JavaScript Lecture PDF

1. Introduction to JavaScript

Definition:

JavaScript is a powerful, lightweight, and interpreted programming language that enables dynamic and interactive content on web pages. It works in web browsers and allows developers to create functionalities such as animations, form validations, and user interactions.

Features of JavaScript:

- Client-side scripting language
- Interpreted language
- Event-driven programming
- Supports Object-Oriented Programming (OOP)
- Used in both frontend (React, Angular, Vue) and backend (Node.js)

Example:

```
<!DOCTYPE html>

<html>

<head>

   <title>JavaScript Example</title>

</head>

<body>

   <script>

      alert("Hello, Welcome to JavaScript!");

   </script>

</body>

</html>
```

2. JavaScript Basics

2.1 Variables and Data Types in JavaScript

1. Variables in JavaScript

A **variable** is a container for storing data. JavaScript provides three ways to declare variables:

- var (old way, function-scoped)
- let (modern way, block-scoped)
- const (constant, block-scoped)

Variable Declaration Syntax-

```
var variableName = value;      // Using var (old way)
let variableName = value;     // Using let (modern way)
const variableName = value;   // Using const (cannot be changed)
```

Example

```
var name = "John";     // Declares a variable with var
let age = 25;              // Declares a variable with let
const country = "India"; // Declares a constant with const

console.log(name);    // Output: John
console.log(age);     // Output: 25
console.log(country); // Output: India
```

Differences between var, let, and const

Feature	var	let	const
Scope	Function-scoped	Block-scoped	Block-scoped
Reassignable	Yes	Yes	No
Redeclarable	Yes	No	No
Hoisting	Yes, but undefined	Yes, but not initialized	Yes, but not initialized

---

2. Data Types in JavaScript

JavaScript has **two types of data types**:

1. **Primitive Data Types** (Stores single value)
2. **Non-Primitive (Reference) Data Types** (Stores collections of values)

Primitive Data Types

1. **String** - Represents text data
2. **Number** - Represents numeric values (integer & floating point)
3. **Boolean** - Represents true/false values
4. **Undefined** - Variable is declared but not assigned a value
5. **Null** - Represents empty or unknown value
6. **BigInt** - Used for very large numbers
7. **Symbol** - Unique and immutable value

Syntax and Example of Primitive Data Types

```
let str = "Hello World";       // String
let num = 100;                 // Number
let isJavaScriptFun = true;  // Boolean
let notDefined;                // Undefined
let emptyValue = null;         // Null
let bigNumber = 1234567890123456789012345678901234567890n; // BigInt
let uniqueId = Symbol("id");     // Symbol

console.log(str);   // Output: Hello World
console.log(num);   // Output: 100
console.log(isJavaScriptFun); // Output: true
console.log(notDefined); // Output: undefined
console.log(emptyValue); // Output: null
console.log(bigNumber);  // Output: 1234567890123456789012345678901234567890n
console.log(uniqueId);   // Output: Symbol(id)
```

Non-Primitive (Reference) Data Types

1. **Object** - Stores key-value pairs
2. **Array** - Stores multiple values in a list
3. **Function** - A block of reusable code

Syntax and Example of Non-Primitive Data Types

```
// Object
let person = {
```

```
  name: "Alice",
  age: 30,
  city: "New York"
};
console.log(person.name); // Output: Alice

// Array
let fruits = ["Apple", "Banana", "Mango"];
console.log(fruits[1]); // Output: Banana

// Function
function greet() {
  return "Hello, Welcome!";
}
console.log(greet()); // Output: Hello, Welcome!
```

2.2 Operators in JavaScript

Operators in JavaScript are symbols that perform operations on variables and values. JavaScript has different types of operators, each used for specific purposes.

Types of Operators in JavaScript

1. Arithmetic Operators

Used to perform mathematical operations.

Operator	Description	Example
`+`	Addition	`5 + 3 = 8`
`-`	Subtraction	`10 - 4 = 6`
`*`	Multiplication	`6 * 2 = 12`
`/`	Division	`15 / 3 = 5`
`%`	Modulus (Remainder)	`10 % 3 = 1`
`**`	Exponentiation	`2 ** 3 = 8`

Example:

```
let a = 10, b = 5;
console.log(a + b);  // Output: 15
console.log(a - b);  // Output: 5
console.log(a * b);  // Output: 50
console.log(a / b);  // Output: 2
```

```
console.log(a % b);  // Output: 0
console.log(2 ** 3); // Output: 8
```

---

2. Assignment Operators

Used to assign values to variables.

