JAVASCRIPT

VARIABLES

1. JavaScript var keyword

var is a keyword in JavaScript used to declare variables and it is Function-scoped and hoisted, allowing redeclaration but can lead to unexpected bugs.

```
var a = "Hello Geeks";
var b = 10;
console.log(a);
console.log(b);
```

2. JavaScript let keyword

let is a keyword in JavaScript used to declare variables and it is Block-scoped and not hoisted to the top, suitable for mutable variables

```
let a = 12
let b = "gfg";
console.log(a);
console.log(b);
```

3. JavaScript const keyword

const is a keyword in JavaScript used to declare variables and it is Block-scoped, immutable bindings that can't be reassigned, though objects can still be mutated.

```
const a = 5
let b = "gfg";
console.log(a);
console.log(b);
```

Rules for Naming Variables

When naming variables in JavaScript, follow these rules

- Variable names must begin with a letter, underscore (_), or dollar sign ($).
- Subsequent characters can be letters, numbers, underscores, or dollar signs.
- Variable names are case-sensitive (e.g., age and Age are different variables).
- Reserved keywords (like function, class, return, etc.) cannot be used as variable names.

Variable Shadowing in JavaScript

Variable shadowing occurs when a variable declared within a certain scope (e.g., a function or block) has the **same name** as a variable in an outer scope. The inner variable **overrides** the outer variable within its scope.

```
let n = 10; // Global scope

function gfg() {
    let n = 20;  // Shadows the global 'n' inside this function
    console.log(n);  // Output: 20
}

gfg();
console.log(n);  // Output: 10 (global 'n' remains unchanged)
```

2. Function Scope

Variables declared inside a function are accessible only within that function. This applies to var, let, and const:

```
function test() {
    var localVar = "I am local";
    let localLet = "I am also local";
    const localConst = "I am local too";
}
console.log(localVar); // Error: not defined
```

3. Block Scope

Variables declared with let or const inside a block (e.g., inside {}) are block-scoped, meaning they cannot be accessed outside the block. var, however, is not block-scoped and will leak outside the block.

```
{
    let blockVar = "I am block-scoped";
    const blockConst = "I am block-scoped too";
}
console.log(blockVar); // Error: not defined
```

Feature	`var`	`let`	`const`
Scope	Function-scoped	Block-scoped	Block-scoped
Hoisting	Hoisted (initialized as `undefined`)	Hoisted (but not initialized)	Hoisted (but not initialized)
Can be Reassigned?	✅ Yes	✅ Yes	❌ No (assignment is constant)
Can be Redeclared?	✅ Yes	❌ No	❌ No
Temporal Dead Zone	❌ No	✅ Yes	✅ Yes
Global Object Property?	✅ Yes (in non-strict mode)	❌ No	❌ No
Use in Loops (safe)?	❌ Problematic in closures	✅ Safer in closures	✅ Safer in closures (if no reassignment)

CURRYING

Currying is the process of transforming a function that takes multiple arguments into a series of functions that each take one argument at a time.

- EXAMPLE :-

```
function add(a, b) {
  return a + b;
}
add(2, 3);
```

- 🔁 With Currying:

```
function add(a) {
  return function(b) {
    return a + b;
  };
}
add(2)(3); // 👉 5
```

The curried version lets you call the function one argument at a time. 📦 Why Use Currying?

1. Reusability: You can create specialized versions of functions.
2. Composition: Works well with functional pipelines.
3. Partial Application: Fix one or more arguments and reuse.

```
const multiply = a => b => a \* b;
```

CLOSURE

Closure tab banta hai jab ek inner function, apne outer function ke variables ko yaad rakhta hai even after outer function execution is complete.

👉 JavaScript mein functions ke andar function bana sakte ho, aur inner function outer function ke scope ke variables ko access kar leta hai — yeh hi closure hai.

🔍 Example:

1. Simple closure

```
function outer() {
let name = "Amit"; // outer variable

function inner() {
    console.log("Hello " + name); // inner can access outer's variable
}

return inner;
}

const greet = outer(); // outer() run hua, inner function return hua
greet(); // ✅ Output: Hello Amit
```

2. Data Privacy using Closure:

```
function secretNumber() {
let number = 42; // private

return {
getNumber: function () {
return number;
},
setNumber: function (newNum) {
number = newNum;
}
};
}

const secret = secretNumber();
console.log(secret.getNumber()); // 42
secret.setNumber(100);
console.log(secret.getNumber()); // 100
```

3. Counter Function (Classic Closure Example)

```
function createCounter() {
let count = 0; // private variable

return function () {
count++;
console.log(count);
};
}

const counter = createCounter();

counter(); // 1
counter(); // 2
counter(); // 3
```

FUNCTION

1. First-Class Functions

In JavaScript, functions are first-class citizens, meaning they can:

- Be stored in variables
- Be passed as arguments to other functions
- Be returned from other functions

```
📌 Example:
function greet() {
  return "Hello!";
}

// Storing in a variable
let sayHello = greet;

// Passing as argument
function callFunction(fn) {
  console.log(fn());
}
callFunction(sayHello); // Output: Hello!
```

2. Function Expression

A function expression is when a function is assigned to a variable. It can be named or anonymous.

```
(Anonymous Function Expression):

const add = function(a, b) {
  return a + b;
};
```

```
(Named Function Expression):

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

3. Function Statement (a.k.a. Function Declaration)

A function statement defines a function with the function keyword, followed by a name.

```
function subtract(a, b) {
  return a - b;
}
```

🔷 Key difference from expression: Function declarations are hoisted, meaning they can be used before they are defined in the code.

4. Anonymous Function

An anonymous function is a function without a name. It's often used in function expressions or as arguments to other functions (like callbacks).

```
setTimeout(function() {
  console.log("This is an anonymous function");
}, 1000);
```

Term	When it's used	Example in code
Parameter	When defining the function	function greet(name)
Argument	When calling the function	greet("Alice")

👉 **Parameters** are like empty boxes □.
👉 **Arguments** are the toys □ you put in them.

Callback Function

What is a Callback Function?

A callback function is a function that is passed as an argument to another function and executed later.

- A function can accept another function as a parameter.
- Callbacks allow one function to call another at a later time.
- A callback function can execute after another function has finished.

How Do Callbacks Work in JavaScript?

JavaScript executes code line by line (synchronously), but sometimes we need to delay execution or wait for a task to complete before running the next function. Callbacks help achieve this by passing a function that is executed later.

```
Callbacks for Asynchronous Execution
console.log("Start");
setTimeout(function () {
    console.log("Inside setTimeout");
}, 2000);
console.log("End");
```

Callbacks are widely used in

- API requests (fetching data)
- Reading files (Node.js file system)
- Event listeners (clicks, keyboard inputs)
- Database queries (retrieving data)
- Callbacks in Event Listeners
    - JavaScript is event-driven, and callbacks handle user interactions like clicks and key presses.

example of EventListener + closure +callback


```
document.getElementById("myButton")
.addEventListener("click", function () {
    console.log("Button clicked!");
});
```

Callbacks in API Calls (Fetching Data)

Callbacks are useful when retrieving data from APIs.

