JAVASCRIPT

THEORY ASSIGNMWNT :-

• Question 1: What is JavaScript? Explain the role of JavaScript in web development.

Ans:- Meaning

JavaScript is a high-level, interpreted programming language primarily used to create interactive and dynamic content on websites.

Role of JavaScript in Web Development:

1. **Client-Side Scripting**:

JavaScript runs in the user's browser , enabling interactive feature like image sliders , form validation , dropdown menus , and more without needing to reload the page .

2. **Dynamic Content Manipulation**:
   It allows developers to modify HTML and CSS on the fly using the Document Objects Model (DOM) This means on a web page can be changed dynamically based on user actions .

3. **Event Handling**:
   JavaScript can respond to user action such as click , key presses , mouse movement etc. enhancing user experience through real time feedback .

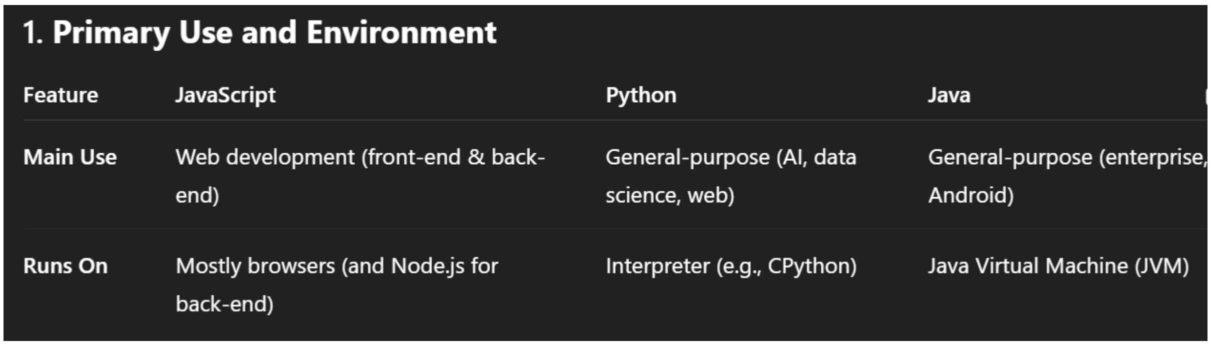
4. **Form Validation**:
   Before sending data to the server, JavaScript can validate user inputs (e.g., checking if a field is empty or if an email address is valid), improving efficiency and reducing server load.

5. **Browser and Feature Detection**:
   JavaScript can detect the user's browser type or features and adapt the functionality accordingly for compatibility.

6. **Framework and Library Support**:
   JavaScript supports popular libraries and frameworks like **React**, **Angular**, and **Vue.js**, which streamline the process of building complex and scalable web applications.

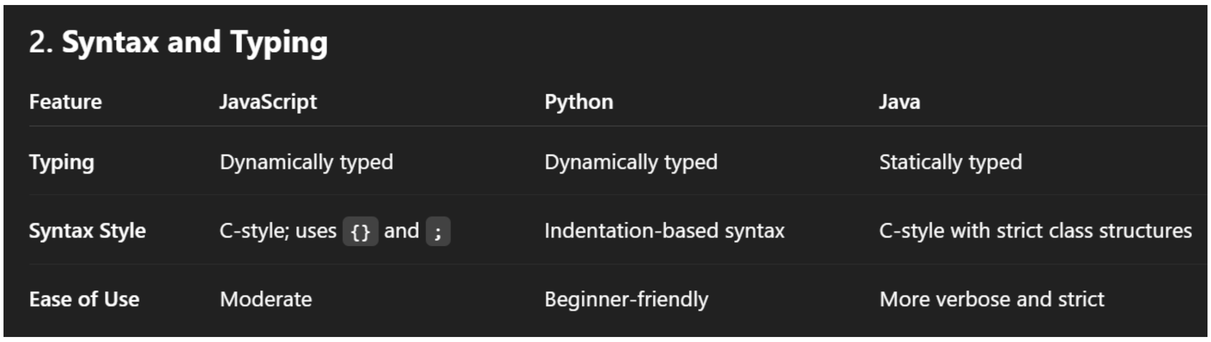
• Question 2: How is JavaScript different from other programming languages like Python or Java?

Ans:-



1. Primary Use and Environment

Feature	JavaScript	Python	Java
Main Use	Web development (front-end & back-end)	General-purpose (AI, data science, web)	General-purpose (enterprise, Android)
Runs On	Mostly browsers (and Node.js for back-end)	Interpreter (e.g., CPython)	Java Virtual Machine (JVM)



2. Syntax and Typing

Feature	JavaScript	Python	Java
Typing	Dynamically typed	Dynamically typed	Statically typed
Syntax Style	C-style; uses `{}` and `;`	Indentation-based syntax	C-style with strict class structures
Ease of Use	Moderate	Beginner-friendly	More verbose and strict

3. **Compilation and Execution**

. JavaScript: interpreted at runtime in the browser or by Node.js.

.Python: interpreted at runtime using interpreters like CPython.

.Java: Compiled to bytecode , then run on the JVM.

4. **Concurrency and Execution Model**

. **JavaScript**: Uses a **single-threaded event loop** with asynchronous callbacks (non-blocking I/O).

. **Python**: Supports multi-threading and multiprocessing, though the Global Interpreter Lock (GIL) limits true parallelism in CPython.

. Java: strong multi-threading support; widely used in concurrent enterprise applications.

5. **Platform Dependence**

.JavaScript: Cross-platform via browsers or Node.js.

.Python: Cross-platform , but depends on having a Pyhton interpreter installed .

.Java: write once, run anywhere (JVM-based)

• Question 3: Discuss the use of <script> tag in HTML. How can you link an external JavaScript file to an HTML document?

Ans:- The <script> tag in HTML is used to **embed or reference JavaScript code** in an HTML document. It tells the browser to **execute the JavaScript** either inline or from an external file.