Operator	Description	Example
`=`	Assign value	`x = 10`
`+=`	Add and assign	`x += 5` (same as `x = x + 5`)
`-=`	Subtract and assign	`x -= 3` (same as `x = x - 3`)
`*=`	Multiply and assign	`x *= 2` (same as `x = x * 2`)
`/=`	Divide and assign	`x /= 4` (same as `x = x / 4`)
`%=`	Modulus and assign	`x %= 3` (same as `x = x % 3`)

Example:

```
let x = 10;
x += 5; // x = x + 5 → 15
console.log(x);
```

---

3. Comparison Operators

Used to compare two values and return a Boolean result (`true` or `false`).

Operator	Description	Example	Result
`==`	Equal to	`5 == "5"`	`true`
`===`	Strict equal to (checks type too)	`5 === "5"`	`false`
`!=`	Not equal	`10 != 5`	`true`
`!==`	Strict not equal	`10 !== "10"`	`true`
`>`	Greater than	`10 > 5`	`true`
`<`	Less than	`3 < 7`	`true`
`>=`	Greater than or equal	`10 >= 10`	`true`
`<=`	Less than or equal	`5 <= 8`	`true`

Example:

```
console.log(5 == "5");  // Output: true
console.log(5 === "5"); // Output: false
console.log(10 > 5);    // Output: true
console.log(3 <= 7);    // Output: true
```

---

4. Logical Operators

Used to perform logical operations, often used in decision-making conditions.

Operator	Description	Example	Result
`&&`	AND (both true)	`true && false`	`false`
`|`		`|`	OR (either true)
`!`	NOT (reverse)	`!true`	`false`

Example:

```
let a = true, b = false;
console.log(a && b); // Output: false
console.log(a || b); // Output: true
console.log(!a);     // Output: false
```

---

5. Bitwise Operators

Used to perform operations at the bit level.

Operator	Description	Example
`&`	AND	`5 & 1` (0101 & 0001 → 0001)
`|`	`|`	OR
`^`	XOR	`5 ^ 1` (0101 ^ 0001 → 0100)
`~`	NOT	`~5` (Inverts bits)
`<<`	Left Shift	`5 << 1` (0101 → 1010)
`>>`	Right Shift	`5 >> 1` (0101 → 0010)

---

6. Ternary Operator (Conditional Operator)

Shorthand for `if-else` conditions.
Syntax:

```
condition ? expression_if_true : expression_if_false;
```

Example:

```
let age = 20;
let status = (age >= 18) ? "Adult" : "Minor";
console.log(status); // Output: Adult
```

---

7. Type Operators

Used to check data types.

Operator	Description	Example
`typeof`	Returns the type of a variable	`typeof "hello"` → `"string"`

Operator	Description	Example
`instanceof`	Checks if an object belongs to a class	`obj instanceof Object`

Example:

```
console.log(typeof 42);       // Output: number
console.log(typeof "Hello"); // Output: string

let arr = [1, 2, 3];
console.log(arr instanceof Array); // Output: true
```

2.3 Conditional Statements in JavaScript

Conditional statements in JavaScript allow the execution of different blocks of code based on conditions. They help in decision-making processes in a program.

Types of Conditional Statements

1. `if` statement
2. `if...else` statement
3. `if...else if...else` statement
4. `switch` statement
5. Ternary (`? :`) operator

---

1. if Statement

The `if` statement executes a block of code **only if** the given condition is `true`.

Syntax:

```
if (condition) {
    // Code to execute if the condition is true
}
```

Example:

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

---

2. if...else Statement

The `if...else` statement executes one block if the condition is `true`, and another block if the condition is `false`.

Syntax:

```
if (condition) {
    // Code to execute if condition is true
} else {
    // Code to execute if condition is false
}
```

Example:

```
let num = 10;
if (num % 2 == 0) {
    console.log("Even number");
} else {
    console.log("Odd number");
}
// Output: Even number
```

---

3. if...else if...else Statement

When there are multiple conditions, we use `if...else if...else` to check multiple cases.

Syntax:

```
if (condition1) {
    // Code for condition1
} else if (condition2) {
    // Code for condition2
} else {
    // Code if no conditions are true
}
```

Example:

```
let marks = 85;

if (marks >= 90) {
    console.log("Grade: A");
} else if (marks >= 75) {
    console.log("Grade: B");
} else if (marks >= 50) {
    console.log("Grade: C");
} else {
    console.log("Grade: F");
}
// Output: Grade: B
```

---

4. switch Statement

The `switch` statement is used when we have multiple possible values for a variable. Instead of using multiple `if...else` statements, `switch` provides a cleaner way.

