function: (a, b) => a - b

Let’s see what happens:

| **Pair Compared** | **Calculation** | **Result** | **Action** |
| --- | --- | --- | --- |
| 100, 5 | 100 - 5 = 95 | > 0 | Swap (5 comes before 100) |
| 5, 20 | 5 - 20 = -15 | < 0 | Keep (5 stays before 20) |
| 20, 15 | 20 - 15 = 5 | > 0 | Swap (15 comes before 20) |

Final order after sorting → [5, 15, 20, 100]

**🧩 2. Using Math.min() and Math.max() with Spread Operator**

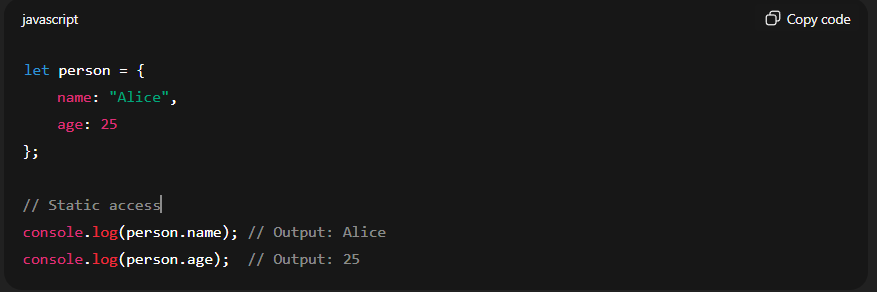
**Object Mystry:**

Can be accessed using dot(.) and [] operator

**All object keys are actually stored as strings internally**, even if you use a number.

**1️⃣ Static Access (Dot Notation)**

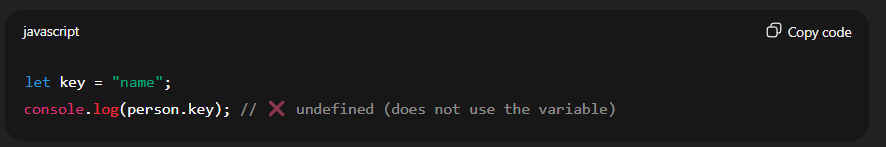
**Definition:**  
Static access uses **dot notation (.)** to access object properties when the **property name is known and fixed** at the time of writing the code.



**Key Points:**

1. Property name **must be a valid identifier** (no spaces or starting with a number).
2. **Cannot use variables** in dot notation.

**Invalid Example:**



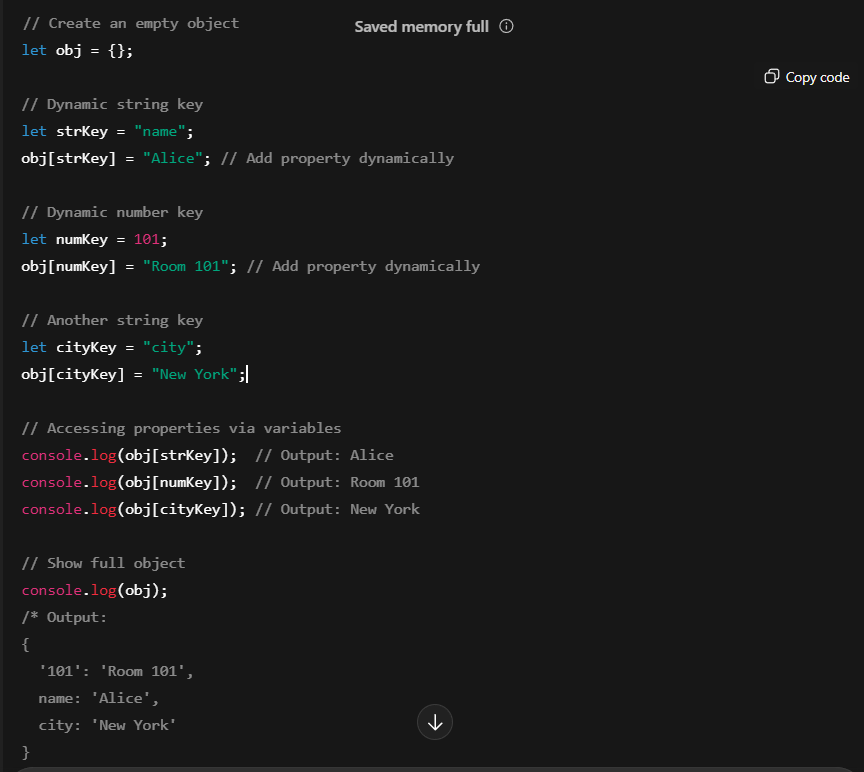
**Add like this**

obj.role = “Developer”

console.log(role) //Developer

**2️⃣ Dynamic Access (Bracket Notation)**

**Definition:**  
Dynamic access uses **bracket notation ([])**, which allows you to use **variables, strings, or expressions** as property names.