Using Inline With JavaScript :-

```
<!DOCTYPE html>
<html>
<head>
  <title>Inline Script</title>
</head>
<body>
  <h1>Hello!</h1>

  <script>
    alert("Welcome to the page!");
  </script>
</body>
</html>
```

2. Linking to an External JavaScript file :

```
<!DOCTYPE html>
<html>
<head>
  <title>External JS</title>
</head>
<body>
  <h1>External JavaScript Example</h1>

  <script src="script.js"></script>
</body>
```

</html>

External JavaScript File (script.js)

alert("This is from an external JS file!");

VARIABLE AND DATATYPE

THEORY ASSIGNMENT :-

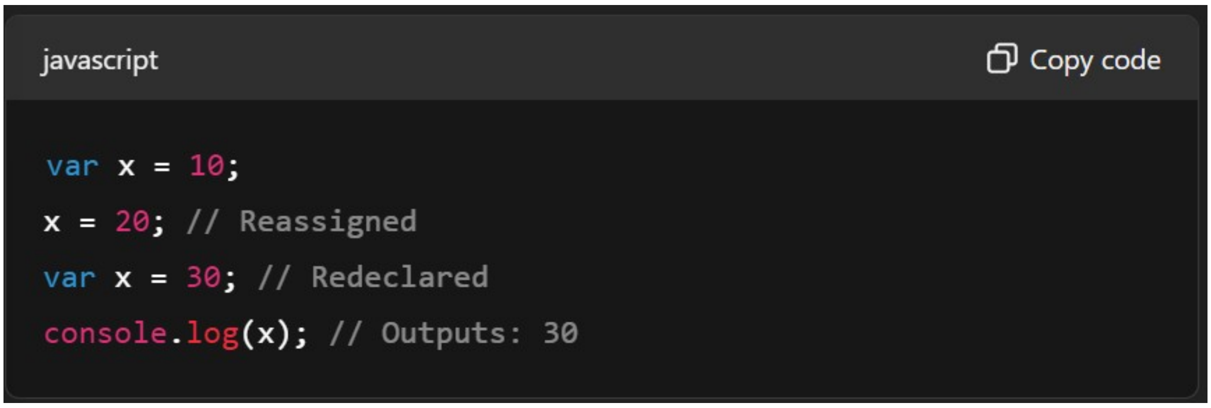
• Question 1: What are variables in JavaScript? How do you declare a variable using var, let and const ?

Ans:- **Variables in JavaScript**

In JavaScript, **variables** are used to **store data values**. A variable acts as a container for holding information that can be referenced and manipulated in a program.

1. var – Function or Global Scope

- **Scope**: Function-scoped or globally scoped.
- **Reassignable**: Yes.
- **Redeclarable**: Yes, within the same scope.
- **Hoisting**: Variables are hoisted and initialized with undefined

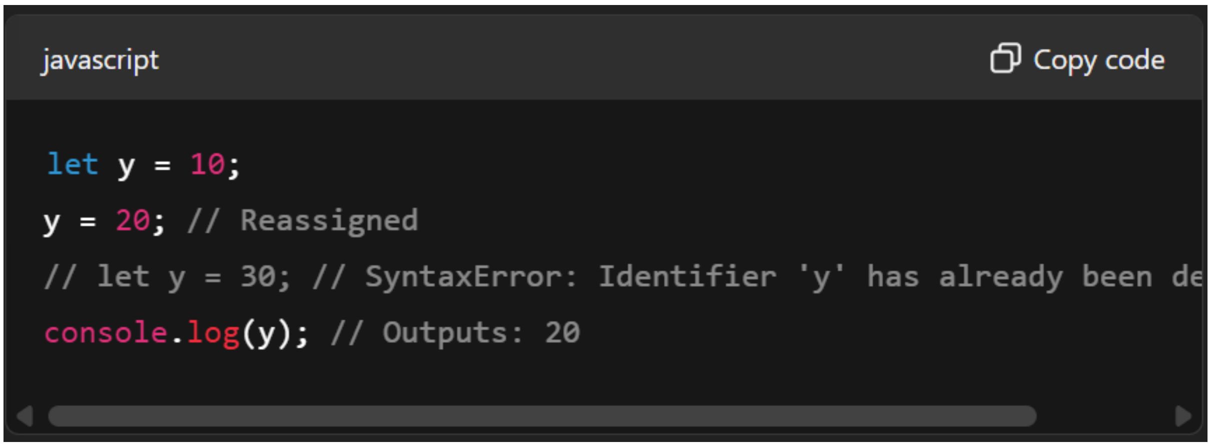


```javascript
var x = 10;
x = 20; // Reassigned
var x = 30; // Redeclared
console.log(x); // Outputs: 30
```

2. let – Block Scope

- **Scope**: Block-scoped (limited to the block, statement, or expression where it's used).
- **Reassignable**: Yes.

- **Redeclarable**: No, within the same scope.



```javascript
let y = 10;
y = 20; // Reassigned
// let y = 30; // SyntaxError: Identifier 'y' has already been de
console.log(y); // Outputs: 20
```

3. const – Block Scope with Constant Value

- **Scope**: Block-scoped.
- **Reassignable**: No.
- **Redeclarable**: No, within the same scope.
- **Hoisting**: Variables are hoisted but not initialized, leading to a "temporal dead zone" until the declaration is encountered.

  Example :-

```
const z = 10;
// z = 20; // TypeError: Assignment to constant variable.
// const z = 30; // SyntaxError: Identifier 'z' has already been declared
console.log(z); // Outputs: 10
```

• Question 2: Explain the different data types in JavaScript. Provide examples for each.

Ans :- Primitive data type –

They represent single values and are immutable . They include :

1. **String –** Represent a sequence orf characters.

   Example :-

   let greeting = "Hello, World!";

2. **Number-** Represent both integer and floating-point numbers.

   Example:-

   let age = 25;

   let pi = 3.14;

3. **BigInt-** Introduced in ES2020, they allow representation of integers larger than Number.MAX_SAFE_INTEGER

Example:-

```
let bigNumber = 1234567890123456789012345678901234567890n;
```

4. **Boolean-** Represent a logical entity with two values true and false.
   Example :-
   ```
   let isActive = true;
   let isCompleted = false;
   ```
5. **undefined**- Indicate that a variable has been declared but not assigned a value.
   Example :-
   ```
       let user;
       console.log(user); // undefined
   ```
6. **null-** Represent the intentional absence of any object value .
   Example :-
   ```
   let selectedItem = null;
   ```
7. symbol- introduced in ES6 , they represent a unique and immutable value, often used as object property identifiers.
   Example:-
   ```
   let sym1 = Symbol('description');
   let sym2 = Symbol('description');
   console.log(sym1 === sym2); // false
   ```

   Non-Primitive (Reference) Data Types:-
   1. **Object-** A collection of key-values pairs .
      Example:-
      ```
      let person = {
        name: "Alice",
        age: 30,
        isEmployed: true
      };
      ```
   2. Array :- A special type of object used for storing ordered collections.
      Example:-
      ```
      let fruits = ["Apple", "Banana", "Cherry"];
      ```
   3. Function :- A block of code designed to perform a particular task.
      Example :-
      ```
      function greet(name) {
        return `Hello, ${name}!`;
      }
      ```

Question 3: What is the difference between undefined and null in JavaScript?

