Module 17) Javascript for Full Stack Course

JavaScript Introduction

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

Answer: JavaScript is a high-level, interpreted programming language that is primarily used to create dynamic and interactive content on websites. It is one of the three core technologies of web development, alongside HTML and CSS. While HTML provides the structure and CSS defines the style, JavaScript adds behaviour and interactivity to web pages.

Role of JavaScript in Web Development

JavaScript plays a crucial role in the development of modern websites. Its main contributions are outlined below:

1. **Client-Side Interactivity:**
   JavaScript enables real-time user interactions on web pages. For example, it can be used to create image sliders, toggle menus, and validate form inputs without needing to reload the page.
2. **DOM Manipulation:**
   JavaScript can interact with and modify the Document Object Model (DOM) of a webpage. This allows developers to change the content and style of elements dynamically, based on user actions or other events.
3. **Event Handling:**
   Through JavaScript, developers can respond to user actions such as mouse clicks, keyboard inputs, and form submissions. This makes the user experience more interactive and responsive.
4. **Asynchronous Communication:**
   JavaScript supports asynchronous operations using techniques such as AJAX and the Fetch API. This allows data to be exchanged with a server in the background, enabling features like live search and real-time notifications without refreshing the page.
5. **Client-Side Storage:**
   JavaScript provides web storage capabilities such as localStorage and sessionStorage, which allow websites to store data on the user's browser. This can be used for saving user preferences, caching data, or maintaining session state.

6. **Use of Frameworks and Libraries:**
   Many JavaScript frameworks and libraries, such as React, Angular, Vue.js, and jQuery, are widely used in modern web development. These tools simplify complex tasks, promote efficient coding practices, and help build scalable single-page applications.
7. **Server-Side Development:**
   With the introduction of Node.js, JavaScript can also be used for server-side programming. This allows developers to use JavaScript for both the front-end and back-end, enabling full-stack development with a single language.

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

Answer: JavaScript, Python, and Java are all high-level programming languages, but they differ significantly in terms of design, use cases, execution environment, and syntax. Below is a detailed comparison highlighting the key differences:

1. Execution Environment

- **JavaScript**: Primarily runs in web browsers (client-side), although it can also run on servers using environments like Node.js.
- **Python**: Runs on the server or local machine. It is not natively supported by browsers for front-end development.
- **Java**: Runs on the Java Virtual Machine (JVM), which allows it to be platform-independent. Java applications are compiled into bytecode before execution.

2. Use Cases

- **JavaScript**: Mainly used for web development to create interactive front-end features. With Node.js, it is also used for back-end development.
- **Python**: Known for its simplicity and readability, Python is used in data science, artificial intelligence, machine learning, web development (using frameworks like Django or Flask), and automation.
- **Java**: Widely used in enterprise applications, Android app development, and large-scale systems requiring high performance and scalability.

3. Syntax and Language Design

- **JavaScript**: Dynamically typed, uses prototype-based inheritance, and is event-driven, especially in the browser environment.
- **Python**: Dynamically typed, emphasizes readability with strict indentation rules. Follows a simple and clean syntax.
- **Java**: Statically typed, uses class-based object-oriented programming. Requires more boilerplate code compared to JavaScript and Python.

4. Compilation and Interpretation

- **JavaScript**: Interpreted (or Just-In-Time compiled) by the browser's JavaScript engine (e.g., V8 in Chrome).
- **Python**: Interpreted by the Python interpreter. Can be compiled to bytecode, but typically run as scripts.
- **Java**: Compiled to bytecode, which is then executed by the JVM.

5. Concurrency Model

- **JavaScript**: Uses an event-driven, non-blocking model with an event loop and async/await or callbacks.
- **Python**: Traditionally uses multi-threading or multiprocessing for concurrency, although async features are available (e.g., asyncio).
- **Java**: Has a robust concurrency model with multi-threading and built-in thread management features.

6. Type System

Language	Typing	Example
JavaScript	Dynamic	let x = 10; x = "text";
Python	Dynamic	x = 10; x = "text"
Java	Static	int x = 10; String y = "text";

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

Answer: The <script> tag in HTML is used to define client-side JavaScript code within an HTML document. It allows developers to either write JavaScript code directly in the HTML file or link to an external JavaScript file.

1. Purpose of the <script> Tag

The <script> tag enables the execution of JavaScript code that can control the behavior of web pages. It is typically placed within the <head> or at the end of the <body> section of an HTML document. The timing of the script loading and execution can affect the page's performance and interactivity.

2. Using Inline JavaScript

Developers can write JavaScript code directly inside the <script> tag, which is known as inline scripting.

