**📘 Module 1: Introduction to JavaScript**

* What is JavaScript?
* Setting up the development environment (browser + editor)
* JavaScript in HTML: <script> tag
* Console & debugging basics

**📗 Module 2: Basic JavaScript Syntax**

* Variables: var, let, const
* Data types: String, Number, Boolean, Null, Undefined
* Operators: Arithmetic, Assignment, Comparison, Logical
* Comments

**📙 Module 3: Control Flow**

* Conditional statements: if, else, switch
* Loops: for, while, do...while
* break and continue

**📕 Module 4: Functions**

* Declaring and calling functions
* Function expressions
* Arrow functions
* Parameters and return values
* Scope (local vs. global)

**📘 Module 5: Arrays and Objects**

* Creating arrays and objects
* Array methods: push, pop, shift, unshift, map, filter, reduce
* Object properties and methods
* Looping through arrays/objects (for...of, for...in)

**📗 Module 6: DOM Manipulation**

* The Document Object Model (DOM) explained
* Selecting elements (getElementById, querySelector)
* Changing content and styles
* Event handling: addEventListener, click, input, etc.

**📙 Module 7: ES6+ Features**

* Template literals
* Destructuring
* Spread and rest operators
* Default parameters
* Modules (import, export)
* let, const, and block scoping

**📕 Module 8: Asynchronous JavaScript**

* Callbacks
* Promises
* async / await
* Fetch API and working with JSON

**📘 Module 9: Error Handling & Debugging**

* Try...catch
* Console tools
* Common JS errors and how to resolve them

**📗 Module 10: Mini Projects & Practice**

* To-do list
* Calculator
* Weather app using Fetch API
* Simple game (e.g., Rock, Paper, Scissors)

**JAVASCRIPT**

JavaScript is a high-level, dynamic, and interpreted programming language that is widely used for web development. It was originally created to enable interactive web pages and is an essential part of web applications. Here are some key features and aspects of JavaScript:

1. **Client-Side Scripting**: JavaScript is primarily used as a client-side scripting language, meaning it runs in the user's web browser. This allows developers to create interactive and dynamic web pages.
2. **Event-Driven**: JavaScript can respond to user actions, such as clicks, mouse movements, and keyboard input, making it ideal for creating interactive user interfaces.
3. **Object-Oriented**: JavaScript supports object-oriented programming concepts, allowing developers to create and manipulate objects, which can represent real-world entities.
4. **Cross-Platform**: JavaScript can run on various platforms and devices, including desktops, tablets, and smartphones, making it a versatile choice for web development.
5. **Asynchronous Programming**: JavaScript supports asynchronous programming through callbacks, promises, and async/await syntax, allowing for non-blocking operations, such as fetching data from a server.
6. **Frameworks and Libraries**: There are numerous frameworks and libraries built on top of JavaScript, such as React, Angular, and Vue.js, which simplify the development of complex web applications.
7. **Server-Side Development**: With the advent of Node.js, JavaScript can also be used for server-side development, allowing developers to use the same language for both client-side and server-side code.
8. **Standardization**: JavaScript is standardized by the ECMAScript specification, which ensures consistency across different implementations of the language.

Why high-level

1. **Abstraction from Hardware**: High-level languages provide a level of abstraction from the underlying hardware and machine code. This means that developers do not need to manage memory manually or deal with low-level operations, such as CPU instructions. Instead, they can focus on writing code that is more understandable and closer to human language.
2. **Ease of Use**: High-level languages are designed to be easier to read and write. JavaScript syntax is relatively simple and resembles natural language, which makes it more accessible for beginners and allows developers to express complex ideas with fewer lines of code.
3. **Built-in Functions and Libraries**: JavaScript comes with a rich set of built-in functions and libraries that simplify common programming tasks, such as manipulating the Document Object Model (DOM), handling events, and making network requests. This reduces the amount of code developers need to write and helps them accomplish tasks more efficiently.
4. **Automatic Memory Management**: JavaScript includes automatic garbage collection, which means that the language automatically manages memory allocation and deallocation. Developers do not need to manually free up memory, which reduces the risk of memory leaks and other related issues.
5. **Platform Independence**: JavaScript code can run on any platform that has a compatible web browser or JavaScript engine, making it highly portable. This allows developers to write code once and have it run on various devices and operating systems without modification.
6. **Dynamic Typing**: JavaScript is dynamically typed, meaning that variable types are determined at runtime rather than at compile time. This flexibility allows for more rapid development and easier prototyping.

