|  |  |  |  |
| --- | --- | --- | --- |
| **S.No** | **Video name** | **URLs** | **Status** |
| 1. | **JavaScript Crash Course: Master the Basics** | https://youtu.be/htznIeWKgg8?si=IN1VuMYMjTXk9PUx |  |
| 2. | **JavaScript Advance Crash Course: Level Up Your Coding Skills** | https://youtu.be/EgDmCbhmstU?si=\_44UB3-w\_jqrcP0M |  |
| 3. | **Master Advanced JavaScript Concepts and Become a JavaScript Ninja** | https://youtu.be/ks4MFTHLfyg?si=IgD5FyaNO7C77A2k |  |
| 4. | **Master Async JavaScript** | https://youtu.be/6kE8lrqfwHo?si=0atm-xEp8E0zEIDi |  |
| 5. | **JavaScript DOM Manipulation** | https://youtu.be/2IPEp\_4obGw?si=wfCaXHHudAme8jyn |  |
| 6. | **Mind-Blowing ES6 JavaScript Techniques** | https://youtu.be/D8HyB-W7ToI?si=kr6Ru-lEQGtExyfX |  |
|  |  |  |  |

**JavaScript - 1**

**Compiler vs Interpreter**

A compiler and interpreter are both programs that translate source code into a low-level language that computers can understand.

**Main differences**

* **Compiler:** A compiler translates the entire program into machine code (bytecode) before execution.
* **Compiler based language:** C, C++, Java, Python.
* **Interpreter:** An interpreter executes the source code line by line or statement by statement. It processes and runs the code directly.
* **Interpreter based language:** JavaScript, PHP, Perl, Ruby.

**JavaScript ES5 (ECMAScript 5) and ES6 (ECMAScript 6 [2015])**

JavaScript ES5 and ES6 are two major versions of the JavaScript language. ES6 introduced significant improvements and new features over ES5, making JavaScript more powerful and easier to work with.

**The key differences between ES5 and ES6:**

|  |  |  |
| --- | --- | --- |
| **JavaScript Keywords** | **ES5** | **ES6** |
| Variable Declarations | - Only **var** was available for variable declarations.  - var is function-scoped and can lead to issues like hoisting and unintended behavior. | - introduced **let and const** for block-scoped variables.  - let and const are block-scoped, reducing bugs related to variable hoisting and scope. |
| Arrow Function | - Function declared using function keyword. | - Introduced arrow function (**=>**), which provides a shorter syntax and lexically bind **this**.  - Arrow functions do not have their own **this** context, they inherit **this** fromthe parent scope. |
| Template Literals | String concatenation was done using **+**. | Introduced template literals **(backtick `)** for easier string interpolation. |
| Default Parameter | - Default parameter had to be handled manually. | - Default parameters can be set directly in the function definition. |

**Words vs Keyword**

* **Word:** A word is a sequence of characters that don’t have meaning in a language but use as an identify anything.
* **Keyword:** A keyword is a reserved word in JavaScript that has a specific meaning and purpose in the language.

**Var, Let and Const**

* **Variables:** A named container that stores data which can change during the program.

**Variable declared in 3 ways:**

* Var (Variable)
* Let
* Const (Constant)
* **Const (Constant):** Anamedcontainer that stored data which cannot change once declared.
* Variable defined with const cannot be redeclared, cannot be reassigned and have block scope.
* **Let:** Let is used to declare a block – scoped (braces scoped) variable, must be reassigned, must be declared before used, cannot be redeclared in the same scoped variable.

**DOM (Document Object Model)**

* The DOM represents the structured content of an HTML document, allowing JavaScript to manipulate elements, styles, and attributes dynamically.

**5 DOM Methods:**

1. document.getElementById("id") – Selects an element by its ID.
2. document.querySelector("selector") – Selects the first matching element.
3. document.createElement("tag") – Creates a new HTML element.
4. document.appendChild(element) – Adds a new child element to a parent.
5. document.removeChild(element) – Removes a child element from a parent.

**BOM (Browser Object Model)**

* The BOM provides browser-related functionalities, allowing JavaScript to interact with the browser environment beyond the document structure.

**5 BOM Methods:**

1. window.open("url", "\_blank") – Opens a new browser tab or window.
2. window.alert("message") – Displays an alert popup.
3. window.history.back() – Navigates to the previous page.
4. window.location.reload() – Reloads the current page.
5. navigator.geolocation.getCurrentPosition(successCallback) – Gets the user's location.

**Window Object**

* The **Window object** represents an open window in a browser and serves as the global object for JavaScript code running in the browser. It provides methods, properties, and events for interacting with the browser.

**5 Common Methods of the window Object:**

1. alert(message)
2. confirm(message)
3. prompt(message, default)
4. setTimeout(function, milliseconds)
5. setInterval(function, milliseconds)
6. open(url, target, features)

**Browser Context API**

* The **Browser Context API** allows control over different browser contexts, such as opening new tabs, pop-up windows, or managing multiple browsing sessions.

**5 Methods of the Browser Context API**

* window.open(url, target, features)
* window.close()
* window.postMessage(message, targetOrigin)
* window.frames[index]
* window.parent

**Heap Memory**

* Heap memory is a dynamically allocated memory space where objects, variables, and functions are stored during program execution. It is managed by JavaScript’s garbage collector.

**Execution Context**

* **Execution context** is a container where the function’s code is executed and it’s created whenever a function is called, it contains 3 things: Variables, Functions and Lexical environment.

**JavaScript creates three types of execution contexts:**

1. **Global Execution Context (GEC)** – The default context when a script runs.
2. **Function Execution Context (FEC)** – Created whenever a function is invoked.
3. **Eval Execution Context** – Created inside the eval() function.

**Lexical Environment**

* A **Lexical Environment** is a structure that holds variable and function references in JavaScript. It is created whenever an execution context is formed.

**Each function or block creates its own lexical environment, which contains:**

1. **Local memory** – Stores variables and function declarations.
2. **Reference to parent lexical environment** – Allows access to outer scopes (Closures).

|  |
| --- |
| **Example:**  function gathering() {  var hostName = "Amandeep"; // Host of the party  console.log("🎉 Gathering Started by: " + hostName);  function invitedPerson1() {  var name1 = "Neelam";  console.log("✅ " + name1 + " is invited by " + hostName);  }  function invitedPerson2() {  var name2 = "Rizwan";  console.log("✅ " + name2 + " is invited by " + hostName);  }  invitedPerson1();  invitedPerson2();  // console.log(name1); // ❌ This will give an error (name1 is not accessible here)  }  gathering();  **Output:**  🎉 Gathering Started by: Amandeep  ✅ Neelam is invited by Amandeep  ✅ Rizwan is invited by Amandeep |

**The key differences between ES5 and ES6:**

**Variable Declarations:**

* **ES5**: Only var was available for variable declarations.

var is function-scoped and can lead to issues like hoisting and unintended behavior.

var x = 10;

* **ES6**: Introduced let and const for block-scoped variables.

let and const are block-scoped, reducing bugs related to variable hoisting and scope.

let x = 10; // Mutable

const y = 20; // Immutable

**Arrow Functions\***

* **ES5**: Functions are declared using the function keyword.

function add(a, b) {

return a + b;

}

* **ES6**: Introduced arrow functions (=>), which provide a shorter syntax and lexically bind this. Arrow functions do not have their own this context; they inherit this from the parent scope.

