

# 1. Introduction to JavaScript

## ✓ What is JavaScript?

### Definition:

JavaScript (JS) is a high-level, interpreted programming language used to create interactive web applications. It is widely used for front-end (client-side) and back-end (server-side) development.

### Key Features of JavaScript:

- **Dynamic Typing** – Variables do not have fixed data types.
- **Prototype-Based OOP** – Uses prototypes instead of traditional class-based inheritance.
- **First-Class Functions** – Functions can be assigned to variables, passed as arguments, and returned from other functions.
- **Event-Driven & Asynchronous** – Supports event handling and asynchronous programming using callbacks, promises, and `async/await`.
- **Runs on Browsers & Servers** – JavaScript runs in browsers using the **JavaScript Engine** (like V8 in Chrome) and on servers using **Node.js**.

### Example:

```
console.log("Hello, JavaScript!"); // Output: Hello, JavaScript!
```

---

## ✓ History and Evolution (ES5, ES6, and beyond)

### Brief History of JavaScript:

- **1995** – JavaScript was created by **Brendan Eich** at Netscape in just 10 days.
- **1996-1997** – JavaScript was standardized as **ECMAScript (ES1)**.
- **2009 (ES5)** – Introduced `strict mode`, JSON support, and better array methods.
- **2015 (ES6/ES2015)** – Introduced `let`, `const`, arrow functions, classes, and template literals.
- **2016-2023 (ES7-ES14)** – Continued improvements like optional chaining, `async/await`, and `BigInt`.

### Major ECMAScript (ES) Features Over Time:

Version	Year	Major Features
ES5	2009	<code>strict mode</code> , <code>JSON.parse()</code> , <code>Array.map()</code> , <code>Array.reduce()</code>

ES6	2015	<code>let, const</code> , arrow functions, classes, template literals, promises
ES7	2016	<code>Array.includes()</code> , Exponentiation Operator ( <code>**</code> )
ES8	2017	<code>async/await</code> , <code>Object.entries()</code> , <code>Object.values()</code>
ES9+	2018+	Rest/Spread operators, Optional Chaining ( <code>?.</code> ), Nullish Coalescing ( <code>??</code> )

### Example (ES6 Features):

```
let name = "Swaraj";
const greet = () => `Hello, ${name}!`;
console.log(greet()); // Output: Hello, Swaraj!
```

---

## ✓ How JavaScript Works in Browsers and Node.js

### 1 JavaScript in Browsers

- JavaScript runs inside browsers using **JavaScript engines** like:
  - **V8** (Chrome, Edge, Node.js)
  - **SpiderMonkey** (Firefox)
  - **JavaScriptCore** (Safari)
- The browser provides **Web APIs** (like DOM, Fetch, Storage) to interact with HTML, CSS, and other web elements.

### 2 JavaScript in Node.js

- **Node.js** allows JavaScript to run outside the browser (on servers).
- Uses **V8 Engine** but provides additional APIs (File System, HTTP, Database support).
- Enables **backend development**, API creation, and package management with **npm**.

### Example of JavaScript in the Browser (Console in Chrome DevTools):

```
document.body.style.backgroundColor = "lightblue";
console.log("JavaScript running in the browser!");
```

### Example of JavaScript in Node.js:

```
// Run this in Node.js
console.log("JavaScript running in Node.js!");
```

---

## ✓ Setting Up the Development Environment (VS Code, Node.js)

### Step 1: Install VS Code

- Download and install **VS Code** from: <https://code.visualstudio.com/>

### Step 2: Install Node.js

- Download and install **Node.js** from: <https://nodejs.org/>

Verify installation:

```
node -v    # Check Node.js version
npm -v     # Check npm version
```

- 

### Step 3: Create a JavaScript File

1. Open **VS Code**.
2. Create a new file **script.js**.

Write the following code:

```
console.log("JavaScript is working!");
```

- 3.

Run it in **Node.js Terminal**:

```
node script.js
```

---

## ✓ Writing Your First JavaScript Program (**console.log**)

The `console.log()` method is used to print messages to the console.

### Example 1 – Simple Output:

```
console.log("Welcome to JavaScript!");
```

### Output:

Welcome to JavaScript!

### Example 2 – Logging Variables:

```
let name = "Swaraj";  
console.log("My name is", name);
```

### Output:

My name is Swaraj

### Example 3 – Debugging with console.log:

```
let a = 10;  
let b = 20;  
console.log("Sum:", a + b); // Output: Sum: 30
```



## Summary

Topic	Key Takeaways
<b>What is JavaScript?</b>	A powerful scripting language for web development, frontend & backend.
<b>History &amp; Evolution</b>	JavaScript evolved from ES1 to modern ES14 with advanced features.
<b>How JS Works</b>	Runs in browsers via JavaScript engines; can also run on servers via Node.js.
<b>Development Setup</b>	Install <b>VS Code</b> & <b>Node.js</b> to write and execute JS programs.
<b>First JS Program</b>	Use <code>console.log()</code> to display output in the console.

## 2. JavaScript Basics

### ✓ Variables (**var**, **let**, **const**)

#### What are Variables?

A **variable** is a container for storing data values. JavaScript allows us to declare variables using **var**, **let**, and **const**.

#### 1 **var** (Old way, avoid using)

- Variables declared with **var** are **function-scoped**.
- Can be **redeclared** and **updated**.
- Can be accessed **before declaration** due to **hoisting** (with **undefined** value).

```
var x = 10;  
console.log(x); // 10
```

```
var x = 20; // Redeclaration allowed  
console.log(x); // 20
```

---

#### 2 **let** (Modern way, use this for changeable values)

- **Block-scoped** (Exists only inside **{ }**).
- Cannot be **redeclared** in the same scope.
- Can be **updated**.

```
let name = "Swaraj";  
console.log(name); // Swaraj
```

```
// let name = "Jadhav"; // ✗ Error: Cannot redeclare  
name = "Jadhav"; // ✓ Allowed  
console.log(name); // Jadhav
```

---

### 3 **const** (Use for constants, cannot change)

- **Block-scoped** like **let**.
- Cannot be **redeclared** or **updated**.

```
const PI = 3.1416;  
// PI = 3.14; // ❌ Error: Cannot assign to 'PI'  
console.log(PI); // 3.1416
```

#### 📌 Best Practice:

- Use **const** when the value won't change.
  - Use **let** when you need to update values.
  - Avoid **var** because of scoping issues.
- 

## ✅ Data Types (Primitive & Non-Primitive)

### 1 Primitive Data Types (Immutable - Stores Single Value)

Primitive types are stored **by value**.

Type	Example
String	"Hello"
Number	100, 3.14
Boolean	true, false
Undefined	let x; (default value is undefined)
Null	let y = null; (intentional absence of value)
BigInt	BigInt(12345678901234567890n) (for large numbers)
Symbol	Symbol('id') (unique identifier)

```
let a = "JavaScript"; // String
let b = 42;           // Number
let c = true;         // Boolean
let d;                // Undefined
let e = null;         // Null
let f = 12345678901234567890n; // BigInt
let g = Symbol("id"); // Symbol

console.log(typeof a, typeof b, typeof c, typeof d, typeof e, typeof f, typeof g);
```

---

## 2 Non-Primitive Data Types (Reference Types - Mutable)

Non-primitive types are stored **by reference** (stored in memory and referenced).

Type	Example
Object	<code>{name: "Swaraj", age: 25}</code>
Array	<code>["JS", "React", "Node"]</code>
Function	<code>function greet() { return "Hello"; }</code>

```
let person = { name: "Swaraj", age: 25 }; // Object
let numbers = [1, 2, 3, 4, 5]; // Array
let greet = function() { return "Hello, World!"; }; // Function

console.log(typeof person, typeof numbers, typeof greet);
```

---

## ✓ Operators in JavaScript

### 1 Arithmetic Operators

Used for mathematical calculations.

Operator	Description	Example
+	Addition	10 + 5 // 15
-	Subtraction	10 - 5 // 5
*	Multiplication	10 * 5 // 50
/	Division	10 / 5 // 2
%	Modulus (Remainder)	10 % 3 // 1
**	Exponentiation	2 ** 3 // 8

```
let x = 10;
let y = 3;
console.log(x + y, x - y, x * y, x / y, x % y, x ** y);
```

---

## 2 Logical Operators

Used to perform logical operations.

Operator	Description	Example
&&	AND	(true && false) // false
,	,	,
!	NOT	!true // false

```
let isJS = true;
let isPython = false;
```



```
console.log(isJS && isPython, isJS || isPython, !isJS);
```

---

### 3 Comparison Operators

Used to compare values.

Operator	Description	Example
<code>==</code>	Equal to (loose comparison)	<code>"5" == 5 // true</code>
<code>===</code>	Strict Equal (checks type too)	<code>"5" === 5 // false</code>
<code>!=</code>	Not Equal	<code>"5" != 5 // false</code>
<code>!==</code>	Strict Not Equal	<code>"5" !== 5 // true</code>
<code>&gt;</code>	Greater than	<code>10 &gt; 5 // true</code>
<code>&lt;</code>	Less than	<code>10 &lt; 5 // false</code>

```
console.log(5 == "5", 5 === "5", 5 !== "5", 10 > 5, 10 < 5);
```

---

### 4 Bitwise Operators

Work at the binary level.

Operator	Example
<code>&amp;</code> (AND)	<code>5 &amp; 1 // 1</code>
<code> </code>	<code> </code> (OR)

```
^ (XOR)      5 ^ 1 //
              4

<< (Left Shift) 5 << 1 //
              10

>> (Right Shift) 5 >> 1 //
              2
```

---

## ✓ Type Conversion & Type Coercion

### Type Conversion (Explicit Conversion)

- Manually convert one type to another using `String()`, `Number()`, `Boolean()`.

```
let num = "10";
console.log(Number(num), String(10), Boolean(1));
```

### Type Coercion (Implicit Conversion)

- JavaScript automatically converts types.





```
console.log("5" + 3); // "53" (Number converted to String)
console.log("5" - 3); // 2 (String converted to Number)
console.log(true + 1); // 2 (Boolean converted to Number)
```

---

## ✓ Template Literals (``)

Allows embedding variables inside strings using **backticks** ( ``` ) and `${}`.

```
let name = "Swaraj";
let age = 25;
console.log(`Hello, my name is ${name} and I am ${age} years old.`);
```

-  **Benefits of Template Literals:**  Multi-line strings
-  Variable embedding
  -  Easy-to-read syntax

```
let message = `This is  
a multi-line  
string!`;  
console.log(message);
```

---

## Summary

Topic	Key Takeaways
Variables	Use <code>let</code> for changeable values, <code>const</code> for constants, avoid <code>var</code> .
Data Types	Primitive types store single values, non-primitive types store references.
Operators	Arithmetic, logical, comparison, and bitwise operators are used for operations.
Type Conversion	JavaScript automatically converts types, but explicit conversion is recommended.
Template Literals	Use <code>` `</code> for string interpolation and multi-line strings.

# 3. Control Flow & Conditions in JavaScript

Control flow in JavaScript determines the **order of execution** of statements. Conditional statements allow us to execute **different code blocks based on conditions**.

---

## ✓ **if, else if, else** Statements

### What is an **if** statement?

An **if** statement runs a block of code **only if a condition is true**.

### Syntax

```
if (condition) {  
    // Code executes if condition is true  
}
```

### Example

```
let age = 18;  
  
if (age >= 18) {  
    console.log("You are eligible to vote.");  
}
```

- ◆ Here, since `age >= 18` is **true**, the message is printed.
- 

### **if...else** Statement

An **else** block executes when the **if** condition is **false**.

### Syntax

```
if (condition) {  
    // Executes if condition is true  
} else {  
    // Executes if condition is false  
}
```

## Example

```
let num = 10;  
  
if (num > 0) {  
    console.log("Positive number");  
} else {  
    console.log("Negative number");  
}
```

---

## if...else if...else (Multiple Conditions)

When there are **multiple conditions**, we use **else if**.

## Syntax

```
if (condition1) {  
    // Executes if condition1 is true  
} else if (condition2) {  
    // Executes if condition2 is true  
} else {  
    // Executes if none of the above conditions are true  
}
```

## Example

```
let score = 85;  
  
if (score >= 90) {
```

```
    console.log("Grade: A");  
  } else if (score >= 80) {  
    console.log("Grade: B");  
  } else if (score >= 70) {  
    console.log("Grade: C");  
  } else {  
    console.log("Grade: F");  
  }  
}
```

- ♦ The second condition `score >= 80` is `true`, so "Grade: B" is printed.
- 

## ✅ Ternary Operator (`? :`)

A **ternary operator** is a shorthand for `if...else` statements. It **returns a value** based on a condition.

### Syntax

```
condition ? value_if_true : value_if_false;
```

### Example

```
let age = 20;  
let canVote = (age >= 18) ? "Yes, you can vote" : "No, you cannot  
vote";  
  
console.log(canVote);
```

- ♦ If `age >= 18` is `true`, it returns "Yes, you can vote", otherwise "No, you cannot vote".
- 

## Using Ternary for Multiple Conditions

```
let score = 85;
let grade = (score >= 90) ? "A" :
            (score >= 80) ? "B" :
            (score >= 70) ? "C" : "F";

console.log(`Grade: ${grade}`);
```

- ♦ Equivalent to `if...else if...else`, but **more concise**.
- 

## ✓ **switch Statement**

A **switch statement** is used when multiple conditions depend on a **single value**.

### **Syntax**

```
switch (expression) {
  case value1:
    // Code block for value1
    break;
  case value2:
    // Code block for value2
    break;
  default:
    // Default code (if no cases match)
}
```

### **Example**

```
let day = 3;

switch (day) {
  case 1:
    console.log("Monday");
    break;
  case 2:
```

```
        console.log("Tuesday");
        break;
    case 3:
        console.log("Wednesday");
        break;
    case 4:
        console.log("Thursday");
        break;
    default:
        console.log("Invalid day");
}
```

- ◆ Since `day = 3`, "Wednesday" is printed.
- 

## switch vs. if...else

- ✓ Use **switch** when comparing one variable against multiple values.
- ✓ Use **if...else** when comparing different conditions.