**What is dynamic typing**?

In programming, the terms "dynamic" and "static" can refer to various concepts, particularly in the context of typing, memory allocation, and behavior during execution. Here’s a breakdown of what these terms mean in relation to programming languages, including JavaScript:

**Dynamic**

1. **Dynamic Typing**:
   * In dynamically typed languages, variable types are determined at runtime rather than at compile time. This means you can assign a value of any type to a variable, and the type can change during the execution of the program.
   * **Example in JavaScript**:

javascript2 lines

Click to close

let x = 5; // x is a number

x = "Hello"; // now x is a string

1. **Dynamic Memory Allocation**:
   * Memory is allocated at runtime as needed. This allows for more flexible data structures, such as linked lists or dynamic arrays, which can grow or shrink as required.
   * In JavaScript, objects and arrays can be modified dynamically, allowing you to add or remove properties and elements at runtime.
2. **Dynamic Behavior**:
   * Dynamic behavior refers to the ability of a program to change its structure or behavior at runtime. For example, JavaScript allows you to add methods to objects or modify functions while the program is running.

**Static**

1. **Static Typing**:
   * In statically typed languages, variable types are determined at compile time. This means that you must declare the type of a variable when you create it, and it cannot change during the program's execution.
   * **Example in TypeScript (a statically typed superset of JavaScript)**:

typescript

RunCopy code

1let x: number = 5; // x is explicitly declared as a number

2x = "Hello"; // This will cause a compile-time error

1. **Static Memory Allocation**:
   * Memory is allocated at compile time, and the size of data structures must be known beforehand. This can lead to more efficient memory usage but less flexibility.
   * In languages like C or C++, arrays have a fixed size that must be defined when they are declared.
2. **Static Behavior**:
   * Static behavior refers to the fixed structure of a program that does not change at runtime. For example, the methods and properties of a class in a statically typed language are defined at compile time and cannot be altered during execution.

**Summary**

* **Dynamic**: Refers to flexibility and runtime determination, allowing for changes in types, memory allocation, and behavior during execution. JavaScript is a dynamically typed language.
* **Static**: Refers to fixed structures and compile-time determination, requiring explicit type declarations and fixed memory allocation. Statically typed languages enforce type constraints at compile time.

**client-side and server-side**

The terms **client-side** and **server-side** refer to where certain processes or operations occur in a web application. Understanding the differences between these two concepts is crucial for web development. Here’s a breakdown of the key differences:

**1. Location of Execution**

* **Client-Side**:
  + Code is executed in the user's web browser.
  + The client (browser) processes the code after receiving it from the server.
* **Server-Side**:
  + Code is executed on the web server.
  + The server processes requests and sends the results back to the client.

**2. Languages Used**

* **Client-Side**:
  + Commonly uses languages like **JavaScript**, HTML, and CSS.
  + Frameworks and libraries such as React, Angular, and Vue.js are also used for client-side development.
* **Server-Side**:
  + Can use a variety of programming languages, including **Node.js (JavaScript)**, **Python**, **Ruby**, **PHP**, **Java**, **C#**, and others.
  + Each language may have its own frameworks (e.g., Express for Node.js, Django for Python, Ruby on Rails for Ruby).

**3. User Interaction**

* **Client-Side**:
  + Directly interacts with the user, providing immediate feedback and dynamic content updates.
  + Examples include form validation, animations, and interactive elements.
* **Server-Side**:
  + Handles requests from the client, processes data, and sends responses back to the client.
  + Examples include database queries, user authentication, and data processing.