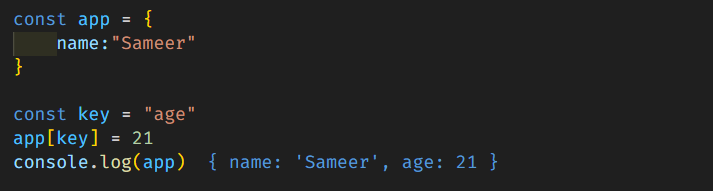


**✅ Key Points Illustrated:**

1. **Both string and number keys** can be added dynamically.
2. Numbers are automatically converted to **strings internally** (101 becomes '101').
3. Access via **variables** is easy using **bracket notation**.
4. Dot notation **cannot** be used for number keys or variable keys.

Note : static vs dynamic access

If user want to access the age from the object , so we can store age **as let key = “age”** and can access using the **obj[key]** that is **not possible in the static access**.

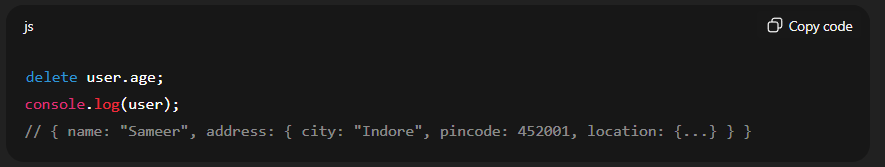


* Can set the value to the variable as considering it as a key

**Delete from object**

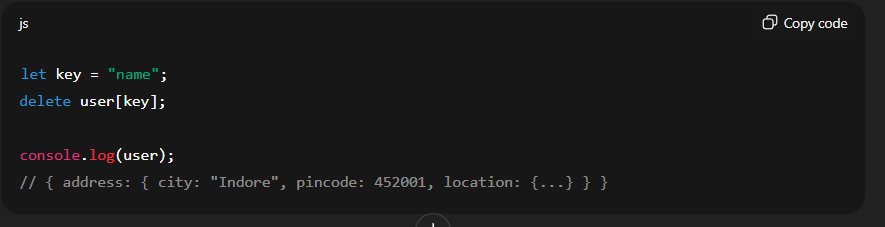
**✅ 1. Remove Static Property**

If you know the property name:

****

**✅ 2. Remove Property Dynamically (Single Level)**

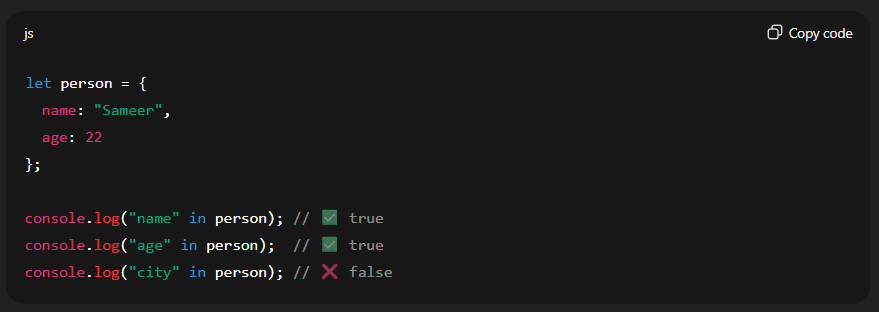
If you have the key stored in a variable:

****

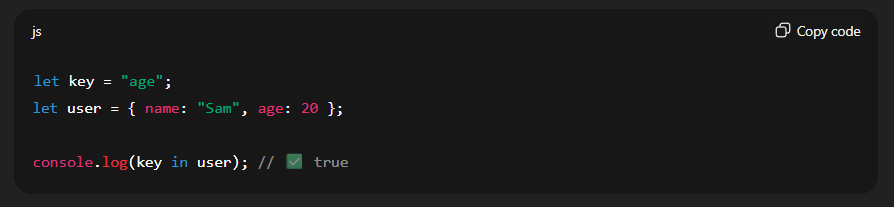
**In operator**

To check whether a key exists in object or not.