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

**Template Literals**

* **ES5**: String concatenation was done using +.

var name = "John";

console.log("Hello, " + name + "!");

* **ES6**: Introduced template literals (backticks `) for easier string interpolation.

const name = "John";

console.log(`Hello, ${name}!`);

**Default Parameters**

* **ES5**: Default parameters had to be handled manually.

function greet(name) {

name = name || "Guest";

console.log("Hello, " + name);

}

* **ES6**: Default parameters can be set directly in the function definition.

function greet(name = "Guest") {

console.log(`Hello, ${name}`);

}

**Destructuring Assignment\***

* **ES5**: Extracting values from objects or arrays required manual assignment.

var person = { name: "John", age: 30 };

var name = person.name;

var age = person.age;

* **ES6**: Introduced destructuring for easier extraction.

const person = { name: "John", age: 30 };

const { name, age } = person;

**Classes\***

* **ES5**: Used constructor functions and prototypes for object-oriented programming.

function Person(name) {

this.name = name;

}

Person.prototype.greet = function() {

console.log("Hello, " + this.name);

};

* **ES6**: Introduced class syntax for cleaner and more intuitive OOP.

class Person {

constructor(name) {

this.name = name;

}

greet() {

console.log(`Hello, ${this.name}`);

}

}

**Modules\***

* **ES5**: No native module system. Developers relied on libraries like CommonJS or AMD.

// CommonJS

var module = require('module');

* **ES6**: Introduced native module support with import and export.

// Export

export const name = "John";

// Import

import { name } from './module';

**Promises\***

* **ES5**: Callbacks were used for asynchronous operations, leading to "callback hell."

setTimeout(function() {

console.log("Done");

}, 1000);

* **ES6**: Introduced Promise for better handling of asynchronous operations.

const promise = new Promise((resolve, reject) => {

setTimeout(() => resolve("Done"), 1000);

});

promise.then(result => console.log(result));

**Block-Scoped Constructs**

* **ES5**: Only function scope was available.

if (true) {

var x = 10;

}

console.log(x); // 10 (no block scope)

* **ES6**: Introduced block scope with let and const.

if (true) {

let x = 10;

}

console.log(x); // ReferenceError: x is not defined

**Enhanced Object Literals\***

* **ES5**: Object literals were straightforward.

var obj = {

name: "John",

greet: function() {

console.log("Hello");

}

};

* **ES6**: Added shorthand syntax and computed property names.

const name = "John";

const obj = {

name, // Shorthand

greet() { // Method shorthand

console.log("Hello");

},

[ "key\_" + (1 + 1) ]: "value" **// Computed property\***

};

**Spread and Rest Operators\***

* **ES5**: No spread or rest operators.

function sum(a, b, c) {

return a + b + c;

}

var args = [1, 2, 3];

sum.apply(null, args);

* **ES6**: Introduced ... for spread and rest operations.

const sum = (a, b, c) => a + b + c;

const args = [1, 2, 3];

sum(...args); // Spread

const restFunc = (...args) => console.log(args); // Rest

**Iterators and Generators\***

* **ES5:** No native support for iterators or generators.
* **ES6:** Introduced iterators and function\* for generators.

function\* generator() {

yield 1;

yield 2;

}

const iterator = generator();

console.log(iterator.next().value); // 1

**New Data Structures\***

* **ES5:** Limited to basic data structures like arrays and objects.
* **ES6:** Introduced Map, Set, WeakMap, and WeakSet.

const map = new Map();

map.set("key", "value");

**Symbols\***

* **ES5**: No symbol type.
* **ES6**: Introduced Symbol for unique identifiers.

const sym = Symbol("description");

**Hoisting vs Hosting (Both are far different)**

* **Hosting:** Hosting refers to storing a website or Web Application on a server so that it is accessible online.
* **Hoisting:** Hoisting is a JavaScript behavior where variableand function declarations are moved to the top of their scope before the code is executed.

**Undefined vs Not defined**

* **Undefined:** Variable is declared but Not Assigned a value
* If a variable is declared but not initialized, JavaScript automatically assigns it the undefined value.
* **Not defined:** Variable is not declared at all.
* If you try to access a variable that was never declared, JavaScript throws a **ReferenceError.**

**Types in JavaScript**

**Primitive and Reference**

* **Primitive Types:** Primitive types are simple, immutable values. They are not objects and are passed by value.

**Example:** string, number, boolean, null, undefined, symbol, BigInt.

* **Reference Types:** Reference types are objects, arrays, functions, and other complex data types. They are passed by reference, meaning that when assigned or passed to a function, the reference (memory address) of the object is passed, not a copy of the object.

**Example:** object, array, function.

**How to copy reference values in array and object**

You need to create a new copy (not a reference) use spread operator:

* **For Array**

|  |
| --- |
| let arr1 = [1, 2, 3];  let arr2 = [...arr1]; // Spread operator  arr2.push(4);  console.log(arr1); // Output: [1, 2, 3]  console.log(arr2); // Output: [1, 2, 3, 4] |

* **For Object**

|  |
| --- |
| let obj1 = { name: "Amandeep", age: 25 };  let obj2 = { ...obj1 }; // Spread operator  obj2.age = 26;  console.log(obj1); // Output: { name: "Amandeep", age: 25 }  console.log(obj2); // Output: { name: "Amandeep", age: 26 } |

**Conditionals**

* **If else else-if:** In JavaScript if, else-if, else are used for conditional statements to execute different blocks of code based on conditions.
* **If Statement:** Executes a block of code if a specified condition is true.
* **Else Statement:** Executes a block of code if the if condition is false than it execute in else condition.
* **Else if Statement:** Checks multiple conditions. If the first, if condition is false, it checks the else-if condition. If none are true, the else block runs.

**Falsey vs truthy**

* In **JavaScript**, **Falsy Values** are those values that are automatically converted to **false** when evaluated in a **Boolean context** (like in an if statement, condition, or comparison). There are exactly **8 Falsy Values** in JavaScript. If any of these values are encountered in a condition, they will be treated as **false**.

**List of 7 Falsy Values in JavaScript:**

|  |  |
| --- | --- |
| **Falsy Value** | **Explanation** |
| false | The Boolean value false itself is always falsy. |
| 0 | The number 0 (zero) is always considered falsy. |
| -0 | The number -0 (negative zero) is also falsy. |
| "" (empty string) | Any empty string ("" or '') without any space is considered falsy. |
| null | It represents the absence of any value, hence treated as falsy. |
| undefined | It represents a variable that has been declared but not assigned any value, hence falsy. |
| NaN (Not a Number) | Any invalid mathematical operation or not a number result is falsy. |
| document.all | document.all is always considered falsy. |

**Switch Statement**

* The switch statement is used to perform different actions based on different conditions. Use the switch statement to select one of many code blocks to be executed.

|  |
| --- |
| switch (expression) {  case value1:  // Code to execute if expression === value1  break;  case value2:  // Code to execute if expression === value2  break;  default:  // Code to execute if no case matches  } |

**Loops (For While)**

* In JavaScript, loops are used to execute a block of code repeatedly until a specified condition is met. Two common types of loops are the for loop and the while loop.