Syntax:

```
switch(expression) {
    case value1:
        // Code to execute if expression === value1
        break;
    case value2:
        // Code to execute if expression === value2
        break;
    default:
        // Code to execute if no match is found
}
```

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;
    case 5:
        console.log("Friday");
        break;
    case 6:
        console.log("Saturday");
        break;
    case 7:
        console.log("Sunday");
        break;
    default:
        console.log("Invalid day");
}
// Output: Wednesday
```

- **Note:** `break` is used to exit the `switch` after a match is found. Without `break`, the next cases will also execute.

---

5. Ternary (`? :`) Operator

A shorthand way of writing `if...else` statements.

Syntax:

```
condition ? expression_if_true : expression_if_false;
```

Example:

```
let age = 20;
let status = (age >= 18) ? "Adult" : "Minor";
console.log(status); // Output: Adult
```

2.4 Loops in JavaScript

Loops in JavaScript allow executing a block of code multiple times. They are used when we need to perform repetitive tasks efficiently.

Types of Loops in JavaScript

1. `for` loop
2. `while` loop
3. `do...while` loop
4. `for...in` loop (used for objects)
5. `for...of` loop (used for arrays and iterables)

---

1. for Loop

The `for` loop is used when the number of iterations is known.

Syntax:

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

Example:

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

---

2. while Loop

The `while` loop is used when the number of iterations is **not known in advance**. It executes as long as the condition is `true`.

Syntax:

```
while (condition) {
    // Code to execute
}
```

Example:

```
let num = 1;
while (num <= 5) {
    console.log("Number: " + num);
    num++;
}
// Output: Number: 1, Number: 2, Number: 3, Number: 4, Number: 5
```

---

3. do...while Loop

The `do...while` loop is similar to `while`, but it **executes at least once** before checking the condition.

Syntax:

```
do {
    // Code to execute
} while (condition);
```

Example:

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

- **Difference between `while` and `do...while`:**
  - `while` **checks** the condition **first**, then executes the code.
  - `do...while` **executes** the code **first**, then checks the condition.

---

4. for...in Loop (Used for Objects)

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

Syntax:

```
for (let key in object) {
    // Code to execute for each property
}
```

Example:

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

for (let key in person) {
    console.log(key + ": " + person[key]);
}
// Output:
// name: Alice
// age: 25
// city: New York
```

---

5. for...of Loop (Used for Arrays and Iterables)

The `for...of` loop is used to iterate over **arrays, strings, maps, sets**, etc.

Syntax:

```
for (let element of iterable) {
    // Code to execute for each element
}
```

Example (Array):

```
let fruits = ["Apple", "Banana", "Cherry"];

for (let fruit of fruits) {
    console.log(fruit);
}
// Output:
// Apple
// Banana
// Cherry
```

Example (String):

```
let text = "JavaScript";

for (let char of text) {
    console.log(char);
}
// Output: J a v a S c r i p t (each letter printed separately)
```

---

6. break and continue Statements

- **break** Statement

The `break` statement **stops the loop** immediately when encountered.

Example:

```
for (let i = 1; i <= 5; i++) {
    if (i === 3) {
        break; // Stops loop when i = 3
    }
    console.log(i);
}
// Output: 1, 2
```

◆ 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 when i = 3
    }
    console.log(i);
}
// Output: 1, 2, 4, 5
```

---

3. JavaScript Functions

A **function** in JavaScript is a block of reusable code designed to perform a specific task. Functions help in writing **clean, modular, and reusable** code.

Types of Functions in JavaScript

1. **Function Declaration (Named Function)**
2. **Function Expression (Anonymous Function)**
3. **Arrow Function (ES6+)**
4. **Immediately Invoked Function Expression (IIFE)**
5. **Callback Function**
6. **Recursive Function**

---

1. Function Declaration (Named Function)

A function is defined using the `function` keyword and can be called multiple times.

Syntax:

```
function functionName(parameters) {
    // Function body
    return value; // (optional)
}
```

Example:

```
function greet(name) {
    return "Hello, " + name + "!";
}

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

---

2. Function Expression (Anonymous Function)

A function can be assigned to a variable without a name.

Syntax:

```
let variableName = function(parameters) {
    // Function body
    return value; // (optional)
};
```

Example:

```
let sum = function(a, b) {
    return a + b;
};

console.log(sum(5, 10)); // Output: 15
```

🔹 **Key Difference:** Function expressions **must be defined before calling them**, unlike function declarations.

---

3. Arrow Function (ES6+)

A shorter syntax for function expressions using `=>`.

Syntax:

```
const functionName = (parameters) => {
    // Function body
    return value;
};
```

Example:

```
const multiply = (a, b) => a * b;

console.log(multiply(4, 5)); // Output: 20
```

🔹 **Benefits:**
✅ Shorter syntax
✅ `this` behaves differently (more useful in objects and classes)

---

4. Immediately Invoked Function Expression (IIFE)

An **IIFE** is a function that runs **immediately after being defined**.

Syntax:

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

Example:

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

🔹 **Use Case:**

- Avoids polluting the global scope.