**4. Performance and Load**

* **Client-Side**:
  + Reduces server load by offloading tasks to the client’s browser.
  + Performance can vary based on the client’s device and browser capabilities.
* **Server-Side**:
  + Can handle complex computations and data processing, but may increase server load.
  + Performance is generally consistent, as it depends on server resources rather than client devices.

**5. Security**

* **Client-Side**:
  + More vulnerable to security risks, such as code injection and cross-site scripting (XSS), since the code is exposed to the user.
  + Sensitive operations (like database access) should not be performed on the client side.
* **Server-Side**:
  + Generally more secure, as the code and data are not exposed to the client.
  + Sensitive operations, such as database queries and user authentication, are typically handled on the server side.

**6. Data Handling**

* **Client-Side**:
  + Can manipulate and display data received from the server but does not typically store data permanently.
  + Uses local storage or session storage for temporary data storage.
* **Server-Side**:
  + Responsible for data storage, retrieval, and manipulation, often interacting with databases.
  + Can perform complex data operations and maintain state across user sessions.

**7. Examples of Use Cases**

* **Client-Side**:
  + Form validation before submission.
  + Dynamic content updates (e.g., loading new content without refreshing the page).
  + User interface enhancements (e.g., dropdown menus, sliders).
* **Server-Side**:
  + User authentication and authorization.
  + Database interactions (e.g., retrieving user data, storing new records).
  + Processing business logic and generating dynamic web pag

**JAVASCRIPT VARIABLES**

Great — you're diving into **JavaScript variables**. Here's a clear breakdown of var, let, and const, which are used to **declare variables** (i.e., names that store values):

**✅ var (Old Way — Not Recommended Now)**

* **Function-scoped**: Only works within the function where it's declared.
* **Can be redeclared and updated**.
* **Hoisted**: Gets moved to the top of its scope, but without its value.

javascript

CopyEdit

var x = 10;

x = 20; // ✅ OK

var x = 30; // ✅ OK (but confusing)

**✅ let (Modern and Recommended)**

* **Block-scoped**: Works only within the { } block it’s in.
* **Can be updated**, but **not redeclared in the same scope**.
* **Also hoisted**, but **not initialized** (will give error if accessed before declared).

javascript

CopyEdit

let y = 10;

y = 20; // ✅ OK

let y = 30; // ❌ Error: already declared

**✅ const (For Constants)**

* **Block-scoped**, like let.
* **Cannot be updated or redeclared**.
* Must be **initialized when declared**.

javascript

CopyEdit

const z = 10;

z = 20; // ❌ Error: can't change a const

const z = 30; // ❌ Error: already declared

📌 Note: If const holds an object or array, **you can still change its contents**, just not reassign it.

javascript

CopyEdit

const person = { name: "Alice" };

person.name = "Bob"; // ✅ OK

person = {}; // ❌ Error

**🔑 Summary Table:**

| **Feature** | **var** | | **let** | | **const** |
| --- | --- | --- | --- | --- | --- |
| Scope | Function | | Block | | Block |
| Can reassign | Yes | | Yes | | No |
| Can redeclare | Yes | | No | | No |
| Hoisting | Yes (undefined) | | Yes (but not usable before) | | Yes (but not usable before) |
|  | |  | |

**DATATYPES**

In **JavaScript**, a **data type** defines the **kind of value** a variable can hold. It tells the interpreter what type of **operations** can be performed on the value and how it should be **stored** and **handled** in memory.

A **data type** in JavaScript is a classification that describes the type of data a variable holds, such as a number, string, boolean, object, etc.