Example:

```html
<!DOCTYPE html>
<html>
<head>
    <title>Inline JavaScript Example</title>
</head>
<body>
    <h1>Hello World</h1>
    <script>
        alert("Welcome to the website!");
    </script>
</body>
</html>
```

3. Linking an External JavaScript File

To separate JavaScript logic from the HTML structure, it is common practice to place JavaScript code in an external file and link it to the HTML document using the src attribute of the <script> tag.

Syntax:

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

- src: Specifies the path to the external JavaScript file.
- The file typically has a .js extension.

Example:
Assume there is a file named script.js :

```html
<!DOCTYPE html>
<html>
<head>
    <title>External JavaScript Example</title>
</head>
<body>
    <h1>Hello World</h1>
    <script src="script.js"></script>
</body>
</html>
```

4. Placement Best Practices

- Placing the <script> tag **at the end of the <body>** is a common practice to ensure that the HTML content loads before the script executes.
- Alternatively, the defer or async attribute can be used in the <script> tag in the <head> section to control the loading behavior.

Example with defer:

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

Variables and Data Types

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

Answer: In JavaScript, **variables** are containers used to store data values. They are fundamental for managing and manipulating data within a program. Variables can hold various data types, including numbers, strings, arrays, objects, and more.

1. Variables

Variables act as symbolic names for values in a program. Once a value is assigned to a variable, it can be accessed or modified throughout the code. JavaScript is a dynamically typed language, which means a variable can hold different data types at different times.

2. Declaring Variables

JavaScript provides three keywords to declare variables: var, let, and const. Each has different characteristics related to scope, hoisting, and mutability.

3. Using var

- Introduced in older versions of JavaScript (ES5 and earlier).
- Function-scoped.
- Can be redeclared and updated.
- Hoisted (moved to the top of the function scope), but **not initialized** until the actual line of code is executed.

Example:

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

---

4. Using let

- Introduced in ES6 (2015).
- Block-scoped (limited to {} block where it is declared).
- Can be updated but **not redeclared** within the same scope.
- Hoisted but **not initialized** (accessing before declaration results in a ReferenceError).

Example:

```
let city = "New York";
city = "Los Angeles"; // valid
```

---

5. Using const

- Also introduced in ES6.
- Block-scoped.
- Cannot be updated or redeclared (for primitive values).
- Must be initialized at the time of declaration.
- For objects and arrays, the reference cannot change, but the contents can be modified.

Example:

```
const country = "India";
country = "USA"; // Error: Assignment to constant variable
```

With an object:

```
const person = { name: "John" };
person.name = "Mike"; // Allowed: modifying property
```

---

6. Comparison Table

Keyword	Scope	Redeclaration	Reassignment	Hoisting	Initialization Required
var	Function scope	Yes	Yes	Yes (undefined)	No
let	Block scope	No	Yes	Yes (but not initialized)	No
const	Block scope	No	No (for primitives)	Yes (but not initialized)	Yes

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

Answer: In JavaScript, data types define the kind of values a variable can hold. Understanding data types is essential for writing efficient and error-free code. JavaScript data types are broadly categorized into **primitive types** and **non-primitive (reference) types**.

1. Primitive Data Types

Primitive types are immutable and store single values. JavaScript has the following primitive data types:

a) String

Represents a sequence of characters enclosed in single ('), double ("), or backtick (`) quotes.

Example:

```
let name = "Alice";
```

---

b) Number

Represents both integer and floating-point numbers.

Example:

```
let age = 25;
let price = 99.99;
```

---

c) Boolean

Represents a logical entity having two values: true or false.

Example:

```
let isLoggedIn = true;
```

---

d) Undefined

A variable that has been declared but not assigned a value is undefined.

Example:

```
let x;
console.log(x); // undefined
```

---

e) Null

Represents an intentional absence of any object value.

Example:

```
let data = null;
```

---

f) Symbol (ES6)

Represents a unique and immutable identifier.

Example:

```
let id = Symbol("userID");
```

---

g) BigInt (ES11)

Used to represent integers larger than the Number type can safely store.

Example:

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

---

2. Non-Primitive (Reference) Data Types

These types can hold multiple values and are mutable. They include:

a) Object

Used to store collections of key-value pairs.

Example:

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

---

b) Array

A special type of object used to store ordered collections of values.

Example:

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

---

c) Function

Functions are objects in JavaScript and can be assigned to variables.