Ans :- **undefined**

- **Type**: undefined is a **type** and also a **value**.
- **Meaning**: It indicates that a variable has been declared but **has not been assigned a value**.

  Example:-

let a;

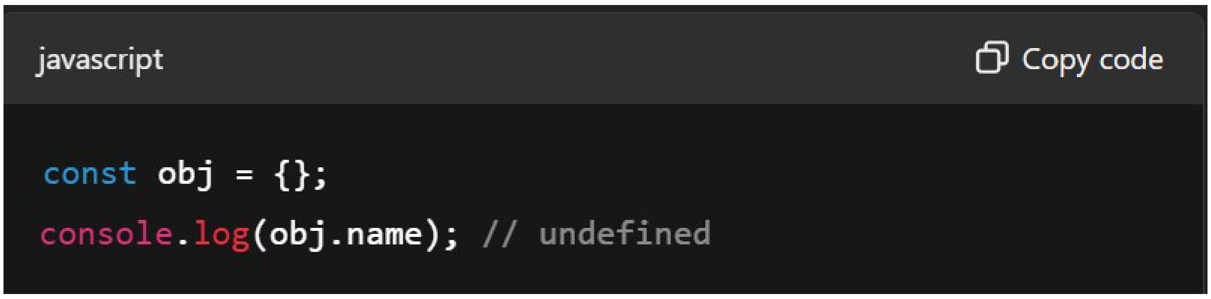
console.log(a); // undefined

Also returned when accessing a non-existent object property or array element:

Example:-

const obj = {};

console.log(obj.name); // undefined



```javascript
const obj = {};
console.log(obj.name); // undefined
```

null

- **Type**: null is a **value**, and its type is technically "object" (this is a long-standing quirk in JavaScript).
- **Meaning**: It represents an **intentional absence** of any object value. Developers typically assign null to a variable to indicate it should be empty.

Example :-

let a = null;

console.log(a); // null

JAVASCRIPT OPERATORS

Question 1: What are the different types of operators in JavaScript? Explain with examples. • Arithmetic operators • Assignment operators • Comparison operators • Logical operators

Ans:- 1. Arithmetic Operators

Operator	Description	Example	Result
`+`	Addition	`5 + 2`	`7`
`-`	Subtraction	`5 - 2`	`3`
`*`	Multiplication	`5 * 2`	`10`
`/`	Division	`10 / 2`	`5`
`%`	Modulus (remainder)	`10 % 3`	`1`
`++`	Increment	`let x = 1; x++`	`2`
`--`	Decrement	`let x = 2; x--`	`1`

2. Assignment Operators

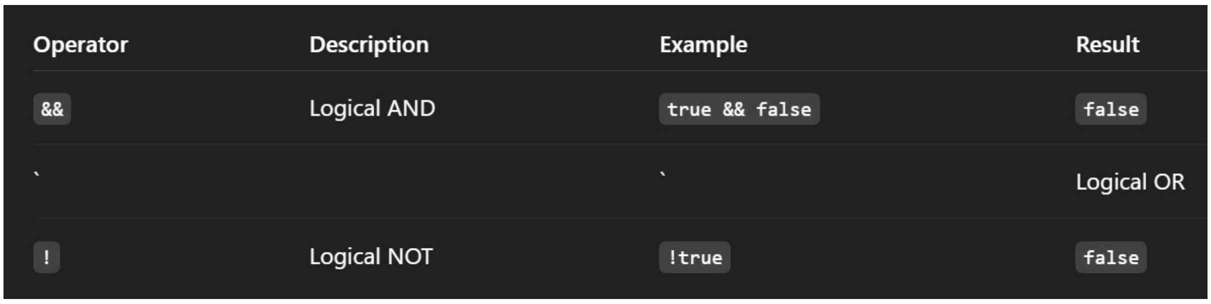
Operator	Description	Example	Result
`=`	Assign	`x = 5`	`x = 5`
`+=`	Add and assign	`x += 2`	`x = x + 2`
`-=`	Subtract and assign	`x -= 2`	`x = x - 2`
`*=`	Multiply and assign	`x *= 3`	`x = x * 3`
`/=`	Divide and assign	`x /= 2`	`x = x / 2`
`%=`	Modulus and assign	`x %= 2`	`x = x % 2`

3. Comparison Operators

Operator	Description	Example	Result
`==`	Equal to (type coerced)	`5 == '5'`	`true`
`===`	Strict equal (no coercion)	`5 === '5'`	`false`
`!=`	Not equal (type coerced)	`5 != '5'`	`false`
`!==`	Strict not equal	`5 !== '5'`	`true`
`>`	Greater than	`5 > 3`	`true`
`<`	Less than	`5 < 3`	`false`
`>=`	Greater than or equal	`5 >= 5`	`true`
`<=`	Less than or equal	`3 <= 5`	`true`

4. Logical Operators

Used to combine multiple conditions (mostly in control flow statements like if).



Operator	Description	Example	Result
`&&`	Logical AND	`true && false`	`false`
`		`	Logical OR
`!`	Logical NOT	`!true`	`false`

Question 2: What is the difference between == and === in JavaScript?