**Types**

1. **1.** **Primitive Data Types**: These are the basic types of data that are not objects and have immutable values. They include:
   * **String**: Represents text.
2. / string enclosed within single quotes  
   let *course*="javascript"  
   // string enclosed within double quotes  
   let *fee*='20000'  
   // string enclosed within backticks  
   let *name*=`Nithya`  
   console.log(*course*)  
   console.log(*fee*)  
   console.log(name)  
     
     
   let *text* = "JavaScript is Fun";  
     
   console.log(*text*.toUpperCase()); // "JAVASCRIPT IS FUN"  
   console.log(*text*.toLowerCase()); // "javascript is fun"  
   console.log(*text*.includes("Fun")); // true  
   console.log(*text*.indexOf("Script")); // 4  
   console.log(*text*.replace("Fun", "Awesome")); // "JavaScript is Awesome"  
   console.log(*text*.slice(0, 10));// "JavaScript"  
     
     
     
   let *name* = "Sara";  
   let *welcome* = `Hi, ${name}! Welcome back.`;  
   console.log(*welcome*); // "Hi, Sara! Welcome back."  
     
     
   let *message* = "Hello, world!";  
   console.log(*message*.length); // 13  
     
     
     
   let *word* = "JavaScript";  
   console.log(*word*[0]); // "J"  
   console.log(*word*.charAt(4)); // "S"
   * **Number**: Represents numeric values (both integers and floats).
3. //🔢 Number Data Type in JavaScript  
   //✅ Definition:  
   //The Number data type in JavaScript is used to represent both integers and floating-point (decimal) values. Unlike some languages, JavaScript has only one number type, and it's a 64-bit floating-point value (IEEE 754).  
   //  
   //📌 Examples of Numbers  
     
   let *age* = 30; // Integer  
   let *price* = 19.99; // Decimal (floating-point)  
   let *negative* = -100; // Negative number  
   let *zero* = 0;  
     
     
     
   console.log(1 / 0); // Infinity  
   console.log("abc" \* 3); // NaN  
   //🧮 Basic Arithmetic Operations  
     
   let *x* = 10;  
   let *y* = 3;  
     
   console.log(*x* + *y*); // 13  
   console.log(*x* - *y*); // 7  
   console.log(*x* \* *y*); // 30  
   console.log(*x* / *y*); // 3.333...  
   console.log(*x* % *y*); // 1 (modulus, remainder)  
     
     
   let *num* = 3.14159;  
     
   console.log(*num*.toFixed(2)); // "3.14" – rounds to 2 decimal places  
   console.log(*num*.toString()); // "3.14159" – converts to string  
   console.log(Number("123")); // 123 – converts string to number  
   console.log(parseInt("100px")); // 100 – extracts number from string  
   console.log(parseFloat("12.5kg")); // 12.5  
   //🧠 Type Checking  
     
   let *value* = 42;  
   console.log(typeof *value*); // "number"
   * **Boolean**: Represents a logical value (**true** or **false**).
4. let *isSunny* = true;  
   let *hasPermission* = false;  
     
     
   🔁 Boolean in Conditional Logic  
   Booleans are often used in if statements:  
     
   j  
   let isRaining = false;  
     
   if (isRaining) {  
    console.log("Take an umbrella!");  
   } else {  
    console.log("Enjoy the sunshine!");  
   }  
   📊 Comparison Returns Boolean  
     
   console.log(5 > 3); // true  
   console.log(10 === 8); // false  
   🔍 Convert Other Types to Boolean  
   You can convert any value using Boolean():  
     
   js  
   Copy  
   Edit  
   Boolean(0); // false  
   Boolean(1); // true  
   Boolean(""); // false  
   Boolean("Hi"); // true  
   Boolean(null); // false
   * **Undefined**: Represents a variable that has been declared but not assigned a value.
5. //🟡 Undefined in JavaScript – Explained  
   //In JavaScript, undefined is a primitive data type and also a special value that means:  
   //  
   //"A variable has been declared but not assigned any value."  
   //  
   //✅ Definition:  
   //undefined indicates absence of a value — but not because you set it to nothing. It means JavaScript hasn't assigned a value yet.  
   //  
   //🔸 Examples:  
     
   let *x*;  
   console.log(*x*); // undefined (no value assigned)
   * **Null**: Represents the intentional absence of any object value.
6. In JavaScript, null is a special primitive value that means:  
     
   "Intentional absence of any value."  
     
   ✅ Definition:  
   null is explicitly assigned by a developer to indicate that a variable is empty on purpose.  
     