```
function fetch(callback) {
    fetch("https://jsonplaceholder.typicode.com/todos/1")
        .then(response => response.json())
        .then(data => callback(data))
        .catch(error => console.error("Error:", error));
}

function handle(data) {
    console.log("Fetched Data:", data);
}

fetch(handle);
```

MAP / FILTER / REDUCE

Method	What it does	Example Use	Output Type
`filter`	**Keeps** items that match	Keep even numbers	New **array**
`map`	**Changes** each item	Double each number	New **array**
`reduce`	**Combines** all into one	Add all numbers together	**Single value**

Example:

For `[1, 2, 3, 4]`:

Method	Code	Result
filter	`[1,2,3,4].filter(n => n % 2 == 0)`	`[2, 4]`
map	`[1,2,3,4].map(n => n * 2)`	`[2, 4, 6, 8]`
reduce	`[1,2,3,4].reduce((a,b) => a + b)`	`10`

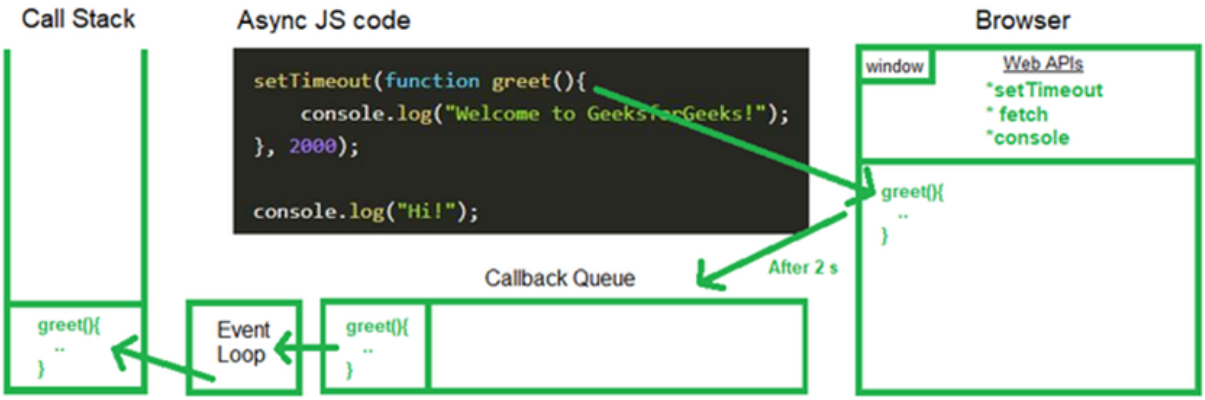
```
var personnel = [
  {
    id: 5,
    name: "Luke Skywalker",
    pilotingScore: 98,
    shootingScore: 56,
    isForceUser: true,
  },
```

```
    {
      id: 82,
      name: "Sabine Wren",
      pilotingScore: 73,
      shootingScore: 99,
      isForceUser: false,
    },
    {
      id: 11,
      name: "Caleb Dume",
      pilotingScore: 71,
      shootingScore: 85,
      isForceUser: true,
    },
  ];

  Our objective: get the total score of force users only
```

```
const totalForceUserScore = personnel
  .filter(person => person.isForceUser)
  .map(person => person.pilotingScore + person.shootingScore)
  .reduce((acc, score) => acc + score, 0);

console.log(totalForceUserScore); // Output: 420
```

Difference Between Microtask Queue and Callback Queue in asynchronous JavaScript

Sometimes this callback queue is also referred as Task Queue or Macrotask queue.



Microtask Queue

Microtask Queue is like the Callback Queue, but Microtask Queue has **higher priority**.

Callback Queue	Microtask Queue
Callback Queue gets the ordinary callback functions coming from the *setTimeout()* API after the timer expires.	Microtask Queue gets the callback functions coming through Promises and Mutation Observer.
Callback Queue has lesser priority than Microtask Queue of fetching the callback functions to Event Loop.	Microtask Queue has higher priority than Callback Queue of fetching the callback functions to Event Loop.

Microtask Queue Examples

Code Snippet	Description
`Promise.resolve().then(...)`	Runs `.then()` callback as microtask
`queueMicrotask(() => {...})`	Explicitly adds to microtask queue
`async function` + `await`	`await` pauses, then resumes as microtask
`MutationObserver`	DOM changes callback runs as microtask

SETTIMEOUT VS SETINTERVAL

Feature	`setTimeout`	`setInterval`
Purpose	Runs **once** after a delay	Repeats execution at **regular intervals**
Runs After	A specified time (one-time)	Every specified time (looping)
Syntax	`setTimeout(fn, delay)`	`setInterval(fn, delay)`
Stops Automatically	✅ Yes	❌ No (must stop manually)
Use Case Example	Show message after 3 seconds	Clock ticking every second
Cancel Function	`clearTimeout(timeoutID)`	`clearInterval(intervalID)`

Callback Hell (Pyramid of Doom)

When multiple asynchronous operations depend on each other, callbacks get deeply nested, making the code hard to read and maintain.

```
getUser(userId, (user) => {
    getOrders(user, (orders) => {
        processOrders(orders, (processed) => {
            sendEmail(processed, (confirmation) => {
                console.log("Order Processed:", confirmation);
            });
        });
    });
});
```

🔴 Demerit	📝 Explanation
❌ Hard to Read	Deep nesting and indentation (a "pyramid shape") makes code confusing
🐞 Difficult to Debug	Hard to trace the flow and find where an error occurs in nested callbacks
🔁 Poor Maintainability	Adding, changing, or removing logic is risky and error-prone
📦 Tight Coupling	Each callback depends on the previous one — changing one may break the chain
🔄 Callback Inversion	Logic runs in reverse order — what happens later is written first (inverted logic)
🚫 No Clean Error Handling	Each callback needs its own `try/catch`; no centralized way to handle errors

✅ How to Overcome Callback Hell

1. Use Promises

js Copy Edit

```js
doSomething()
  .then(result1 => doNext(result1))
  .then(result2 => doAnother(result2))
  .then(result3 => finish(result3))
  .then(result4 => console.log("All done!"))
  .catch(error => console.error(error));
```

2. Use `async/await`

js Copy Edit

```js
async function runTasks() {
  try {
    const result1 = await doSomething();
    const result2 = await doNext(result1);
    const result3 = await doAnother(result2);
    const result4 = await finish(result3);
    console.log("All done!");
  } catch (error) {
    console.error(error);
  }
}
```

PROMISES

A Promise is an object representing the eventual completion (or failure) of an asynchronous operation.

main.js

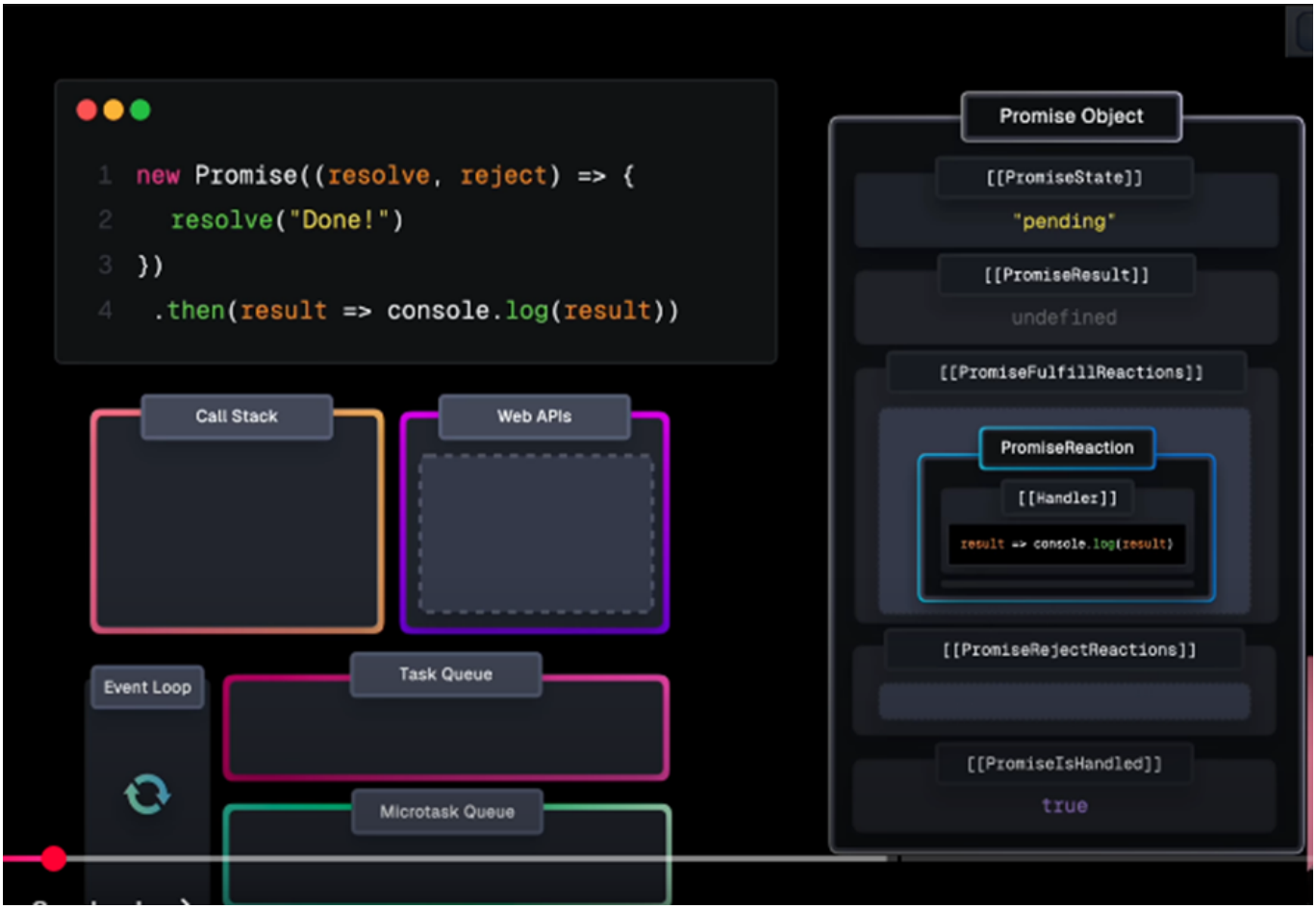
Share Run