### Example (Using if...else)

```
let fruit = "Apple";

if (fruit === "Apple") {
    console.log("It is an Apple");
} else if (fruit === "Banana") {
    console.log("It is a Banana");
} else {
    console.log("Unknown fruit");
}
```

### Example (Using switch)

```
let fruit = "Apple";
```



```
switch (fruit) {  
  case "Apple":  
    console.log("It is an Apple");  
    break;  
  case "Banana":  
    console.log("It is a Banana");  
    break;  
  default:  
    console.log("Unknown fruit");  
}
```

---

## Summary

Feature	Description
<code>if...else</code>	Executes code based on conditions
<b>Ternary (<code>? :</code>)</b>	Shorter <code>if...else</code>
<code>switch</code>	Best for multiple fixed-value cases

## 4. Loops & Iteration in JavaScript

Loops allow us to execute a block of code **multiple times**. They are used to iterate over **arrays**, **objects**, or a **set number of times**.

---

### ✓ for, while, do-while Loops

#### 1 for Loop

A **for** loop is used when we know **how many times** to iterate.

##### Syntax

```
for (initialization; condition; increment/decrement) {  
    // Code to execute  
}
```

##### Example

```
for (let i = 1; i <= 5; i++) {  
    console.log("Iteration:", i);  
}
```

♦ Outputs:

Makefile

```
Iteration: 1  
Iteration: 2  
Iteration: 3  
Iteration: 4  
Iteration: 5
```

---

#### 2 while Loop

A **while** loop **runs until** a condition becomes **false**.

## Syntax

```
while (condition) {  
    // Code executes as long as condition is true  
}
```

## Example

```
let count = 1;  
  
while (count <= 5) {  
    console.log("Count:", count);  
    count++;  
}
```

- ♦ Runs until **count** exceeds 5.
- 

## 3 do-while Loop

A **do-while** loop **always runs at least once**, because the condition is checked **after** execution.

## Syntax

```
do {  
    // Code executes at least once  
} while (condition);
```

## Example

```
let number = 1;  
  
do {
```

```
    console.log("Number:", number);  
    number++;  
} while (number <= 5);
```

- ◆ Even if the condition is **false** initially, the loop runs **once**.
- 

## ✓ **forEach, for...in, for...of Loops**

### 4 **forEach()** (for Arrays)

The **forEach()** method is used to iterate over **arrays**.

#### **Syntax**

```
array.forEach((element, index) => {  
    // Code to execute  
});
```

#### **Example**

```
let fruits = ["Apple", "Banana", "Mango"];  
  
fruits.forEach((fruit, index) => {  
    console.log(`Index ${index}: ${fruit}`);  
});
```

- ◆ Outputs:

```
Index 0: Apple  
Index 1: Banana  
Index 2: Mango
```

---

## 5 `for...in` (for Objects & Arrays)

The `for...in` loop is used for **iterating over object properties**.

### Syntax

```
for (key in object) {  
  // Code executes for each property  
}
```

### Example

```
let person = { name: "John", age: 30, city: "New York" };  
  
for (let key in person) {  
  console.log(`${key}: ${person[key]}`);  
}
```

♦ Outputs:

```
name: John  
age: 30  
city: New York
```

♦ **For Arrays?**

```
let colors = ["Red", "Green", "Blue"];  
  
for (let index in colors) {  
  console.log(`Index ${index}: ${colors[index]}`);  
}
```

♦ Not recommended for arrays as it **iterates over indices** (use `forEach` or `for...of` instead).

---

## 6 `for...of` (for Arrays & Iterables)

The `for...of` loop is **best for iterating over array elements**.

### Syntax

```
for (let item of iterable) {  
  // Code executes for each element  
}
```

### Example

```
let numbers = [10, 20, 30];  
  
for (let num of numbers) {  
  console.log(num);  
}
```

- ◆ Outputs

```
10  
20  
30
```

- ◆ Works on **strings** too:

```
let word = "JavaScript";  
  
for (let char of word) {  
  console.log(char);  
}
```

- ◆ Outputs each character of "JavaScript".

---

## ✓ break & continue Statements

### 7 break Statement

The **break** statement **stops the loop immediately**.

#### Example

```
for (let i = 1; i <= 5; i++) {  
  if (i === 3) {  
    break; // Stops loop at 3  
  }  
  console.log(i);  
}
```

♦ Outputs:

1  
2

Loop **stops** when **i = 3**.

---

### 8 continue Statement

The **continue** statement **skips the current iteration** and moves to the next.

#### Example

```
for (let i = 1; i <= 5; i++) {  
  if (i === 3) {  
    continue; // Skips iteration 3  
  }  
  console.log(i);  
}
```

◆ Outputs:

1  
2  
4  
5

Iteration 3 is **skipped**.

---

## Summary Table

Loop Type	Used For	Stops When
<b>for</b>	Fixed number of iterations	Condition is <b>false</b>
<b>while</b>	Unknown number of iterations	Condition is <b>false</b>
<b>do-while</b>	Runs at least once	Condition is <b>false</b>
<b>forEach( )</b>	Arrays only	Iterates over all elements
<b>for...in</b>	Objects & Arrays (indices)	Iterates over property keys
<b>for...of</b>	Arrays, Strings, Iterables	Iterates over values
<b>break</b>	Exits the loop	Stops immediately
<b>continue</b>	Skips current iteration	Moves to next iteration



# 5. Functions in JavaScript

Functions are **blocks of reusable code** that perform a specific task. They help in making the code **modular, readable, and maintainable**.

---

## ✓ Function Declaration & Expression

### 1 Function Declaration (Named Function)

A **function declaration** is a named function that can be **hoisted** (used before it's defined).

#### Syntax

```
function functionName(parameters) {  
    // Code to execute  
}
```

#### Example

```
function greet(name) {  
    return `Hello, ${name}!`;  
}  
  
console.log(greet("Alice")); // Hello, Alice!
```

- ♦ **Hoisting**: Function declarations are hoisted, meaning they can be called **before** their definition.

---

### 2 Function Expression (Anonymous Function)

A **function expression** is stored inside a variable. It **cannot be hoisted**.

#### Syntax

```
const functionName = function(parameters) {
```

```
// Code to execute  
};
```

## Example

```
const add = function(a, b) {  
  return a + b;  
};  
  
console.log(add(3, 4)); // 7
```

- ◆ **No Hoisting:** Function expressions cannot be called before their definition.
- 

## ✓ Arrow Functions `(( )=>{ })`

Arrow functions provide a **shorter syntax** and do **not have their own `this` context**.

## Syntax

```
const functionName = (parameters) => {  
  // Code to execute  
};
```

## Example

```
const multiply = (a, b) => a * b;  
  
console.log(multiply(5, 3)); // 15
```

- ◆ If there is **only one parameter**, parentheses `()` are optional:

```
const square = num => num * num;  
console.log(square(4)); // 16
```

- ♦ If there is **no parameter**, empty `()` are required:

```
const greet = () => "Hello!";  
console.log(greet()); // Hello!
```

🚀 **Arrow functions do NOT have their own `this`, which makes them useful in callbacks.**

---

## ✓ Parameters & Arguments

### 3 Default Parameters

If no argument is passed, a default value is used.

```
function greet(name = "Guest") {  
  return `Hello, ${name}!`;  
}  
  
console.log(greet()); // Hello, Guest!  
console.log(greet("Alice")); // Hello, Alice!
```

---

### 4 Rest Parameters (`...args`)

The **rest operator** (`...`) allows a function to accept **multiple arguments as an array**.

```
function sum(...numbers) {  
  return numbers.reduce((acc, num) => acc + num, 0);  
}  
  
console.log(sum(1, 2, 3, 4, 5)); // 15
```

---

## 5 Spread Operator in Function Calls

The **spread operator** (`...`) expands an array into individual values.

```
const numbers = [10, 20, 30];

function sum(a, b, c) {
  return a + b + c;
}

console.log(sum(...numbers)); // 60
```

---

## ✓ Callbacks & Higher-Order Functions

### 6 Callbacks

A **callback function** is a function passed as an argument to another function.

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

function showMessage() {
  console.log("Welcome to JavaScript!");
}

greet("Alice", showMessage);
```

- ♦ `showMessage` is passed as a **callback** to `greet`.
- 

### 7 Higher-Order Functions

A **higher-order function** takes another function as a parameter or returns a function.

```
function applyOperation(a, b, operation) {  
  return operation(a, b);  
}  
  
const add = (x, y) => x + y;  
const multiply = (x, y) => x * y;  
  
console.log(applyOperation(5, 3, add)); // 8  
console.log(applyOperation(5, 3, multiply)); // 15
```

♦ `applyOperation` is a **higher-order function** that accepts another function (`add` or `multiply`) as an argument.

---

## ✓ **this** Keyword Inside Functions

### 8 **this** in Regular Functions

In a **regular function**, `this` refers to the **object that calls the function**.

```
const person = {  
  name: "Alice",  
  greet: function () {  
    console.log(`Hello, my name is ${this.name}`);  
  }  
};
```

```
person.greet(); // Hello, my name is Alice
```

♦ `this.name` refers to `person.name`.


---


### 9 **this** in Arrow Functions

Arrow functions **do not have their own `this`**. They inherit `this` from their surrounding scope.

```
const person = {
  name: "Alice",
  greet: () => {
    console.log(`Hello, my name is ${this.name}`);
  }
};

person.greet(); // Hello, my name is undefined
```

 **Problem:** Arrow functions take `this` from the outer scope, which in this case is the global object.

 **Solution:** Use a normal function instead.

---

## Immediately Invoked Function Expressions (IIFE)

### 10 What is IIFE?

An **Immediately Invoked Function Expression (IIFE)** runs **immediately after it's defined**.

### Syntax

```
(function() {
  // Code executes immediately
})();
```

### Example

```
(function() {
  console.log("This runs immediately!");
})();
```

- ♦ Useful for **creating private variables**.

### IIFE with Arrow Function

```
(() => {  
  console.log("This is an IIFE using arrow function!");  
})();
```

---

## Summary Table

Concept	Description	Example
<b>Function Declaration</b>	Regular function (hoisted)	<code>function greet() {}</code>
<b>Function Expression</b>	Function stored in a variable	<code>const add = function() {};</code>
<b>Arrow Function</b>	Shorter syntax, no <code>this</code>	<code>const sum = (a, b) =&gt; a + b;</code>
<b>Default Parameters</b>	Assign default values	<code>function greet(name = "Guest") {}</code>
<b>Rest Parameters</b>	Collects multiple arguments as an array	<code>function sum(...nums) {}</code>
<b>Spread Operator</b>	Expands an array into individual values	<code>sum(...arr);</code>
<b>Callback Function</b>	A function passed as an argument	<code>greet("Alice", showMessage);</code>
<b>Higher-Order Function</b>	A function that accepts/returns a function	<code>applyOperation(5, 3, add);</code>
<b>this in Regular Function</b>	Refers to the calling object	<code>this.name</code> in <code>person.greet()</code>
<b>this in Arrow Function</b>	Inherits <code>this</code> from outer scope	<code>this</code> is <code>undefined</code> if used inside an object
<b>IIFE</b>	Runs immediately after definition	<code>(function() {   console.log("Hello!"); })();</code>

## 6. Objects & OOP Concepts in JavaScript

Objects are **collections of key-value pairs**, where keys are strings (or symbols) and values can be of any data type. JavaScript is **prototype-based**, meaning it does not have traditional class-based inheritance but uses **prototypes** to enable object-oriented programming (OOP).

---

### ✓ Creating Objects

#### 1 Object Literals

The simplest way to create an object is by using **object literals**.

```
const person = {
  name: "Alice",
  age: 25,
  greet: function () {
    return `Hello, my name is ${this.name}`;
  }
};

console.log(person.name); // Alice
console.log(person.greet()); // Hello, my name is Alice
```

- ♦ **this** inside `greet()` refers to the `person` object.

---

#### 2 Constructor Functions

A **constructor function** is used to create multiple objects with the same structure.

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



```
const person1 = new Person("Alice", 25);
const person2 = new Person("Bob", 30);
```

```
console.log(person1.name); // Alice
console.log(person2.age); // 30
```

- ♦ `new` creates a **new object** and assigns `this` to it.
- 

### 3 Classes (ES6 Syntax)

Classes provide a **cleaner** way to define constructor functions.

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  greet() {
    return `Hello, my name is ${this.name}`;
  }
}
```

```
const person3 = new Person("Charlie", 28);
console.log(person3.greet()); // Hello, my name is Charlie
```

---

## ✓ Object Methods

### 4 Useful Built-in Object Methods

Method	Description	Example
<code>Object.keys(obj)</code>	Returns an array of keys	<code>Object.keys(person)</code>

<code>Object.values(obj)</code>	Returns an array of values	<code>Object.values(person)</code>
<code>Object.entries(obj)</code>	Returns key-value pairs as an array	<code>Object.entries(person)</code>

## Examples

```
const person = { name: "Alice", age: 25, city: "New York" };

console.log(Object.keys(person)); // ["name", "age", "city"]
console.log(Object.values(person)); // ["Alice", 25, "New York"]
console.log(Object.entries(person)); // [["name", "Alice"], ["age", 25], ["city", "New York"]]
```

---

## ✓ **this** Keyword in Objects

The **this** keyword refers to the **current object** in which the function is executed.

### 5 **this** in Object Methods

```
const user = {
  name: "Alice",
  greet: function() {
    console.log(`Hello, I am ${this.name}`);
  }
};
```

```
user.greet(); // Hello, I am Alice
```

- ◆ Inside objects, **this** refers to the **object itself**.
- 

## ✓ **Prototypes & Prototype Inheritance**

JavaScript **does not use classical inheritance (like Java or C#)**. Instead, it uses **prototypes**.