**For loop and While loop**

* **for Loop:** The for loop is used when you know how many times you want to execute a block of code. It consists of three parts: initialization, condition, and iteration.

|  |
| --- |
| **Example:**  for (let i = 1; i <= 5; i++) {  console.log("Iteration:", i);  }  **Output:**  Iteration: 1  Iteration: 2  Iteration: 3  Iteration: 4  Iteration: 5 |

* **while Loop:** The while loop is used when you want to execute a block of code as long as a specified condition is true. The condition is evaluated before each iteration.

|  |
| --- |
| **Example:**  let count = 1;  while (count <= 5) {  console.log("Count:", count);  count++; // Increment count  }  **Output:**  Count: 1  Count: 2  Count: 3  Count: 4  Count: 5 |

* **forEach Loop:** The forEach() method is used to iterate over an array and execute a function for each element.

|  |
| --- |
| **Example:** const numbers = [1, 2, 3, 4, 5];  numbers.forEach(function(num) {  console.log(num \* 2)  });  **Output:**  2, 4, 6, 8, 10 |

**✅ Modify Original Array →** Use forEach() with arr[index] = value;

**✅ Create a New Array →** Use map()

**❌ Reassigning** num **inside** forEach() **does nothing.**

* **For In Loop:** The JavaScript for in statement loops through the properties of an Object.

|  |
| --- |
| **Example**  **Iterating Over an Object** const person = {  name: "Amandeep",  age: 25,  profession: "Software Developer"  };  for (let key in person) {  console.log(key + ": " + person[key]);  }  **Output:**  name: Amandeep  age: 25  profession: Software Developer  **Using for in loop with an Array**  const numbers = [10, 20, 30, 40];  for (let index in numbers) {  console.log(index + ": " + numbers[index]);  }  **Output:**  0: 10  1: 20  2: 30  3: 40 |

* **For Of Loop (Optional):** The for of loop is used to iterate over values in iterable objects like arrays, strings, maps, sets, and more. It directly accesses values instead of indexes (unlike for...in):

|  |
| --- |
| **Example**  **Iterating Over an Array**  const numbers = [10, 20, 30, 40];  for (let num of numbers) {  console.log(num);  }  **Output:**  10  20  30  40  **Iterating Over a String**  const name = "Amandeep";  for (let char of name) {  console.log(char);  }  **Output:**  A  m  a  n  d  e  e  p |

* **Do while loop (Optional):** The do...while loop executes the code at least once and then continues running as long as the condition is true.

|  |
| --- |
| **Example:**  let count = 1;  do {  console.log("Count:", count);  count++;  } while (count <= 5);  **Output:**  Count: 1  Count: 2  Count: 3  Count: 4  Count: 5 |

|  |  |
| --- | --- |
| **Loop Type** | **Best For** |
| for | Fixed number of iterations |
| while | Unknown iterations (condition-based) |
| do...while | Ensuring at least one execution |
| for...in | Looping through object properties |
| for...of | Looping through arrays/iterables |
| .forEach() | Applying a function to each array element |

**Functions**

* **Function:** In JavaScript, a function is reusable block of code that performs a specific task or calculates a value. Functions allow you to encapsulate logic, making your code more modular, organized and easier to maintain. You can define a function once and call it multiple times throughout your program.

**Parameter:** Parameter is defined in the function.

**Argument:** Argumentpassed when the function is called.

**Syntax:** function functionName(parameters) {

// Code to be executed

return result; // Optional

}

functionName(arguments)

**Returns**

* In JavaScript, the return statement is used to send a value back from a function to the caller. It immediately stops the execution of the function and returns the specified value.

|  |
| --- |
| **Example:**  function add(a, b) {  return a + b; // Returns the sum of a and b  }  const result = add(5, 10);  console.log(result);  **Output:**  15 |

**Callback Function**

* **Callback Function:** A callback function is a function that is passed as an argument to another function and is executed later, usually after some operation is completed.

**Why Use Callback Functions?**

✅ **Handles asynchronous operations** (e.g., API calls, file reading).  
✅ **Allows code reusability** (pass different functions to achieve different results).  
✅ **Improves modularity** (separates logic into smaller, manageable functions).

|  |
| --- |
| **Example:**  function greet(name, callback) {  console.log("Hello, " + name);  callback();  }  function sayGoodbye() {  console.log("Goodbye!");  }  greet("Aman", sayGoodbye);  **Output:**  Hello, Aman  Goodbye! |

✔️ sayGoodbye() is passed as a callback and executed after greet().

**First-Class Function**

* In JavaScript, **first-class functions** mean that **functions are treated like variables**. You can:  
  ✅ Assign functions to variables.  
  ✅ Pass functions as arguments.  
  ✅ Return functions from other functions.

This makes JavaScript a **functional programming** language.

|  |
| --- |
| **Example:**  function greetUser(a) {  console.log("Executing callback function...");  a();  }  greetUser(function() {  console.log("Hello, welcome to JavaScript!");  });  **Output:**  Executing callback function...  Hello, welcome to JavaScript! |

**Arrays**

* **Arrays:** An array is a data structure that stores a collection of elements in a specific order. Each element in an array is identified by a unique index or key, and the position of each element can be calculated using a mathematical formula.

**Example:** const fruits = ["Apple", "Banana", "Orange"];

console.log(fruits[0]); // Output: Apple

console.log(fruits[2]); // Output: Orange

**How array is made behind the scene**

* JavaScript arrays are implemented as **objects with optimized indexing**. arrays are a type of **object** with integer-based keys.

|  |
| --- |
| **Example:**  let arr = [10, 20, 30];  console.log(typeof arr); // "object" |

Internally, JavaScript stores arrays as objects where:

* **Indexes (0, 1, 2, …)** are stored as keys.
* **Values (10, 20, 30, …)** are stored as properties.

|  |
| --- |
| **Equivalent object representation:**  let arr = {  0: 10,  1: 20,  2: 30,  length: 3 // Special property  }; |

**Array.isArray():**

* The Array.isArray() method checks whether a given value is an array. It returns true if the value is an array and false otherwise.

|  |
| --- |
| **Example:**  console.log(Array.isArray([1, 2, 3])); // true  console.log(Array.isArray([])); // true  console.log(Array.isArray("hello")); // false  console.log(Array.isArray({})); // false |

**Push, Pop, Shift, Unshift**

* **push():** push() is used for adding one or more elements to the end of an array. Return the new length of the array. Mutates the original array.

**Syntax:** array.push(element1, element2, ..., elementN);

* **pop():** pop() is used for removes the last element from an array. Returns the removed element. Mutates the original arrays.

**Syntax:** array.pop();

* **shift():** shift() is used for removes the first element from an array. Return the removed element. Mutates the original array.

**Syntax:** array.shift();

* **unshift():** unshift() is used for adding one or more elements to the beginning of an array. Return the new length of the array. Mutates the original array.

**Syntax:** array.unshift(element1, element2, ..., elementN);

* **splice():** In JavaScript, The splice() method is a powerful array method used to add, remove, and replace elements in array. It modifies the original array and returns an array of the add, remove and replace elements (if any).

**Syntax:** array.splice(start, deleteCount, item1, item2, ..., itemN);

1. **Removing Elements**

To remove elements, specify the start index and the number of elements to remove (deleteCount).