---

5. Callback Function

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

Example:

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

function sayBye() {
    console.log("Goodbye!");
}

greet("Alice", sayBye);
// Output:
// Hello, Alice
// Goodbye!
```

- **Use Case:**
  - Used in **asynchronous programming**, e.g., setTimeout(), API calls.

---

6. Recursive Function

A function that calls itself to solve a problem.

Example (Factorial Calculation):

```
function factorial(n) {
    if (n === 0) return 1; // Base case
    return n * factorial(n - 1);
}

console.log(factorial(5)); // Output: 120
```

- **Use Case:**
  - Useful for tasks like factorials, tree traversals, etc.

Function Parameters & Default Values

Functions can take parameters, and we can also assign default values.

Example:

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

greet(); // Output: Hello, Guest!
greet("Bob"); // Output: Hello, Bob!
```

Return Statement

A function **returns** a value using the `return` keyword.

Example:

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

let result = add(3, 7);
console.log(result); // Output: 10
```

- **Important Notes:**
  - If a function **does not return** a value, it returns `undefined` by default.
  - A function **stops execution** when `return` is encountered.

---

4. JavaScript Arrays and Objects

Arrays and objects are two fundamental data structures in JavaScript. Arrays store collections of values, while objects store data in key-value pairs.

---

1. JavaScript Arrays

An **array** is a special variable that can hold multiple values in a single reference. Each value in an array is identified by an index.

Creating an Array

1. Using Square Brackets (Recommended)

```
let fruits = ["Apple", "Banana", "Cherry"];
```

2. Using `new Array()` Constructor

```
let numbers = new Array(10, 20, 30);
```

- **Note:** The square bracket notation is preferred for simplicity.

Accessing Array Elements

Array elements are accessed using **indexing**, starting from `0`.

```
let fruits = ["Apple", "Banana", "Cherry"];

console.log(fruits[0]); // Output: Apple
console.log(fruits[2]); // Output: Cherry
```

Modifying Array Elements

We can change an existing element using its index.

```
let fruits = ["Apple", "Banana", "Cherry"];
fruits[1] = "Mango"; // Replacing "Banana" with "Mango"
```

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

Array Methods

Method	Description	Example
`push()`	Adds an element at the end	`arr.push(4);`
`pop()`	Removes the last element	`arr.pop();`
`shift()`	Removes the first element	`arr.shift();`
`unshift()`	Adds an element at the beginning	`arr.unshift(0);`
`splice()`	Adds/removes elements	`arr.splice(1, 2, "New");`
`slice()`	Extracts elements into a new array	`arr.slice(1, 3);`
`concat()`	Merges two arrays	`arr1.concat(arr2);`
`indexOf()`	Finds the index of an element	`arr.indexOf(3);`
`includes()`	Checks if an element exists	`arr.includes(5);`
`sort()`	Sorts an array	`arr.sort();`
`reverse()`	Reverses an array	`arr.reverse();`

Example:

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

numbers.push(40); // Adds 40 at the end
console.log(numbers); // [10, 20, 30, 40]

numbers.pop(); // Removes the last element
console.log(numbers); // [10, 20, 30]

numbers.shift(); // Removes the first element
console.log(numbers); // [20, 30]
```

Looping Through an Array

1. Using `for` Loop

```
let colors = ["Red", "Green", "Blue"];
for (let i = 0; i < colors.length; i++) {
    console.log(colors[i]);
}
```

2. Using `forEach()` Method

```
colors.forEach((color) => {
    console.log(color);
});
```

3. Using `for...of` Loop

```
for (let color of colors) {
    console.log(color);
}
```

---

2. JavaScript Objects

An **object** is a collection of key-value pairs. It is used to store related data.

Creating an Object

1. Using Object Literal (Recommended)

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

2. Using `new Object()` Constructor

```
let person = new Object();
person.name = "Alice";
person.age = 25;
person.city = "New York";
```

Accessing Object Properties

1. Dot Notation (Recommended)

```
console.log(person.name); // Output: Alice
```

2. Bracket Notation

```
console.log(person["age"]); // Output: 25
```

- **Bracket notation is useful when accessing properties dynamically.**

Modifying Object Properties

```
person.age = 30;
console.log(person.age); // Output: 30
```

Adding New Properties

```
person.country = "USA";
console.log(person);
```

Removing Properties

```
delete person.city;
console.log(person);
```

Object Methods

Objects can have functions as properties, called **methods**.

```
let car = {
    brand: "Toyota",
    model: "Corolla",
    start: function () {
        return "Car started!";
    }
};

console.log(car.start()); // Output: Car started!
```

Looping Through an Object

1. Using `for...in` Loop

```
for (let key in person) {
    console.log(key + ": " + person[key]);
}
```

---

3. Array of Objects

An **array of objects** stores multiple objects in a single array.

```
let students = [
    { name: "Alice", age: 22 },
    { name: "Bob", age: 24 },
    { name: "Charlie", age: 21 }
];

students.forEach((student) => {
    console.log(student.name + " is " + student.age + " years old.");
});
```