## 6 Prototype Example

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

// Adding a method to the prototype
Person.prototype.greet = function () {
  return `Hello, my name is ${this.name}`;
};

const person1 = new Person("Alice");
console.log(person1.greet()); // Hello, my name is Alice
```

- ◆ The `greet()` method is **not inside the object**, but inside its **prototype**.
- 

## 7 Prototype Inheritance

Objects can inherit methods and properties from another object's prototype.

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

Animal.prototype.speak = function () {
  return `${this.name} makes a sound`;
};

function Dog(name, breed) {
  Animal.call(this, name);
  this.breed = breed;
}

// Inheriting Animal prototype
```

```
Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;

Dog.prototype.bark = function () {
  return `${this.name} barks`;
};

const dog1 = new Dog("Buddy", "Golden Retriever");

console.log(dog1.speak()); // Buddy makes a sound
console.log(dog1.bark()); // Buddy barks
```

- ◆ `Dog` inherits from `Animal`, but also has its own method (`bark`).

---

## ✓ Classes & Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) is a programming paradigm based on **objects and classes**. JavaScript **supports OOP** through prototypes and classes.

### 8 Encapsulation

Encapsulation is the **hiding of internal implementation** details and exposing only necessary functionalities.

```
class BankAccount {
  #balance; // Private property (ES2020)

  constructor(initialBalance) {
    this.#balance = initialBalance;
  }

  deposit(amount) {
    this.#balance += amount;
  }

  getBalance() {
```

```
        return this.#balance;
    }
}

const account = new BankAccount(1000);
account.deposit(500);
console.log(account.getBalance()); // 1500
console.log(account.#balance); // ❌ Error: Private field
```

- ◆ `#balance` is a **private variable**, meaning it cannot be accessed outside the class.
- 

## 9 Abstraction

Abstraction is **hiding complex details** and showing only what is necessary.

```
class Car {
    startEngine() {
        console.log("Engine started");
    }

    drive() {
        console.log("Car is moving");
    }
}

const myCar = new Car();
myCar.drive(); // Car is moving
```

- ◆ The **user doesn't need to know** how `startEngine()` works internally.
- 

## 10 Inheritance

Inheritance allows a class to **inherit properties and methods** from another class.

```
class Animal {
  constructor(name) {
    this.name = name;
  }

  speak() {
    return `${this.name} makes a sound`;
  }
}

class Dog extends Animal {
  bark() {
    return `${this.name} barks`;
  }
}

const dog = new Dog("Buddy");
console.log(dog.speak()); // Buddy makes a sound
console.log(dog.bark());  // Buddy barks
```

- ♦ Dog **inherits** from `Animal` using `extends`.
- 

## Polymorphism

Polymorphism allows methods to be **overridden in subclasses**.

```
class Animal {
  speak() {
    return "Animal makes a sound";
  }
}

class Dog extends Animal {
  speak() {
    return "Dog barks";
  }
}
```

```
}

const animal = new Animal();
const dog = new Dog();

console.log(animal.speak()); // Animal makes a sound
console.log(dog.speak());    // Dog barks
```

- ◆ Dog **overrides** `speak()` from `Animal`.

---

## ✓ Getter & Setter Methods

### 12 Getters & Setters in JavaScript

Getters and setters allow **controlled access** to properties.

```
class Person {
  constructor(name, age) {
    this.name = name;
    this._age = age; // Private-like variable (not truly private)
  }

  get age() {
    return this._age;
  }

  set age(value) {
    if (value < 0) {
      console.log("Age cannot be negative");
    } else {
      this._age = value;
    }
  }
}

const person = new Person("Alice", 25);
```

```
console.log(person.age); // 25
person.age = -5; // Age cannot be negative
person.age = 30;
console.log(person.age); // 30
```

- ♦ `get` retrieves `_age`, and `set` validates the input.

---

## Summary Table

Concept	Description	Example
<b>Object Literals</b>	Create objects directly	<code>{ name: "Alice" }</code>
<b>Constructor Function</b>	Function to create objects	<code>function Person(name) {}</code>
<b>Classes</b>	ES6 way to create objects	<code>class Person {}</code>
<b>Object Methods</b>	Built-in object operations	<code>Object.keys(obj)</code>
<b>Prototypes</b>	Inherit properties/methods	<code>Person.prototype.greet = function () {}</code>
<b>Encapsulation</b>	Hide implementation details	Private variables in classes
<b>Abstraction</b>	Expose only necessary details	<code>class Car { drive() {} }</code>
<b>Inheritance</b>	Subclass inherits properties	<code>class Dog extends Animal {}</code>
<b>Polymorphism</b>	Overriding methods	<code>Dog overrides speak()</code>
<b>Getters &amp; Setters</b>	Control access to properties	<code>get age() {}</code>



# 7. Arrays & Array Methods in JavaScript

Arrays are **ordered collections** of elements. JavaScript provides **various built-in methods** to manipulate arrays efficiently.

---

## ✓ Creating & Accessing Arrays

### 1 Creating Arrays

Arrays can be created using **array literals** or the **Array** constructor.

```
const fruits = ["Apple", "Banana", "Cherry"]; // Array literal
const numbers = new Array(1, 2, 3, 4, 5);    // Array constructor
```

```
console.log(fruits[0]); // Apple (Accessing first element)
console.log(numbers[2]); // 3 (Accessing third element)
```

- ♦ Arrays are **zero-indexed**, meaning the first element is at index **0**.
- 

## ✓ Array Methods: push(), pop(), shift(), unshift()

### 2 Adding & Removing Elements

Method	Description	Example
<code>push()</code>	Adds an element to the <b>end</b>	<code>arr.push(6)</code>
<code>pop()</code>	Removes and returns the <b>last</b> element	<code>arr.pop()</code>
<code>unshift()</code>	Adds an element to the <b>beginning</b>	<code>arr.unshift(0)</code>
<code>shift()</code>	Removes and returns the <b>first</b> element	<code>arr.shift()</code>

## Examples

```
const nums = [1, 2, 3];

nums.push(4);    // [1, 2, 3, 4]
nums.pop();      // [1, 2, 3]
nums.unshift(0); // [0, 1, 2, 3]
nums.shift();    // [1, 2, 3]

console.log(nums);
```

---

## Iterating Arrays: map(), filter(), reduce()

### **map()** – Transform Each Element

Creates a **new array** by applying a function to each element.

```
const numbers = [1, 2, 3, 4];
const doubled = numbers.map(num => num * 2);

console.log(doubled); // [2, 4, 6, 8]
```

---

### **filter()** – Keep Elements That Match a Condition

Creates a **new array** with elements that **pass a condition**.

```
const ages = [10, 20, 30, 40];
const adults = ages.filter(age => age >= 18);

console.log(adults); // [20, 30, 40]
```

---

### **reduce()** – Accumulate a Value

Reduces an array to a **single value**.

```
const numbers = [1, 2, 3, 4];
const sum = numbers.reduce((acc, curr) => acc + curr, 0);

console.log(sum); // 10
```

- ♦ `reduce()` starts with `0` (initial value) and **adds each number**.
- 

## ✓ **find(), findIndex(), some(), every()**

### 6 **find()** – Find the First Matching Element

```
const users = [{ name: "Alice", age: 25 }, { name: "Bob", age: 30 }];
const user = users.find(user => user.age > 25);

console.log(user); // { name: "Bob", age: 30 }
```

- ♦ **Returns only the first match.**
- 

### 7 **findIndex()** – Find the Index of the First Match

```
const numbers = [5, 12, 8, 130, 44];
const index = numbers.findIndex(num => num > 10);

console.log(index); // 1 (First number > 10 is 12, at index 1)
```

---

### 8 **some()** – Check If At Least One Element Matches

```
const scores = [50, 40, 90, 30];
const hasHighScore = scores.some(score => score > 80);
```

```
console.log(hasHighScore); // true
```

- ◆ Returns **true** if at least one element satisfies the condition.
- 

## 9 **every()** – Check If All Elements Match

```
const scores = [90, 85, 88, 92];  
const allPassed = scores.every(score => score >= 80);  
  
console.log(allPassed); // true
```

- ◆ Returns **true** only if ALL elements match the condition.
- 

## ✓ **sort(), reverse(), splice(), slice()**

### 10 **sort()** – Sorts an Array (Mutates Original)

```
const numbers = [40, 100, 1, 5, 25, 10];  
numbers.sort((a, b) => a - b); // Ascending order  
  
console.log(numbers); // [1, 5, 10, 25, 40, 100]
```

- ◆ **sort()** mutates the array, so it changes the original.
- 

### 11 **reverse()** – Reverse an Array

```
const arr = [1, 2, 3, 4];  
arr.reverse();  
  
console.log(arr); // [4, 3, 2, 1]
```

---

## 12 splice() – Modify Array (Add/Remove Items)

```
const colors = ["Red", "Green", "Blue"];

// Remove 1 element at index 1
colors.splice(1, 1); // ["Red", "Blue"]

// Add "Yellow" at index 1
colors.splice(1, 0, "Yellow"); // ["Red", "Yellow", "Blue"]

console.log(colors);
```

- ◆ **Changes the original array.**

---

## 13 slice() – Extract a Portion of an Array

```
const animals = ["Lion", "Tiger", "Elephant", "Giraffe"];
const sliced = animals.slice(1, 3); // ["Tiger", "Elephant"]

console.log(sliced);
```

- ◆ **Does NOT modify the original array.**

---

## ✓ concat(), flat(), includes(), indexOf()

### 14 concat() – Merge Arrays

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

const merged = arr1.concat(arr2);
```

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

- ◆ Returns a new array.
- 

## 15 flat() – Flatten Nested Arrays

```
const nested = [1, [2, [3, [4]]]];
const flatArr = nested.flat(2);

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

- ◆ flat(n) flattens up to n levels.
- 

## 16 includes() – Check If an Element Exists

```
const fruits = ["Apple", "Banana", "Cherry"];
console.log(fruits.includes("Banana")); // true
console.log(fruits.includes("Mango"));  // false
```

---

## 17 indexOf() – Find Index of an Element

```
const numbers = [10, 20, 30, 40];
console.log(numbers.indexOf(30)); // 2
console.log(numbers.indexOf(50)); // -1 (not found)
```

---

## Summary Table

Method	Purpose	Example
push()	Add element at the end	arr.push(6)

<code>pop()</code>	Remove last element	<code>arr.pop()</code>
<code>unshift()</code>	Add element at the start	<code>arr.unshift(0)</code>
<code>shift()</code>	Remove first element	<code>arr.shift()</code>
<code>map()</code>	Transform array elements	<code>arr.map(x =&gt; x * 2)</code>
<code>filter()</code>	Keep elements that match a condition	<code>arr.filter(x =&gt; x &gt; 10)</code>
<code>reduce()</code>	Accumulate values	<code>arr.reduce((a, b) =&gt; a + b, 0)</code>
<code>find()</code>	Find first matching element	<code>arr.find(x =&gt; x &gt; 10)</code>
<code>findIndex()</code>	Find index of first match	<code>arr.findIndex(x =&gt; x &gt; 10)</code>
<code>some()</code>	Check if at least one element matches	<code>arr.some(x =&gt; x &gt; 10)</code>
<code>every()</code>	Check if all elements match	<code>arr.every(x =&gt; x &gt; 10)</code>
<code>sort()</code>	Sort array (mutates)	<code>arr.sort((a, b) =&gt; a - b)</code>
<code>reverse()</code>	Reverse array order	<code>arr.reverse()</code>
<code>splice()</code>	Add/remove elements	<code>arr.splice(1, 2, "X")</code>
<code>slice()</code>	Extract portion of array	<code>arr.slice(1, 3)</code>
<code>concat()</code>	Merge arrays	<code>arr1.concat(arr2)</code>
<code>flat()</code>	Flatten nested arrays	<code>arr.flat(2)</code>
<code>includes()</code>	Check if element exists	<code>arr.includes(10)</code>
<code>indexOf()</code>	Find index of element	<code>arr.indexOf(10)</code>

## 8. Strings & String Methods in JavaScript

Strings in JavaScript are **sequences of characters** used for storing text. JavaScript provides **various built-in methods** to manipulate strings efficiently.

---

### ✓ String Manipulation

#### 1 **length** – Find the Length of a String

```
const text = "Hello, World!";  
console.log(text.length); // 13
```

- ♦ Spaces and punctuation are counted in the length.
- 

#### 2 **toUpperCase()** – Convert to Uppercase

```
const message = "hello";  
console.log(message.toUpperCase()); // "HELLO"
```

---

#### 3 **toLowerCase()** – Convert to Lowercase

```
const greeting = "GOOD MORNING";  
console.log(greeting.toLowerCase()); // "good morning"
```

---

### ✓ String Methods: **substring()**, **slice()**, **split()**, **join()**

#### 4 **substring(start, end)** – Extract Part of a String

- Extracts characters between **start** (inclusive) and **end** (exclusive).



```
const str = "JavaScript";
console.log(str.substring(0, 4)); // "Java"
console.log(str.substring(4));    // "Script" (if end is omitted)
```

---

## 5 **slice(start, end)** – Extract Substring with Negative Index

- Works like `substring()`, but supports **negative indices**.

```
const str = "JavaScript";
console.log(str.slice(0, 4)); // "Java"
console.log(str.slice(-6));   // "Script" (Negative index starts from end)
```

---

## 6 **split(separator)** – Convert String to an Array

```
const sentence = "apple,banana,grape";
const fruits = sentence.split(",");

console.log(fruits); // ["apple", "banana", "grape"]
```

- ♦ `split()` is useful for **breaking strings into arrays**.
- 

## 7 **join(separator)** – Convert Array to String

```
const words = ["Hello", "World"];
console.log(words.join(" ")); // "Hello World"
```

- ♦ Opposite of `split()`.
-

## ✓ Template Literals & String Interpolation

### 8 Template Literals (`` ` `) – Embed Variables & Expressions

```
const name = "Alice";
const age = 25;

console.log(`Hello, my name is ${name} and I am ${age} years old.`);
// Output: Hello, my name is Alice and I am 25 years old.
```

- ◆ Supports **multiline strings** without `\n`:

```
const message = `Hello,
This is a multiline string!`;
console.log(message);
```

---

## ✓ Searching in Strings

### 9 `indexOf()` – Find the First Occurrence of a Substring

```
const str = "Hello World";
console.log(str.indexOf("World")); // 6
console.log(str.indexOf("o"));      // 4
console.log(str.indexOf("Java"));   // -1 (Not Found)
```

- ◆ Returns `-1` if the substring is **not found**.
- 

### 10 `lastIndexOf()` – Find the Last Occurrence

```
const str = "Hello World, Hello Again";
console.log(str.lastIndexOf("Hello")); // 13
console.log(str.lastIndexOf("o"));     // 20
```

---

## **includes()** – Check if a Substring Exists

```
const text = "Learning JavaScript";
console.log(text.includes("JavaScript")); // true
console.log(text.includes("Python"));    // false
```

- ◆ Returns **true** if the substring is found.