```
function getData(success = true) {
  return new Promise((resolve, reject) => {
    if (!success) {
      return reject("Error: Manual failure triggered");
    }

    fetch("https://api.github.com/users/KANT-LAXMI")
      .then(response => {
        if (!response.ok) {
          throw new Error(`HTTP error! Status: ${response.status}`);
        }
        return response.json();
      })
      .then(data => resolve(data))
      .catch(err => reject(`Fetch failed: ${err.message}`));
  });
}

// ✅ Case 1: Successful fetch
getData(true)
  .then(data => {
    console.log("Data received:", data);
  })
  .catch(err => {
    console.error(err);
  });

```

main.js

Share Run

```js
function getData(success = true) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (success) {
        resolve("Data received");
      } else {
        reject("Error: Data not found");
      }
    }, 2000);
  });
}
// Case 1: Successful data fetch
getData(true)
  .then(data => {
    console.log(data); // "Data received"
  })
  .catch(err => {
    console.error(err);
  });

```



ASYNC AWAIT

async/await is syntax sugar over Promises, introduced in ES2017, to make asynchronous code look and behave more like synchronous code.

Why async/await was introduced:

- Promises made async code more manageable than callbacks.
- But .then() chains can still become messy for multiple sequential async operations.
- async/await makes it cleaner and easier to write and read.

main.js Share Run

```
async function getData(success = true) {
  if (!success) {
    throw new Error("Error: Manual failure triggered");
  }

  try {
    const response = await fetch("https://api.github.com/users/KANT-LAXMI");

    if (!response.ok) {
      throw new Error(`HTTP error! Status: ${response.status}`);
    }

    const data = await response.json();
    return data;

  } catch (err) {
    throw new Error(`Fetch failed: ${err.message}`);
  }
}

// ✅ Case 1: Successful fetch
getData(true)
  .then(data => {
    console.log("Data received:", data);
  })
  .catch(err => {
    console.error(err.message);  // Note: we access err.message here
  });

```

🧠 Key Differences:

Feature	Promises	`async/await`
Syntax	`.then()` and `.catch()` to handle results and errors.	Use `await` to wait for a Promise to resolve inside an `async` function.
Readability	More complex; requires chaining `.then()` and `.catch()`.	Cleaner and more like synchronous code.
Error Handling	`.catch()` for errors.	`try/catch` for errors, making it look like regular code.
Return Value	Returns a Promise.	Returns a Promise, but you write code that looks synchronous.
Use Case	Good for chaining multiple async actions.	Good for simplifying complex async code.

STRICT MODE

use strict is a special directive in JavaScript that enables strict mode, making the code safer and cleaner by catching silent errors and preventing bad practices.

✅ How to Enable Strict Mode

```
Place this line at the top of your script or function:
"use strict";
Example:"use strict";
let x = 10;
It must be the first line — or it won't work!
```

⚠️ What Does Strict Mode Do?

Behavior	Without "use strict"	With "use strict"
❌ Undeclared variables	Allowed (auto-globals)	❌ Throws an error
❌ Assign to read-only props	Fails silently	❌ Throws an error
❌ Duplicated parameter names	Allowed	❌ Throws an error
❌ this in functions	Refers to window (or global)	❌ undefined if not bound
❌ Deletes non-deletables	Fails silently	❌ Throws an error

1. **Prevents the use of undeclared variables:**
   - Without strict mode, you can accidentally create global variables (even if they are not declared).
   - In strict mode, this will throw an error.

```js
"use strict";
x = 10;  // ❌ ReferenceError: x is not defined
```

2. **Disallows duplicate parameters** in functions:
   - In non-strict mode, you can have functions with **duplicate parameter names**. In strict mode, this will throw an error.

```js
"use strict";
function example(a, a) {  // ❌ SyntaxError: Duplicate parameter name not allowed in this con
  return a;
}
```

4. **Eliminates `this` coercion:**
   - In non-strict mode, if you call a function without a specific context (i.e., not as an object), the `this` keyword points to the **global object** (in browsers, `window`).
   - In strict mode, `this` will be `undefined` if no explicit context is provided.

```js
"use strict";
function showThis() {
  console.log(this);  // ❌ undefined (instead of global object)
}
showThis();
```

5. **Prevents `with` statements:**
   - The `with` statement, which allows you to extend the scope of an object, is **disallowed** in strict mode because it makes code harder to optimize and understand.

```js
"use strict";
with (Math) {  // ❌ SyntaxError: Strict mode code may not include a with statement
  console.log(sqrt(16));
}
```

6. **Makes `eval` safer:**
   - In strict mode, `eval` behaves differently: it doesn't create new variables in the surrounding scope.

```js
"use strict";
eval("var x = 10");
console.log(x);  // ❌ ReferenceError: x is not defined
```

7. **Disallows `delete` of variables:**
   - You cannot delete a variable, function, or function argument in strict mode.

```js
"use strict";
var a = 10;
delete a;  // ❌ SyntaxError: Delete of an unqualified identifier in strict mode
```

📋 Summary Table

Behavior	Without `"use strict"`	With `"use strict"`
🔴 Undeclared Variables	Allowed (automatic global)	❌ Throws an error
🔴 Duplicate Function Parameters	Allowed	❌ Throws an error
🔴 Assignment to Read-only Properties	Allowed	❌ Throws an error
🔴 `this` in Global Functions	Refers to global object (`window`)	`this` is `undefined`
🔴 Use of `with` Statement	Allowed	❌ Throws an error
🔴 Global `eval` Variables	Allowed	Creates new variables only inside `eval()`
🔴 Delete Variables	Allowed	❌ Throws an error

IIFE (Immediately Invoked Function Expression)?

An IIFE is a function in JavaScript that is defined and executed immediately. The key idea behind an IIFE is that it runs right after it's created, and is often used to create a local scope for variables to avoid polluting the global scope.

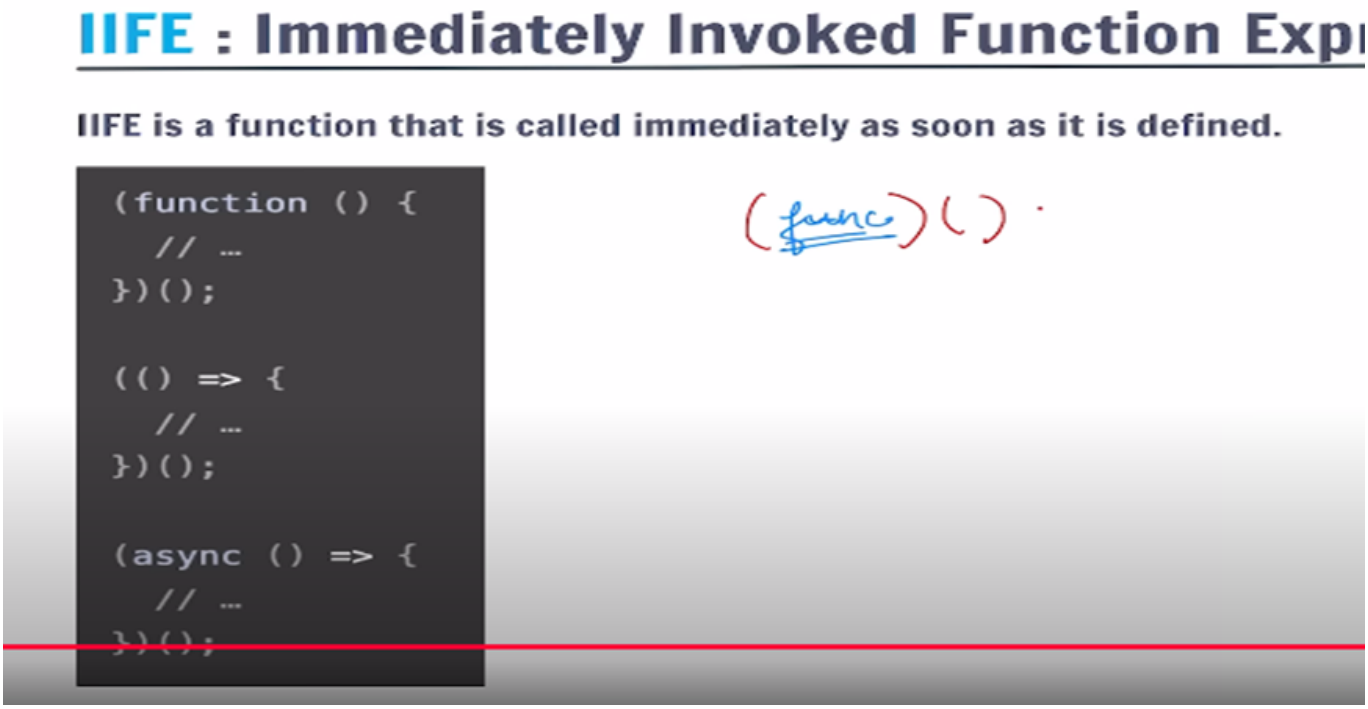
- IIFE Syntax