   🔸 Example:  
     
   let *user* = null;  
   console.log(user); // null
   * **Symbol**: Represents a unique and immutable value, often used as object property keys.
7. 🟣 Symbol in JavaScript – Explained  
   A Symbol is a primitive data type introduced in ES6 (ECMAScript 2015). It's used to create unique and immutable identifiers.  
     
   ✅ Definition:  
   A Symbol is a unique value often used as a key for object properties to avoid name clashes.  
     
   🔸 Creating a Symbol  
   js  
   Copy  
   Edit  
   let id = Symbol();  
   console.log(typeof id); // "symbol"  
   You can also add a description (for debugging only):  
     
   let userId = Symbol("user");  
   Even if two symbols have the same description, they are not equal:  
     
     
   let a = Symbol("id");  
   let b = Symbol("id");  
     
   console.log(a === b); // false  
   📦 Using Symbols as Object Keys  
     
   let user = {};  
   let id = Symbol("id");  
     
   user[id] = 101;  
     
   console.log(user[id]); // 101
   * **BigInt**: Represents integers with arbitrary precision.

🟠 BigInt in JavaScript – Explained  
BigInt is a primitive data type that allows you to work with integers larger than the safe limit of the regular Number type.  
  
✅ Definition:  
BigInt is used to represent whole numbers larger than 2^53 - 1 (which is 9007199254740991) – the maximum safe integer for regular numbers in JavaScript.  
  
🔢 Creating a BigInt  
Add n at the end of the number:  
  
  
let big = 1234567890123456789012345678901234567890n;  
Or use the BigInt() constructor:  
  
  
let big = BigInt("1234567890123456789012345678901234567890");

1. **Non-Primitive (Reference) Data Types:** These are more complex types that can hold collections of values or more complex entities. They include:
   * **Object**: A collection of key-value pairs.
2. let *student*={  
    'name':"Suriya",  
    'age':34,  
    'fees':20000,  
    'hobby': "Dancing",  
   }  
   *student*.hobby = "Singing";  
   *student*.rollNo = 14;  
   *student*.faculty = "Science";  
   delete *student*.hobby;  
     
   console.log(*student*)  
   console.log(typeof(*student*))  
   console.log(*student*.name,*student*.age);
   * **Array**: A special type of object used to store ordered collections of values.
3. const *lists*=[1,3,4,,5,'java','python','html']  
     
   console.log(*lists*.length)  
   console.log(*lists*)  
   console.log(typeof(*lists*))  
   for(i=0;i<*lists*.length;i++)  
   {  
    console.log(*lists*[i]);  
   }

**MODULE 3: CONTROL FLOW**

In JavaScript, **control statements**—also known as **control flow statements**—are constructs that manage the execution order of code. They enable programs to make decisions, repeat tasks, and jump between different parts of the code based on specific conditions. Without these statements, code would execute sequentially from top to bottom, limiting its functionality.

**🧭 What Are Control Statements?**

Control statements determine the flow of a program by allowing it to:

* **Make decisions**: Execute certain blocks of code based on conditions.
* **Repeat tasks**: Loop through code blocks multiple times.
* **Alter execution flow**: Jump to different parts of the code as needed.

These statements are fundamental for creating dynamic and responsive applications.

**🔑 Types of Control Statements in JavaScript**

1. **Conditional Statements**: Execute code blocks based on specific conditions.
   * if, else if, else
   * switch

**if Statement**

The if statement evaluates a condition; if the condition is true, the associated code block executes.

let *age*=18;  
if(*age*>=18){  
 console.log("Eligible for votting")  
}

**else Statement**

The if...else statement provides an alternative path of execution when the if condition is false

let *age*=1;  
if(*age*>=18){  
 console.log("Eligible for votting")  
}  
else  
{  
 console.log("not eligible for votting")  
}

**🔄 3. if...else if...else Statement**

This structure allows multiple conditions to be checked sequentially.