---

## **Summary Table**

Method	Purpose	Example
<code>length</code>	Get string length	<code>"Hello".length // 5</code>
<code>toUpperCase()</code>	Convert to uppercase	<code>"hello".toUpperCase() // "HELLO"</code>
<code>toLowerCase()</code>	Convert to lowercase	<code>"HELLO".toLowerCase() // "hello"</code>
<code>substring(start, end)</code>	Extract part of string	<code>"JavaScript".substring(0,4) // "Java"</code>
<code>slice(start, end)</code>	Extract using negative index	<code>"JavaScript".slice(-6) // "Script"</code>
<code>split(separator)</code>	Convert string to array	<code>"a,b,c".split(",") // ["a", "b", "c"]</code>
<code>join(separator)</code>	Convert array to string	<code>["a", "b"].join("-") // "a-b"</code>
<code>indexOf(substring)</code>	Find first occurrence	<code>"Hello".indexOf("l") // 2</code>

<code>lastIndexOf(substring)</code>	Find last occurrence	<code>"Hello".lastIndexOf("l") // 3</code>
<code>includes(substring)</code>	Check if exists	<code>"Hello".includes("lo") // true</code>

# 9. JavaScript ES6+ Features

JavaScript ES6+ introduced **powerful features** that improve code readability, maintainability, and efficiency. Let's explore them with examples.

---

## ✓ 1. **let** and **const** vs **var**

### ♦ **var** (Function Scoped, Hoisting)

- **var** is **function-scoped** and can be **re-declared**.
- It gets **hoisted** but is initialized with **undefined**.

```
console.log(a); // undefined (Hoisting)
var a = 10;
console.log(a); // 10
```

---

### ♦ **let** (Block Scoped, No Re-declaration)

- **let** is **block-scoped** (inside `{}`).
- Cannot be **re-declared** in the same scope.

```
let x = 5;
{
  let x = 10; // Block scope
  console.log(x); // 10
}
console.log(x); // 5
```

---

### ♦ **const** (Block Scoped, No Re-assignment)

- **const** is **block-scoped** and **cannot be reassigned**.

```
const PI = 3.14;  
PI = 3.14159; // ❌ Error: Assignment to constant variable.
```

---

## ✅ 2. Spread Operator (...)

The **spread operator** (...) allows expanding elements of an array or object.

### ♦ Copying Arrays

```
const arr1 = [1, 2, 3];  
const arr2 = [...arr1]; // Creates a copy  
console.log(arr2); // [1, 2, 3]
```

### ♦ Merging Arrays

```
const numbers = [1, 2, 3];  
const moreNumbers = [4, 5, 6];  
const merged = [...numbers, ...moreNumbers];  
  
console.log(merged); // [1, 2, 3, 4, 5, 6]
```

### ♦ Copying Objects

```
const user = { name: "Alice", age: 25 };  
const newUser = { ...user, city: "New York" };  
  
console.log(newUser); // { name: "Alice", age: 25, city: "New York" }
```

---

## ✅ 3. Destructuring (Array & Object)

### ♦ Array Destructuring

```
const numbers = [10, 20, 30];
```

```
const [first, second] = numbers;
```

```
console.log(first); // 10  
console.log(second); // 20
```

### ◆ Object Destructuring

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

```
console.log(name); // "John"  
console.log(age); // 30
```

---

## ✓ 4. Default Parameters

Allows setting **default values** for function parameters.

```
function greet(name = "Guest") {  
    console.log(`Hello, ${name}!`);  
}
```

```
greet();           // "Hello, Guest!"  
greet("Alice");   // "Hello, Alice!"
```

---

## ✓ 5. Template Literals

Allows **string interpolation** using backticks (`).

```
const name = "John";  
const message = `Hello, my name is ${name}!`;  
  
console.log(message); // "Hello, my name is John!"
```

Supports **multiline strings**:

```
const paragraph = `This is line 1  
This is line 2`;  
  
console.log(paragraph);
```

---

## ✓ 6. Rest Operator (...)

- **Collects multiple values** into an array.

### ◆ Function Arguments

```
function sum(...numbers) {  
    return numbers.reduce((acc, num) => acc + num, 0);  
}  
  
console.log(sum(1, 2, 3, 4)); // 10
```

### ◆ Object Rest

```
const user = { name: "Alice", age: 25, city: "Paris" };  
const { name, ...rest } = user;  
  
console.log(name); // "Alice"  
console.log(rest); // { age: 25, city: "Paris" }
```

---

## ✓ 7. Arrow Functions (()=>{})

- **Shorter syntax** for functions.
- Automatically binds `this`.

### ◆ Basic Arrow Function



```
const add = (a, b) => a + b;  
console.log(add(5, 3)); // 8
```

### ♦ Arrow Function with **this**

```
const person = {  
  name: "Alice",  
  greet: function() {  
    setTimeout(() => {  
      console.log(`Hello, ${this.name}`);  
    }, 1000);  
  }  
};
```

```
person.greet(); // "Hello, Alice" (Arrow function captures `this`)
```

---

## ✅ 8. Modules (import & export)

ES6 introduces **modules** for reusability.

### ♦ Exporting

📌 math.js

```
export const add = (a, b) => a + b;  
export const subtract = (a, b) => a - b;
```

### ♦ Importing

📌 app.js

```
import { add, subtract } from './math.js';
```

```
console.log(add(5, 3)); // 8
```

```
console.log(subtract(10, 4)); // 6
```

#### ◆ Default Export

```
export default function greet() {  
  console.log("Hello!");  
}
```

```
import greet from './greet.js';  
greet(); // "Hello!"
```

---

## ✓ 9. Optional Chaining (?.)

Prevents **errors** when accessing deeply nested properties.

```
const user = { name: "Alice", address: { city: "Paris" } };
```

```
console.log(user.address?.city);    // "Paris"  
console.log(user.contact?.email);  // undefined (No error)
```

---

## Summary Table

Feature	Example
let & const vs var	let a = 5; const b = 10;
Spread Operator	const arr2 = [...arr1];
Destructuring	const {name, age} = user;
Default Parameters	function greet(name = "Guest") {}

Template Literals	<code>`Hello, \${name}!`</code>
Rest Operator	<code>function sum(...nums) {}</code>
Arrow Functions	<code>const add = (a, b) =&gt; a + b;</code>
Modules	<code>import { add } from './math.js';</code>
Optional Chaining	<code>user?.profile?.email;</code>

# 10. Asynchronous JavaScript

Asynchronous programming in JavaScript allows code to be executed in a non-blocking manner, enabling efficient handling of operations like **API calls**, **file reading**, and **delays**. Below are the concepts and examples related to asynchronous JavaScript:

---

## ✓ 1. Callbacks

A **callback** is a function passed into another function as an argument and executed after the completion of the operation.

### ◆ Example:

```
function fetchData(callback) {
    setTimeout(() => {
        const data = { name: "Alice", age: 25 };
        callback(data); // Callback is invoked after 2 seconds
    }, 2000);
}

function displayData(data) {
    console.log(data);
}

fetchData(displayData); // { name: "Alice", age: 25 }
```

---

## ✓ 2. Promises

A **Promise** is an object representing the eventual completion (or failure) of an asynchronous operation. It has three possible states:

1. **Pending**
2. **Fulfilled (Resolved)**
3. **Rejected**

### ◆ Creating a Promise

A Promise takes an executor function with two arguments: `resolve` and `reject`.

```
const myPromise = new Promise((resolve, reject) => {
  const success = true;

  if (success) {
    resolve("Data fetched successfully!");
  } else {
    reject("Error fetching data.");
  }
});

myPromise
  .then(result => console.log(result)) // Success handler
  .catch(error => console.log(error)) // Error handler
  .finally(() => console.log("Operation complete.")).finally(); // Cleanup
```

#### ◆ States of a Promise

- **Pending:** Initial state when the promise is not fulfilled.
- **Resolved (Fulfilled):** When the operation is successful and `resolve` is called.
- **Rejected:** When the operation fails and `reject` is called.

---

## ✓ 3. Async & Await

`async` and `await` provide a more **synchronous-like** behavior for handling asynchronous operations. `await` can only be used inside an `async` function.

#### ◆ Example:

```
async function fetchData() {
  try {
    const data = await myPromise; // Wait until the promise is
    resolved
    console.log(data); // Data fetched successfully!
  } catch (error) {
```

```
        console.error(error); // Handle error
    }
}

fetchData();
```

- **async**: Marks a function as asynchronous and ensures it returns a **promise**.
  - **await**: Pauses the execution of the **async** function until the **promise** is resolved or rejected.
- 

## ✓ 4. fetch() API & HTTP Requests

The **fetch()** API is used to make **HTTP requests**. It returns a **Promise** that resolves to the response of the request.

### ♦ Example: Basic GET Request

```
fetch('https://jsonplaceholder.typicode.com/users')
    .then(response => response.json()) // Parse JSON response
    .then(data => console.log(data)) // Process data
    .catch(error => console.error("Error:", error)); // Handle errors
```

### ♦ Example: POST Request

```
const newUser = {
    name: "John Doe",
    email: "johndoe@example.com"
};

fetch('https://jsonplaceholder.typicode.com/users', {
    method: 'POST',
    headers: {
        'Content-Type': 'application/json'
    },
    body: JSON.stringify(newUser)
})
```

```
.then(response => response.json())
.then(data => console.log("User Created:", data))
.catch(error => console.error("Error:", error));
```

---

## ✓ 5. Handling API Errors

It is important to handle errors when working with **API requests** to ensure the app can respond to issues gracefully.

### ◆ Common API Errors:

1. **Network Issues:** Request fails due to server unavailability.
2. **HTTP Status Codes:** 404 Not Found, 500 Internal Server Error, etc.

### ◆ Example: Handling Errors with **fetch()**

```
fetch('https://jsonplaceholder.typicode.com/users')
  .then(response => {
    if (!response.ok) { // Check for successful response
      throw new Error('Network response was not ok');
    }
    return response.json();
  })
  .then(data => console.log(data))
  .catch(error => console.error("There was a problem with the fetch
operation:", error));
```

### ◆ Example: Handling Errors with **async/await**

```
async function fetchData() {
  try {
    const response = await
fetch('https://jsonplaceholder.typicode.com/users');

    if (!response.ok) {
      throw new Error('Failed to fetch data');
    }
  }
}
```

```

    }

    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error("Error fetching data:", error.message);
  }
}

fetchData();

```

---

## Summary Table

Concept	Description	Example
<b>Callbacks</b>	Functions passed as arguments to be executed later.	<code>fetchData(displayData);</code>
<b>Promises</b>	Objects representing eventual completion (or failure) of an async operation.	<code>myPromise.then(...).catch(...);</code>
<b>Async/Await</b>	Makes async code look synchronous, improving readability.	<code>await myPromise;</code>
<b>fetch() API</b>	API for making HTTP requests. Returns a promise that resolves to response.	<code>fetch(url).then(...).catch(...);</code>
<b>Handling API Errors</b>	Catching errors like network issues or HTTP status errors.	<code>catch(error =&gt; console.log(error));</code>



# 11. JavaScript Event Loop & Execution Context

The **JavaScript Event Loop** is a fundamental concept for understanding how asynchronous operations are handled in JavaScript. It allows non-blocking behavior by managing the execution of tasks like API calls, user events, and timers.

---

## ✓ 1. Call Stack & Execution Context

### ♦ Call Stack

The **Call Stack** is a stack data structure that keeps track of the function calls in a program. Every time a function is invoked, it is added (pushed) to the stack. When the function finishes executing, it is removed (popped) from the stack.

- **LIFO (Last In, First Out)**: The last function called is the first one to finish.

### ♦ Execution Context

The **Execution Context** is an environment in which the code is executed. There are three types:

1. **Global Execution Context**: The outermost context, where the JavaScript engine starts.
2. **Function Execution Context**: Created when a function is invoked.
3. **Eval Execution Context**: Created for code executed inside an `eval()` function.

### ♦ Example: Call Stack and Execution Context

```
function firstFunction() {  
    console.log("First Function");  
    secondFunction();  
}  
  
function secondFunction() {  
    console.log("Second Function");  
}  
  
firstFunction();
```

1. **Global Execution Context** is created first.
  2. When `firstFunction()` is called, a new **Function Execution Context** is created.
  3. Inside `firstFunction()`, `secondFunction()` is called, which adds another **Function Execution Context** to the stack.
  4. After `secondFunction()` completes, it's popped from the stack, then `firstFunction()` completes and is also popped.
- 

## ✓ 2. Synchronous vs Asynchronous Code

### ♦ Synchronous Code

Synchronous code is executed line by line, blocking the execution of the next line until the current one is completed.