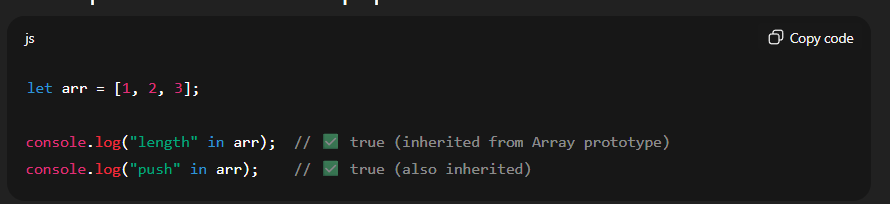
**🔹 Example 1: Basic Usage**



**🔹 Example 2: Dynamic key checking**

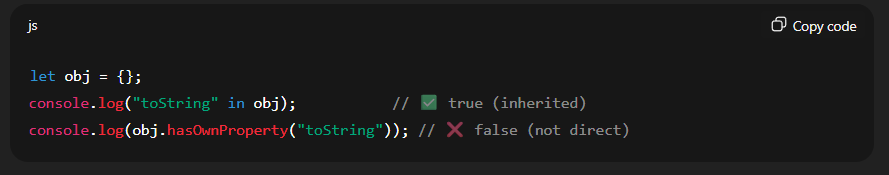
****

**🔹 Example 3: Works with inherited properties**

****

**✅ Summary**

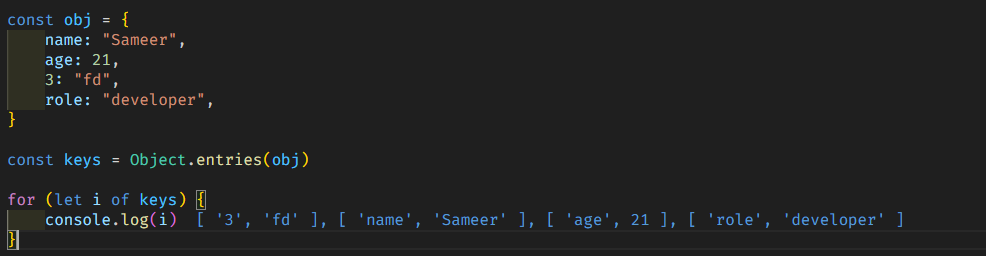
| **Purpose** | **Operator** |
| --- | --- |
| Check if property/key exists in object (including inherited) | in |
| Check only own property | .hasOwnProperty() |

****

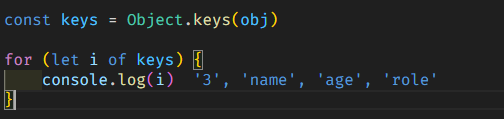
**Convert objects into arrays as key value**

* Object.entries()
* Object.keys()
* Object.values()

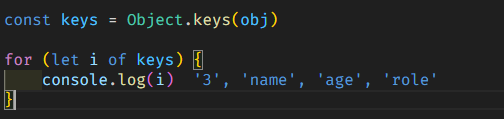
**Object.entries()**



**Object.keys() :** make array of all keys



**Object.values() :**  make array of all values

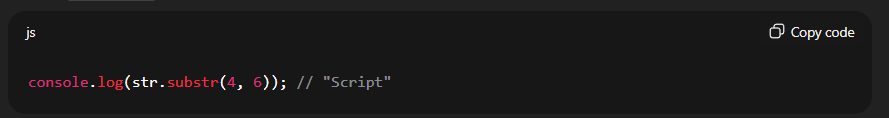


**Object Destructuring**

****

**Substr vs substring vs slice**

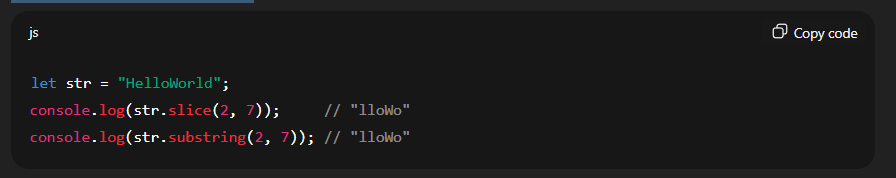
**🔹 substr()**

****

👉 Starts at index 4 and takes 6 characters

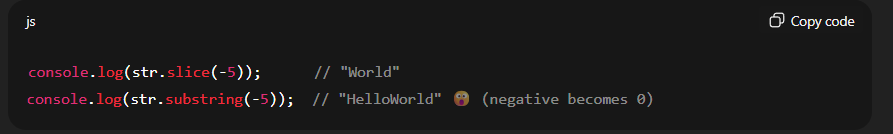
**🔹 Substring() and slice()**

**⚙️ Example 1: Normal Use**



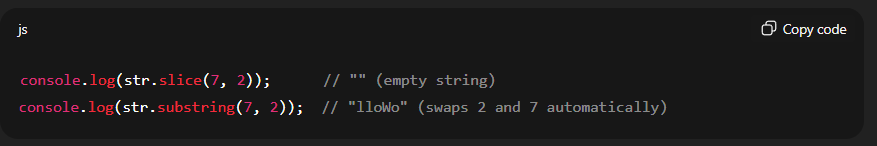
Both behave the same here.