Ans:- **== (Equality Operator)**

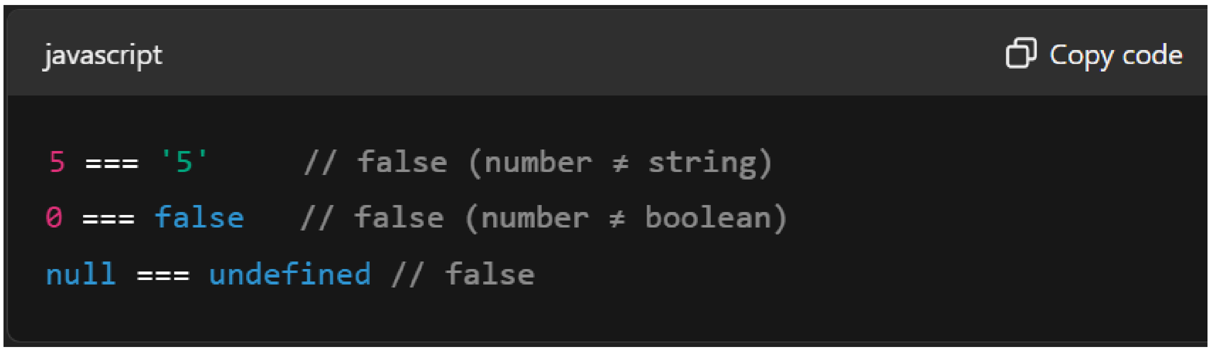
- **Performs type coercion** before comparison.
- Converts operands to the same type **if they are of different types**.
- **Looser comparison**.

Example :-

```
5 == '5'    // true (string '5' is coerced to number 5)
0 == false   // true (false is coerced to 0)
null == undefined // true
```

=== (Strict Equality Operator)

- **Does not perform type coercion**.
- Both value **and type must be the same**.
- **Stricter and more predictable**.



javascript   Copy code

```javascript
5 === '5'      // false (number ≠ string)
0 === false    // false (number ≠ boolean)
null === undefined // false
```

Control Flow (If-Else, Switch)

Question 1: What is control flow in JavaScript? Explain how if-else statements work with an example.

Ans :- **Control flow** in JavaScript refers to the **order in which code is executed**. By default, JavaScript runs code **from top to bottom, line by line**.

if-else Statements Work

The if-else statement is a control structure that **executes different blocks of code based on a condition**.

Example :-

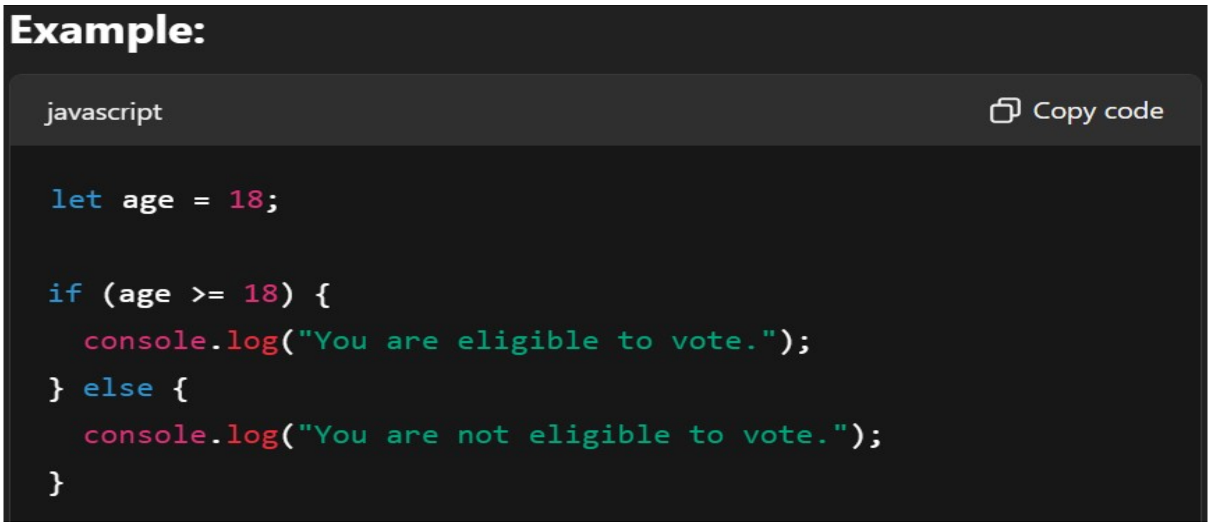
```
if (condition) {

  // Block of code if condition is true

} else {

  // Block of code if condition is false

}
```



Example:

javascript

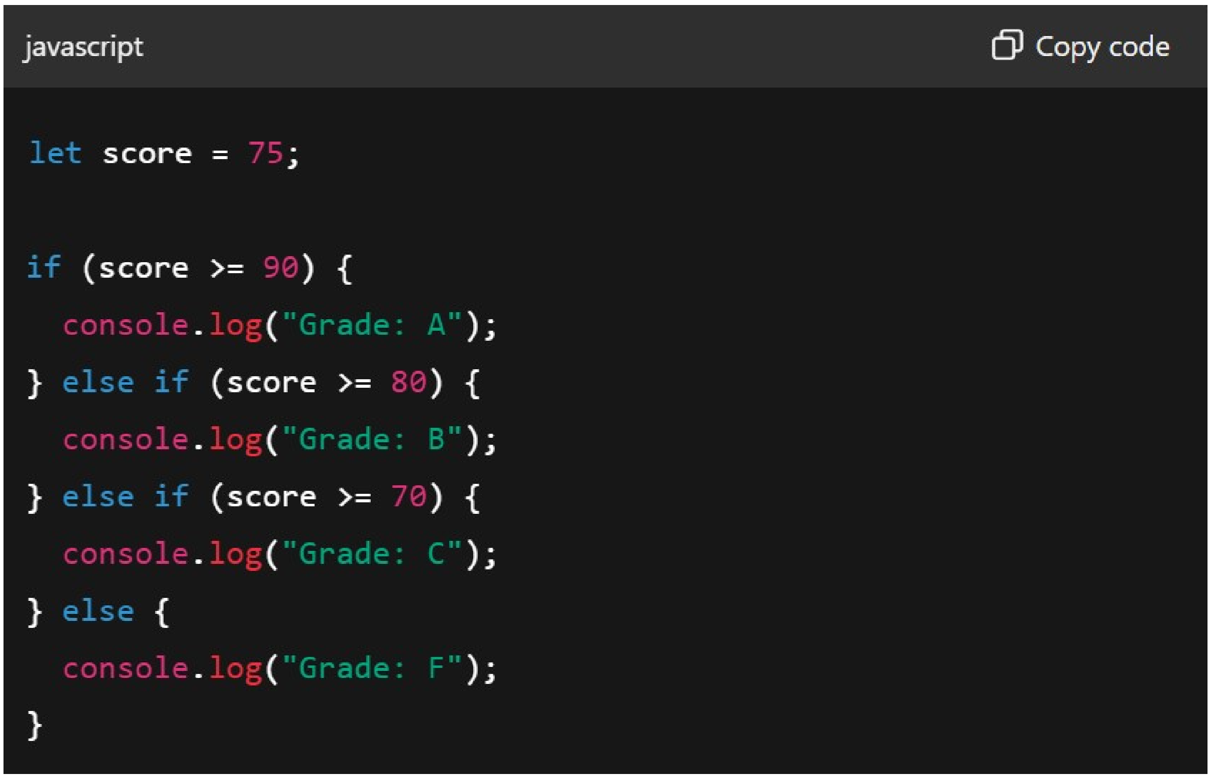
Copy code

```javascript
let age = 18;

if (age >= 18) {
  console.log("You are eligible to vote.");
} else {
  console.log("You are not eligible to vote.");
}
```