**Example:** const fruits = ["Apple", "Banana", "Orange", "Mango"];

const removed = fruits.splice(1, 2); // Remove 2 elements starting at index 1

console.log(fruits); // Output: ["Apple", "Mango"]

console.log(removed); // Output: ["Banana", "Orange"]

1. **Adding Elements**

To add elements, specify the start index, set deleteCount to 0, and provide the elements to add.

**Example:** const fruits = ["Apple", "Mango"];

fruits.splice(1, 0, "Banana", "Orange"); // Add elements at index 1

console.log(fruits); // Output: ["Apple", "Banana", "Orange", "Mango"]

1. **Replacing Elements**

To replace elements, specify the start index, the number of elements to remove (deleteCount), and the new elements to add.

**Example:** const fruits = ["Apple", "Banana", "Orange"];

fruits.splice(1, 1, "Mango"); // Replace 1 element at index 1 with "Mango"

console.log(fruits); // Output: ["Apple", "Mango", "Orange"]

1. **Removing All Elements After a Specific Index:**

If deleteCount is omitted, all elements from start to the end of the array are removed.

**Example:**

const fruits = ["Apple", "Banana", "Orange", "Mango"];

const removed = fruits.splice(2); // Remove all elements starting at index 2

console.log(fruits); // Output: ["Apple", "Banana"]

console.log(removed); // Output: ["Orange", "Mango"]

1. **Using Negative Index:**

A negative start index counts from the end of the array.

**Example:**

const fruits = ["Apple", "Banana", "Orange", "Mango"];

fruits.splice(-2, 1); // Remove 1 element at the 2nd index from the end

console.log(fruits); // Output: ["Apple", "Banana", "Mango"]

**Object**

* In JavaScript, an object is a collection of key-value pairs, where each key (also called a property) is a string (or symbol) and each value can be any data type, including other objects, arrays, functions, etc. Objects are one of the most important data structures in JavaScript and are used to represent real-world entities, store data and organize code.

**Syntax:**

const person = {

name: "John",

age: 30,

isStudent: false,

};

**Properties vs Methods**

* **Properties:** Properties are key-value pairs where the value can be any data type (e.g., string, number, Boolean, array, object, etc). Properties store data or state associated with an object. Accessed using dot notation or bracket notation.

**Example:**   
const person = {

name: "John", // Property

age: 30, // Property

};

console.log(person.name); // Output: John

console.log(person["age"]); // Output: 30

* **Methods:** Methods are functions that are associated with an object. They are properties where the value is a function. Methods define behaviour or actions that an object can perform. Accesssedusingdot notation or bracket notation, and invoked using parentheses().

**Example:**

const person = {

name: "John",

greet: function () { // Method

console.log("Hello, " + this.name + "!");

},

};

person.greet(); // Output: Hello, John!

**Update an object**

* In JavaScript, you can update an object by modifying its existing properties, adding new properties, or deleting properties. Objects are mutable, meaning you can change them after they are created.

1. **Updating Existing Properties:**

You can update the value of an existing property using **dot notation** or **bracket notation**.

**Example:**

const person = {

name: "John",

age: 30,

};

// Update the 'age' property

person.age = 31;

console.log(person); // Output: { name: "John", age: 31 }

1. **Adding New Properties:**

You can add new properties to an object by assigning a value to a new key.

**Example:**

const person = {

name: "John",

};

// Add a new property

person.age = 30;

console.log(person); // Output: { name: "John", age: 30 }

1. **Deleting Properties:**

You can remove a property from an object using the delete operator.

**Example:**

javascript

Copy

const person = {

name: "John",

age: 30,

};

// Delete the 'age' property

delete person.age;

console.log(person); // Output: { name: "John" }

**Typeof**

* In JavaScript, typeof is an operator used to determine the type of a given variable or expression. It returns a string indicating the type of the operand.

|  |
| --- |
| **Example:**  console.log(typeof 42); // "number"  console.log(typeof "hello"); // "string"  console.log(typeof true); // "boolean"  console.log(typeof {}); // "object"  console.log(typeof []); // "object" (arrays are objects)  console.log(typeof function(){});// "function" |

**Higher-Order Functions**

* A **Higher-Order Function (HOF)** is a function that can do **one or both** of the following:

1. **Takes one or more functions as arguments.**
2. **Returns a function as its result.**

* JavaScript treats functions as **first-class citizens**, meaning functions can be passed around like variables — enabling higher-order functions.

|  |
| --- |
| **Example:**  function operate(a, b, callback) {  return callback(a, b);  }  function add(x, y) {  return x + y;  }  function multiply(x, y) {  return x \* y;  }  console.log(operate(5, 3, add)); // Output: 8  console.log(operate(5, 3, multiply)); // Output: 15 |

**Constructor Function**

* A constructor function in JavaScript is a special function used to create and initialize objects. It acts like a blueprint for creating multiple objects with the same properties and methods.
* In JavaScript, constructor functions are usually named with a capital letter to distinguish them from regular functions.

|  |
| --- |
| **Example:**  function Person(name, age) {  this.name = name; // 'this' refers to the new object  this.age = age;  console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);  }  const person1 = new Person("Aman", 25);  const person2 = new Person("Sneha", 27);  **Output:**  Hello, my name is Aman and I am 25 years old.  Hello, my name is Sneha and I am 27 years old. |

**Prototype**

* Using **prototype** is not mandatory, but it is a more efficient way to manage memory and structure your code.

**Memory Efficiency**

* When you define methods inside a constructor function without using prototype, every object created will get its **own copy** of the method.
* This wastes memory if you create many objects.

|  |
| --- |
| **Example:** function Person(name) {  this.name = name;  }  Person.prototype.greet = function() {  console.log(`Hello, my name is ${this.name}`);  };  const person1 = new Person("Aman");  const person2 = new Person("Neha");  console.log(person1.greet === person2.greet);  **Output:**  true (Shared method) |

✔️ **Solution:** Both objects share the same function from the prototype, saving memory.

**First-Class Function**

* In JavaScript, **first-class functions** mean that **functions are treated like variables**. You can:  
  ✅ Assign functions to variables.  
  ✅ Pass functions as arguments.  
  ✅ Return functions from other functions.

This makes JavaScript a **functional programming** language.

|  |
| --- |
| **Example:**  function greetUser(a) {  console.log("Executing callback function...");  a();  }  greetUser(function() {  console.log("Hello, welcome to JavaScript!");  });  **Output:**  Executing callback function...  Hello, welcome to JavaScript! |

**new Keyword**

* The **new** keyword in JavaScript is used to **create a new object** from a **constructor function** or a class. It performs a series of steps to set up the object and return it.

|  |
| --- |
| **Example:**  function Person(name, age) {  this.name = name;  this.age = age;  }  const person1 = new Person("Aman", 25);  const person2 = new Person("Sneha", 27);  console.log(person1);  console.log(person2);  **Output:**  Person { name: 'Aman', age: 25 }  Person { name: 'Sneha', age: 27 } |

**IIFE (Immediately Invoked Function Expression)**

* An IIFE (Immediately Invoked Function Expression) is a function that is executed immediately after it is defined. It is often used to create a private scope and prevent variable pollution in the global scope.

|  |
| --- |
| **Syntax of IIFE**  (function() {  console.log("IIFE executed!");  })();  ✔️ **Explanation:**   * **(function() {})** → Function Expression (Wrapped in parentheses). * **()()** → Immediately Invoked (The second pair of parentheses calls the function). |