- Example: **Blocking**

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

**Output:** The output will always be:

```
Start  
Middle  
End
```

- 

### ♦ Asynchronous Code

Asynchronous code allows the program to continue executing other tasks while waiting for operations like **API calls** or **setTimeout** to complete.

- Example: **Non-blocking**

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

```
setTimeout(() => {  
    console.log("Middle");  
}, 2000);
```

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

**Output:**

```
Start  
End  
Middle
```

---

## ✓ 3. Event Loop & Task Queue

The **Event Loop** is responsible for executing the code, handling events, and executing tasks in the **Task Queue**.

### ◆ How Event Loop Works

1. **Call Stack**: Contains the current executing functions.
2. **Task Queue (Callback Queue)**: Holds all tasks like `setTimeout()`, `setInterval()`, and events that are waiting to be processed.
3. **Event Loop**: Continuously checks if the **Call Stack** is empty and moves tasks from the **Task Queue** to the **Call Stack** for execution.

### ◆ Example: Event Loop

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

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

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

1. `console.log("Start")` is executed and removed from the **Call Stack**.

2. `setTimeout()` is executed and moves to the **Web API** environment.
3. `console.log("End")` is executed and removed from the **Call Stack**.
4. Once the **Call Stack** is empty, the **Event Loop** picks up the `setTimeout()` callback from the **Task Queue** and executes it.

**Output:**

Start  
End  
Middle

---

## ✓ 4. Microtasks vs Macrotasks

### ◆ Microtasks

Microtasks are small tasks that are executed after the currently executing script has completed, but before the next event loop iteration.

- **Examples of Microtasks:**
  - Promises (`then()`, `catch()`, `finally`)
  - `queueMicrotask()`

### ◆ Macrotasks

Macrotasks are larger tasks, executed after all microtasks have completed, such as I/O events, timers, and user interactions.

- **Examples of Macrotasks:**
  - `setTimeout()`, `setInterval()`
  - I/O events (e.g., HTTP requests)
  - UI events (e.g., mouse click, key press)

### ◆ Execution Order:

1. Synchronous code executes first.
2. Microtasks are executed after the synchronous code completes.
3. Macrotasks are processed after all microtasks have been processed.

### ◆ Example: Microtasks vs Macrotasks

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

setTimeout(() => {
  console.log("Macrotask 1");
}, 0);

Promise.resolve().then(() => {
  console.log("Microtask 1");
});

setTimeout(() => {
  console.log("Macrotask 2");
}, 0);

Promise.resolve().then(() => {
  console.log("Microtask 2");
});

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

#### Execution Order:

```
Start
End
Microtask 1
Microtask 2
Macrotask 1
Macrotask 2
```

- 
- **Explanation:**
  - The synchronous code (**Start**, **End**) runs first.
  - All **microtasks** (**Promise.then()**) are executed before the **macrotasks** (**setTimeout()**).

---

## 5. Web APIs & Callback Queue

## ♦ Web APIs

Web APIs are built-in browser APIs that allow JavaScript to interact with the browser and the outside world. These include things like **DOM manipulation**, **AJAX requests**, **setTimeout()**, and more.

## ♦ Callback Queue

The **Callback Queue** is where tasks that are ready to be executed, such as event handlers, are placed after they are triggered.

- After a task is completed, the event or callback is pushed to the **Callback Queue**.
- The **Event Loop** picks up tasks from the **Callback Queue** once the **Call Stack** is empty.

## ♦ Example: Using setTimeout with Web APIs

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

setTimeout(() => {
  console.log("This is a delayed message.");
}, 2000);

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

- **Explanation:**
    - `setTimeout()` is part of the **Web API**, which asynchronously schedules the callback.
    - The callback moves to the **Callback Queue** once the specified time (2000ms) elapses.
    - The **Event Loop** picks up the callback after the **Call Stack** is clear.
- 

## Summary Table

Concept	Description	Example
<b>Call Stack</b>	Stack of function calls executed in order.	<code>firstFunction()</code> , <code>secondFunction()</code>

<b>Execution Context</b>	The environment in which code is executed (Global, Function, Eval).	Global Execution Context
<b>Synchronous Code</b>	Code executed line by line, blocking the next operation.	<code>console.log("Start")</code>
<b>Asynchronous Code</b>	Code that runs without blocking subsequent operations.	<code>setTimeout()</code>
<b>Event Loop</b>	Manages execution of code and moves tasks from <b>Task Queue</b> to <b>Call Stack</b> .	Event Loop checks the Task Queue
<b>Microtasks</b>	Small tasks that run after the current script completes, before macrotasks.	<code>Promise.then()</code>
<b>Macrotasks</b>	Larger tasks like I/O events and timers.	<code>setTimeout()</code>
<b>Web APIs</b>	Browser APIs to handle asynchronous operations like DOM manipulation.	<code>setTimeout()</code> , <code>fetch()</code>

# 12. DOM Manipulation

The **DOM (Document Object Model)** is an interface that allows programming languages (like JavaScript) to interact with and manipulate the content, structure, and style of a webpage. The DOM represents the HTML document as a tree structure where each element is a node, allowing you to access and modify various parts of the webpage.

---

## ✓ 1. What is the DOM?

The **DOM** is a programming interface for web documents. It represents the document as a tree of nodes, where each node is an object representing a part of the page. JavaScript can be used to access and modify the DOM to create dynamic and interactive web pages.

- **HTML to DOM Mapping:** Each HTML element (like `<div>`, `<p>`, etc.) corresponds to a node in the DOM tree.
  - The DOM allows interaction with elements such as creating, deleting, and modifying content and styles.
- 

## ✓ 2. Selecting Elements

There are various methods for selecting HTML elements in JavaScript. The most common methods include:

### ♦ `getElementById()`

Selects an element by its `id` attribute. It returns the first element with the matching `id`.

```
let element = document.getElementById("myElement");
console.log(element);
```

### ♦ `querySelector()`

Selects the first element that matches a specified CSS selector (e.g., class, id, tag).

```
let element = document.querySelector(".myClass");
```



```
console.log(element);
```

- ◆ **querySelectorAll()**

Selects all elements that match a specified CSS selector and returns a **NodeList**.

```
let elements = document.querySelectorAll("p");  
console.log(elements);
```

- ◆ **getElementsByClassName()**

Selects elements by class name.

```
let elements = document.getElementsByClassName("myClass");  
console.log(elements);
```

- ◆ **getElementsByTagName()**

Selects elements by their tag name (e.g., `<div>`, `<p>`).

```
let elements = document.getElementsByTagName("div");  
console.log(elements);
```

---

## ✓ 3. Changing HTML & CSS with JavaScript

- ◆ **Changing HTML**

You can change the HTML content of an element using the `innerHTML` property.

```
document.getElementById("myElement").innerHTML = "New Content";
```

- ◆ **Changing CSS**

You can change the style of an element using the `style` property.

```
document.getElementById("myElement").style.color = "red";  
document.getElementById("myElement").style.fontSize = "20px";
```

---

## ✓ 4. Creating, Appending, and Removing Elements

### ♦ Creating Elements

To create new HTML elements, you use `document.createElement()`.

```
let newElement = document.createElement("div");  
newElement.innerHTML = "This is a new div element";
```

### ♦ Appending Elements

To append an element to another element, use `appendChild()`.

```
let parentElement = document.getElementById("parentDiv");  
parentElement.appendChild(newElement);
```

### ♦ Removing Elements

To remove an element, use `removeChild()`.

```
let parentElement = document.getElementById("parentDiv");  
parentElement.removeChild(newElement);
```

Alternatively, you can remove an element directly using `remove()`:

```
newElement.remove();
```

---

## ✓ 5. Event Handling

Events in JavaScript can be triggered by user actions like clicks, key presses, or mouse movements. To handle events, you attach event listeners to elements.

### ♦ **addEventListener()**

This method allows you to attach an event listener to an element. It is used to specify the type of event (e.g., `click`, `keydown`) and the function to run when the event occurs.

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

### ♦ **removeEventListener()**

This method removes an event listener from an element. It requires the same arguments as `addEventListener()`.

```
function clickHandler() {
    alert("Button clicked!");
}
```

```
document.getElementById("myButton").addEventListener("click",
clickHandler);
```

```
// To remove the event listener
document.getElementById("myButton").removeEventListener("click",
clickHandler);
```

---

## ✓ 6. Event Delegation & Bubbling

### ♦ **Event Bubbling**

When an event is triggered on an element, it bubbles up through its ancestors in the DOM hierarchy. This means that an event on a child element can be caught by its parent element.

```
document.getElementById("parentDiv").addEventListener("click",
function() {
    alert("Parent clicked");
});
```

```
document.getElementById("childDiv").addEventListener("click",
function() {
    alert("Child clicked");
});
```

- **Output:** If you click on `childDiv`, both the child and the parent click events will be triggered (because of event bubbling).

### ◆ Event Delegation

Event delegation is the practice of attaching a single event listener to a parent element that will handle events for child elements. This approach is particularly useful for dynamically added elements.

```
document.getElementById("parentDiv").addEventListener("click",
function(event) {
    if (event.target && event.target.matches("button")) {
        alert("Button clicked!");
    }
});
```

In this case, you only need one event listener on the `parentDiv`, and it will handle clicks on any `button` inside it, including dynamically added buttons.

---

## ✓ 7. localStorage & sessionStorage

Both `localStorage` and `sessionStorage` are used for storing data on the client-side in a browser.

## ♦ localStorage

- **Persistent storage:** Data stored in `localStorage` persists even when the browser is closed and reopened.
- Data is stored as key-value pairs.

```
// Store data
localStorage.setItem("username", "John");

// Retrieve data
let username = localStorage.getItem("username");
console.log(username); // John

// Remove data
localStorage.removeItem("username");

// Clear all data
localStorage.clear();
```

## ♦ sessionStorage

- **Temporary storage:** Data stored in `sessionStorage` is cleared when the browser session ends (when the tab is closed).
- Data is also stored as key-value pairs.

```
// Store data
sessionStorage.setItem("sessionUser", "Alice");

// Retrieve data
let sessionUser = sessionStorage.getItem("sessionUser");
console.log(sessionUser); // Alice

// Remove data
sessionStorage.removeItem("sessionUser");

// Clear all data
sessionStorage.clear();
```

---

## Summary Table

Concept	Description	Example
<b>What is the DOM?</b>	The DOM represents the page structure as a tree, which JavaScript can manipulate.	<code>document.getElementById()</code>
<b>Selecting Elements</b>	Methods for selecting DOM elements.	<code>document.querySelector()</code>
<b>Changing HTML &amp; CSS</b>	Modifying content and styles of elements.	<code>element.style.color = "red"</code>
<b>Creating &amp; Appending Elements</b>	Creating and adding new elements to the DOM.	<code>document.createElement()</code>
<b>Removing Elements</b>	Removing elements from the DOM.	<code>parentElement.removeChild()</code>
<b>Event Handling</b>	Handling events like <code>click</code> , <code>keydown</code> , etc.	<code>addEventListener()</code>
<b>Event Delegation</b>	Handling events from child elements using a single listener on a parent.	<code>event.target.matches()</code>
<b>localStorage</b>	Persistent client-side storage for key-value pairs.	<code>localStorage.setItem()</code>
<b>sessionStorage</b>	Temporary client-side storage for key-value pairs during a session.	<code>sessionStorage.setItem()</code>

# 13. Error Handling

Error handling is a critical part of writing reliable and maintainable JavaScript code. It helps catch unexpected situations (like user errors, server errors, or bugs in the code) and ensures the program can handle them gracefully without crashing.

---

## ✓ 1. try...catch...finally

The **try...catch...finally** statement is used to handle exceptions (runtime errors) in JavaScript. It allows the program to continue running even if an error occurs, preventing the script from terminating abruptly.

### ♦ try Block

The **try** block contains the code that might throw an error. If an error is thrown inside this block, control is transferred to the **catch** block.

### ♦ catch Block

The **catch** block contains the code that handles the error. It catches the error thrown from the **try** block.

### ♦ finally Block

The **finally** block is optional and contains code that will always run, regardless of whether an error occurred or not. It is usually used for cleanup activities (like closing a file or network connection).

```
try {  
    // Code that may throw an error  
    let result = 10 / 0;  
    console.log(result);  
} catch (error) {  
    // Handling the error  
    console.log("An error occurred:", error.message);  
} finally {  
    // Code that will always run, regardless of whether an error  
    occurred
```

```
    console.log("This will run no matter what!");  
}
```

### Output:

```
An error occurred: Infinity  
This will run no matter what!
```

In the example above, we try to divide by zero, which results in `Infinity`. The `catch` block catches this as an error, logs the message, and the `finally` block runs regardless of the error.

---

## 2. Throwing Custom Errors (throw new Error())

You can throw your own errors in JavaScript using the `throw` statement, which allows you to generate custom error messages. This is particularly useful for creating meaningful exceptions in your program's flow.

### Syntax:

```
throw new Error("Custom error message");
```

### Example:

```
function checkAge(age) {  
    if (age < 18) {  
        throw new Error("You must be 18 or older.");  
    }  
    return "Access granted";  
}  
  
try {  
    let access = checkAge(16);  
    console.log(access);  
} catch (error) {  
    console.log("Error:", error.message);  
}
```



## Output:

Error: You must be 18 or older.

In this example, the `checkAge` function throws an error if the age is less than 18. The error is caught and logged in the `catch` block.

---

## ✓ 3. onerror Event

The `onerror` event handler is a global error handler for handling errors that occur during the execution of scripts, including errors in images, scripts, and other resources. It can be attached to the `window` object to handle any uncaught JavaScript errors.

### Syntax:

```
window.onerror = function(message, source, lineno, colno, error) {  
    // Error handling code  
    console.log("Error: " + message);  
    return true; // Prevent the default browser error handler  
};
```

### Example:

```
window.onerror = function(message, source, lineno, colno, error) {  
    console.log("Error occurred: " + message);  
    console.log("Source: " + source);  
    console.log("Line number: " + lineno);  
    console.log("Column number: " + colno);  
    return true; // Stops the default error handling  
};  
  
// Trigger an error  
let result = nonexistentFunction();
```

## Output:

```
Error occurred: nonExistentFunction is not defined
Source: <script source URL>
Line number: 10
Column number: 5
```

In this example, when a function is called that does not exist (`nonExistentFunction`), the `onerror` event handler is triggered, logging the error message, source, line, and column of the error.