if-else if-else

You can also chain multiple conditions using else if.



```javascript
let score = 75;

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");
}
```

Question 2: Describe how switch statements work in JavaScript. When should you use a switch statement instead of if-else?

Ans :- A switch statement in JavaScript is used to perform **different actions based on different conditions**, particularly when comparing the same variable or expression against multiple values.

Syntax :-

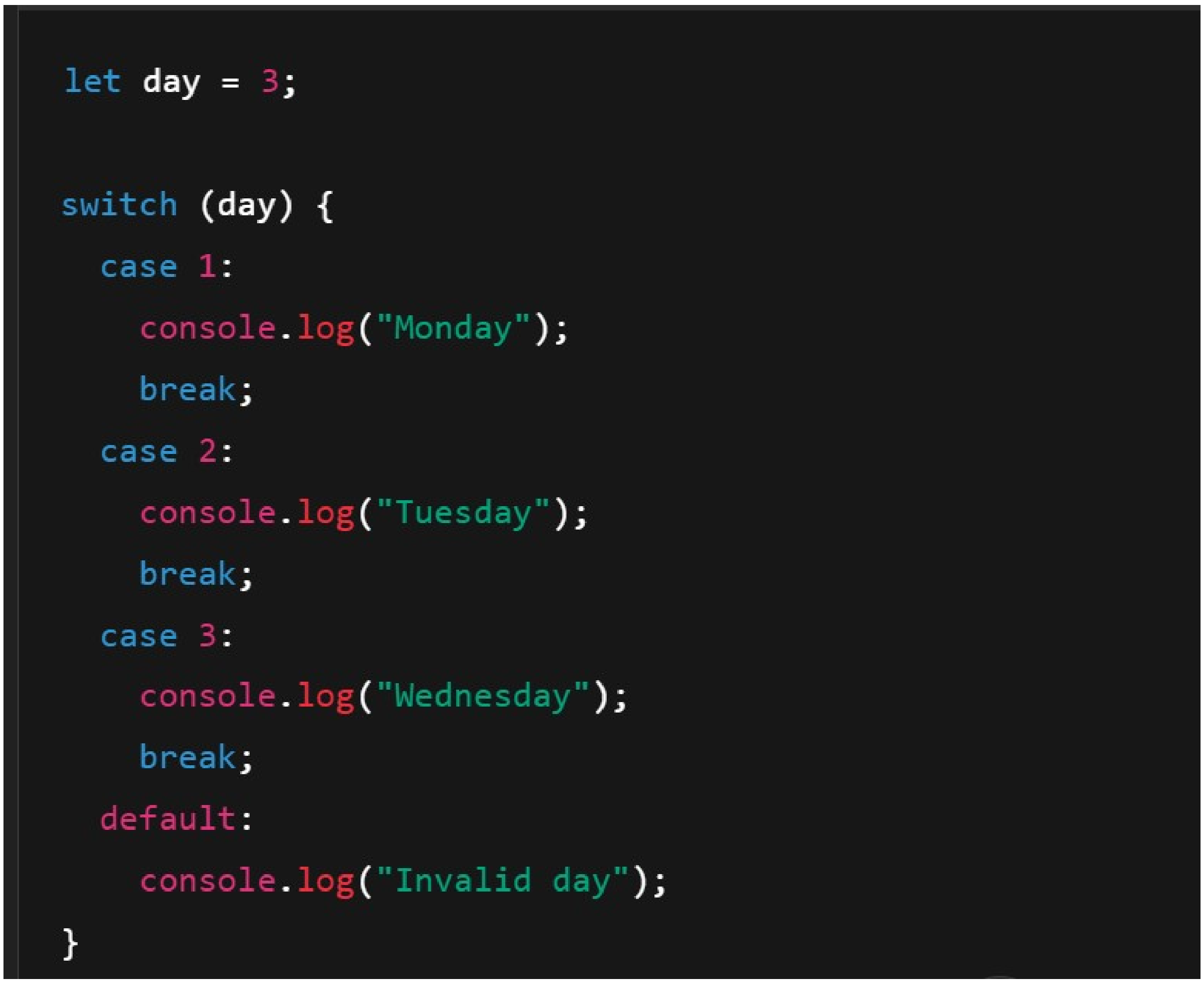
```
   switch (expression) {
 case value1:
   // Code to run if expression === value1
   break;
 case value2:
   // Code to run if expression === value2
   break;
 default:
   // Code to run if no cases match
}
```

expression: The variable or value being tested.

case: A possible value for the expression.

break: Ends the case. Without it, execution "falls through" to the next case.

default: (Optional) Runs if none of the cases match.



```
let day = 3;

switch (day) {
  case 1:
    console.log("Monday");
    break;
  case 2:
    console.log("Tuesday");
    break;
  case 3:
    console.log("Wednesday");
    break;
  default:
    console.log("Invalid day");
}
```

Loops (For , While , Do-While)

Theory Assignment

• Question 1: Explain the different types of loops in JavaScript (for, while, do-while). Provide abasic example of each .

Ans :- **1. for loop**

Use case: When you know **exactly how many times** you want to repeat something.

Syntax :-

```
for (initialization; condition; update) {
  // code to run
}
```

Example :-

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

2. while loop

Use case: When you want to repeat code **as long as a condition is true**, but don't know how many times.