```
(function() {
// code inside the function
console.log("This is an IIFE!");
})();
```

- 🧩 Breaking It Down:

1.
```
(function() { ... }): The function is wrapped in parentheses to treat it as
an expression, not a declaration.
```

2. () after the parentheses: This immediately invokes the function.



1. Creating Private Variables (Module Pattern):

```js
const counter = (function() {
  let count = 0;  // This variable is private to the IIFE

  return {
    increment: function() {
      count++;
      console.log(count);
    },
    decrement: function() {
      count--;
      console.log(count);
    },
    getCount: function() {
      return count;
    }
  };
})();

counter.increment();  // 1
counter.increment();  // 2
console.log(counter.getCount());  // 2
```

2. Passing Arguments to an IIFE:

```js
(function(a, b) {
  console.log(a + b);  // 5
})(2, 3);
```

This example demonstrates how you can **pass parameters** to an IIFE when invoking it.

3. Using IIFE in Loops (Avoiding Global Scope):

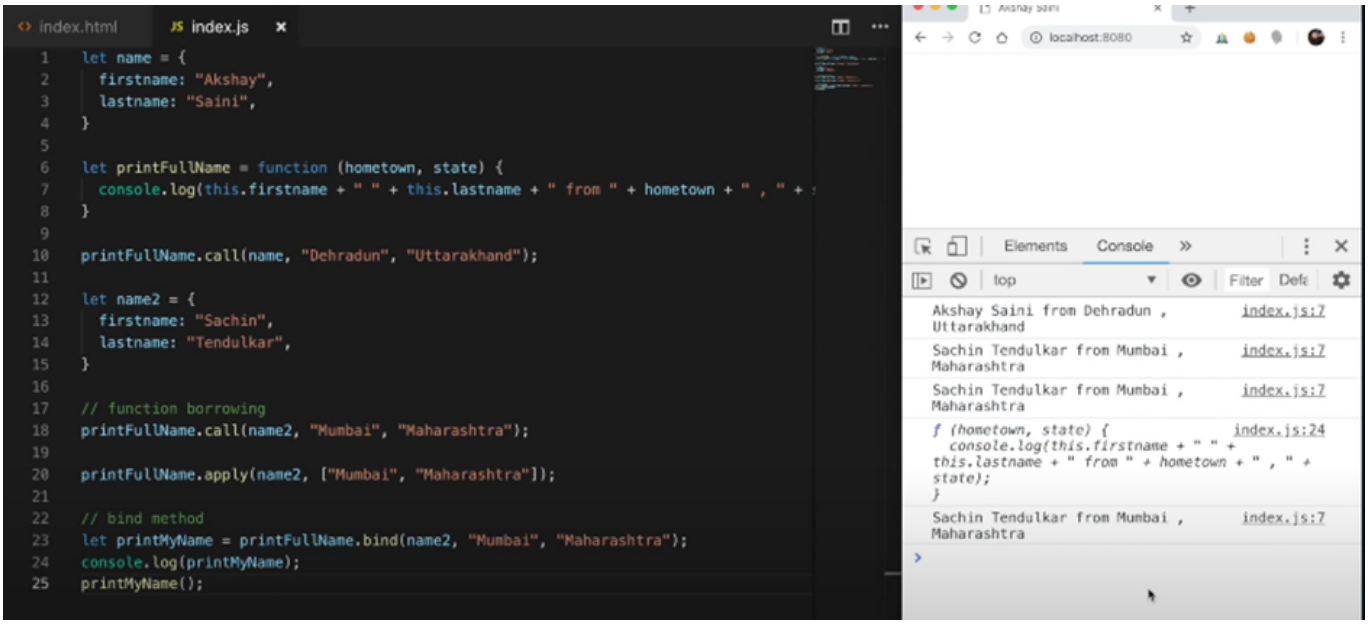
In older JavaScript versions, IIFEs were commonly used inside loops to **capture the current value** of a loop variable.

```js
for (var i = 0; i < 3; i++) {
  (function(i) {
    setTimeout(function() {
      console.log(i);  // Outputs: 0, 1, 2 (not 3, 3, 3 as you might expect with regular var)
    }, 1000);
  })(i);
}
```

Without the IIFE, `i` would have been shared across all iterations, and after the `setTimeout` execution, `i` would have been `3` in all the console logs. The IIFE creates a **new scope for each iteration**, so each callback sees the correct value of `i`.

CALL APPLY BIND

1. call() - Immediate Invocation with Individual Arguments
    - The call() method calls a function immediately and allows you to specify the value of this and pass individual arguments to the function.
2. apply() - Immediate Invocation with Arguments in an Array
    - The apply() method is similar to call(), but instead of passing individual arguments, we pass them as an array.
3. bind() - Returns a New Function with Fixed this (Not Invoked Immediately)
    - The bind() method returns a new function with a fixed this value, but it does not invoke the function immediately. You can call this new function later.



📊 Comparison Table for Call, Apply, and Bind:

Method	Invocation	Arguments	Returns
`call()`	Immediate	Pass arguments individually	Function result
`apply()`	Immediate	Pass arguments as an array	Function result
`bind()`	Deferred (returns a new function)	Pass arguments to preset	New function

🚀 Summary:

- `call()`: Immediately invoke a function with `this` set to a specific object and individual arguments.
- `apply()`: Immediately invoke a function with `this` set to a specific object and arguments passed as an array.
- `bind()`: Create a new function with `this` fixed, but do not invoke the function immediately. You can call this new function later.

ARRAY REVERSE

The Array.reverse() method in JavaScript reverses the order of elements in an array in place, meaning it modifies the original array and also returns it.

- The reverse() method reverses the order of the elements in an array.
- The reverse() method overwrites the original array.