---

## Summary Table

Concept	Description	Example
<b>try...catch...finally</b>	A block to handle errors by catching them and optionally running code afterward.	<pre>try { ... } catch { ... } finally { ... }</pre>
<b>throw new Error()</b>	Used to throw custom errors with specific messages.	<pre>throw new Error("Custom error message")</pre>
<b>onerror Event</b>	A global event handler for uncaught errors in JavaScript.	<pre>window.onerror = function() { ... }</pre>

# 14. JavaScript Advanced Topics

These advanced JavaScript topics help developers understand how the language works behind the scenes, optimize performance, and build more efficient and effective applications. Let's dive into each of them:

---

## ✓ 1. Closures

A **closure** is a function that remembers its lexical scope (the environment in which it was defined) even when it's executed outside that scope. This allows inner functions to access variables from their outer functions even after the outer function has finished executing.

**Example:**

```
function outerFunction() {
    let outerVariable = "I'm from the outer function";

    function innerFunction() {
        console.log(outerVariable); // Closure: Accessing variable
        from outer function
    }

    return innerFunction;
}

const closureFunction = outerFunction();
closureFunction(); // Output: "I'm from the outer function"
```

Here, `innerFunction` is a closure because it still has access to `outerVariable` even after `outerFunction` has finished execution.

---

## ✓ 2. Hoisting

**Hoisting** is JavaScript's behavior of moving variable and function declarations to the top of their containing scope during the compile phase. This means that variables and functions can be

used before they are declared, but only function declarations and `var` variables (not `let` and `const`).

### Example:

```
console.log(x); // Output: undefined (due to hoisting)
var x = 5;

foo(); // Output: "Hello!" (hoisted function declaration)

function foo() {
  console.log("Hello!");
}
```

However, `let` and `const` do not get hoisted in the same way:

```
console.log(y); // ReferenceError: Cannot access 'y' before
initialization
let y = 10;
```

---

## 3. this Keyword in Different Contexts

The `this` keyword refers to the object that is executing the current function. It behaves differently depending on the context in which it is used:

1. **Global Context:** `this` refers to the global object (in browsers, it's `window`).
2. **Object Method:** `this` refers to the object calling the method.
3. **Constructor Function:** `this` refers to the newly created object.
4. **Arrow Functions:** `this` inside arrow functions is lexically bound to the surrounding context.

### Example:

```
// Global Context
console.log(this); // In browsers, it refers to `window`
```

```
// Object Method
const person = {
  name: 'John',
  greet: function() {
    console.log(this.name); // `this` refers to `person` object
  }
};
person.greet(); // Output: John

// Arrow Function
const greetArrow = () => {
  console.log(this); // `this` refers to the surrounding lexical
  context (global in this case)
};
greetArrow();
```

---

## ✓ 4. Currying Functions

**Currying** is a technique where a function is transformed into a sequence of functions, each taking one argument and returning a new function. This helps to create reusable functions that can be partially applied.

### Example:

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

const addFive = add(5);
console.log(addFive(3)); // Output: 8
```

In this example, the `add` function is curried, allowing the creation of a partially applied function `addFive`.

---

## ✓ 5. call(), apply(), bind()

These methods allow you to control the value of `this` within functions, making them useful for invoking functions in different contexts:

- **call()**: Immediately invokes the function with `this` set to the specified object and passes arguments individually.
- **apply()**: Like `call()`, but arguments are passed as an array.
- **bind()**: Returns a new function with `this` bound to the specified object (but does not invoke it immediately).

### Example:

```
function greet(greeting, name) {  
    console.log(greeting + ", " + name + "!");  
}  
  
greet.call(null, "Hello", "John"); // Output: Hello, John!  
greet.apply(null, ["Hi", "Jane"]); // Output: Hi, Jane!  
  
const boundGreet = greet.bind(null, "Hey");  
boundGreet("Alice"); // Output: Hey, Alice!
```

---

## ✓ 6. Generators & Iterators

A **generator** is a special function that can pause its execution and resume it later, allowing you to iterate over a sequence of values. It is created using the `function*` syntax, and you can use `yield` to return a value.

- **Iterator**: An object with a `next()` method that returns a sequence of values.

### Example:

```
function* numberGenerator() {  
    yield 1;  
    yield 2;  
    yield 3;  
}
```

```
const gen = numberGenerator();
console.log(gen.next()); // Output: { value: 1, done: false }
console.log(gen.next()); // Output: { value: 2, done: false }
console.log(gen.next()); // Output: { value: 3, done: false }
console.log(gen.next()); // Output: { value: undefined, done: true }
```

---

## ✓ 7. WeakMap & WeakSet

**WeakMap** and **WeakSet** are collections that hold objects, but unlike regular **Map** and **Set**, their keys or values are weakly referenced. This means that if there are no other references to the object, it can be garbage collected.

- **WeakMap**: Similar to **Map**, but keys must be objects.
- **WeakSet**: Similar to **Set**, but values must be objects.

### Example:

```
let obj = {name: "John"};
let weakMap = new WeakMap();
weakMap.set(obj, "some value");
console.log(weakMap.get(obj)); // Output: "some value"
```

---

## ✓ 8. Memoization

**Memoization** is an optimization technique used to improve the performance of functions by caching previously computed results based on the input arguments. It is useful for functions that are computationally expensive and called multiple times with the same arguments.

### Example:

```
function memoize(fn) {
  const cache = {};
  return function(...args) {
    const key = JSON.stringify(args);
    if (cache[key]) {
```

```

        return cache[key];
    }
    const result = fn(...args);
    cache[key] = result;
    return result;
};
}

const slowFunction = (n) => {
    console.log("Calculating...");
    return n * 2;
};

const fastFunction = memoize(slowFunction);

console.log(fastFunction(5)); // Output: Calculating... 10
console.log(fastFunction(5)); // Output: 10 (cached result)

```

---

## 9. Debouncing & Throttling

**Debouncing** and **throttling** are techniques used to limit the frequency of function calls, often used for events like scrolling, resizing, or typing.

- **Debouncing**: Ensures a function is only called after a certain amount of time has passed since the last call.
- **Throttling**: Ensures a function is called at most once every specified period of time.

### Example (Debouncing):

```

function debounce(func, delay) {
    let timer;
    return function(...args) {
        clearTimeout(timer);
        timer = setTimeout(() => func(...args), delay);
    };
}

const logInput = debounce((input) => console.log(input), 500);

```



```
document.getElementById("input").addEventListener("input", (e) =>
logInput(e.target.value));
```

---

## ✓ 10. Web Workers

**Web Workers** allow you to run JavaScript code in the background on a separate thread, which helps improve performance by preventing the main thread from being blocked (useful for heavy computations or I/O operations).

### Example:

```
// main.js
const worker = new Worker('worker.js');
worker.postMessage('Start computation');

worker.onmessage = function(event) {
    console.log('Result from worker:', event.data);
};

// worker.js
onmessage = function(event) {
    const result = event.data + " completed!";
    postMessage(result);
};
```

---

## Summary Table

Concept	Description	Example
<b>Closures</b>	Functions that remember their lexical scope even after execution completes.	<pre>function outer() {   return function()   {...}}</pre>
<b>Hoisting</b>	JavaScript moves declarations to the top during the compile phase.	<pre>var x; console.log(x)</pre>

<b>this Keyword</b>	Refers to the object in the current context (global, object, or constructor).	<code>this</code> in methods, constructors, etc.
<b>Currying Functions</b>	Creating functions that take one argument at a time.	<pre>function add(a) {   return function(b)     {...}}}</pre>
<b>call(), apply(), bind()</b>	Methods to control <code>this</code> context and pass arguments.	<pre>greet.call(this,   'Hello', 'John')</pre>
<b>Generators &amp; Iterators</b>	Functions that yield values one at a time, allowing iteration.	<pre>function* gen() {   yield 1; yield 2; }</pre>

## 15. JavaScript & Browser APIs

Browser APIs allow JavaScript to interact with the browser environment, providing functionality like timing events, reading files, or interacting with the user's device. Let's explore some important ones:

---

## ✓ 1. `setTimeout()`, `setInterval()`

These are two commonly used timing functions in JavaScript for scheduling tasks.

- **`setTimeout()`**: Executes a function after a specified delay (in milliseconds).
- **`setInterval()`**: Repeatedly executes a function at specified intervals (in milliseconds).

### Example:

```
// setTimeout() example
setTimeout(() => {
  console.log("This message is displayed after 2 seconds.");
}, 2000);

// setInterval() example
let count = 0;
const intervalId = setInterval(() => {
  count++;
  console.log(`This message is displayed every second. Count: ${count}`);
  if (count === 5) clearInterval(intervalId); // Stop after 5 iterations
}, 1000);
```

- `setTimeout()` runs once after the specified time.
  - `setInterval()` keeps running at the given interval until stopped with `clearInterval()`.
- 

## ✓ 2. `navigator` API

The **`navigator` API** allows access to the browser's environment and system information, such as the browser version, language, platform, and online/offline status.

## Common Methods/Properties:

- **navigator.geolocation**: Access the user's geographic position.
- **navigator.language**: Get the preferred language of the browser.
- **navigator.onLine**: Check if the browser is online or offline.

## Example:

```
// Check if the user is online
if (navigator.onLine) {
    console.log("You are online!");
} else {
    console.log("You are offline.");
}

// Get browser language
console.log(navigator.language); // Output: e.g., "en-US"
```

---

## ✓ 3. Geolocation API

The **Geolocation API** allows you to get the geographic position of the user, including latitude and longitude, with their consent. It is commonly used in location-based applications like maps or weather services.

## Example:

```
if (navigator.geolocation) {
    navigator.geolocation.getCurrentPosition((position) => {
        const { latitude, longitude } = position.coords;
        console.log(`Latitude: ${latitude}, Longitude: ${longitude}`);
    }, (error) => {
        console.log("Error getting geolocation:", error);
    });
} else {
    console.log("Geolocation is not supported by this browser.");
}
```

You can also watch the user's position over time with `watchPosition()`, which continuously updates the location.

---

## ✓ 4. FileReader API

The **FileReader API** allows JavaScript to read files from the user's local system (such as images or documents) and use the content in the browser without needing to upload the file to a server.

### Common Methods:

- **readAsText()**: Reads a file as a text string.
- **readAsDataURL()**: Reads a file as a base64-encoded data URL.
- **readAsArrayBuffer()**: Reads a file as a binary array buffer.

### Example:

```
// Input element to select a file
const fileInput = document.getElementById("fileInput");
fileInput.addEventListener("change", (event) => {
    const file = event.target.files[0];
    const reader = new FileReader();

    // Read the file as text
    reader.onload = () => {
        console.log(reader.result); // File content as text
    };
    reader.readAsText(file);
});
```

This example reads a text file selected by the user and logs its content.

---

## ✓ 5. Canvas API

The **Canvas API** allows you to draw graphics, animations, and images directly onto a `<canvas>` element. It's useful for creating dynamic graphics like games, charts, and more.

## Common Methods:

- **getContext():** Returns a drawing context (e.g., `2d` or `webgl`).
- **fillRect(), strokeRect(), arc():** Methods for drawing shapes.
- **drawImage():** Draws an image onto the canvas.

## Example:

```
<canvas id="myCanvas" width="200" height="200"></canvas>

<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");

  // Draw a rectangle
  ctx.fillStyle = "blue";
  ctx.fillRect(50, 50, 100, 100);

  // Draw a circle
  ctx.beginPath();
  ctx.arc(100, 100, 50, 0, Math.PI * 2);
  ctx.fillStyle = "red";
  ctx.fill();
</script>
```

In this example, we draw a blue square and a red circle on the canvas.

---

## Summary Table

API	Description	Example
<b>setTimeout()</b>	Executes a function after a specified delay.	<code>setTimeout(() =&gt; console.log("Hello"), 1000);</code>
<b>setInterval()</b>	Repeatedly executes a function at a specified interval.	<code>setInterval(() =&gt; console.log("Tick"), 1000);</code>

<b>navigator API</b>	Provides access to browser/system information.	<code>navigator.language, navigator.onLine</code>
<b>Geolocation API</b>	Gets the user's geographic location with their consent.	<code>navigator.geolocation.getCurrentPosition()</code>
<b>FileReader API</b>	Allows reading files from the user's local system.	<code>reader.readAsText(file)</code>
<b>Canvas API</b>	Enables drawing graphics directly in the browser.	<code>ctx.fillRect(50, 50, 100, 100)</code>

## 16. JavaScript Testing

Testing is a crucial part of development that helps ensure the functionality of your code and catch any bugs early. In JavaScript, there are various ways to test your code, and in this section, we'll cover **unit testing with Jest** and **debugging with Console & DevTools**.

---

## ✓ 1. Unit Testing with Jest

Unit testing involves testing individual units (functions, methods, etc.) of your code to ensure they work as expected. **Jest** is a popular JavaScript testing framework that simplifies the process of writing and running tests.

### Setting Up Jest:

Install Jest in your project:

```
npm install --save-dev jest
```

1.

In the `package.json`, add a test script:

```
"scripts": {  
  "test": "jest"  
}
```

2.

3. Create a test file (e.g., `sum.test.js`) in your `__tests__` folder or alongside the file you're testing.

### Writing Tests:

Jest provides functions like `test()`, `it()`, and `expect()` to write and run your tests.

- **test()**: Defines a test.
- **expect()**: Defines an expectation that the code should meet.

### Example:

```
// Function to test  
function sum(a, b) {  
  return a + b;  
}
```



```
// Unit test
test('adds 1 + 2 to equal 3', () => {
  expect(sum(1, 2)).toBe(3);
});

test('adds -1 + 2 to equal 1', () => {
  expect(sum(-1, 2)).toBe(1);
});
```

- `expect(sum(1, 2)).toBe(3)` checks if `sum(1, 2)` equals 3.
- Jest also provides matchers like `.toBe()`, `.toEqual()`, `.toBeTruthy()`, `.toBeFalsy()`, etc., to validate different kinds of values.

## Running Tests:

Run the tests using the following command:

```
npm test
```

This will run Jest and output the results of the tests.

---

## ✓ 2. Debugging with Console & DevTools

Debugging is the process of identifying and fixing issues in your code. JavaScript provides a variety of tools for debugging.

### Using **console** Methods:

**console.log()**: Logs any information to the console.