var *mark1*=450;  
if(*mark1*>=450 && *mark1*<=500){  
 console.log("distinction");  
}  
else if(*mark1*>=400 && *mark1*<=449){  
 console.log("first class");  
}  
else if(*mark1*>=300 && *mark1*<=399){  
 console.log("second class")  
}  
else{  
 console.log("fail");  
}

**switch Statement**

The switch statement evaluates an expression and executes code blocks based on matching case values.

let *day* = "Monday";  
switch (*day*) {  
 case "Monday":  
 console.log("Start of the work week.");  
 break;  
 case "Friday":  
 console.log("End of the work week.");  
 break;  
 default:  
 console.log("Midweek days.");  
}

The **switch** statement in JavaScript is a control flow statement that allows you to execute different blocks of code based on the value of a specific expression. It is often used as an alternative to multiple **if...else if** statements when you need to compare the same expression against different values.

let *rechargePlan* = "Data Pack";  
  
switch (*rechargePlan*) {  
 case "Talktime":  
 console.log("You have selected the Talktime plan: ₹10 for ₹7.47 talktime.");  
 break;  
 case "Data Pack":  
 console.log("You have selected the Data Pack: ₹299 for 1.5GB/day for 28 days.");  
 break;  
 case "Unlimited":  
 console.log("You have selected the Unlimited plan: ₹719 for unlimited calls and 1.5GB/day for 84 days.");  
 break;  
 case "International":  
 console.log("You have selected the International plan: ₹1,098 for international roaming.");  
 break;  
 default:  
 console.log("Invalid selection. Please choose a valid recharge plan.");  
}

**🔸 Comparison Table**

| **Feature** | **if...else if...else** | **switch** |
| --- | --- | --- |
| Condition Types | Complex expressions, ranges, multiple vars | Single variable equality checks |
| Syntax Complexity | Can become lengthy with many conditions | More concise for multiple specific cases |
| Evaluation | Conditions evaluated in order | Expression evaluated once |
| Use Cases | Complex logic, ranges | Multiple specific values |
| Readability | Less readable with many conditions | More readable for multiple discrete cases |
| Performance | May be slower with many conditions | Generally faster with many cases |

1. **Looping Statements**: Repeat code blocks while certain conditions are met.
   * For

## 🔁 JavaScript Looping Statements

### 1. for ****Loop****

**Purpose:** Used when the number of iterations is known.

**Syntax:**

javascript

CopyEdit

for (initialization; condition; increment) {

// code block to be executed

}

let =10  
for(i=0;i<=10;i++)  
{  
 if(i%2!==0)  
 console.log(i)  
}

**while Loop**

**Purpose:** Used when the number of iterations is not known and depends on a condition.

**Syntax:**

javascript

CopyEdit

while (condition) {

// code block to be executed

}

let *i*=0  
while (*i*<10){  
 console.log(*i*)  
 *i*+=2;  
  
}

In JavaScript, a do...while loop is a control structure that executes a block of code once before checking the condition, and then repeatedly executes the loop as long as the specified condition evaluates to true. This guarantees that the code block will run at least once, regardless of the condition.[GeeksforGeeks+4Programiz+4W3Schools.com+4](https://www.programiz.com/javascript/while-loop?utm_source=chatgpt.com)

**🔄 Syntax of a do...while Loop**

javascript

CopyEdit

do {

// Code to be executed

} while (condition);

* **Code Block**: The code inside the do block is executed once before the condition is tested.
* **Condition**: After each execution, the condition is evaluated. If it evaluates to true, the loop continues; if false, the loop terminates.

**🧪 Example: Counting with a do...while Loop**

javascript

CopyEdit

let i = 0;

do {

console.log("Count:", i);

i++;

} while (i < 5);

**Output:**

makefile

CopyEdit

Count: 0

Count: 1

Count: 2

Count: 3

Count: 4

In this example, the loop starts with i = 0 and continues to execute as long as i is less than 5. After each iteration, i is incremented by 1.