Syntax :-

```
while (condition) {
  // code to run
}
```

Example :-

```
let i = 1;
while (i <= 3) {
  console.log("While Loop:", i);
  i++;
}
```

3. do...while loop

Use case: Similar to while, but this loop will always run at least once, even if the condition is false.

Syntax :-

```
do {
  // code to run
} while (condition);
```

Example :-

```
let i = 1;
```

```
do {
  console.log("Do While Loop:", i);
  i++;
} while (i <= 2);
```

• Question 2: What is the difference between a while loop and a do-while loop?

Ans :- **1. while loop**

- **Condition is checked first**, before the loop body runs.
- If the condition is **false at the beginning**, the loop **will not run at all**

Syntax

```
while (condition) {
  // code to run
}
```

Example :-

```
let x = 5;
while (x < 5) {
  console.log("This will NOT run");
}
```

2. do...while loop

- The **loop body runs once first**, and then the condition is checked.
- This means it will **always run at least once**, even if the condition is false.

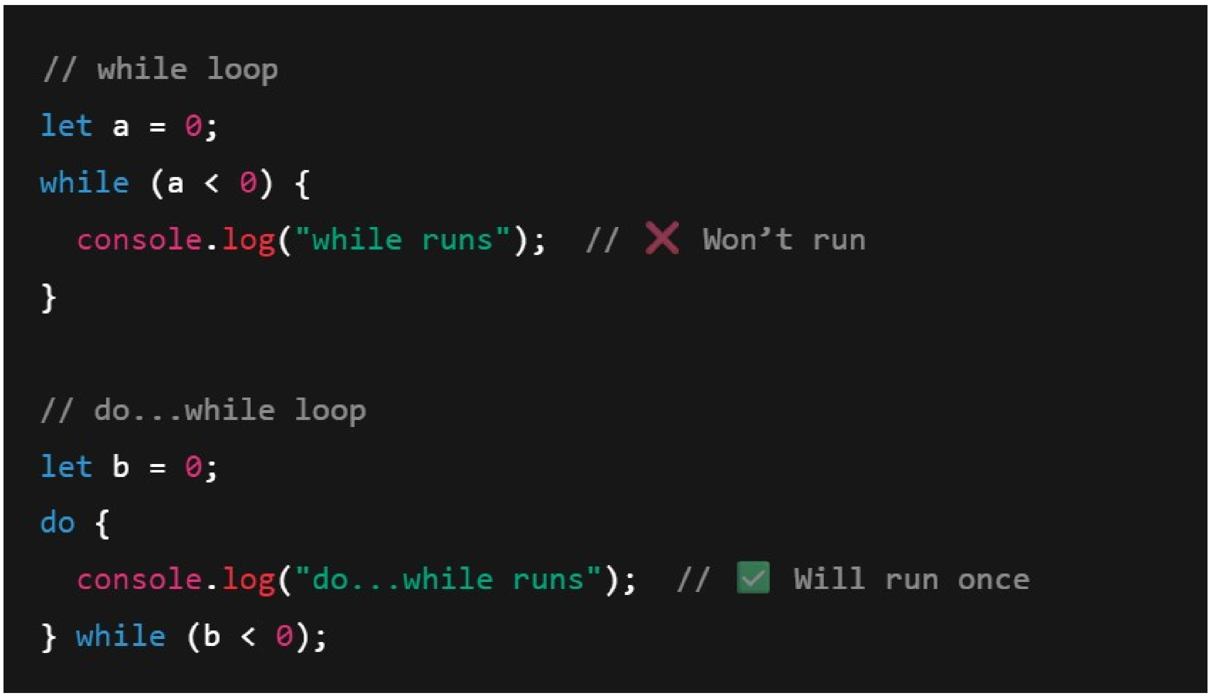
Syntax :-

```
do {
  // code to run
} while (condition);
```

Example :-

```
let y = 5;
```

```
do {
  console.log("This WILL run once");
} while (y < 5);
```



```
// while loop
let a = 0;
while (a < 0) {
  console.log("while runs");  // ❌ Won’t run
}

// do...while loop
let b = 0;
do {
  console.log("do...while runs");  // ✅ Will run once
} while (b < 0);
```

FUNCTIONS

• Question 1: What are functions in JavaScript? Explain the syntax for declaring and calling a function.

Ans :- ◇ **Definition:**

A **function** in JavaScript is a **block of code** designed to perform a **specific task**. You can **reuse** this code by "calling" the function whenever you need it.

1. Function Declaration :-

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

2. **Function Call**

```
functionName(arguments);
```

Example 1: A Simple Function

```
function greet() {
  console.log("Hello, World!");
}

greet();  // Calling the function
```

Output :-

```
Hello, World!
```

Example 2: Function with Parameters

```
function add(a, b) {
  console.log(a + b);
}

add(5, 3);  // Output: 8
```

Example 3: Function with Return Value

```
function multiply(x, y) {
  return x * y;
}

let result = multiply(4, 6);
console.log(result);  // Output: 24
```

• Question 2: What is the difference between a function declaration and a function expression?

Ans :- **Function Declaration**

A **function declaration** defines a named function using the function keyword:

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

Hoisted: Function declarations are **hoisted** completely. This means you can call the function **before** its definition in the code.

```
greet(); // ✅ Works fine


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

Function Expression

A **function expression** creates a function and assigns it to a variable:

```
const greet = function() {
  console.log("Hello!");
};
```

Function expressions are **not hoisted** like declarations. You **cannot** call the function before it is defined.

```
greet(); // ❌ Error: Cannot access 'greet' before initialization


const greet = function() {
  console.log("Hello!");
};
```

• Question 3: Discuss the concept of parameters and return values in functions.

Ans :- 🔹 **Parameters**

- Parameters are **placeholders** used in a function definition.
- They allow you to **pass data into a function** so it can work with different values.

Example :-

```
function greet(name) {
  console.log("Hello, " + name);
}
greet("Ankit");  // Output: Hello, Anushka
```

◇ Return Values

- A function can **return a value** using the return keyword.
- This value can be used wherever the function is called.

Example :-

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


let sum = add(5, 3);  // sum = 8
console.log(sum);     // Output: 8
```

The function add takes two parameters (a and b) and **returns** their sum.

The return value is stored in a variable and can be used later.

ARRAY

Theory Assignment

• Question 1: What is an array in JavaScript? How do you declare and initialize an array?