Example:

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

---

3. Type Summary Table

Data Type	Description	Example
String	Textual data	"Hello"
Number	Numeric values	42, 3.14
Boolean	Logical true/false	true, false
Undefined	Declared but unassigned variable	let x;
Null	Explicitly no value	let y = null;
Symbol	Unique identifier (ES6)	Symbol("id")
BigInt	Very large integers (ES11)	12345678901234567890n
Object	Collection of key-value pairs	{ name: "Alice", age: 25 }
Array	Ordered list of values	["a", "b", "c"]
Function	Callable object	function greet() {}

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

Answer: In JavaScript, both undefined and null are primitive values that represent the absence of a meaningful value. However, they are used in different contexts and have distinct meanings and behaviours.

1. Definition

- **undefined**: A variable that has been declared but has not yet been assigned a value automatically gets the value undefined.
- **null**: Represents the intentional absence of any object value. It is explicitly assigned by the programmer.

2. Type

- **undefined**: Has the type "undefined".
- **null**: Has the type "object" (this is a known quirk in JavaScript).

3. Usage Context

- **undefined** is usually the default value given by JavaScript to:
    - Declared variables that haven't been assigned a value
    - Function parameters that are not provided
    - Missing object properties
- **null** is used by developers to explicitly indicate that a variable should hold "no value" or is "empty".

4. Equality Comparison

- **Loose Equality (==)**: undefined == null is true
- **Strict Equality (===)**: undefined === null is false (because their types differ)

5. Summary Table

Feature	undefined	null
Type	"undefined"	"object" (typeof quirk)
Assigned by	JavaScript (automatically)	Developer (explicitly)
Meaning	Variable declared but not assigned	Intentional absence of value
Equality (==)	Equal to null	Equal to undefined
Strict Equality (===)	Not equal to null	Not equal to undefined

JavaScript Operators

Question 1: What are the different types of operators in JavaScript? Explain with examples.

→ Arithmetic operators
→ Assignment operators
→ Comparison operators
→ Logical operators

Answer: In JavaScript, **operators** are special symbols or keywords used to perform operations on variables and values. Operators help build expressions and control logic in programs. The most common types include arithmetic, assignment, comparison, and logical operators.

1. Arithmetic Operators

These operators are used for performing mathematical operations.

Operator	Description	Example	Result
+	Addition	5 + 3	8
-	Subtraction	10 - 4	6
*****	Multiplication	6 * 2	12
/	Division	10 / 2	5
%	Modulus (remainder)	7 % 3	1

Pre- and Post-Increment/Decrement Operators

These are used to increase or decrease a variable's value by one.

Operator	Type	Example	Explanation	Result
++x	Pre-increment	let x = 5; ++x;	Increments before using value	x = 6
x++	Post-increment	let x = 5; x++;	Uses value, then increments	x = 6 (after)
--x	Pre-decrement	let x = 5; --x;	Decrements before using value	x = 4
x--	Post-decrement	let x = 5; x--;	Uses value, then decrements	x = 4 (after)

Example:

```
let a = 10, b = 3;
console.log(a + b); // Output: 13

console.log(a % b); // Output: 1
console.log(++a); // Output: 10 (pre-increment)
console.log(b++); // Output: 3 (post-increment), b is now 4
```

2. Assignment Operators

Used to assign values to variables.

Operator	Description	Example	Equivalent to
=	Assignment	x = 10	Assign 10 to x
+=	Addition assignment	x += 5	x = x + 5
-=	Subtraction assignment	x -= 2	x = x - 2
*=	Multiplication assignment	x *= 3	x = x * 3
/=	Division assignment	x /= 2	x = x / 2
%=	Modulus assignment	x %= 4	x = x % 4

Example:

```
let x = 10;
x += 5; // Equivalent to x = x + 5
console.log(x); // Output: 15
```

---

3. Comparison Operators

Used to compare two values, returning a Boolean (true or false).

Operator	Description	Example	Result
==	Equal to (loose)	5 == "5"	true
===	Equal to (strict)	5 === "5"	false
!=	Not equal (loose)	4 != 3	true
!==	Not equal (strict)	4 !== "4"	true
>	Greater than	6 > 3	true
<	Less than	2 < 5	true
>=	Greater than or equal to	7 >= 7	true
<=	Less than or equal to	3 <= 2	false

Example:

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

---

4. Logical Operators

Logical operators in JavaScript are used to combine or manipulate Boolean (true/false) values. They are vital in decision-making and controlling program flow.

a) Logical AND (&&)

Returns true only if **both operands** are true.

b) Logical OR (||)

Returns true if **at least one operand** is true.

c) Logical NOT (!)

Inverts the Boolean value: true becomes false, and false becomes true.