```
Syntax:
array.reverse()
```

```
Example:
let numbers = [1, 2, 3, 4, 5];
numbers.reverse();
console.log(numbers); // [5, 4, 3, 2, 1]
```

Key Points:

1. It does not create a new array.
2. Useful for reversing the display or processing order of elements.
3. If you want a reversed copy of the array without changing the original:

```
let original = [1, 2, 3];
let reversedCopy = [...original].reverse();
```

ES6 Features

ECMAScript 6 (ES6), also known as ECMAScript 2015, introduced many powerful features to JavaScript that make code more concise, readable, and robust. Here are some of the most important ES6 features:

```
let , var / => / `  ${expression}` / promises / async await / ... /default value
to parameter ,class , modules
```

Key Connections & Memory Hooks

1. let/const → Everything
    - Base for all other features (block scope).
    - Remember: const for constants, let for reassignables.

2. Arrow Functions (=>) → Promises/Arrays

Arrow functions (`=>`) are a concise syntax for writing functions in ES6, with two key differences from regular functions:

1. **Shorter syntax** (often one-liners)
2. **Lexical** `this` (inherits `this` from surrounding scope)

---

1. Basic Syntax

Regular Function:

javascript Copy Download

```javascript
function add(a, b) {
  return a + b;
}
```

Arrow Function Equivalent:

javascript Copy Download

```javascript
const add = (a, b) => a + b;
```

2. Lexical `this` (Key Difference)

Problem with Regular Functions:

javascript Copy

```javascript
const person = {
  name: "Alice",
  greet: function() {
    setTimeout(function() {
      console.log(`Hi, I'm ${this.name}`); // ❌ `this` is undefined (or window)
    }, 1000);
  }
};
person.greet(); // "Hi, I'm undefined"
```

Fixed with Arrow Function:

javascript Copy

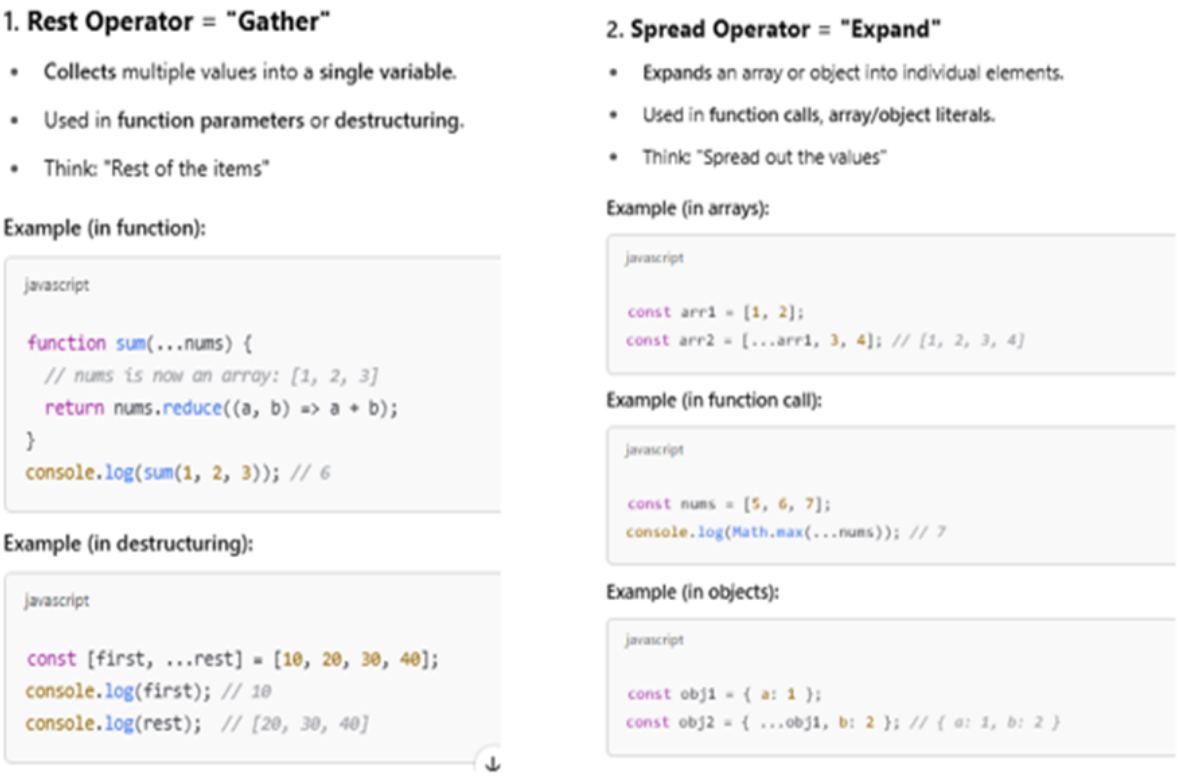
```javascript
const person = {
  name: "Alice",
  greet: function() {
    setTimeout(() => {
      console.log(`Hi, I'm ${this.name}`); // ✅ `this` refers to `person`
    }, 1000);
  }
};
person.greet(); // "Hi, I'm Alice"
```

Why?

- Arrow functions do not bind their own this—they inherit it from the parent scope.
- Regular functions have their own this, which depends on how they're called.
- Used in .then() chains (fetch) and array methods (reduce).

3. Rest/Spread (...) → Functions/Arrays

```
Spread ===== Unpacking elements.
Rest ===== Packing elements.
```



1. Rest Operator = "Gather"

- Collects multiple values into a single variable.
- Used in function parameters or destructuring.
- Think: "Rest of the items"

Example (in function):

```javascript
function sum(...nums) {
  // nums is now an array: [1, 2, 3]
  return nums.reduce((a, b) => a + b);
}
console.log(sum(1, 2, 3)); // 6
```

Example (in destructuring):

```javascript
const [first, ...rest] = [10, 20, 30, 40];
console.log(first); // 10
console.log(rest);  // [20, 30, 40]
```

2. Spread Operator = "Expand"

- Expands an array or object into individual elements.
- Used in function calls, array/object literals.
- Think: "Spread out the values"

Example (in arrays):

```javascript
const arr1 = [1, 2];
const arr2 = [...arr1, 3, 4]; // [1, 2, 3, 4]
```

Example (in function call):

```javascript
const nums = [5, 6, 7];
console.log(Math.max(...nums)); // 7
```

Example (in objects):

```javascript
const obj1 = { a: 1 };
const obj2 = { ...obj1, b: 2 }; // { a: 1, b: 2 }
```

4. Destructuring

Destructuring in JavaScript is a concise way to extract values from arrays and objects into distinct variables. It simplifies code by providing a cleaner syntax for accessing and assigning values, improving readability and reducing boilerplate.

How it works:

Array Destructuring:

You can unpack values from an array into individual variables using the destructuring syntax. For example: `const [a, b, c] = [1, 2, 3];` This assigns 1 to `a`, 2 to `b`, and 3 to `c`.

Object Destructuring:

You can extract properties from an object into variables. For example: `const { name, age } = { name: "John", age: 30 };` This assigns "John" to `name` and 30 to `age`.

Example (Array):

JavaScript

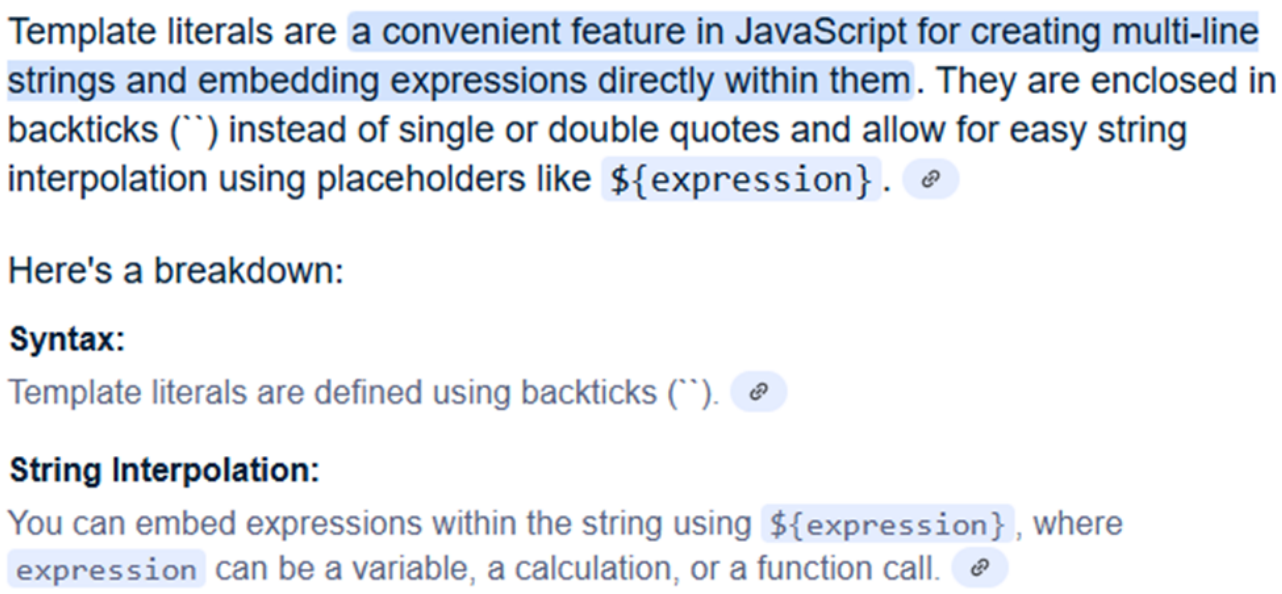
```
const fruits = ["apple", "banana", "cherry"];
const [first, second, third] = fruits;
console.log(first); // Output: apple
console.log(second); // Output: banana
console.log(third); // Output: cherry
```

Example (Object):

JavaScript