**⚙️ Example 2: Negative Index**

****

substring() ignores negative numbers and treats them as 0.

**⚙️ Example 3: Reversed Arguments**



substring() automatically swaps arguments if start > end.

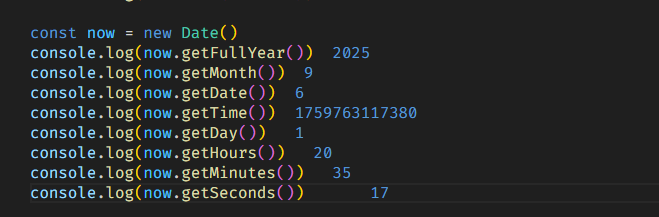
**🧩 Summary Comparison**

| **Feature** | **slice()** | **substring()** | **substr() (deprecated)** |
| --- | --- | --- | --- |
| 2nd Argument | End index | End index | Length |
| Negative Index | ✅ Yes | ❌ No | ✅ Yes |
| Swaps start/end if reversed | ❌ No | ✅ Yes | ❌ No |
| Recommended | ✅ Yes | ✅ Okay | ❌ No (deprecated) |

**✅ Recommendation**

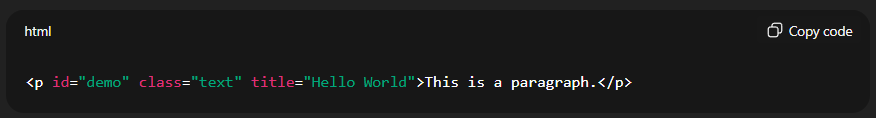
Use **slice()** when working with substrings in modern JavaScript.  
Use **substring()** only if you need automatic swapping of indexes.  
Avoid **substr()** — it’s deprecated.

**Date**



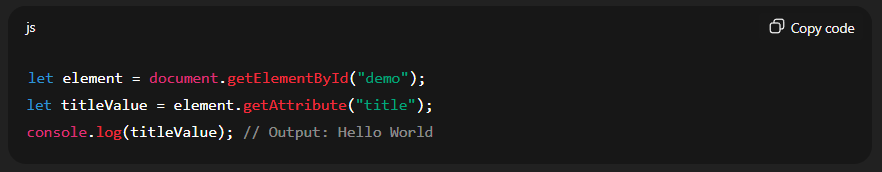
**Handling Attributes**

**🧩 Example HTML**



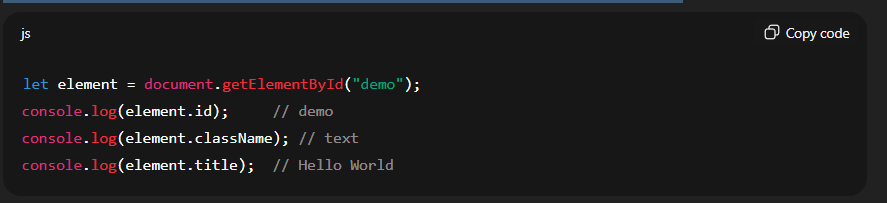
**✅ 1. Using getAttribute()**

The most common method.



**✅ 2. Direct Property Access**

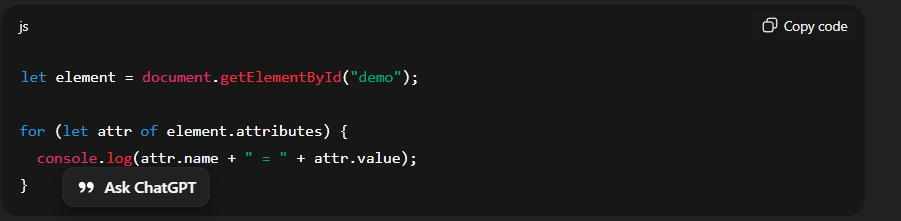
If the attribute is a **standard HTML attribute**, you can access it directly like a property.



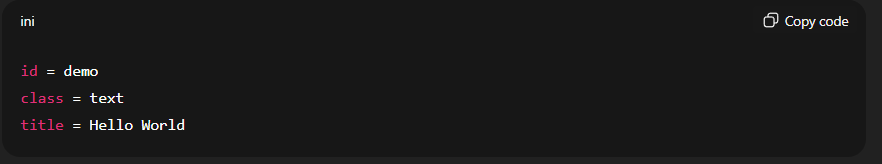
⚠️ Works only for known properties like id, className, src, href, title, etc.

**✅ 3. Access All Attributes**

You can access all attributes of an element using the attributes property.