**IIFE with Setter and Getter**

* You can use IIFE (Immediately Invoked Function Expression) to create private variables using closures, and then provide getter and setter functions to access and modify them.

|  |
| --- |
| **Example:**  var ans = (function() {  var privateVal = 12; // Private variable  return {  // Getter function to display the value  getter: function() {  console.log("Current Value:", privateVal);  },  // Setter function to update the value  setter: function(val) {  privateVal = val;  console.log("Value Updated!");  }  };  })();  // Using the Getter and Setter  ans.getter();  **Output:** Current Value: 12  ans.setter(30);  **Output:** Value Updated!  ans.getter();  **Output:** Current Value: 30 |

**[[Prototype]]: Object in JavaScript**

* In JavaScript, every object has a hidden internal property called **[[Prototype]]**.
* It refers to the **prototype** from which the object inherits properties and methods.
* This is part of JavaScript’s **prototypal inheritance** model.
* You can access an object's prototype using Object.getPrototypeOf() or .prototype for constructor functions.

|  |
| --- |
| **Example:**  const obj = { name: 'Amandeep' };  console.log(obj.name); // Output: Amandeep  console.log(Object.getPrototypeOf(obj)); // Output: [Object: null prototype] {} |

**Explanation:**

* obj is a simple object with a name property.
* Object.getPrototypeOf(obj) shows its prototype.
* The [[Prototype]] is set to Object.prototype by default.

**What is Object.prototype?**

* Object.prototype is the **parent object** of all objects in JavaScript.
* It contains built-in methods that every object can access.

**Common Methods in Object.prototype**

* Here are some commonly used methods inherited from Object.prototype:

|  |  |  |
| --- | --- | --- |
| Method | Description | Example |
| hasOwnProperty() | Checks if an object has a specific property. | obj.hasOwnProperty('name'); // true |
| toString() | Converts the object to a string. | obj.toString(); // [object Object] |
| valueOf() | Returns the primitive value of the object. | obj.valueOf(); // { name: 'Amandeep' } |
| isPrototypeOf() | Checks if an object is in the prototype chain. | Object.prototype.isPrototypeOf(obj); // true |
| propertyIsEnumerable() | Checks if a property is enumerable. | obj.propertyIsEnumerable('name'); // true |
| defineProperty() | Adds or modifies a property on an object. | Object.defineProperty(obj, 'age', { value: 25 }); |
| keys() | Returns an array of the object's keys. | Object.keys(obj); // ['name'] |
| entries() | Returns an array of key-value pairs. | Object.entries(obj); // [['name', 'Amandeep']] |

|  |
| --- |
| **Example:**  const person = { name: 'Amandeep' };  // Even though we didn't define a method on 'person', it can use Object.prototype methods  console.log(person.hasOwnProperty('name')); // true  console.log(person.toString()); // [object Object]  console.log(Object.getPrototypeOf(person) === Object.prototype); // true  **Explanation:**   * Since **person** inherits from **Object.prototype**, it has access to these built-in methods. * The **[[Prototype]]** is like a chain leading back to **Object.prototype**. |

**Prototypal Inheritance**

* Prototypal Inheritance is a feature in JavaScript that allows objects to inherit properties and methods from another object through their prototype.

**How It Works**

* Every object in JavaScript has a hidden property called [[Prototype]].
* This property links to another object (its prototype).
* If a property or method is not found on an object, JavaScript looks up the prototype chain to find it.

|  |
| --- |
| **Example:** var human = {  canFly: false,  canTalk: true,  canWalk: true,  haveEmotions: true,  hasFourLegs: false  };  var Amandeep = {  canMakeWebsite: true,  canMakeAnimations: true  };  // Set prototype using \_\_proto\_\_  Amandeep.\_\_proto\_\_ = human;  **Output:**  // Access inherited properties  console.log(Amandeep.canTalk); // true (inherited from human)  console.log(Amandeep.canFly); // false (inherited from human)  console.log(Amandeep.canMakeWebsite); // true (own property)  console.log(Amandeep.haveEmotions); // true (inherited from human) |

**this Keyword**

* In JavaScript, this refers to the current object that is executing the code. The value of this depends on how and where the function is called.

**1. Global Context (Outside Any Function)**

In the global context, this refers to the **global object**.

* In browsers, it’s **window**.
* In Node.js, it’s **global**.

|  |
| --- |
| **Example:**  **console.log(this); // In browser: Window object** |

**2. Inside an Object Method**

When used inside an object’s method, this refers to the **object itself**.

|  |
| --- |
|  |

**3. In a Constructor Function**

In constructor functions, this refers to the **new object** being created.

|  |
| --- |
| **Example:**  function Person(name) {  this.name = name;  this.sayName = function() {  console.log(`My name is ${this.name}`);  };  }  const person1 = new Person("Amandeep");  person1.sayName();  Output:  My name is Amandeep |

**Explanation:**

* this.name refers to person1.name.

**4. In Arrow Functions**

* Arrow functions **don’t have their own this**.
* They inherit this from their surrounding scope.

|  |
| --- |
| **Example:**  const person = {  name: "Amandeep",  greet: () => {  console.log(`Hello, I am ${this.name}`);  }  };  person.greet();  Output:  Hello, I am undefined |

**Explanation:**

* Arrow functions use the this from their **parent scope**, which in this case is the global object.

**5. Using this in Event Handlers**

* In DOM event handlers, this refers to the element that triggered the event.

|  |
| --- |
| **Example:**  <button id="btn">Click Me</button>  <script>  document.getElementById('btn').onclick = function() {    this.style.backgroundColor = 'green'; // Refers to the button element    console.log(this); // Refers to the button element  };  </script> |

**Summary of this**

|  |  |
| --- | --- |
| Context | Value of this |
| Global (Browser) | window |
| Object Method | Object itself |
| Constructor Function | New Object created |
| Arrow Function | Parent Scope this |
| Event Handler | Element that triggered the event |

**call() Method**

* The **call()** method is used to **invoke a function** with a specific this value and pass arguments **individually**.

|  |
| --- |
| **Example:**  const person1 = { name: 'Amandeep', age: 25 };  const person2 = { name: 'Sneha', age: 27 };  function greet() {  console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);  }  // Using call()  greet.call(person1);  greet.call(person2);  **Output:**  Hello, my name is Amandeep and I am 25 years old.  Hello, my name is Sneha and I am 27 years old. |

**Explanation:**

* call() sets this to person1 and person2, allowing the greet function to access their name properties.

|  |
| --- |
| **Using call() with Multiple Arguments**  **Example:**  function introduce(city, country) {  console.log(`Hi, I'm ${this.name} from ${city}, ${country}.`);  }  const person = { name: "Amandeep" };  // Using call() with multiple arguments  introduce.call(person, "Pune", "India");  **Output:**  Hi, I'm Amandeep from Pune, India. |

**bind() Method**

* The **bind()** method creates a **new function** that, when called, has its this value set to a specified object.
* Unlike call() and apply(), it **does not execute the function immediately**.
* Instead, it **returns a new function** that you can call later.

|  |
| --- |
| **Example 1: Using bind() to Set this** const person = {  name: "Amandeep",  greet: function() {  console.log(`Hello, my name is ${this.name}`);  }  };  person.greet()  // Creating person2 const person2 = { name: "Sneha" };  // Bind person2 to the greet function  const greetSneha = person.greet.bind(person2);  greetSneha();  **Output:** Hello, my name is Amandeep  Hello, my name is Sneha |

|  |
| --- |
| **Example 2: Using bind() with Arguments**  function introduce(city, country) {  console.log(`Hi, I'm ${this.name} from ${city}, ${country}.`);  }  const person = { name: "Amandeep" };  // Bind with arguments  const intro = introduce.bind(person, "Pune", "India");  intro();  Output:  Hi, I'm Amandeep from Pune, India. |

**Explanation:**

* bind() pre-fills city and country arguments, known as **partial application**.