- **Output:**

```
Alice is 22 years old.
Bob is 24 years old.
Charlie is 21 years old.
```

---

4. Object Methods (`Object.keys()`, `Object.values()`, `Object.entries()`)

Method	Description	Example
`Object.keys(obj)`	Returns an array of keys	`Object.keys(person);`
`Object.values(obj)`	Returns an array of values	`Object.values(person);`
`Object.entries(obj)`	Returns an array of key-value pairs	`Object.entries(person);`

Example:

```
console.log(Object.keys(person)); // ["name", "age"]
console.log(Object.values(person)); // ["Alice", 30]
console.log(Object.entries(person)); // [["name", "Alice"], ["age", 30]]
```

---

5. JavaScript DOM Manipulation

The **Document Object Model (DOM)** is a programming interface that allows JavaScript to interact with and manipulate HTML and CSS. It represents a webpage as a hierarchical tree of elements.

---

1. What is DOM?

The **DOM (Document Object Model)** is a representation of the structure of an HTML document. Using JavaScript, we can:

✅ Change the content of an element
✅ Modify styles dynamically
✅ Add, remove, or replace elements
✅ Handle user interactions (events)

---

2. Selecting Elements in the DOM

Before modifying an element, we must **select** it. There are multiple ways to do this.

1. `getElementById()` – Select by ID

```
let title = document.getElementById("main-title");
console.log(title);
```

- **Returns:** The first element with the specified ID.

---

2. `getElementsByClassName()` – Select by Class

```
let items = document.getElementsByClassName("list-item");
```

```
console.log(items);
```

- **Returns:** A collection (`HTMLCollection`) of elements with the specified class.

---

3. `getElementsByTagName()` – Select by Tag

```
let paragraphs = document.getElementsByTagName("p");
console.log(paragraphs);
```

- **Returns:** A collection (`HTMLCollection`) of all matching tag elements.

---

4. `querySelector()` – Select the First Matching Element

```
let firstItem = document.querySelector(".list-item");
console.log(firstItem);
```

- **Returns:** The first matching element.

---

5. `querySelectorAll()` – Select All Matching Elements

```
let allItems = document.querySelectorAll(".list-item");
console.log(allItems);
```

- **Returns:** A **NodeList** of all matching elements.

---

3. Changing Content and Attributes

1. Changing Inner HTML (`innerHTML`)

```
document.getElementById("main-title").innerHTML = "Welcome to JavaScript!";
```

2. Changing Text Content (`textContent`)

```
document.getElementById("main-title").textContent = "Hello World!";
```

3. Changing Attributes (`setAttribute` and `getAttribute`)

```
let link = document.querySelector("a");
link.setAttribute("href", "https://www.google.com");
console.log(link.getAttribute("href")); // Output: https://www.google.com
```

4. Changing Styles (`style` Property)

```
document.getElementById("main-title").style.color = "blue";
document.getElementById("main-title").style.fontSize = "24px";
```

---

4. Adding and Removing Elements

1. Creating a New Element (`createElement()`)

```
let newParagraph = document.createElement("p");
newParagraph.textContent = "This is a new paragraph.";
```

2. Appending the Element (`appendChild()`)

```
document.body.appendChild(newParagraph);
```

3. Removing an Element (`remove()`)

```
let item = document.querySelector(".list-item");
item.remove();
```

4. Replacing an Element (`replaceChild()`)

```
let newHeading = document.createElement("h2");
newHeading.textContent = "New Heading";
let oldHeading = document.getElementById("main-title");

document.body.replaceChild(newHeading, oldHeading);
```

---

5. Event Handling in JavaScript

JavaScript can respond to user actions (e.g., clicks, typing, mouse movements) using **event listeners**.

1. Adding an Event Listener (`addEventListener()`)

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

2. Removing an Event Listener (`removeEventListener()`)

```
function sayHello() {
    alert("Hello!");
}

document.getElementById("btn").addEventListener("click", sayHello);
document.getElementById("btn").removeEventListener("click", sayHello);
```

---

6. Common DOM Events

Event Type	Description	Example
`click`	Fires when an element is clicked	`element.addEventListener("click", func);`
`mouseover`	Fires when mouse is over an element	`element.addEventListener("mouseover", func);`
`keydown`	Fires when a key is pressed	`element.addEventListener("keydown", func);`
`load`	Fires when a page loads	`window.addEventListener("load", func);`

---

7. Example: Interactive Button

```
<!DOCTYPE html>
<html>
<head>
    <title>DOM Manipulation</title>
</head>
<body>
    <h1 id="main-title">Hello, JavaScript!</h1>
    <button id="btn">Click Me</button>

    <script>
        document.getElementById("btn").addEventListener("click", function
() {
            document.getElementById("main-title").textContent = "Button
Clicked!";
            document.getElementById("main-title").style.color = "red";
        });
    </script>
</body>
</html>
```

🔹 **What Happens?**
✅ Clicking the button changes the heading text and color dynamically.