Output:



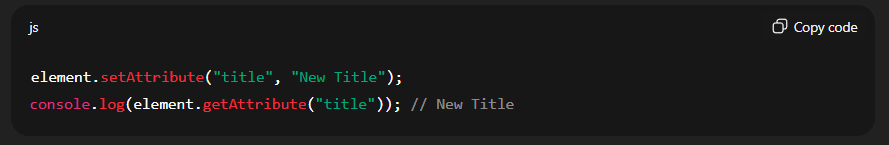
**✅ 4. For Custom Attributes (data-attributes)**

If you have custom attributes (like data-\*), you can access them using dataset.

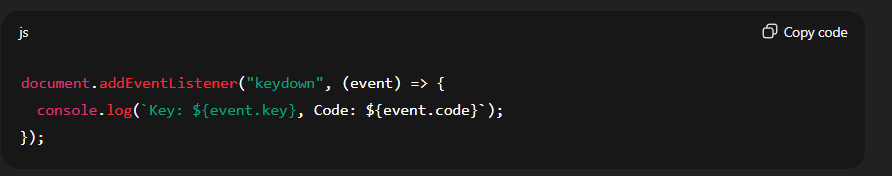


**✅ 6. Getting and Setting Together**

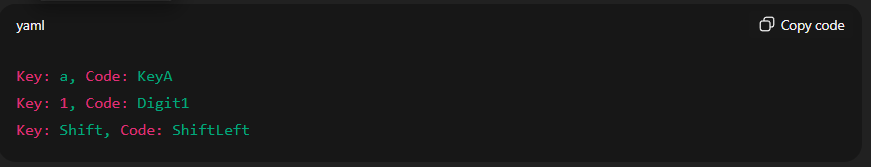
You can also modify attributes easily:



**Show Key + Code**



Output:

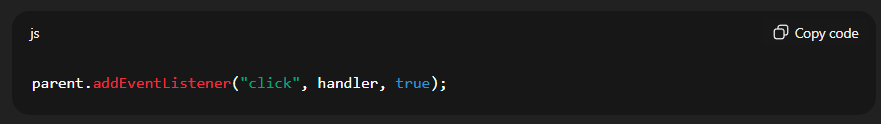


**Bubbling vs Capturing**

**🎯 1️⃣ Event Capturing Phase**

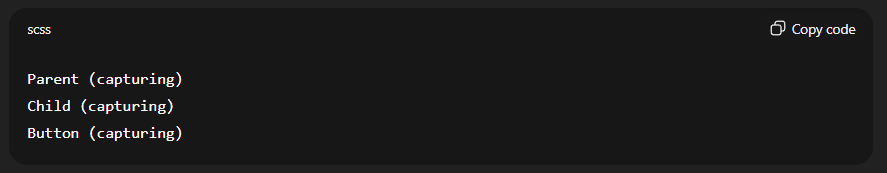
* The event starts from the **top of the DOM (window → document → parent → child → target)**.
* It **moves down** the tree **toward the element you clicked**.
* This is also called **trickling**.

If you add an event listener like this 👇



The true means → **listen during capturing**.  
So, the parent will catch the event **on its way down** before the button itself gets it.

🧠 **Order of execution when you click the button:**

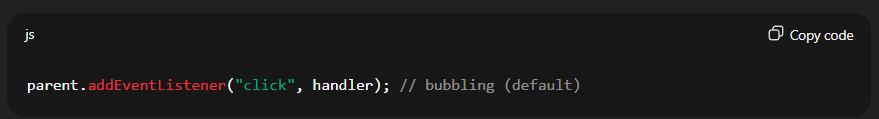


**💥 2️⃣ Target Phase**

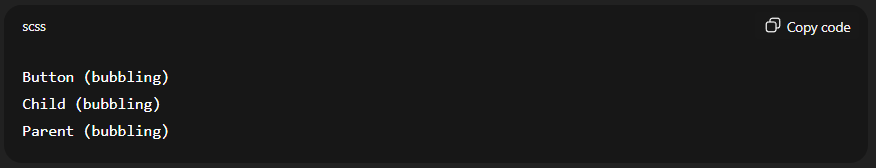
* The event reaches the **actual element** you clicked (<button> in this case).
* The event fires on the **target itself**.

**🌊 3️⃣ Event Bubbling Phase**

* After the target element handles the event,  
  it **bubbles up** from **child → parent → document → window**.