**apply() Method**

* + The **apply()** method is similar to call(), but it takes arguments as an **array** instead of individual values.

|  |
| --- |
| **Example 1: Using apply() to Call a Function**  function introduce(city, country) {  console.log(`Hi, I'm ${this.name} from ${city}, ${country}.`);  }  const person = { name: "Amandeep" };  // Using apply() with an array of arguments  introduce.apply(person, ["Pune", "India"]);  **Output:**  Hi, I'm Amandeep from Pune, India. |

|  |
| --- |
| **Example 2: Function Borrowing with apply()**  const student = {  name: "Aman",  age: 22,  showDetails: function(city) {  console.log(`${this.name} is ${this.age} years old and lives in ${city}.`);  }  };  const teacher = {  name: "Mr. Raj",  age: 45  };  // Borrowing showDetails using apply  student.showDetails.apply(teacher, ["Delhi"]);  **Output:**  Mr. Raj is 45 years old and lives in Delhi. |

**Explanation:** The showDetails method is borrowed from student and applied to teacher.

**Key Differences Between bind(), call(), and apply()**

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Execution** | **Arguments** | **Returns** |
| **bind()** | Doesn't execute immediately | Passed individually or none | New function with bound this |
| **call()** | Executes immediately | Passed individually | Result of the function |
| **apply()** | Executes immediately | Passed as an array | Result of the function |

**Pure and Impure Functions**

* In JavaScript, functions are categorized into Pure and Impure functions based on their behaviour.

**1. Pure Functions**

A **pure function** is a function that:

* **Always returns the same output** for the same input.
* **Has no side effects** (doesn’t modify external data or state).

|  |
| --- |
| **Example:**  function add(a, b) {  return a + b; // No external changes, same output for same input  }  console.log(add(2, 3)); // output: 5  console.log(add(2, 3)); // output: 5 |

**2. Impure Functions**

An **impure function** is a function that:

* **May return different outputs** for the same input.
* **Has side effects** (modifies external variables, logs data, or performs DOM manipulations).

|  |
| --- |
| **Example:** let total = 10;  function addToTotal(value) {  total += value; // Modifies external state  return total;  }  console.log(addToTotal(5)); // 15  console.log(addToTotal(5)); // 20 (Different output for same input) |

**Explanation:**

* The function modifies the global variable total.
* The output is not consistent for the same input.

**Closure in JavaScript**

* A **closure** is created when a function "remembers" and has access to its **outer variables** even after the outer function has finished executing.
* A **closure** is a combination of a function and its **lexical environment** (the surrounding state or variables) in which it was created.

|  |
| --- |
| **Example:**  function outerFunction(outerVariable) {  return function innerFunction(innerVariable) {  console.log(`Outer: ${outerVariable}, Inner: ${innerVariable}`);  };  }  const closureExample = outerFunction("Hello");  // Calling the inner function  closureExample("World");  Output:  Outer: Hello, Inner: World |

**Synchronous and Asynchronous**

**Synchronous**

* In JavaScript, synchronous means tasks are executed one after another in a sequential order. Each operation waits for the previous one to complete before moving on to the next.

|  |
| --- |
| **Example:**  console.log("Start");  function greet() {  console.log("Hello, Amandeep!");  }  greet();  console.log("End");  **Output:** Start  Hello, Amandeep!  End  **Explain:**   * Each line is executed **one by one** in a blocking manner. |

**Asynchronous in JavaScript**

* In **JavaScript**, **asynchronous** programming allows tasks to run **in the background** without blocking the execution of the rest of the code.
* This is useful when tasks like API calls, file reading, or database queries are involved since they can take time.

|  |
| --- |
| **Example:**  console.log("Start");  // Asynchronous operation using setTimeout  setTimeout(() => {  console.log("This is an asynchronous task.");  }, 2000);  console.log("End");  **Output:** |

* **Key Differences Between Synchronous and Asynchronous Code**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Synchronous** | **Asynchronous** |
| **Execution** | Blocks further code until task completes | Allows other code to run while waiting |
| **Performance** | Slower for time-consuming tasks | Faster for tasks like API calls |
| **Complexity** | Easier to understand | May involve callbacks, promises, or async/await |
| **Use Cases** | Simple calculations, loops | Network requests, file I/O, timers |

* **Synchronous Vs Asynchronous Code (Identify Synchronous vs Asynchronous Code)**
* Without asynchronous methods, everything is synchronous.
* Use synchronous code only when it won’t cause performance issues.
* For network calls, file handling, or database operations, prefer using async/await, Promises, or callbacks.
* **Asynchronous methods or techniques**

|  |  |  |
| --- | --- | --- |
| **Method** | **Category** | **Description** |
| **setTimeout()** | Web API (Timer) | Runs a function once after a delay |
| **setInterval()** | Web API (Timer) | Runs a function repeatedly at intervals |
| **Promise** | Async Management | Represents a future value |
| **async/await** | Async Management | Cleaner way to handle promises |
| **Callback** | Async Control Flow | Executes a function after a task completes |
| **fetch()** | HTTP Request (Web API) | Makes API requests using promises |
| **axios()** | HTTP Request (Library) | Makes API requests (easier than fetch) |
| **XMLHttpRequest** | HTTP Request (Legacy API) | Old method for making API calls |

* **Synchronous:**if you don't use any asynchronous methods or techniques, then your code will be synchronous.
* **JS is not Asynchronous**

JavaScript itself is not asynchronous. It is a single-threaded, synchronous language, meaning it executes one task at a time in a sequential manner.

However, JavaScript can behave asynchronously using the Web APIs provided by the browser or the Node.js runtime. These APIs allow asynchronous tasks like network requests, timers, or file reading to be handled without blocking the main thread.

* **Why Does JavaScript Appear Asynchronous?**

**JavaScript uses:**

1. Event Loop
2. Callback Queue
3. Task Queue and Microtask Queue

(These components enable non-blocking operations by offloading tasks to the browser or Node.js environment.)

* **How It Works?**

1. **Single-Threaded Execution:**

JavaScript runs code line by line.

1. **Delegation to Web APIs:**

When an asynchronous function is called (setTimeout(), fetch(), XMLHttpRequest()), it is handled by browser Web APIs, not JavaScript itself. Web APIs manage the asynchronous operations in the background without blocking the main thread.

* **Task Queue (Callback Queue)**

Once an asynchronous task is completed, its callback function is added to the Task Queue, waiting for execution. It waits here until the Call Stack is free.

* **Microtask Queue**

- Callbacks from Promises and async/await are placed in the Microtask Queue.

- Microtasks have higher priority than normal callbacks (Task Queue).