```
const person = { name: "Alice", age: 30, city: "Pune" };
const { name, age } = person;
console.log(name); // Output: Alice
console.log(age); // Output: 30
```

5. Template Literals (${}) → Everywhere



6. Classes ↔ Modules
    - ```
      Classes often exported via modules (export class).
      ```
    - Hook: "Classes are blueprints; modules are filing cabinets."
7. Promises → Async/Await (ES8)

SHALLOW VS DEEP COPY

Shallow Copy

A shallow copy creates a new object or array. In this process, only the top-level properties are copied, while nested objects or arrays still reference the original memory location. This means that if you change the nested properties in one object, those changes will reflect in the other because they share the same memory reference.
      ```

```javascript
const original = {
    name: 'John',
    age: 30,
    address: {
        city: 'New York',
        country: 'USA'
    }
};

// Creating a shallow copy
const shallowCopy = { ...original };

// Modifying the nested object in shallow copy
shallowCopy.name = 'Laxmi Kant';
shallowCopy.address.city = 'Bareilly';

console.log(original.address.city); // Output: Bareilly
console.log(original.name); // Output: John
```

Common Shallow Copy Methods:

1. Spread operator (...)
2. Object.assign({}, obj)
3. Array.prototype.slice()
4. Array.from()

✅ 1. Spread Operator (`...`)

```js
const copyWithSpread = { ...original };
console.log(copyWithSpread);
```

- **Note:** This is a **shallow copy**. If you change `copyWithSpread.address.city`, it will also affect `original.address.city`.

✅ 2. `Object.assign()`

```js
const copyWithAssign = Object.assign({}, original);
console.log(copyWithAssign);
```

- Also a **shallow copy**. Nested objects (like `address`) are shared between `original` and `copyWithAssign`.

✅ 3. `Array.prototype.slice()` (Used for arrays)

```js
const originalArray = [original];
const copyWithSlice = originalArray.slice();
console.log(copyWithSlice);
```

- `slice()` makes a **shallow copy** of the array. But if the array contains objects (like `original`), those objects are not cloned — only the reference is copied.

✅ 4. `Array.from()`

```js
const copyWithArrayFrom = Array.from(originalArray);
console.log(copyWithArrayFrom);
```

- Also makes a **shallow copy** of the array.

Deep Copy

A deep copy, on the other hand, creates a completely new object or array and recursively copies all nested objects and arrays found in the original. This means that any changes made to the deep copy do not affect the original structure

```
const original = {
    name: 'Alice',
    age: 25,
    address: {
        city: 'London',
        country: 'UK'
    }
};

// Creating a deep copy using JSON methods
const deepCopy = JSON.parse(JSON.stringify(original));

// Modifying the nested object in deep copy
deepCopy.address.city = 'Manchester';

console.log(original.address.city); // Output: London
```

Common deep Copy Methods:

1. structuredClone()
2. JSON.parse(JSON.stringify(obj))
3. lodash.cloneDeep() (using lodash library)

✅ 1. `structuredClone()` *(Modern and best for most cases)*

```javascript
const original = { a: 1, b: { c: 2 } };
const copy = structuredClone(original);

copy.b.c = 99;
console.log(original.b.c); // 2 (unaffected)
```

✔ Supports most types: `Date`, `Map`, `Set`, `ArrayBuffer`
✖ Not supported in older browsers (but widely supported in modern JS)

✅ 2. `JSON.parse(JSON.stringify(obj))`

```javascript
const original = { a: 1, b: { c: 2 } };
const copy = JSON.parse(JSON.stringify(original));
```

✔ Simple and quick
✖ Loses functions, `undefined`, `Infinity`, `Symbol`, `Date`, `Map`, `Set`
✖ Fails on circular references

✅ 3. `lodash.cloneDeep()` (using lodash library)

```javascript
import cloneDeep from 'lodash/cloneDeep';

const original = { a: 1, b: { c: 2 } };
const copy = cloneDeep(original);
```

✔ Handles all types (even complex/circular structures)
✖ Requires installing `lodash`

```bash
npm install lodash
```

1. `JSON.stringify(original)`
   - This converts the `original` JavaScript object into a **JSON string**.
   - Example:

```javascript
'{"name":"John","age":30,"address":{"city":"New York","country":"USA"}}'
```

2. `JSON.parse(...)`
   - This takes the JSON string and converts it **back into a new JavaScript object**.
   - The result is a **completely new object** in memory with the same structure and values.

PROTOTYPES AND PROTOTYPES INHERITANCE

Prototypes in JavaScript are a mechanism for objects to inherit properties and methods from other objects. Every object in JavaScript has an associated prototype, which is another object that it inherits from. When you try to access a property on an object, JavaScript first looks for the property directly on the object itself. If it doesn't find it, it then looks for the property on the object's prototype, and so on, up the prototype chain.

Prototypes enable code reuse and inheritance. By defining properties and methods on a prototype, you can make them available to all objects that inherit from that prototype. This can help to reduce code duplication and make your code more maintainable.

JavaScript

```javascript
function Person(name) {
  this.name = name;
}

Person.prototype.greet = function() {
  console.log("Hello, my name is " + this.name);
};

const person1 = new Person("John");
person1.greet(); // Output: Hello, my name is John