🧠 **Order of execution when you click the button:**



**⚖️ Difference Summary**

| **Feature** | **Capturing** | **Bubbling** |
| --- | --- | --- |
| Direction | Top → Bottom | Bottom → Top |
| Default behavior | ❌ Disabled | ✅ Enabled |
| Syntax | addEventListener(event, fn, true) | addEventListener(event, fn, false) |
| When triggered | Before target | After target |
| Common use | Intercept or prevent event early | Most general event handling |

**Example : (for both)**

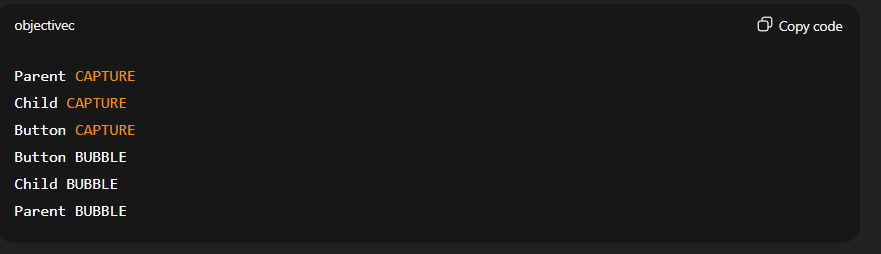
HTML :



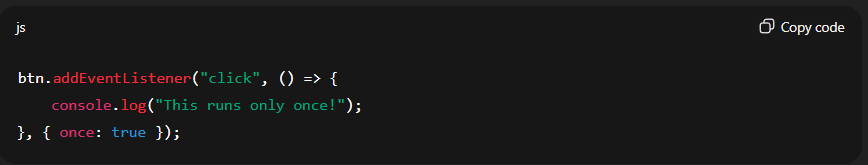
Js:



🧠 Output when clicking the button:



**Using One-Time Event Listener**

****

The listener automatically removes itself after executing once.

**🧩 What is Event Delegation?**

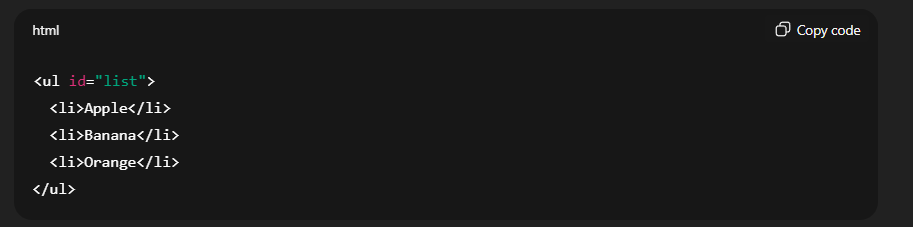
**Event Delegation** is a technique in JavaScript where you **attach a single event listener to a parent element** instead of adding separate listeners to each child element.

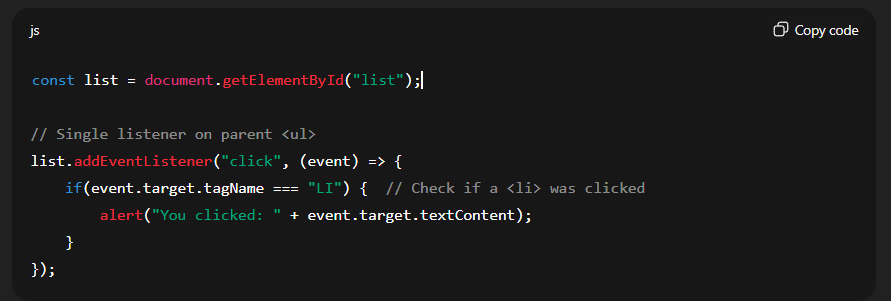
* The parent **“delegates”** the handling of events to its children.
* It works because of **event bubbling**: the event triggered on a child element bubbles up to the parent.

**🧠 How It Works**

1. **Event happens** on a child element.
2. **Event bubbles up** to the parent element.
3. The parent’s listener checks **which child triggered the event** using event.target.
4. Performs the action on the correct child element.

**🔹 Example**





✅ Here, only **one event listener** handles all <li> items.

**💡 Advantages of Event Delegation**

1. **Fewer event listeners** → better performance.
2. **Works for dynamic elements** added later.
3. **Cleaner code** → no repetitive listeners on each child.