- Before moving to the Task Queue, JavaScript clears all pending Microtasks.

1. **Event Loop Management:**

The event loop monitors the call stack(Main Stack) and moves completed tasks from the task queue(Side Stack) to the call stack(Main Stack) when it’s empty.

1. **Non-Blocking Execution:**

While waiting for asynchronous tasks, JavaScript continues executing other code.

A diagram of a software project

AI-generated content may be incorrect.

|  |
| --- |
| **Example:**  console.log("Start");  // Asynchronous Task using setTimeout (Task Queue)  setTimeout(() => console.log("setTimeout Callback"), 0);  // Microtask using Promise (Microtask Queue)  Promise.resolve().then(() => console.log("Promise Resolved"));  console.log("End");  **Output:** |

**Single Thread vs Multi Thread**

**Single Thread**

* JavaScript is single threaded, meaning it can execute one task at a time using a single Call Stack(Main Stack).

**Behavior:**

* One task must complete before the next task begins.
* It follows a synchronous execution model by default.
* Asynchronous operations are handled using Web APIs and the Event Loop.

**Use Case:**

* Ideal for tasks that involve UI interactions, light data processing, or web applications.

|  |
| --- |
| **Example:**  console.log("Start");  console.log("Processing...");  console.log("End");  **Output:**  Start  Processing...  End |

**Multi Thread**

* Multi-threaded languages like Java, C++, Python can run multiple tasks simultaneously by using multiple threads.

**Behavior:**

* Tasks run in parallel.
* Suitable for CPU-intensive operations like video rendering, gaming, or machine learning.

**Use Case:**

* Useful for applications that require high computation or background processing.

**Example (Conceptual)**:

* **Thread 1** → Processing User Input
* **Thread 2** → Running Complex Calculation
* **Thread 3** → Fetching Data from API

(Multiple threads operate without blocking each other.)

**Key Differences Between Single Thread and Multi Thread**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Single Thread** | **Multi Thread** |
| **Execution** | One task at a time | Multiple tasks run in parallel |
| **Concurrency** | Achieved using asynchronous operations | Achieved using multiple threads |
| **Efficiency** | Suitable for I/O operations and UI management | Ideal for CPU-intensive tasks |
| **Blocking** | Blocking behavior possible without async code | Less blocking due to parallelism |
| **Memory Usage** | Uses less memory | Requires more memory for multiple threads |
| **Complexity** | Easier to manage and debug | More complex to manage and synchronize |

**Is JavaScript Always Single Threaded?**

* JavaScript is single threaded by design for simplicity and performance.
* However, it supports asynchronous operations using the Event Loop and Web APIs.
* For true multi-threading, JavaScript uses Web Workers which allow running scripts in background threads without blocking the main thread.

**Callbacks**

* A **callback** is a **function** that is passed as an argument to another function and is executed **later**.  
  It allows functions to **call back** or run in other functions after completing a task.

**In simple words:**

* A callback is a function inside a function.
* It is commonly used for handling asynchronous operations like fetching data or reading files.

|  |
| --- |
| **Example: Callback Function (Synchronous)**  function greet(name, callback) {  console.log(`Hello, ${name}!`);  callback(); // Calling the callback  }  function sayGoodbye() {  console.log("Goodbye!");  }  greet("Amandeep", sayGoodbye);  **Output:**  Hello, Amandeep!  Goodbye!  **Here, sayGoodbye() is a callback that runs after the greet() function.** |

|  |
| --- |
| **Example: Callback with Asynchronous Code**  function fetchData(callback) {  console.log("Fetching data...");  setTimeout(() => {  console.log("Data fetched successfully.");  callback();  }, 2000);  }  function processData() {  console.log("Processing data...");  }  fetchData(processData);  **Output:**    **The callback function processData() executes after the data is fetched using setTimeout.** |

**Why Use Callbacks?**

* **Handle Asynchronous Tasks:** Callbacks are essential for managing tasks that take time, like API calls.
* **Maintain Code Structure:** They keep code **organized** and readable.
* **Enable Reusability:** You can pass different callbacks for different tasks.

**Callback Hell (Issue with Callbacks)**

* When callbacks are nested too deeply, it creates **Callback Hell**, making the code hard to read and manage.

**Solution:** Use **Promises** or **async/await** instead of nested callbacks.

|  |
| --- |
| **Example:**  setTimeout(() => {  console.log("Task 1 completed");  setTimeout(() => {  console.log("Task 2 completed");  setTimeout(() => {  console.log("Task 3 completed");  }, 1000);  }, 1000);  }, 1000); |

**Promises**

* **A Promise in JavaScript is an object representing the eventual completion (success) or failure (error) of an asynchronous operation.**
* **Pending →** Initial state, neither resolved nor rejected.
* **Fulfilled (Resolved/Responses) →** The operation completed successfully.
* **Rejected →** The operation failed.

**Using .then() and .catch()**

* .then() → Executes when the promise is **fulfilled** (success).
* .catch() → Executes when the promise is **rejected** (error).

|  |
| --- |
| **Example Using .then() and .catch()**  const myPromise = new Promise((resolve, reject) => {  const success = true; // Change this to false to test rejection  setTimeout(() => {  if (success) {  resolve("Data fetched successfully!");  } else {  reject("Failed to fetch data!");  }  }, 2000);  });  myPromise  .then((result) => {  console.log("Success:", result);  })  .catch((error) => {  console.error("Error:", error);  });  **Output:** |

|  |
| --- |
| **Example: Chaining .then() for Multiple Operations**  const fetchDataPromise = new Promise((resolve, reject) => {  setTimeout(() => resolve(5), 1000);  });  fetchDataPromise  .then((num) => {  console.log("First .then:", num);  return num \* 2;  })  .then((num) => {  console.log("Second .then:", num);  return num + 10;  })  .then((num) => {  console.log("Third .then:", num);  })  .catch((error) => {  console.error("Error:", error);  });  **Output:** |

**async/await**

* async/await is a **modern way** to handle asynchronous operations in JavaScript.
* It makes asynchronous code look like **synchronous code**, improving readability and reducing callback nesting (**callback hell**).
* It is a cleaner alternative to using .then() and .catch() with Promises.

**How it Works**

* **async Function:**
  + Declares a function as asynchronous.
  + Always returns a **Promise**.
* **await Keyword:**
  + Pauses the execution until the Promise is **resolved**.
  + Must be used inside an async function.

|  |
| --- |
| **Example with async/await**  async function fetchData() {  try {  const response = await fetch('https://jsonplaceholder.typicode.com/posts/1');  if (!response.ok) {  throw new Error('Network error');  }  const data = await response.json();  console.log("Data:", data);  } catch (error) {  console.error("Error:", error.message);  }  }  fetchData(); |

**Key Benefits of async/await**

* **Improves Readability:** Makes code look cleaner like synchronous code.
* **Better Error Handling:** Using try...catch is straightforward.
* **Avoids Callback Hell:** No nested .then() or .catch().
* **Easier Debugging:** Errors are clearer with clean stack traces.