const person2 = new Person("Jane");
person2.greet(); // Output: Hello, my name is Jane
```

How Prototype Works in JavaScript?

- In JavaScript, each object has an internal [[Prototype]] property, which points to another object. This allows the object to inherit properties and methods from its prototype.
- When you access a property or method on an object, JavaScript first checks the object itself. If it's not found, it looks in the object's prototype and continues up the prototype chain until it either finds the property or reaches null.
- Functions in JavaScript have a prototype property. This is where you add properties or methods that you want to be available to all instances of objects created by that constructor function.
- Objects created using a constructor function inherit properties and methods from the constructor's prototype. This allows for reusable code and shared behaviour across multiple objects.
- You can add new properties or methods to an object's prototype, and all instances of that object will automatically have access to the new functionality. This is a common way to extend built-in objects like Array or Object.

Prototype Inheritance

All JavaScript objects inherit properties and methods from a prototype:

- `Date` objects inherit from `Date.prototype`
- `Array` objects inherit from `Array.prototype`
- `Person` objects inherit from `Person.prototype`

The `Object.prototype` is on the top of the prototype inheritance chain:

`Date` objects, `Array` objects, and `Person` objects inherit from `Object.prototype`.

prototype inheritance where the child constructor inherits from the parent constructor, allowing instances of child to access methods defined in parent's prototype. It also adds a custom caste method in the child

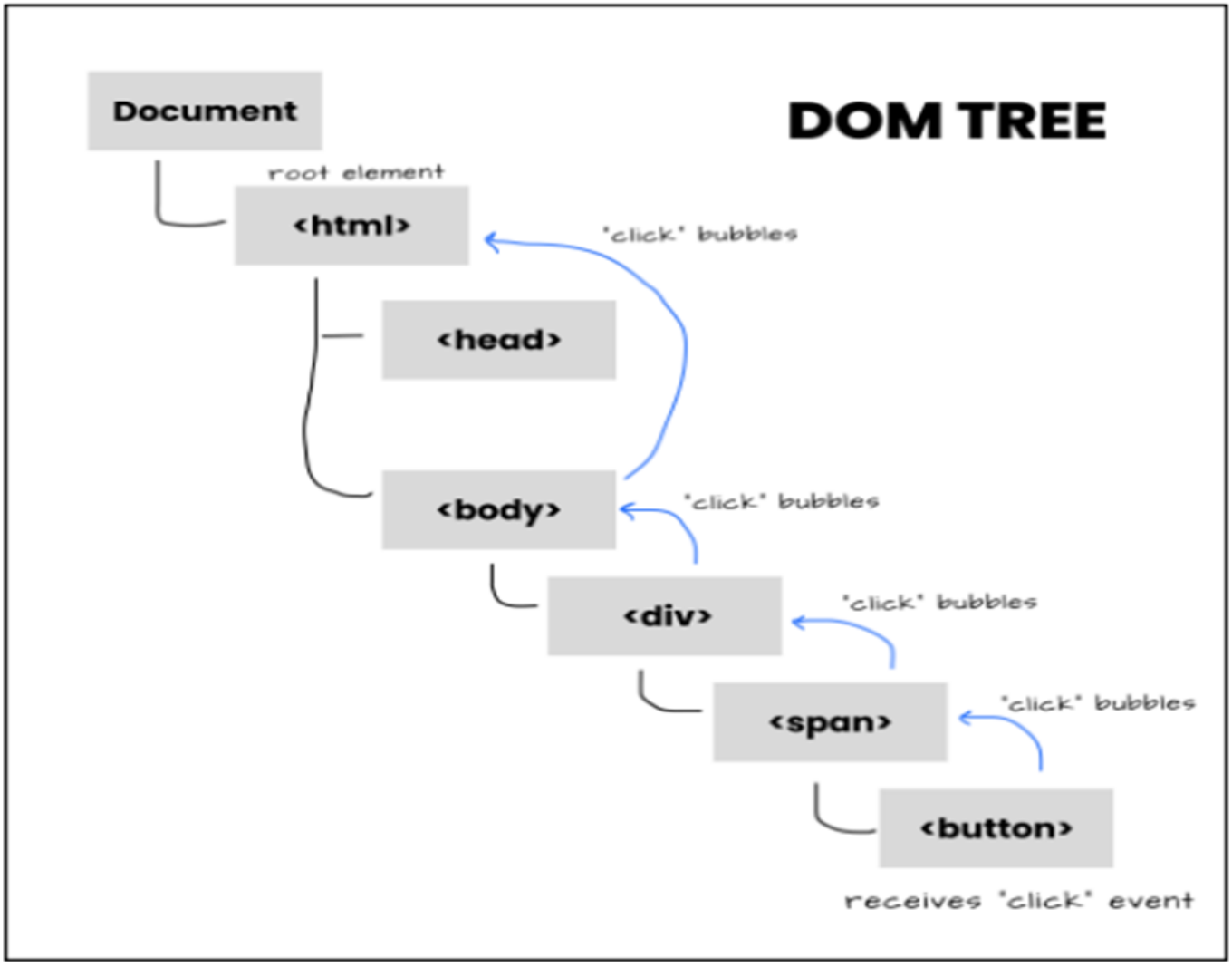
prototype

```
function Animal(name) {
    this.name = name;
}

Animal.prototype.speak = function () {
    console.log(`${this.name} makes a noise.`);
};

function Dog(name) {
    Animal.call(this, name); // Call the parent constructor
}

Dog.prototype = Object.create(Animal.prototype); // Set up inheritance
Dog.prototype.constructor = Dog;

Dog.prototype.speak = function () {
    console.log(`${this.name} barks.`);
};

const dog = new Dog('Rex');
dog.speak(); // Rex barks.
```

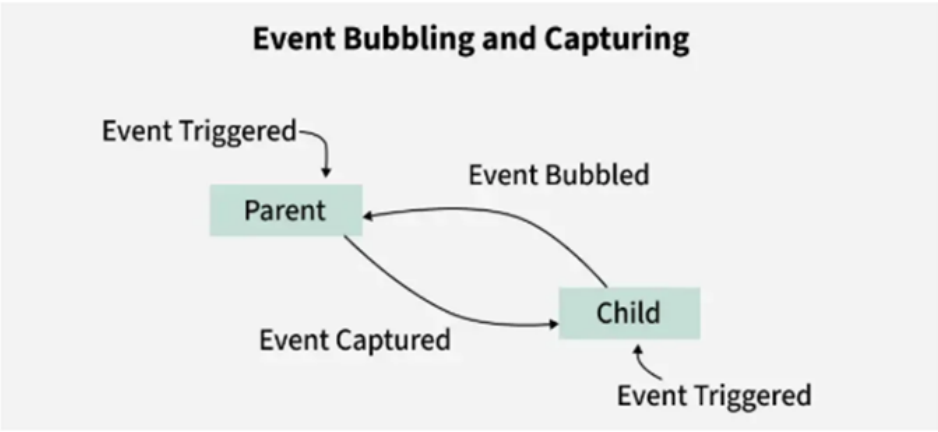
EVENT BUBBLING

Event Bubbling is a concept in the DOM (Document Object Model). It happens when an element receives an event, and that event bubbles up (or you can say is transmitted or propagated) to its parent and ancestor elements in the DOM tree until it gets to the root element.

1. The "Event Bubbling" behavior makes it possible for you to handle an event in a parent element instead of the actual element that received the event.
2. The pattern of handling an event on an ancestor element is called Event Delegation.
3. Event Bubbling Allows events to move up the DOM tree
4. Event Delegation Lets you handle events at a higher level node



How Event Bubbling Works?



Event Bubbling in JavaScript

- **Event Triggering**: The click event is triggered on the child element (button), initiating the event propagation.
- **Event Capturing**: In the capturing phase, the event propagates from the root of the DOM down to the target (child). However, no listeners are explicitly set to handle events in this phase in the given code.
- **Event Bubbling**: After reaching the target element (child), the event enters the bubbling phase, propagating back up through the DOM tree to the parent (parent).
- **Listener Behavior**: Event listeners are attached to both parent and child elements using addEventListener. By default, these listeners respond during the bubbling phase unless the capture option is set to true.
- **Execution Order**: When the button is clicked, the child listener executes first (console.log("Child")), followed by the parent listener (console.log("Parent")) as the event bubbles up.

```javascript
let grandparent = document.getElementById('grandparent');
let parent = document.getElementById('parent');
let child = document.getElementById('child');

grandparent.addEventListener('click', function () {
    console.log("Grandparent Clicked");
});

parent.addEventListener('click', function () {
    console.log("Parent Clicked");
});

child.addEventListener('click', function () {
    console.log("Child Clicked");
});
```

🧪 What Happens When You Click the "Child" Element?

You will see this in the console:

```nginx
Child Clicked
Parent Clicked
Grandparent Clicked
```

This shows **event bubbling**:

- The event starts at the **child** (the actual clicked element),
- Then **bubbles up** to the **parent**,
- And then to the **grandparent**.

🔁 If You Add `stopPropagation()`?

```javascript
child.addEventListener('click', function (e) {
    e.stopPropagation();
    console.log("Child Clicked");
});
```

- Event Bubbling Stopped: e.stopPropagation() prevents the event from bubbling to parent elements.

- Independent Logs: Only the clicked element logs its message (e.g., “Child Clicked”), as bubbling is stopped.

MEMOIZATION

Memoization is a technique for speeding up applications by caching the results of expensive function calls and returning them when the same inputs are used again.

1. Expensive Function Calls: Time and memory are the two most important resources in computer applications. As a result, an expensive function call is one that consumes large amounts of these two resources due to extensive calculation during execution.
2. Cache: A cache is just a temporary data store that stores data in order to serve future requests for that data more quickly.

Importance of Memoization:

When a function is given in input, it performs the necessary computation and saves the result in a cache before returning the value. If the same input is received again in the future, it will not be necessary to repeat the process. It would simply return the cached answer from the memory. This will result in a large reduction in a code’s execution time.

✅ How to Achieve Memoization in JavaScript

🔗 Memoization in 3 Simple Steps

1. Create a cache to store previously computed results.
2. Check the cache before computing.
3. Store the result in the cache if it's not already there.