Try catch methods:

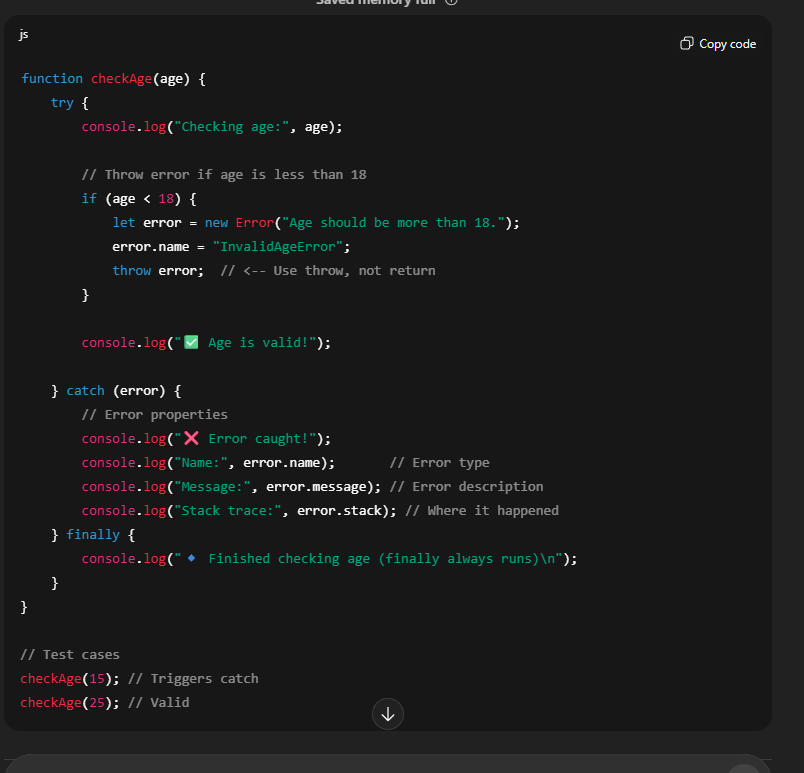
**🧱 3️⃣ Common Error Properties**

Inside the catch(error) block, the **Error object** provides useful properties:

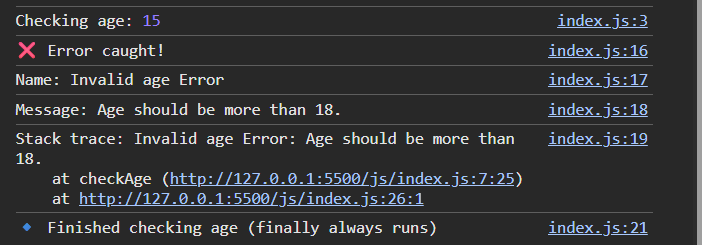
| **Property** | **Description** | **Example** |
| --- | --- | --- |
| error.name | Type of the error | "ReferenceError", "TypeError", "SyntaxError" |
| error.message | Description of the error | "x is not defined" |
| error.stack | Call stack trace | Shows where the error occurred |

**✅ Summary:**

| **Block** | **Purpose** |
| --- | --- |
| try | Wrap risky code |
| catch(error) | Handle the error |
| finally | Run cleanup code |
| throw | Manually generate an error |
| error.name | Error type |
| error.message | Error description |
| error.stack | Where the error happened |
| **Example:** |  |



**Output:**

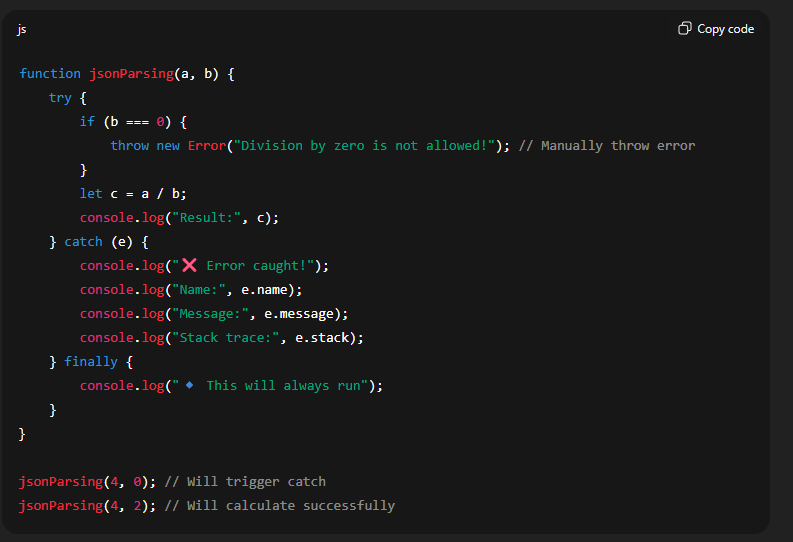


**Check Json parsing:**

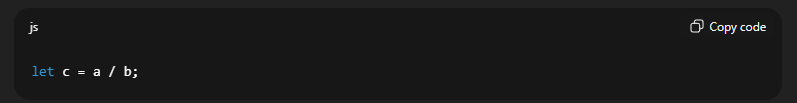


* + Usually an syntax error

**Q. Handle the division by zero error.**

****

* Used additionally if check for the number is that is 0, because **unlike java, js does not throw error on dividing by zero**.