**Sending Requests and Handling Responses in Async JavaScript**

* In **asynchronous JavaScript**, you can send HTTP requests to servers and handle responses using various methods. Here's how it works:

**Step 1: Sending Requests**

When you want to **send data to a server** or **fetch data from a server**, you use HTTP methods like:

* GET → Retrieve data
* POST → Send data
* PUT/PATCH → Update data
* DELETE → Remove data

These requests can be sent using the following methods:

**a) fetch()**

* The most modern and widely used method.
* Returns a **Promise** that resolves with the response.

|  |
| --- |
| **Example:**  fetch('https://api.example.com/data')  .then(response => response.json())  .then(data => console.log(data))  .catch(error => console.error('Error:', error)); |

**b) XMLHttpRequest *(Old Method)***

* Used before fetch() was introduced.
* Requires more code to handle requests and responses.

|  |
| --- |
| **Example:**  const xhr = new XMLHttpRequest();  xhr.open('GET', 'https://api.example.com/data', true);  xhr.onload = function() {  if (xhr.status === 200) {  console.log(JSON.parse(xhr.responseText));  }  };  xhr.onerror = function() {  console.error('Request failed');  };  xhr.send(); |

**c) axios() *(Third-Party Library)***

* Easier than fetch(), with simpler syntax.
* Automatically handles JSON conversion.

|  |
| --- |
| **Example:**  axios.get('https://api.example.com/data')  .then(response => console.log(response.data))  .catch(error => console.error('Error:', error)); |

**Step 2: Handling Responses**

Once the request is completed, you can handle the response using:

* **Callbacks:**
* A traditional way of handling responses.
* The callback function is executed once the data is received.
* **Promises:**
* **.then()** for success and **.catch()** for errors.
* **async/await:**
* **await** waits for the promise to resolve.
* **try...catch** handles errors.

|  |
| --- |
| **Example: Handling with Callbacks**  function fetchData(callback) {  setTimeout(() => {  const data = { name: "Amandeep", age: 25 };  callback(null, data);  }, 2000);  }  function handleResponse(error, data) {  if (error) {  console.error("Error:", error);  } else {  console.log("Data received:", data);  }  }  fetchData(handleResponse);  **Explanation: The callback function handleResponse() handles both errors and data.** |

**Concurrency in JavaScript**

* Concurrency in JavaScript means the ability to execute multiple tasks at the same time, without blocking the main thread.
* JavaScript uses a single-threaded event loop to manage multiple operations asynchronously.
* It doesn't truly run multiple tasks simultaneously like a multi-threaded environment, but it can handle tasks concurrently using callbacks, Promises, async/await, and Web APIs. (use Main stack and Side stack)

**Key Concepts of Concurrency in JavaScript**

1. **Single-Threaded Nature:**

* JavaScript runs on a **single thread** (one line of code at a time).
* However, asynchronous operations like network requests, timers, or file I/O can run concurrently in the background.

1. **Event Loop:**

* The **event loop** ensures that completed tasks from the callback queue (e.g., setTimeout, fetch) are pushed to the call stack when it’s empty.

1. **Web APIs:**

* Functions like setTimeout(), fetch(), and XMLHttpRequest use **Web APIs** to perform tasks in the background without blocking the main thread.

1. **Task Queue & Microtask Queue:**

* **Task Queue:** Handles callbacks from setTimeout() or setInterval().
* **Microtask Queue:** Handles promises (.then() and catch()) and async/await. Microtasks are executed **before** tasks from the Task Queue.

|  |
| --- |
| **Example: Understanding Concurrency**  console.log("Start");  // Using Promise  const promise = new Promise((resolve) => {  setTimeout(() => resolve("Promise Resolved!"), 3000);  });  // Async function using await  async function fetchData() {  const result = await promise;  console.log(result);  }  fetchData();  console.log("End");  **Output:**  Start  End  Promise Resolved! |

**Parallelism in JavaScript**

* **Parallelism** is the ability to execute multiple tasks **at the same time** using multiple **CPU cores**.
* Unlike **concurrency**, which manages multiple tasks at once using a single thread, **parallelism** performs multiple tasks simultaneously using multiple threads or processes.

**Does JavaScript Support True Parallelism?**

* **JavaScript is Single-Threaded** → It has a single **main thread** and uses the **event loop** to manage asynchronous tasks.
* However, JavaScript achieves **parallelism** using:
* **Web Workers**
* **Worker Threads (Node.js)**
* **Offloading Tasks to Web APIs** (e.g., fetch, File I/O)

|  |  |  |
| --- | --- | --- |
| Aspect | Concurrency | Parallelism |
| Definition | Multiple tasks managed at once | Multiple tasks executed at the same time |
| Threads | Single thread with task switching | Multiple threads across different CPU cores |
| Example | Fetching data while displaying UI | Video rendering or image processing |
| Use Case | Non-blocking UI updates | CPU-intensive tasks like AI model training |
| Tools | Event Loop, Callbacks, Promises, async/await | Web Workers, Worker Threads, Cluster Module |

**Throttling in JavaScript**

* **Throttling** is a technique in JavaScript used to **limit the number of times** a function can execute within a specific period of time.
* It ensures that even if an event is triggered multiple times, the function will only execute **once** every defined time interval.
* It is useful for optimizing performance in scenarios where frequent event calls can slow down applications.

**Why Use Throttling?**

* Prevents **performance issues** caused by excessive function execution.
* **Reduces CPU usage** by limiting the number of function calls.
* Helpful for events like:
* **Window resizing**
* **Scrolling**
* **Mouse movements**
* **API calls**

**How Throttling Works**

* It uses **timers** (setTimeout) or **timestamps** to track when the function was last called.
* While a throttled function is waiting to execute again, further calls are **ignored** until the time limit is over.

**Final Thoughts**

* Use **Throttling** when you want to ensure a function is called at consistent intervals.
* It is best for **scrolling**, **resizing**, and **button click** events.
* Combine with other performance techniques like **Debouncing** for better efficiency.

**Debouncing in JavaScript**

* **Debouncing** is a technique used to **delay** the execution of a function until after a specified time has passed **since the last time** the function was called.
* It ensures that a function is executed **only once** after a particular event has **stopped firing** for a defined period.
* **Useful for reducing the number of unnecessary function calls** in scenarios where frequent events occur.

**Why Use Debouncing?**

* **Optimize Performance:** Prevents unnecessary executions during continuous events.
* **Avoid API Overload:** Minimizes the number of API requests during live searches.
* **Reduce Lag:** Ensures smooth UI experience by limiting computations.

**Common Use Cases:**

* **Search Suggestions:** Waiting for users to stop typing before sending an API request.
* **Window Resize Events:** Executing layout adjustments only after resizing stops.
* **Button Click Prevention:** Avoiding multiple rapid clicks leading to duplicate actions.

**How Debouncing Works**

* **Timer-Based Execution:**
* Each time the function is called, a timer (setTimeout) is reset.
* The function only executes if no further calls are made within the specified delay.
* **Prevents Overloading:**
* If the event is continuously triggered, the previous timer is cleared using clearTimeout().

|  |  |  |
| --- | --- | --- |
| Aspect | Debouncing | Throttling |
| Purpose | Executes a function after a delay | Executes a function at regular intervals |
| Event Handling | Best for search inputs, resizing | Best for scrolling, mouse movements |
| Frequency | Function runs once after user stops action | Function runs multiple times at set intervals |
| Control | Provides more control over function execution timing | Ensures a consistent rate of execution |

heap memory

**Event Handlers**

**Json**

**Ajax**

**Settimeout**

**setinterval**

fetch

bom

dom