Ans :- An **array** in JavaScript is a special variable used to **store multiple values** in a single variable.

Instead of declaring separate variables for each value:

```
let item1 = "apple";
let item2 = "banana";
let item3 = "cherry";
```

You can use **one array**:

```
let fruits = ["apple", "banana", "cherry"];
```

☑ How to Declare and Initialize an Array

◇ 1. Using Array Literal (Most Common Way):

```
let colors = ["red", "green", "blue"];
```

◇ 2. **Using new Array() Constructor**:

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

☑ **Array Characteristics**

- Arrays can hold **any data type**: strings, numbers, objects, even other arrays.
- Arrays are **zero-indexed**, meaning the first element is at index 0.

```
 let mix = [10, "hello", true];

        console.log(mix[0]); // Output: 10
```

- Question 2: Explain the method push (), pop (), shift (), and unshift () used in arrays.

Ans :- ◇ **push()**

- Adds **one or more elements to the **end** of the array.
- **Returns** the new length of the array.

```
let fruits = ["apple", "banana"];

           fruits.push("mango");

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

◇ **pop()**

- Removes the **last** element from the array.
- **Returns** the removed element.

```
  let fruits = ["apple", "banana", "mango"];

        let removed = fruits.pop();

           console.log(removed); // "mango"

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

◇ **shift()**

- Removes the **first** element from the array.
- **Returns** the removed element.

```
    let fruits = ["apple", "banana", "mango"];

                let removed = fruits.shift();

          console.log(removed); // "apple"

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

◇ **unshift()**

- Adds **one or more elements** to the **beginning** of the array.
- **Returns** the new length of the array.

```
  let fruits = ["banana", "mango"];

          fruits.unshift("apple");
```

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

OBJECTS

Theory Assignment

- Question 1: What is an object in JavaScript? How are objects different from arrays?

Ans :- Object in JavaScript?

An **object** in JavaScript is a **collection of key-value pairs**. It is used to store **related data and functions** together.

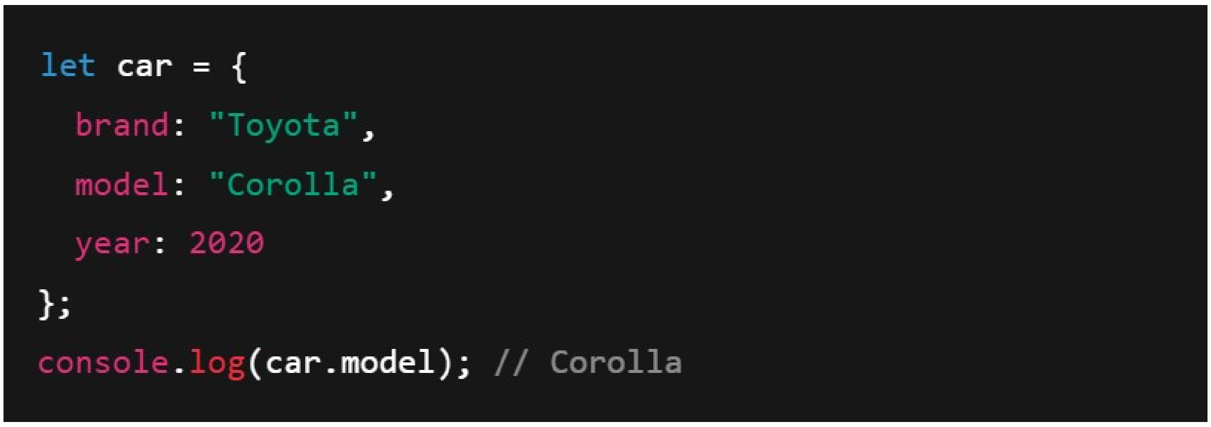
Syntax : -

```
let person = {
name: "John",
age: 30,
isStudent: false
};
```

You can access the values using **dot notation** or **bracket notation**:

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

◇ Object:



```
let car = {
  brand: "Toyota",
  model: "Corolla",
  year: 2020
};
console.log(car.model); // Corolla
```

◇ Array:

```
let car = ["Toyota", "Corolla", 2020];
console.log(car[1]); // Corolla
```

JavaScript Events

Theory Assignment :-

• Question 1: What are JavaScript events? Explain the role of event listeners.

Ans : - JavaScript, **events** are actions or occurrences that happen in the browser, which the JavaScript code can **respond to**.

◇ Examples of events include:

- A user clicking a button (click)
- Typing in a text field (input or keydown)
- Page loading (load)
- Moving the mouse (mousemove)

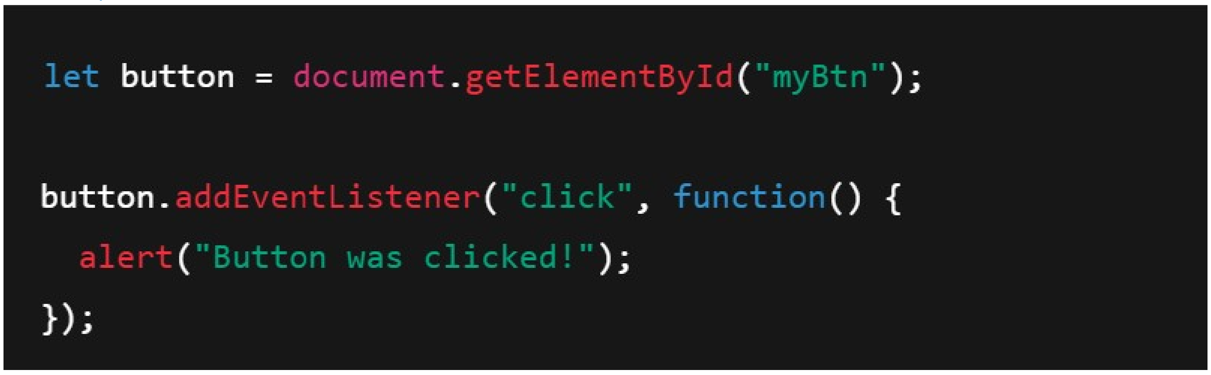
Syntax : -

```
element.addEventListener("event", function);
```

```
let button = document.getElementById("myBtn");


  button.addEventListener("click", function() {
    alert("Button was clicked!");
  });
```

Example : -



```
let button = document.getElementById("myBtn");

button.addEventListener("click", function() {
  alert("Button was clicked!");
});
```

• Question 2: How does the add Event Listener() method work in JavaScript? Provide an example.

Ans : - The add Event Listener() method is used to **attach an event handler** to a specific HTML element.

Syntax :

```
element.addEventListener("event", function, useCapture);
```

Example : -

element: The HTML element you want to attach the event to

"event": A string like "click", "mouseover", "keydown", etc.

function: The function to run when the event occurs useCapture (optional): A boolean value (true or false) that defines the event phase. Usually false.