 Dividing by 0 in JavaScript **does NOT throw an error**.

 Instead, it returns Infinity (or -Infinity if negative).

 So the catch block **won’t run** here.

**Nested try-catch**

 **Inner try-catch** handles a **specific error** (JSON parsing).

 **Outer try-catch** handles **any other unexpected errors**.

**Async/ Await:**

To use the aysnc await you will have to use the promises unless using the promises you can use the settimeout but cannot use the aysnc await.

🗣️ **setTimeout() works without Promises**,  
but **await doesn’t wait** unless you return a Promise.

**🧠 Final Summary**

* setTimeout() alone doesn’t block or delay code — it **schedules a callback**.
* await only pauses on **Promises**.
* To make setTimeout() work with await, wrap it in a Promise.

**⚙️ 1. First key idea:**

setTimeout() and async/await are **two completely different systems** for handling asynchrony.

| **Feature** | **setTimeout()** | **async/await** |
| --- | --- | --- |
| Based on | Callback mechanism | Promise mechanism |
| Returns | undefined | Promise |
| JS waits for it? | ❌ No | ✅ Yes (if Promise is returned) |
| Managed by | Browser / Node timer API | JavaScript engine’s async flow |

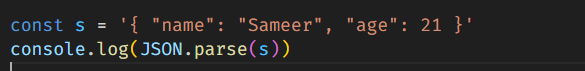
**🧩 Summary**

| **Case** | **setTimeout()** | **Promise?** | **await waits?** | **Works as expected?** |
| --- | --- | --- | --- | --- |
| Just setTimeout() | ✅ Yes | ❌ No | ❌ No | No (async flow breaks) |
| setTimeout() + Promise | ✅ Yes | ✅ Yes | ✅ Yes | ✅ Perfect |
|  |  |  |  |  |
|  |  |  |  |  |

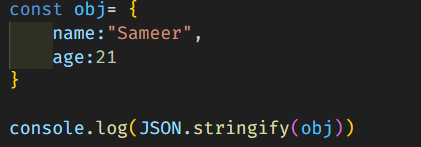
**🧩 One-line answer**

async and await don’t work with plain setTimeout() because setTimeout() returns a timer ID, not a Promise — and await only pauses for Promises.

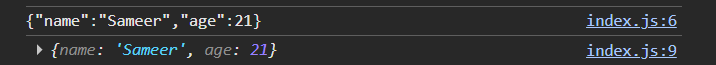
**String to Json**

****

**Json to string**

****

**Output:**

****

**⚙️ 3️⃣ Summary Table**

| **Function** | **Direction** | **Converts** | **Example** |
| --- | --- | --- | --- |
| JSON.stringify() | Object → String | JS Object → JSON text | JSON.stringify({name:"Sameer"}) |
| JSON.parse() | String → Object | JSON text → JS Object | JSON.parse('{"name":"Sameer"}') |

**setItem:**

* Accept: accept the number/Boolean but all stored as string

To store object use JSON.stringify(obj) otherwise stored as [Object Object]

* Return: always string

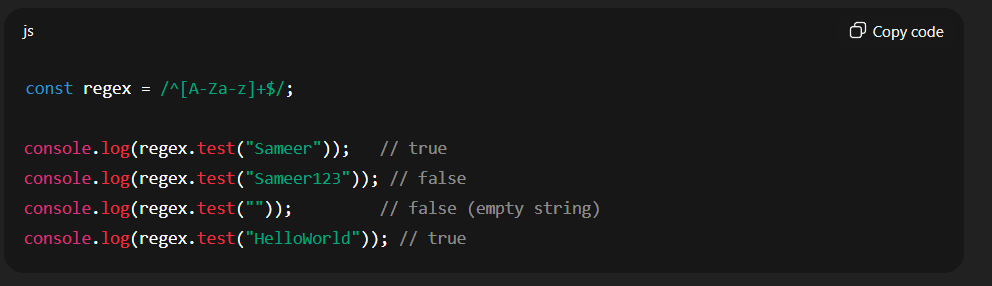
**getItem:**

* Return : always string
* To convert json use JSON.parse()

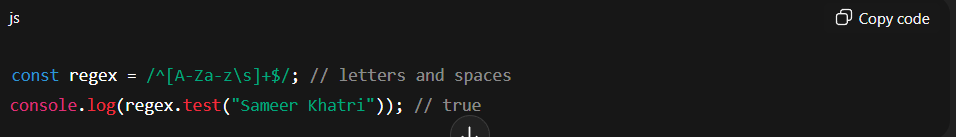
**Regex:**

To check the pattern of the string what it contains

1. contains only letters:



Allow spaces



2. contains only number

const regex = /^[0-9]+$/