```
console.log("Debugging value:", myVariable);
```

**console.error()**: Logs error messages with a red highlight.

javascript

CopyEdit

```
console.error("This is an error message");
```

**console.warn():** Logs warning messages with a yellow highlight.

javascript

CopyEdit

```
console.warn("This is a warning message");
```

**console.table():** Displays an object or array in a tabular format.

javascript

CopyEdit

```
const person = { name: "John", age: 30 };  
console.table(person);
```

**console.trace():** Displays the stack trace for function calls.

javascript

CopyEdit

```
function myFunction() {  
    console.trace();  
}  
myFunction();
```

## Using Browser DevTools:

Most modern browsers come with built-in Developer Tools (DevTools), which can be accessed by right-clicking on a webpage and selecting **Inspect**.

- **Console Tab:** Allows you to view logs, warnings, and errors.
- **Sources Tab:** Lets you debug JavaScript code by setting breakpoints, stepping through code, and inspecting variables at runtime.
- **Network Tab:** Helps monitor network requests, such as API calls, and view their responses.
- **Elements Tab:** Used to inspect and modify the HTML and CSS of a page.

## Setting Breakpoints:

You can set breakpoints directly in the **Sources** tab in DevTools:

1. Open the **Sources** tab in Chrome DevTools.
2. Find the JavaScript file you want to debug.
3. Click on the line number where you want to pause execution.

4. Reload the page or trigger the JavaScript function to hit the breakpoint.
5. Once paused, you can step through your code, inspect variables, and see where the problem lies.

### Example: Debugging with DevTools:

```
function multiply(a, b) {  
  let result = a * b;  
  console.log("Multiplying:", a, "and", b);  
  return result;  
}  
  
multiply(2, 3); // Open DevTools and set a breakpoint here
```

---

## Summary Table

Concept	Description	Example
<b>Unit Testing with Jest</b>	Testing functions or units of code to ensure they work as expected.	<code>test('adds 1 + 2 to equal 3', () =&gt; { expect(sum(1, 2)).toBe(3); });</code>
<b>console.log()</b>	Logs any data to the console for debugging.	<code>console.log("Debugging value:", myVariable);</code>
<b>console.error()</b>	Logs error messages to the console.	<code>console.error("This is an error message");</code>
<b>console.warn()</b>	Logs warning messages to the console.	<code>console.warn("This is a warning message");</code>
<b>console.table()</b>	Logs data in a tabular format.	<code>console.table([{name: "John", age: 30}]);</code>
<b>console.trace()</b>	Displays a stack trace to track where the code was called from.	<code>console.trace();</code>

**DevTools  
(Breakpoints)**

Allows stepping through JavaScript code, inspecting variables, and debugging.

Set breakpoints in the Sources tab of Chrome DevTools.

---

**Conclusion:**

- **Jest** provides a simple way to automate testing and ensure your code is functioning as expected. It helps catch errors early and improves the maintainability of your code.
- **Console & DevTools** are invaluable tools for manual debugging, allowing you to inspect variables, track code execution, and resolve issues interactively.

By using both **unit testing** and **debugging tools**, you ensure your JavaScript code is robust, reliable, and error-free.

## 17. JavaScript Frameworks & Libraries

In modern web development, JavaScript frameworks and libraries play an essential role in creating dynamic and interactive web applications. They provide built-in functionalities, tools, and structures to streamline development and enhance productivity. In this section, we will introduce the three most popular JavaScript frameworks and libraries: **React.js**, **Vue.js**, and **Angular**. Additionally, we will cover **State Management Concepts** and provide an **Overview of Redux**.

---

## ✓ 1. Introduction to React.js, Vue.js, and Angular

### React.js:

- **React.js** is a declarative, efficient, and flexible JavaScript library for building user interfaces (UIs), especially for single-page applications. Developed by Facebook, React allows developers to build reusable UI components that can dynamically update in response to data changes.
- **Key Features:**
  - Component-based architecture
  - Virtual DOM for better performance
  - JSX (JavaScript XML) syntax for defining components
  - Unidirectional data flow
- **When to Use:** Ideal for building complex UIs with reusable components, especially in single-page applications (SPAs).

### Example:

```
import React from 'react';

function Greeting(props) {
  return <h1>Hello, {props.name}!</h1>;
}

export default Greeting;
```

- 

### Vue.js:

- **Vue.js** is a progressive JavaScript framework used for building UIs and SPAs. Vue is lightweight, flexible, and easy to integrate into projects, even with existing codebases. Vue is designed to be incrementally adoptable and is known for its simplicity.
- **Key Features:**

- Two-way data binding (like Angular)
- Component-based architecture
- Reactive and declarative binding
- Detailed and easy-to-understand documentation
- **When to Use:** Great for both small and large-scale projects, especially when simplicity and flexibility are prioritized.

#### Example:

```
<template>
  <div>
    <p>Hello, {{ name }}!</p>
  </div>
</template>
```

```
<script>
export default {
  data() {
    return {
      name: 'John',
    };
  },
};
</script>
```

#### Angular:

- **Angular** is a full-fledged framework developed by Google for building SPAs. It is a robust, opinionated framework that provides a wide range of tools and functionalities out of the box. Angular uses TypeScript, a statically typed superset of JavaScript, for development.
- **Key Features:**
  - Two-way data binding
  - Dependency Injection (DI) for managing services and components
  - Built-in directives and pipes
  - Angular CLI for scaffolding projects and managing builds
  - TypeScript-based development for enhanced tooling and type safety
- **When to Use:** Best suited for large-scale enterprise applications and projects that require structured frameworks with powerful tooling.

### Example:

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  template: '<h1>Hello, {{ name }}!</h1>',
})
export class AppComponent {
  name: string = 'John';
}
```

- 

---

## ✓ 2. State Management Concepts

State management is an essential concept in modern web development, especially in complex applications. State refers to the data or information about the application at any given time. Proper state management ensures that the application can efficiently update, display, and share data across components.

### Types of State:

1. **Local State:** This state is confined to a single component, typically managed within the component itself.
  - Example: A form input field storing a user's input.
2. **Global State:** This state is shared across multiple components in an application. Managing global state ensures that components can access and update data without unnecessary re-rendering.
  - Example: A user authentication status that needs to be accessed across different parts of the app.
3. **Server State:** Data that is fetched from a server (API calls, etc.) and needs to be managed within the app to reflect updates.
  - Example: A list of products fetched from a backend service.

### State Management Challenges:

- Passing state from one component to another through props can become cumbersome in large applications.
- Keeping track of how and where the state is updated can be error-prone.

- In complex apps, state management becomes difficult if the state is not structured efficiently.
- 

## ✓ 3. Redux Overview

**Redux** is a predictable state container for JavaScript applications, commonly used in React.js applications. It allows you to manage the global state of an application in a centralized store, making state transitions predictable and traceable.

### Core Principles of Redux:

1. **Single Source of Truth:** The state of the entire application is stored in a single object called the store.
2. **State is Read-Only:** The only way to change the state is by dispatching actions, which are plain objects describing what happened.
3. **Changes are Made with Pure Functions:** The only way to specify how the state changes is by writing pure reducers, which are functions that return a new state based on the previous state and an action.

### Redux Flow:

1. **Action:** An object that describes a change in the application state (e.g., `{ type: 'ADD_ITEM', payload: { id: 1, name: 'Item 1' } }`).
2. **Reducer:** A function that specifies how the application's state changes in response to an action. It takes the current state and the action and returns a new state.
3. **Store:** The global state container that holds the state and allows access to it via `getState()`.
4. **Dispatch:** The method used to send actions to the store, causing the reducer to run and update the state.
5. **Subscribe:** Allows components to listen to state changes and automatically re-render when the state changes.

### Example (Redux with React):

**Action:**

```
const addItem = (item) => ({
  type: 'ADD_ITEM',
  payload: item,
});
```

- 1.



### Reducer:

```
const initialState = {
  items: [],
};

const itemReducer = (state = initialState, action) => {
  switch (action.type) {
    case 'ADD_ITEM':
      return {
        ...state,
        items: [...state.items, action.payload],
      };
    default:
      return state;
  }
};
```

2.

### Store:

```
import { createStore } from 'redux';
const store = createStore(itemReducer);
```

3.

### Dispatching Actions:

```
store.dispatch(addItem({ id: 1, name: 'Item 1' }));
```

4.

### Connecting Redux with React (via `react-redux`):

```
import { Provider, connect } from 'react-redux';

function App({ items }) {
  return (
    <div>
      <h1>Items</h1>
    </div>
  );
}
```

```

        <ul>
          {items.map((item) => (
            <li key={item.id}>{item.name}</li>
          ))}
        </ul>
      </div>
    );
  }

const mapStateToProps = (state) => ({
  items: state.items,
});

const ConnectedApp = connect(mapStateToProps)(App);

export default function Root() {
  return (
    <Provider store={store}>
      <ConnectedApp />
    </Provider>
  );
}

```

5.

## Advantages of Redux:

- Centralized and predictable state management.
- Makes state transitions easier to track and debug.
- Helps in building scalable applications with complex state logic.

---

## Summary Table

Concept	Description	Example
<b>React.js</b>	A JavaScript library for building user interfaces using components.	<pre>function Greeting(props) {   return &lt;h1&gt;Hello,     {props.name}!&lt;/h1&gt;; }</pre>

<b>Vue.js</b>	A lightweight, flexible JavaScript framework for building SPAs.	<pre>&lt;template&gt;&lt;p&gt;{{ name }}&lt;/p&gt;&lt;/template&gt;</pre>
<b>Angular</b>	A comprehensive TypeScript-based framework for building large-scale SPAs.	<pre>@Component({ selector: 'app-root', template: '&lt;h1&gt;{{ name }}&lt;/h1&gt;' })</pre>
<b>State Management</b>	The process of managing data and state across an application.	Managing authentication state across multiple components.
<b>Redux</b>	A state management library that centralizes state and actions.	<pre>const store = createStore(itemReducer); store.dispatch(addItem(item));</pre>

---

## Conclusion:

- **React.js, Vue.js, and Angular** each have their unique features and advantages for building modern web applications. Choose a framework/library based on project needs, scalability, and developer preferences.
- **State Management** ensures smooth data flow and sharing across components, while **Redux** helps manage global state in a centralized manner for more predictable and debuggable applications.

# 18. JavaScript Design Patterns

Design patterns are reusable solutions to common problems in software design. They help in creating more maintainable and scalable applications by providing standardized approaches to solving recurring issues. In JavaScript, design patterns can be applied to different programming scenarios, such as object creation, state management, and communication between objects. In this section, we'll cover four popular design patterns: **Singleton**, **Factory Pattern**, **Observer Pattern**, and **Module Pattern**.

---

## ✓ 1. Singleton Pattern

### Definition:

The **Singleton** pattern ensures that a class has only one instance, and it provides a global point of access to that instance. This is useful when you need to manage a shared resource (e.g., database connection, logging service) across an application, ensuring that multiple parts of the application don't create multiple instances of the same object.

### When to Use:

- When you need to ensure that a class has only one instance.
- When a global point of access to the instance is required.

### Example:

```
class Singleton {
  constructor() {
    if (!Singleton.instance) {
      this.data = "Singleton Data";
      Singleton.instance = this;
    }

    return Singleton.instance;
  }

  getData() {
    return this.data;
  }
}
```

// Usage

```
const instance1 = new Singleton();
const instance2 = new Singleton();

console.log(instance1 === instance2); // true
console.log(instance1.getData()); // "Singleton Data"
```

---

## ✓ 2. Factory Pattern

### Definition:

The **Factory** pattern provides an interface for creating objects in a super class, but allows subclasses to alter the type of objects that will be created. It is used when you don't know beforehand what class of objects you might need to create.

### When to Use:

- When you have a class that should create different objects based on certain conditions.
- When object creation logic is complex or involves multiple steps.

### Example:

```
// Factory Function
function VehicleFactory(type) {
  if (type === 'car') {
    return new Car();
  } else if (type === 'bike') {
    return new Bike();
  }
}

class Car {
  drive() {
    return "Driving a car";
  }
}

class Bike {
  drive() {
```

```
        return "Riding a bike";
    }
}

// Usage
const car = VehicleFactory('car');
console.log(car.drive()); // "Driving a car"

const bike = VehicleFactory('bike');
console.log(bike.drive()); // "Riding a bike"
```

---

### 3. Observer Pattern

#### Definition:

The **Observer** pattern is used to allow an object (the subject) to notify a list of dependent objects (observers) about state changes, typically by calling one of their methods. It is commonly used in event-driven systems or applications with a need for components to react to changes in other components.

#### When to Use:

- When an object changes state and you need to notify other objects about this change.
- When you have multiple objects that need to be updated automatically when another object's state changes.

#### Example:

```
class Subject {
    constructor() {
        this.observers = [];
    }

    addObserver(observer) {
        this.observers.push(observer);
    }

    notifyObservers() {
```

```
        this.observers.forEach(observer => observer.update(this));
    }
}

class Observer {
    update(subject) {
        console.log(`Observer: Subject state has changed`);
    }
}

// Usage
const subject = new Subject();
const observer1 = new Observer();
const observer2 = new Observer();

subject.addObserver(observer1);
subject.addObserver(observer2);

subject.notifyObservers(); // "Observer: Subject state has changed"
                             (twice)
```

---

## ✓ 4. Module Pattern

### Definition:

The **Module** pattern is used to create private and public encapsulation of functionality. It provides a way to organize and group code together into reusable modules while keeping some of the data and functions private and hidden from the outside world.