```
You can achieve this in two main ways:
•   Using closures
•   Using a higher-order function
```

✅ Example: Memoizing with a Closure

javascript Copy Edit

```javascript
const memoizedAdd = (function () {
    const cache = {};  // This cache lives in the closure

    return function (x, y) {
        const key = `${x},${y}`;

        if (cache[key]) {
            console.log('Fetching from cache for:', key);
            return cache[key];
        }

        console.log('Computing result for:', key);
        const result = x + y;
        cache[key] = result;
        return result;
    };
})();
```

✅ Example: Memoizing a Function with a Higher-Order Function

javascript Copy Edit

```javascript
function memoize(fn) {
    const cache = {}; // Closure to store results

    return function(...args) {
        const key = JSON.stringify(args); // Create a key from arguments

        if (cache[key]) {
            console.log('Fetching from cache for:', args);
            return cache[key];
        }

        console.log('Computing result for:', args);
        const result = fn(...args);
        cache[key] = result;
        return result;
    };
}
```

DEBOUNCE

Debouncing is a technique used to ensure that a function is not called too frequently. Debouncing in JavaScript can be defined as the technique that is used to limit the number of times a function gets executed. Debouncing is useful when the event is frequently being triggered in a short interval of time like typing, scrolling, and resizing.

1. Limit Function Calls: During frequent events like typing, resizing, or scrolling debouncing prevents the frequent function calls.
2. Delays Execution: After the specific delay only the function is executed, ensuring no rapid consecutive calls.
3. Prevents Overload: Efficiently managing high-frequency triggers helps in preventing overloading.

```
// Debounce function
function debounce(func, delay) {
    let timeout;
    return function (...args) {
        clearTimeout(timeout);
        timeout = setTimeout(() => {
            func.apply(this, args);
        }, delay);
    };
}

// Function to be debounced
function search(query) {
    console.log('Searching for:', query);
}

// Create a debounced version of the search function
const dSearch = debounce(search, 100);

// Simulate typing with multiple calls to the debounced function
dSearch('Hello');
dSearch('Hello, ');
dSearch('Hello, World!');  // Only this call will trigger after 100ms
```

In this example

- debounce() function: It is the higher order function that takes (delay) and function(func) as the arguments. It returns a new function that will wait for the specified delay before calling the original function.
- clearTimeout(): It is used to clear any previous set timeout so that if the event is triggered repeatedly the function call does not happen too quickly.
- setTimeout(): This method is used to set the timeout after clearing the previous timeouts.
- Search function: It is the placeholder for the function we want to debounce.

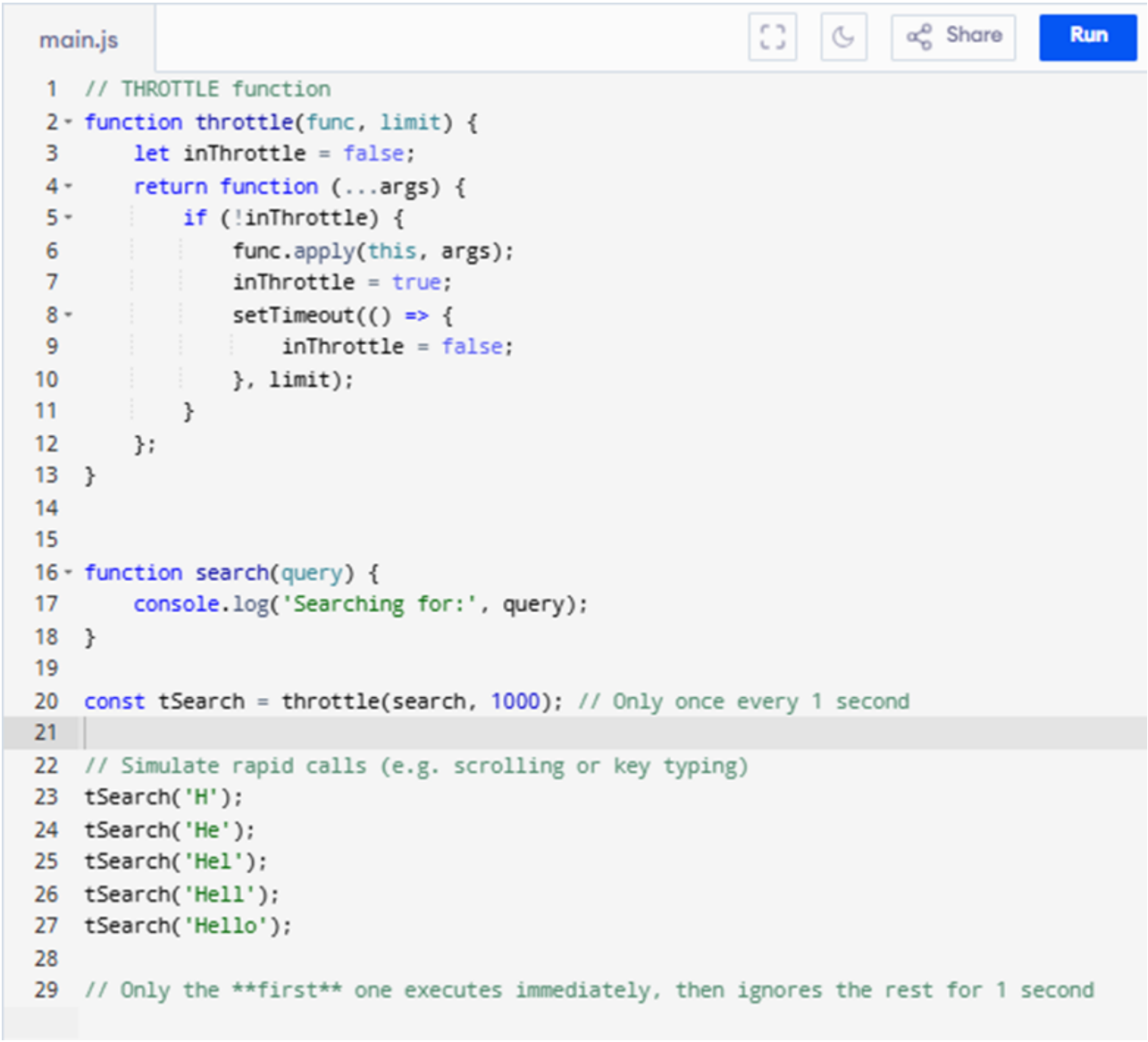
THROTTLE

Throttling is a technique used to limit the number of times a function can be executed in a given time frame. It's extremely useful when dealing with performance-heavy operations, such as resizing the window or scrolling events, where repeated triggers can lead to performance issues.

How Throttling Works

Throttling works by restricting the execution of a function so that it runs at most once every predefined period, even if the event is triggered multiple times within that interval.

1. A function is triggered multiple times due to an event (e.g., scroll, resize).
2. Throttling ensures that the function executes only once within the defined interval.
3. Any additional triggers during the interval are ignored until the next cycle starts.
4. Once the interval is over, the function can execute again if triggered



In this example

- throttle(func, limit) A higher-order function that returns a throttled version of func, allowing it to run at most once every limit milliseconds.
- let inThrottle = false; A flag to prevent repeated calls during the cooldown period.
- return function (...args) Returns the throttled function that receives any number of arguments.

- if (!inThrottle) Only allows func to run if it's not currently throttled.
- func.apply(this, args) Calls the original function with proper context and arguments.
- setTimeout(() => { inThrottle = false }, limit) Resets the throttle flag after the specified limit time, allowing the next execution.

PREVENT DEFAULT

preventDefault() prevents the default behavior of an event from happening.

Why it's useful:

Some events have default actions provided by the browser. preventDefault() stops those actions so you can handle them your own way.

🧪 Examples:

1. Prevent a form from submitting

```javascript
document.querySelector('form').addEventListener('submit', function(e) {
    e.preventDefault();  // Stops form from refreshing the page
    console.log('Form submission stopped!');
});
```

2. Prevent link navigation

```javascript
document.querySelector('a').addEventListener('click', function(e) {
    e.preventDefault();  // Stops link from navigating
    console.log('Link click prevented!');
});
```

⚠️ Common Use Cases:

- Form validation before submit
- Custom button/link behavior
- Drag-and-drop control
- Blocking scrolling (e.g., in modals)

📝 Summary Table

Event Type	Default Action	When to Use `preventDefault()`
`submit`	Submits and reloads page	Validate input first
`click`	Navigates (for `<a href>`)	You want custom routing or logic
`keydown`	Scrolls or triggers shortcuts	Build custom keybindings