6.JavaScript Events

Events in JavaScript are **actions** or **occurrences** that happen in the browser, such as clicking a button, submitting a form, or pressing a key. JavaScript allows us to handle these events using **event listeners** and functions.

---

1. What is an Event?

An **event** is triggered when a user interacts with a web page. JavaScript can "listen" for these events and execute a function when they occur.

Example of an Event

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

📌 **Explanation:** When the button is clicked, an alert message appears.

---

2. Types of JavaScript Events

JavaScript provides different types of events, categorized as follows:

1. Mouse Events

Event	Description
`click`	Fires when an element is clicked
`dblclick`	Fires when an element is double-clicked
`mousedown`	Fires when a mouse button is pressed
`mouseup`	Fires when a mouse button is released
`mousemove`	Fires when the mouse moves over an element
`mouseover`	Fires when the mouse enters an element
`mouseout`	Fires when the mouse leaves an element

Example: Click Event

```
document.getElementById("box").addEventListener("click", function () {
    console.log("Box Clicked!");
});
```

---

2. Keyboard Events

Event	Description
`keydown`	Fires when a key is pressed
`keyup`	Fires when a key is released
`keypress`	Fires when a key is pressed (deprecated)

Example: Keydown Event

```
document.addEventListener("keydown", function (event) {
    console.log("Key Pressed: " + event.key);
});
```

---

3. Form Events

Event	Description
`submit`	Fires when a form is submitted
`change`	Fires when a form input value is changed
`focus`	Fires when an input field is focused
`blur`	Fires when an input field loses focus

Example: Submit Event

```
document.getElementById("myForm").addEventListener("submit", function
(event) {
    event.preventDefault(); // Prevents form submission
    alert("Form Submitted!");
});
```

---

4. Window Events

Event	Description
`load`	Fires when the page has fully loaded
`resize`	Fires when the window is resized
`scroll`	Fires when the user scrolls the page

Example: Window Load Event

```
window.addEventListener("load", function () {
    console.log("Page Loaded!");
});
```

---

3. Ways to Handle Events

1. Inline HTML Event Handling (Not Recommended)

```
<button onclick="alert('Button Clicked!')">Click Me</button>
```

---

2. Using the `onclick` Property

```
document.getElementById("myButton").onclick = function () {
    alert("Button Clicked!");
};
```

---

3. Using `addEventListener()` (Recommended)

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

✅ **Advantages of `addEventListener()`**

- Can attach multiple event handlers
- Supports event propagation

---

4. Event Object (`event`)

When an event occurs, JavaScript provides an `event` object containing information about the event.

Example: Accessing Event Object

```
document.getElementById("myButton").addEventListener("click", function
(event) {
    console.log("Event Type: " + event.type);
    console.log("Target Element: " + event.target);
});
```

---

5. Event Propagation (Bubbling & Capturing)

When an event occurs in a nested element, it propagates through the DOM in two phases:

1. **Bubbling Phase:** The event moves **from the target element up** to the root.
2. **Capturing Phase:** The event moves **from the root down** to the target.

Example: Event Bubbling

```
document.getElementById("parent").addEventListener("click", function () {
    console.log("Parent Div Clicked!");
});

document.getElementById("child").addEventListener("click", function (event)
{
    console.log("Child Div Clicked!");
    event.stopPropagation(); // Stops bubbling
});
```

📌 **Without `event.stopPropagation()`, clicking the child div would also trigger the parent's event.**

---

6. Prevent Default Action (`preventDefault()`)

Some events have default behaviors (e.g., form submission, link navigation). We can prevent these using `preventDefault()`.

Example: Preventing Form Submission

```
document.getElementById("myForm").addEventListener("submit", function
(event) {
    event.preventDefault();
    alert("Form submission prevented!");
});
```

---

7. Example: Interactive Event Handling

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>JavaScript Events</title>
</head>
<body>
    <h1 id="heading">Click the Button</h1>
    <button id="myButton">Click Me</button>

    <script>
        document.getElementById("myButton").addEventListener("click",
function () {
            document.getElementById("heading").textContent = "Button
Clicked!";
            document.getElementById("heading").style.color = "red";
        });
    </script>
</body>
</html>
```

🔹 **What Happens?**
✅ Clicking the button changes the heading text and color dynamically.

---

8. Summary Table

Concept	Description
Event Types	Mouse, Keyboard, Form, Window Events
Event Handling Methods	Inline, `onclick`, `addEventListener()`
Event Object (`event`)	Provides event details like `event.type`, `event.target`
Event Propagation	Bubbling (bottom-up) & Capturing (top-down)
`preventDefault()`	Prevents default behavior (e.g., form submission)

7.JavaScript ES6 Features

ECMAScript 6 (ES6) introduced several powerful features that improved JavaScript's readability, maintainability, and efficiency. Let's explore the key features with syntax and examples.

---

1. let and const (Block Scope Variables)

let (Block Scope)

`let` allows you to declare variables that are **limited to the block scope**.