### When to Use:

- When you need to organize related code into reusable modules.
- When you want to encapsulate the internal workings of a module and expose only necessary functionality.

### Example:

```
const CounterModule = (function () {
```

```

let count = 0; // Private variable

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

// Usage
CounterModule.increment(); // 1
CounterModule.increment(); // 2
CounterModule.decrement(); // 1
console.log(CounterModule.getCount()); // 1

```

---

## Summary Table

Design Pattern	Description	Example Usage
<b>Singleton</b>	Ensures that a class has only one instance and provides global access to it.	Global configuration manager, logging service.
<b>Factory</b>	Provides an interface for creating objects, allowing subclasses to determine the type of object to create.	Vehicle creation, shape drawing (circle, square).
<b>Observer</b>	Allows an object to notify dependent objects about changes to its state.	Event listeners, chat application notifications.



<b>Module</b>	Encapsulates functionality in a module with private and public access.	Organizing code into reusable modules, like a counter.
---------------	--	--

---

## Conclusion:

- **Singleton**, **Factory**, **Observer**, and **Module** patterns provide structured approaches to common problems in JavaScript programming.
- These patterns enhance maintainability, scalability, and reusability of code, helping developers design efficient, organized applications.

# 19. Security & Best Practices

In web development, security is crucial to protect applications and users from various malicious attacks and vulnerabilities. JavaScript, being a key language for both frontend and backend development, has its own set of security concerns. In this section, we'll explore three common web security issues: **Cross-Site Scripting (XSS)**, **Cross-Site Request Forgery (CSRF)**, and **Secure Coding Practices**.

---

## ✓ 1. Cross-Site Scripting (XSS)

### Definition:

**Cross-Site Scripting (XSS)** is a security vulnerability that allows attackers to inject malicious scripts into web pages viewed by users. These scripts can steal cookies, session tokens, or any sensitive information from users, or even perform actions on behalf of the user without their consent.

### Types of XSS:

1. **Stored XSS:** The malicious script is permanently stored on the target server, typically in a database.
2. **Reflected XSS:** The malicious script is reflected off a web server, typically as a URL or query parameter.
3. **DOM-based XSS:** The attack occurs when the malicious script is executed as a result of client-side manipulation of the DOM.

### Prevention:

- **Input Validation and Sanitization:** Always validate and sanitize user input before rendering it on the page.
- **Use Content Security Policy (CSP):** CSP is a browser feature that helps mitigate XSS attacks by restricting the sources from which content can be loaded.
- **Escaping Output:** When rendering dynamic content in the browser, ensure that special characters (like `<`, `>`, etc.) are properly escaped to prevent execution of malicious code.

### Example (Vulnerable Code):

```
<!-- Example of vulnerable code -->
<input type="text" id="username" value="John">
<p>Welcome, <span id="user"></span></p>
```

```
<script>
  var user = document.getElementById('username').value;
  document.getElementById('user').innerHTML = user; // Vulnerable to
XSS
</script>
```

### Prevention (Sanitization):

```
<script>
  var user = document.getElementById('username').value;
  document.getElementById('user').textContent = user; // Safe way to
insert text
</script>
```

---

## 2. Cross-Site Request Forgery (CSRF)

### Definition:

**Cross-Site Request Forgery (CSRF)** is a type of attack where a malicious website or script tricks a user into performing actions on a web application where they are authenticated. The attacker uses the credentials of the authenticated user to make unauthorized requests.

### When It Happens:

- If a user is logged in to a web application, the attacker can use the user's session (e.g., cookies) to perform actions without their knowledge.

### Prevention:

- **Anti-CSRF Tokens:** Include a unique token in every form submitted by the user to verify that the request is legitimate and coming from the user's session.
- **SameSite Cookies:** Use the **SameSite** attribute for cookies to prevent browsers from sending cookies along with cross-site requests.
- **Referer Header Validation:** Ensure that requests come from the intended source by validating the **Referer** header in HTTP requests.

### Example (Vulnerable Code):

```
<!-- Malicious website -->

```

### Prevention (Anti-CSRF Token):

```
// Include CSRF token in each request
fetch('/transferFunds', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    'X-CSRF-Token': 'your-csrf-token-here' // Include CSRF token
  },
  body: JSON.stringify({ amount: 1000, to: 'attacker' })
});
```

---

## ✓ 3. Secure Coding Practices

### Definition:

**Secure coding practices** refer to a set of guidelines, techniques, and tools that developers follow to write secure code that minimizes vulnerabilities and prevents attacks like XSS, CSRF, SQL injection, and others.

### Best Practices:

1. **Use HTTPS:** Always use HTTPS to ensure secure communication between the client and server.
2. **Input Validation & Output Encoding:** Always validate input, sanitize output, and escape any dynamic content.
3. **Use Parameterized Queries:** Prevent **SQL Injection** by using parameterized queries instead of concatenating SQL strings.
4. **Avoid Storing Sensitive Data in Cookies:** Sensitive data, like passwords and session tokens, should not be stored in cookies. Use secure methods for authentication like OAuth or JWT.
5. **Minimize Privileges:** Grant the least privilege possible to the user, limiting access to sensitive data and features.

6. **Use Strong Authentication:** Implement strong authentication mechanisms, such as multi-factor authentication (MFA).
7. **Security Headers:** Implement security headers such as:
  - **Strict-Transport-Security (HSTS):** Forces the browser to use HTTPS only.
  - **X-Content-Type-Options:** Prevents MIME-sniffing attacks.
  - **X-Frame-Options:** Protects against clickjacking.

### Example (Secure HTTP Headers):

```
// Express.js Example: Setting security headers
const helmet = require('helmet');
app.use(helmet()); // Automatically sets common security headers like
XSS Protection, Content-Security-Policy, etc.
```

---

### Summary Table

Security Concern	Description	Prevention Techniques
<b>XSS (Cross-Site Scripting)</b>	Injection of malicious scripts into web pages.	Input validation, output escaping, Content Security Policy (CSP).
<b>CSRF (Cross-Site Request Forgery)</b>	Attacks where unauthorized actions are performed using authenticated users' credentials.	Anti-CSRF tokens, SameSite cookies, referer header validation.
<b>Secure Coding Practices</b>	General best practices to prevent security vulnerabilities.	Use HTTPS, input validation, avoid storing sensitive data in cookies, security headers.

---

### Conclusion:

- **XSS** and **CSRF** are common attack vectors that can compromise the security of your web applications, but they can be mitigated with proper coding practices.
- **Secure coding practices** should be a fundamental part of your development workflow to minimize risks and protect sensitive data.

# 20. JavaScript in the Backend (Node.js & Express.js Basics)

In this section, we will explore how JavaScript can be used in backend development with **Node.js** and **Express.js**. Node.js allows JavaScript to be executed on the server side, while Express.js is a popular framework built on top of Node.js to simplify building web applications and APIs.

---

## ✓ 1. Introduction to Node.js

### Definition:

**Node.js** is an open-source, cross-platform JavaScript runtime environment that allows you to run JavaScript on the server side. Unlike traditional JavaScript, which runs in the browser, Node.js enables JavaScript to interact with files, databases, and other server-side resources.

### Why Use Node.js?:

- **Event-Driven & Non-Blocking:** Node.js is designed around an event-driven, non-blocking I/O model, making it lightweight and efficient for handling multiple concurrent requests.
- **Single Language:** You can use JavaScript for both frontend and backend development, streamlining the development process.
- **NPM (Node Package Manager):** Node.js comes with NPM, which provides a vast collection of libraries and packages for development.

### Example (Simple Node.js Server):

```
// Importing the http module
const http = require('http');

// Creating an HTTP server
const server = http.createServer((req, res) => {
  res.write('Hello, World!');
  res.end();
});

// Starting the server on port 3000
```

```
server.listen(3000, () => {  
  console.log('Server is running on http://localhost:3000');  
});
```

---

## ✓ 2. Using npm & Package.json

### Definition:

- **npm** (Node Package Manager) is a tool that comes with Node.js to manage libraries, dependencies, and packages.
- **package.json** is a file that holds metadata for your project (name, version, dependencies, etc.) and keeps track of all packages required for your project.

### How npm Works:

- **Installing Packages:** You can use npm to install third-party packages, such as Express, MongoDB, etc.
- **Managing Dependencies:** All dependencies used in a project are listed in `package.json` under the `dependencies` section.

### Example (Creating a `package.json` and Installing Packages):

Initialize a new Node.js project:

```
npm init -y
```

1.

Install Express.js:

bash

CopyEdit

```
npm install express
```

2.

`package.json` will be updated with the installed dependencies:

json

CopyEdit

```
{  
  "name": "my-node-app",  
  "version": "1.0.0",
```

```
"dependencies": {  
  "express": "^4.17.1"  
}
```

---

## ✓ 3. Express.js Basics

### Definition:

**Express.js** is a minimal and flexible web application framework for Node.js. It simplifies the development of web applications and APIs by providing a robust set of features such as routing, middleware, templating engines, etc.

### How Express.js Works:

- Express is used to handle HTTP requests and responses, set up routes, and manage middleware.
- It is built on top of Node.js's HTTP module but provides a simpler and more intuitive way of creating server-side logic.

### Example (Basic Express Server):

```
// Importing the express module  
const express = require('express');  
  
// Creating an Express app  
const app = express();  
  
// Setting up a basic route  
app.get('/', (req, res) => {  
  res.send('Hello from Express!');  
});  
  
// Starting the server  
app.listen(3000, () => {  
  console.log('Express server running on http://localhost:3000');  
});
```



---

## ✓ 4. REST API Development

### Definition:

**REST** (Representational State Transfer) is an architectural style for designing networked applications. A **RESTful API** allows clients (usually frontend applications) to interact with a backend server using standard HTTP methods: GET, POST, PUT, DELETE.

### REST API with Express:

Express.js is widely used to create REST APIs because of its simple syntax for defining routes and handling HTTP requests.

### Example (Creating a Basic REST API):

```
const express = require('express');
const app = express();

// Middleware to parse JSON body
app.use(express.json());

// Sample data
let users = [
  { id: 1, name: 'Alice' },
  { id: 2, name: 'Bob' }
];

// GET all users
app.get('/users', (req, res) => {
  res.json(users);
});

// GET a specific user by ID
app.get('/users/:id', (req, res) => {
  const user = users.find(u => u.id == req.params.id);
  if (user) {
    res.json(user);
  }
});
```

```
    } else {
      res.status(404).json({ message: 'User not found' });
    }
  });

// POST new user
app.post('/users', (req, res) => {
  const newUser = req.body;
  users.push(newUser);
  res.status(201).json(newUser);
});

// PUT (update) an existing user
app.put('/users/:id', (req, res) => {
  const user = users.find(u => u.id == req.params.id);
  if (user) {
    user.name = req.body.name;
    res.json(user);
  } else {
    res.status(404).json({ message: 'User not found' });
  }
});

// DELETE a user
app.delete('/users/:id', (req, res) => {
  const index = users.findIndex(u => u.id == req.params.id);
  if (index !== -1) {
    users.splice(index, 1);
    res.status(204).end();
  } else {
    res.status(404).json({ message: 'User not found' });
  }
});

// Starting the server
app.listen(3000, () => {
  console.log('REST API server running on http://localhost:3000');
});
```

---

## ✓ 5. Middleware & Authentication

### Definition:

**Middleware** in Express is a function that is executed during the request-response cycle, before the final request handler. Middleware can be used for a variety of tasks such as logging, authentication, error handling, etc.

**Authentication** is the process of verifying the identity of a user. Common methods include using **JWT (JSON Web Tokens)** or **sessions**.

### Example (Using Middleware for Authentication):

#### 1. Simple Logging Middleware:

```
// Logging middleware
const logRequest = (req, res, next) => {
  console.log(`${req.method} request made to ${req.url}`);
  next(); // Pass the request to the next middleware or route handler
};

app.use(logRequest); // Apply logging middleware globally
```

#### 2. Authentication Middleware (JWT Example):

```
const jwt = require('jsonwebtoken');

// Authentication middleware
const authenticate = (req, res, next) => {
  const token = req.header('Authorization');
  if (!token) {
    return res.status(401).json({ message: 'Access denied' });
  }

  try {
    const verified = jwt.verify(token, 'secretKey');
```

```

    req.user = verified;
    next(); // Continue to the next route handler
  } catch (err) {
    res.status(400).json({ message: 'Invalid token' });
  }
};

// Protecting a route with authentication
app.get('/protected', authenticate, (req, res) => {
  res.send('This is a protected route');
});

```

---

## Summary Table

Topic	Description	Example Code
<b>Node.js Introduction</b>	JavaScript runtime for server-side development.	Simple Node.js HTTP server
<b>Using npm &amp; Package.json</b>	Managing dependencies and packages in Node.js.	<code>npm install express</code> to install packages.
<b>Express.js Basics</b>	Simplifying HTTP request handling with Express.	Basic Express server with routing.
<b>REST API Development</b>	Building RESTful APIs using HTTP methods.	CRUD API with Express ( <code>GET</code> , <code>POST</code> , <code>PUT</code> , <code>DELETE</code> ).
<b>Middleware &amp; Authentication</b>	Middleware for processing requests and handling authentication.	Logging middleware and JWT authentication middleware.

---

## Conclusion:

- **Node.js** allows JavaScript to be used for backend development, enabling server-side functionality.
- **Express.js** simplifies the creation of web applications and APIs with its easy-to-use routing and middleware support.
- Building **RESTful APIs** and handling **middleware** like authentication are fundamental to backend development with Node.js and Express.