```
<button id="myBtn">Click Me</button>

<script>
 // Select the button element
 let btn = document.getElementById("myBtn");

 // Add a click event listener
 btn.addEventListener("click", function() {
   alert("Button was clicked!");
 });
</script>
```

DOM Manipulation

Theory Assignment

• Question 1: What is the DOM (Document Object Model) in JavaScript? How does JavaScript interact with the DOM?

Ans : - DOM (Document Object Model)

The DOM is a programming interface for HTML and XML document It represent the structure of a web page as a tree of object where each element

◇ **Example:**

Given this HTML:

html

Copy code

```
<h1 id="title">Welcome</h1>
```

In the DOM, JavaScript sees it like this:

javascript

Copy code

```
document.getElementById("title");
```

📄 **Example in Action:**

html

Copy code

```
<p id="msg">Hello!</p>
```

```
<button onclick="changeText()">Click Me</button>


<script>
  function changeText() {
    document.getElementById("msg").textContent = "Text changed!";
  }
</script>
```

• Question 2: Explain the methods getElementById(), getElementsByClassName(),and querySelector() used to select elements from the DOM.

Ans : - ◇ **getElementById()**

- Selects **one element** by its **ID**.
- Returns the **first matching element** (IDs should be unique).

📃 **Example:**

html

Copy code

```
<p id="msg">Hello</p>


<script>
  let element = document.getElementById("msg");
  console.log(element.textContent); // Output: Hello
</script>
```

◇ **getElementsByClassName()**

- Selects **all elements** with a **specific class name**.
- Returns an **HTMLCollection** (like an array, but not exactly).

📃 **Example:**

html

Copy code

```
<div class="item">Item 1</div>
<div class="item">Item 2</div>


<script>
  let items = document.getElementsByClassName("item");
  console.log(items[0].textContent); // Output: Item 1
```

</script>

◇ **querySelector()**

- Selects the **first element** that **matches a CSS selector**.
- Very flexible — you can select by ID (#id), class (.class), tag, etc.

📄 **Example:**

html

Copy code

```
<p class="text">First</p>
<p class="text">Second</p>


<script>
 let first Text = document.querySelector(".text");
 console.log(firstText.textContent); // Output: First
</script>
```

JavaScript Timing Events (setTimeout, setInterval)

Theory Assignment

• Question 1: Explain the set Timeout() and set Interval() functions in JavaScript. How are they used for timing events?

Ans : - **1. Set Timeout()**

- **Purpose:** Executes a function **once after a specified delay** (in milliseconds).

Syntax:

```
Set Timeout(function, delay, param1, param2, ...)
```

Example:

```
Set 3Timeout(() => {
console.log("This message appears after 2 seconds");
}, 2000);
```

◇ **setInterval()**

This function is used to execute a piece of code **repeatedly** at specified intervals (in milliseconds).

Syntax:

javascript

Copy code

```
setInterval(function, interval, param1, param2, ...);
```

Example :-

```
const timerId = setTimeout(() => {
  console.log("This will not run");
}, 5000);

clearTimeout(timerId); // Cancels the timeout
```

• Question 2: Provide an example of how to use setTimeout() to delay an action by 2 seconds.

Ans : -   example : -

```
function showMessage() {
  console.log("This message appears after 2 seconds");
}

// Call the function after a 2000ms (2 seconds) delay
setTimeout(showMessage, 2000);
```

Same Example Using an Arrow Function (without separate named function):

javascript

Copy code

```
setTimeout(() => {
  console.log("This message also appears after 2 seconds");
}, 2000);
```

Both methods are valid.

1. showMessage is the function we want to run.

2. setTimeout(showMessage, 2000) tells JavaScript to wait **2000 milliseconds** (or 2 seconds) before executing showMessage.

JavaScript Error Handling

Theory Assignment

• Question 1: What is error handling in JavaScript? Explain the try, catch, and finally blocks with an example.

Ans : - **Error handling** in JavaScript is the process of responding to and managing **runtime errors** (unexpected issues that occur while the script is running)

- **try block**
    - The code that may potentially cause an error goes inside the try block.
- **catch block**

- If an error occurs inside the try, control jumps to the catch block.
- The error object provides information about what went wrong.

◇ **finally block (optional)**

- This block always runs — **whether an error occurred or not**.
- Used for cleanup actions like closing files, stopping a load

- **Example of try-catch-finally**
- javascript
- Copy code
- try {
-   // Code that might throw an error
-   let x = 5;
-   let y = x.toUpperCase(); // Error: toUpperCase() is not a function for number
-   console.log("This line will not run");
- } catch (error) {
-   // Handle the error
-   console.log("An error occurred:", error.message);
- } finally {
-   // Always runs
-   console.log("This block runs no matter what");
- }

• Question 2: Why is error handling important in JavaScript applications?

Ans: -

◇ **1. Prevents Application Crashes**

Without proper error handling, a single unexpected error can crash the entire application or cause it to behave unpredictably.

Example: A broken API response might crash your website if not handled properly.

◇ **2. Improves User Experience**
Good error handling shows meaningful messages to users instead of just failing silently or displaying confusing errors.

Example: Showing "Failed to load data, please try again" instead of a blank page.

◇ **3. Easier Debugging**
By catching and logging errors, developers can track down bugs more easily.

Example: Using console.log(error.message) or sending logs to a monitoring service.

◇ **4. Ensures Application Stability**
finally blocks or custom handlers can ensure that cleanup tasks (like closing popups or loaders) always happen, even if an error occurs.

Example: A loading spinner is removed even if data fetching fails.

◇ **5. Handles Unexpected Situations Gracefully**
Errors from user input, network issues, or third-party APIs can be anticipated and managed without disrupting the entire system.

Example: Catching a JSON.parse() error when receiving bad data from an API.