```
let x = 10;
if (true) {
    let x = 20;
    console.log(x); // 20 (inside block)
}
console.log(x); // 10 (outside block)
```

const (Immutable Variables)

`const` is used for **constants** (values that do not change).

```
const PI = 3.1416;
console.log(PI);
```

📌 **Note:** You cannot reassign a `const` variable.

---

2. Arrow Functions (Shorter Function Syntax)

Arrow functions provide a **shorter** and **more readable** syntax for functions.

Regular Function

```
function add(a, b) {
    return a + b;
}
console.log(add(5, 3)); // 8
```

Arrow Function Equivalent

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

📌 **Benefits:**
✅ Less code
✅ Automatically binds `this`

---

3. Template Literals ( )

Template literals allow us to **embed variables and expressions** inside strings using backticks (`) and `${}`.

Example

```
let name = "John";
let age = 25;

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

📌 **Benefits:**
✅ No need for concatenation (+)
✅ Multiline strings

---

4. Default Parameters

Function parameters can have **default values**.

Example

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

---

5. Spread (...) and Rest (...) Operators

Spread Operator (...)

Expands elements of an array or object.

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

Rest Operator (...)

Collects multiple arguments into an array.

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

---

6. Destructuring Assignment

Destructuring allows extracting values from **arrays** and **objects** easily.

Array Destructuring

```
let [a, b, c] = [10, 20, 30];
console.log(a, b, c); // 10 20 30
```

Object Destructuring

```
let person = { name: "Alice", age: 22 };
let { name, age } = person;
console.log(name, age); // Alice 22
```

---

7. Enhanced Object Literals

ES6 allows **shorthand** property and method definitions in objects.

Example

```
let name = "Emma";
let person = {
    name,  // Same as name: name
    greet() { console.log("Hello!"); } // No need for function keyword
};
person.greet(); // Hello!
```

---

8. ES6 Classes

ES6 introduces **class-based object-oriented programming**.

Example

```
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
    greet() {
        console.log(`Hello, my name is ${this.name}.`);
    }
```

```
}

let person1 = new Person("John", 30);
person1.greet(); // Hello, my name is John.
```

---

9. Promises (Asynchronous Handling)

Promises handle **asynchronous operations** like API calls.

Example

```
let myPromise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("Data loaded!"), 2000);
});

myPromise.then(result => console.log(result));
```

📌 **Benefits:**
✅ Avoids callback hell
✅ Handles errors properly

---

10. Modules (`import` and `export`)

ES6 introduces modules to **split code into separate files**.

Export (module1.js)

```
export const greet = () => console.log("Hello!");
export const name = "Alice";
```

Import (main.js)

```
import { greet, name } from "./module1.js";
greet();  // Hello!
console.log(name);  // Alice
```

Summary of ES6 Features

Feature	Description
`let` & `const`	Block-scoped variables
Arrow Functions	Shorter function syntax (=>)
Template Literals	`${}` for embedding variables in strings
Default Parameters	Assign default values to function arguments
Spread & Rest Operators	Expands/rests arrays/objects (...)

Feature	Description
Destructuring	Extract values from arrays/objects
Enhanced Object Literals	Shorthand property/methods in objects
Classes	Object-oriented programming
Promises	Handles asynchronous tasks
Modules	Import/export functionality

---

8. JavaScript Advanced Concepts

JavaScript has many advanced concepts that help developers build **efficient, scalable, and maintainable** applications. Here, we'll cover some **key advanced topics** with examples.

---

1. Closures

A **closure** is a function that **remembers** the variables from its **outer scope**, even after the outer function has finished execution.

Example

```
function outerFunction(outerVariable) {
    return function innerFunction(innerVariable) {
        console.log(`Outer: ${outerVariable}, Inner: ${innerVariable}`);
    };
}

const newFunction = outerFunction("Hello");
newFunction("World"); // Output: Outer: Hello, Inner: World
```

📌 **Closures are used for:**
✅ Data encapsulation
✅ Memoization (caching results)
✅ Maintaining state in async functions

---

2. Asynchronous JavaScript (Promises & Async/Await)

Promises

A **promise** represents an operation that **hasn't completed yet** but will in the future.

```
let myPromise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("Promise resolved!"), 2000);
});

myPromise.then(result => console.log(result)); // Output after 2 sec:
"Promise resolved!"
```

Async/Await (Better Way to Handle Async Code)

```
function fetchData() {
    return new Promise(resolve => setTimeout(() => resolve("Data fetched"),
2000));
}

async function getData() {
    console.log("Fetching...");
    let data = await fetchData();
    console.log(data);
}
getData();
```

📌 **Benefits of Async/Await:**
✅ More readable than `.then()`
✅ Easier error handling with `try-catch`

---

3. Prototype & Prototypal Inheritance

JavaScript uses **prototypal inheritance**, where objects inherit properties/methods from a prototype.