**🧾 Comparison Table**

| **Feature** | **for Loop** | **while Loop** | **do...while Loop** |
| --- | --- | --- | --- |
| **Condition Check** | Before loop starts | Before loop starts | After loop starts |
| **Guaranteed Run** | No | No | Yes (at least once) |
| **Use Case** | Known number of iterations | Unknown number of iterations | Execute at least once before condition |
| **Syntax** | Compact initialization, condition, and increment in one line | Requires separate initialization and increment | Initialization and increment inside loop body |

1. **Jump Statements**: Alter the normal flow of execution.
   * break
   * continue
   * return
2. **Exception Handling Statements**: Handle errors and exceptions gracefully.
   * try...catch
   * throw
   * finally

**WHAT IS A FUNCTION?**

A function is a block of code designed to perform a particular task. It can take inputs (known as parameters), execute a series of statements, and optionally return a value.

**Defining a Function**

**Function function\_name()**

**{**

**Function definitions**

**}**

**Function calling**

**Parameters and Arguments**

* **Parameters** are the names listed in the function's definition.
* **Arguments** are the actual values you pass to the function when you call it.
* **11. Function Scope**
* Functions have their own scope, and variables declared inside a function are not accessible outside of it.
* javascript
* CopyEdit
* function example() {
* let x = 10; // x is local to this function
* console.log(x);
* }
* example(); // Output: 10
* console.log(x); // Error: x is not defined

***DOM (DOCUMENT OBJECT MODEL)***

The **DOM (Document Object Model)** is a programming interface for web documents. It represents the page so that programs can change the document structure, style, and content. Essentially, the DOM is a tree-like structure where each node represents part of the document (like elements, attributes, or text).

**Why Use the DOM?**

The DOM allows you to interact with web pages using programming languages (like JavaScript) to:

* Change HTML elements.
* Modify CSS styles.
* Add or remove HTML elements.
* Handle events (like clicks or form submissions).

Link file in JAVASCRIPTS

**1. Internal Script**

The JavaScript code is placed within the <script> tag inside the HTML <head> or <body>.

2. External Script (Linked Script)

<script src="script.js" async></script>

1. **What is JavaScript?**
2. **What are the different data types in JavaScript?**
3. **What is the difference between var, let, and const?**
4. **What is hoisting in JavaScript?**
5. **What is the difference between == and ===?**
6. **What is a function in JavaScript?**
7. **What are arrays in JavaScript?**
8. **What is the typeof operator used for?**
9. **What is the difference between null and undefined?**
10. **How do you create an object in JavaScript?**
11. **What is a closure in JavaScript?**
12. **What is a callback function?**
13. **What are arrow functions?**
14. **What is the purpose of the setTimeout() function?**
15. **What is the event loop in JavaScript?**

**🔹 15 Practical JavaScript Interview Questions**

1. **Reverse a String**

*Write a function that takes a string as input and returns the reversed string.*

1. **Check for Palindrome**

*Determine whether a given string is a palindrome (reads the same backward as forward).*

1. **Find the Maximum Number in an Array**

*Write a function that returns the largest number in an array.*

1. **Remove Duplicates from an Array**

*Create a function that removes duplicate values from an array.*

1. **Count Vowels in a String**

*Count the number of vowels (a, e, i, o, u) in a given string.*

1. **Check if Two Strings are Anagrams**

*Determine if two strings are anagrams of each other.*

1. **Factorial of a Number**

*Calculate the factorial of a given non-negative integer.*

1. **Check if a Number is Prime**

*Write a function to check if a number is a prime number.*

1. **Generate Fibonacci Sequence**

*Generate the first n numbers in the Fibonacci sequence.*

1. **Sum of Array Elements**

*Calculate the sum of all elements in an array.*

1. **Find the Missing Number in an Array**

*Given an array containing numbers from 1 to n with one missing number, find the missing number.*

1. **Check if a String Contains Only Digits**

*Determine if a string consists solely of numeric digits.*

1. **Convert Celsius to Fahrenheit**

*Write a function to convert a temperature from Celsius to Fahrenheit.*

1. **Capitalize the First Letter of Each Word**

*Capitalize the first letter of each word in a given string.*

1. **Check if an Object is Empty**

*Determine if a JavaScript object has no properties.*