Example

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

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

let person1 = new Person("John", 30);
person1.greet(); // Hello, my name is John
```

📌 **Prototypes help in:**
✅ Reducing memory usage
✅ Sharing methods between objects

---

4. The `this` Keyword (Context Binding)

The `this` keyword **refers to the object** executing the function.

Global Context

```
console.log(this); // In browser: window object
```

Object Method

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

Arrow Functions & `this`

Arrow functions **do not bind `this`**.

```
let obj = {
    name: "Bob",
    greet: () => console.log(`Hello, ${this.name}`) // 'this' is
window/global
};
obj.greet(); // Undefined
```

📌 **Fix with `function` keyword or `.bind(this)`.**

---

5. Call, Apply, and Bind

These methods allow controlling `this` in function execution.

Call

```
function greet(city) {
    console.log(`Hello, I am ${this.name} from ${city}`);
}
let person1 = { name: "Alice" };
greet.call(person1, "New York"); // Hello, I am Alice from New York
```

Apply (Same as Call, but with an Array)

```
greet.apply(person1, ["Los Angeles"]); // Hello, I am Alice from Los
Angeles
```

Bind (Returns a New Function)

```
let newGreet = greet.bind(person1, "London");
newGreet(); // Hello, I am Alice from London
```

📌 **Use Cases:**
✅ Borrowing functions from other objects
✅ Function execution with specific `this`

---

6. Higher-Order Functions & Functional Programming

A **higher-order function** is a function that takes another function as an argument or returns a function.

Example

```
function operateOnNumbers(arr, operation) {
    return arr.map(operation);
}

let result = operateOnNumbers([1, 2, 3], num => num * 2);
console.log(result); // [2, 4, 6]
```

📌 **Functional Programming Principles:**
✅ Pure functions (no side effects)
✅ Immutability (avoid modifying data)

---

7. Event Loop & Concurrency Model

JavaScript is **single-threaded**, but it uses an **event loop** to handle asynchronous operations.

Execution Order Example

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

setTimeout(() => console.log("Inside setTimeout"), 0);

Promise.resolve().then(() => console.log("Inside Promise"));

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

📌 **Expected Output:**

```
Start
End
Inside Promise
Inside setTimeout
```

✅ **Microtasks (Promises) run before Macrotasks (setTimeout).**

---

8. Debouncing & Throttling (Performance Optimization)

These techniques **limit the number of function calls** for performance improvement.

Debouncing (Delays Execution Until Last Call)

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

window.addEventListener("resize", debounce(() => console.log("Resized!"),
300));
```

Throttling (Limits Execution Frequency)

```
function throttle(func, limit) {
    let lastCall = 0;
    return function (...args) {
        let now = Date.now();
        if (now - lastCall >= limit) {
            lastCall = now;
            func.apply(this, args);
        }
    };
}

window.addEventListener("scroll", throttle(() => console.log("Scrolling!"),
1000));
```

📌 **Use Cases:**
✅ Debouncing → Search bar suggestions
✅ Throttling → Window resize & scroll events

---

9. Currying (Function Transformation)

Currying transforms a function that takes multiple arguments into a **series of functions**.

Example

```
function multiply(a) {
    return function (b) {
        return function (c) {
            return a * b * c;
        };
    };
}
console.log(multiply(2)(3)(4)); // 24
```

📌 **Benefits:**
✅ Code reusability
✅ Partial application of functions

---

10. Memoization (Caching Function Results)

Memoization **stores computed results** to improve performance.

Example

```
function memoize(fn) {
    let cache = {};
    return function (num) {
        if (num in cache) {
            console.log("Fetching from cache...");
            return cache[num];
        } else {
            console.log("Calculating result...");
            let result = fn(num);
            cache[num] = result;
            return result;
        }
    };
}

const factorial = memoize(num => num <= 1 ? 1 : num * factorial(num - 1));
console.log(factorial(5)); // Calculating result...
console.log(factorial(5)); // Fetching from cache...
```

📌 **Benefits:**
✅ Faster execution for repeated calls
✅ Used in complex calculations

Summary Table

Concept	Description
Closures	Functions remember outer variables
Promises & Async/Await	Handle asynchronous code
Prototypes	Object inheritance
`this` Keyword	Refers to the current object
Call, Apply, Bind	Control function execution context
Higher-Order Functions	Functions that take/return other functions
Event Loop	JavaScript concurrency model
Debouncing & Throttling	Optimize function calls
Currying	Transform functions for reusability
Memoization	Cache results for performance