Truth Table for Logical Operators

A (left)	B (right)	A && B	A || B	!A
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

Example:

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

Examples Combining Operators

Arithmetic + Assignment:

```
let num = 10;
num += 5; // Equivalent to num = num + 5
console.log(num); // Output: 15
```

Comparison + Logical:

```
let age = 25;
console.log(age > 18 && age < 30); // Output: true
console.log(age < 18 || age > 30); // Output: false
```

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

Answer: In JavaScript, == and === are both comparison operators used to compare two values, but they differ in how they compare these values.

1. == (Equality Operator - Loose Equality)

- Compares **values** for equality **after performing type coercion** if the types differ.
- JavaScript tries to convert the operands to the same type before making the comparison.
- This can sometimes lead to unexpected results.

Example:

```
5 == "5";     // true, because "5" (string) is converted to 5(number)
0 == false;         // true, because false is converted to 0
null == undefined; // true, these are considered equal by loose
equality
```

---

2. === (Strict Equality Operator)

- Compares **both value and type** without performing type coercion.
- Both the value and the data type must be the same for it to return true.
- This operator is generally preferred to avoid unexpected bugs.

Example:

```
5 === "5"; // false, because one is a number and the other is string
0 === false;// false, because one is number and the other is Boolean
null === undefined; // false, different types
```

Summary Table

Operator	Type Coercion	Compares Type	Example	Result
==	Yes	No	5 == "5"	true
===	No	Yes	5 === "5"	false

Control Flow (If-Else, Switch)

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

Answer: Control flow refers to the order in which individual statements, instructions, or function calls are executed or evaluated in a program. It determines how the program proceeds based on certain conditions or repeated actions.

JavaScript provides various control flow structures, such as:

- Conditional statements (if, if-else, switch)
- Loops (for, while, do-while)

These structures help the program make decisions and execute code blocks selectively.

If-Else Statements

The if-else statement is a fundamental control flow structure that executes a block of code if a specified condition evaluates to true; otherwise, it executes another block of code.

Syntax:

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

- The **condition** inside the parentheses is evaluated.
- If the condition is true, the code inside the first block executes.
- If the condition is false, the code inside the else block executes.

Example:

```
let age = 18;

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

Explanation:

- The condition age >= 18 is checked.
- Since age is 18, the condition is true, so the message "You are eligible to vote." is printed.

- If age were less than 18, the else block would execute instead.

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

Answer: A **switch statement** is a control flow structure that allows a variable or expression to be tested against multiple possible values, called **cases**. It provides a more readable and organized way to perform different actions based on the value of a variable compared to multiple if-else statements.

Syntax of Switch Statement

```
switch (expression) {
  case value1:
    // code to execute if expression === value1
    break;
  case value2:
    // code to execute if expression === value2
    break;
  // more cases...
  default:
    // code to execute if no case matches
}
```

- The expression inside switch is evaluated once.
- It is then compared with each case value using **strict equality (===)**.
- If a matching case is found, the corresponding block executes.
- The break statement prevents fall-through to the next cases.
- The default case runs if no match is found (optional).

Example of Switch Statement

```
let day = 3;
let dayName;

switch (day) {
  case 1:
    dayName = "Monday";
    break;
  case 2:
    dayName = "Tuesday";
    break;
  case 3:
    dayName = "Wednesday";
    break;
  case 4:
    dayName = "Thursday";
    break;
  default:
    dayName = "Unknown day";
```

```
}

console.log(dayName);

// Output: Wednesday
```

When to Use Switch Instead of If-Else

- **Use switch when you need to compare the same variable or expression against multiple discrete values.** This makes the code cleaner, easier to read, and maintain compared to long if-else if chains.
- switch is ideal when there are many conditions based on one variable, such as menu options, days of the week, or command processing.
- **Use if-else** when the conditions involve ranges, complex expressions, or different variables, as switch only supports strict equality with specific values.

Loops (For, While, Do-While)

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

Answer: Loops in JavaScript are control flow statements used to repeatedly execute a block of code as long as the specified condition is true. They help automate repetitive tasks and are fundamental in programming.

1. For Loop

The for loop repeats a block of code a specific number of times. It consists of three parts: initialization, condition, and iteration expression.

Syntax:

```
for (initialization; condition; iteration) {
      // code to execute
   }
```

- Initialization: executed once before the loop starts.
- Condition: checked before each iteration; loop continues while true.
- Iteration: executed after each loop iteration.

Example:

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

```
// Output: 1 2 3 4 5
```

---

2. While Loop

The while loop executes a block of code repeatedly as long as the given condition remains true. The condition is evaluated before each iteration.

Syntax:

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

Example:

```
let i = 1;
while (i <= 5) {
```

```
  console.log(i);
  i++;
}

// Output: 1 2 3 4 5
```

---

3. Do-While Loop

The do-while loop is similar to the while loop, but it guarantees that the block of code executes **at least once** because the condition is checked **after** the execution of the loop body.

Syntax:

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

Example:

```
let i = 1;
do {
  console.log(i);
  i++;
} while (i <= 5);

// Output: 1 2 3 4 5
```

Loop Type	Condition Checked	Executes at Least Once?	Best Use Case
for	Before each loop	No	When the number of iterations is known
while	Before each loop	No	When loop depends on a condition
do-while	After each loop	Yes	When code must run at least once

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

Answer: Both **while** and **do-while** loops in JavaScript are used to repeatedly execute a block of code based on a condition. However, the key difference lies in **when the condition is checked** and **how many times the loop body executes**.

1. While Loop

- The condition is evaluated **before** the execution of the loop body.
- If the condition is false at the beginning, the loop body **may not execute at all**.
- Used when you want to repeat the code **only if** the condition is true from the start.

2. Do-While Loop

- The loop body is executed **once first**, and then the condition is checked.
- The loop body **always executes at least once**, regardless of the condition.
- Useful when the code inside the loop must run **at least once** before checking the condition.

Feature	While Loop	Do-While Loop
Condition checked	Before loop execution	After loop execution
Executes at least once	No	Yes
Use case	When condition must be true to start	When loop body must run once regardless

Functions

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

Answer: In JavaScript, **functions** are reusable blocks of code designed to perform a specific task. Functions are objects in JavaScript and serve as the foundation for both **procedural** and **functional** programming patterns.

Functions help in:

- Organizing code into modular components.
- Improving code readability and reusability.
- Encapsulating logic for specific operations.
- Implementing callbacks and higher-order behaviour.

Syntax for Declaring a Function

General Syntax:

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

- functionName is the identifier for the function.
- parameters are optional inputs.
- Code inside {} is the body of the function.

Syntax for Calling a Function

To **invoke** or **call** a function:

```
functionName(arguments);
```

Example:

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

greet("John"); // Output: Hello, John
```

Types of Functions in JavaScript

JavaScript supports several types of functions, each with unique syntax and use cases.

1. Function Declaration (Named Function)

Declared using the function keyword.

Syntax:

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

Calling:

```
add(2, 3); // Returns 5
```

Hoisting: Function declarations are hoisted to the top of their scope.

---

2. Function Expression

A function assigned to a variable. May be **anonymous** or **named**.

Syntax:

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

Calling:

```
multiply(4, 5); // Returns 20
```

Not hoisted like function declarations.

---

3. Arrow Function Expression (ES6+)

Introduced in ES6, provides a shorter syntax. Ideal for small or inline functions.

Syntax:

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

Calling:

```
subtract(10, 3); // Returns 7
```

Arrow functions do **not bind their own this** and are not suitable for all contexts (e.g., constructors, event handlers with this).

---

4. Anonymous Function

A function without a name, usually used in function expressions or callbacks.

Example:

```
setTimeout(function() {
  console.log("Executed after 2 seconds");
}, 2000);
```

5. Named Function Expression

A function expression with a name for recursion or debugging.

Example:

```
const factorial = function fact(n) {
  return n <= 1 ? 1 : n * fact(n - 1);
};
```

---

6. Immediately Invoked Function Expression (IIFE)

A function that runs as soon as it is defined.

Syntax:

```
(function() {
  console.log("IIFE executed");
})();
```

Often used for scoping and avoiding global namespace pollution.

---

7. Constructor Function

Used with the new keyword to create objects.

Syntax:

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

const p1 = new Person("Alice", 30);
```

---

8. Generator Function (ES6+)

Declared using function*, can pause execution using yield.

Syntax:

```
function* idGenerator() {
  let id = 1;
  while (true) {
    yield id++;
  }
}
```

Calling:

```
const gen = idGenerator();
console.log(gen.next().value); // 1
console.log(gen.next().value); // 2
```

---

9. Async Function (ES8+)

Declared with the async keyword, always returns a Promise.

Syntax:

```
async function fetchData() {
  let response = await fetch('https://api.example.com/data');
  let data = await response.json();
  return data;
}
```

Type	Syntax Example	Key Feature
Function Declaration	function fn() {}	Hoisted, named
Function Expression	const fn = function() {}	Not hoisted, anonymous/named
Arrow Function	const fn = () => {}	Short syntax, no own this
IIFE	(function() {})();	Executes immediately
Constructor Function	function Obj() {} + new Obj()	Used to create objects
Generator Function	function* gen() { yield 1; }	Supports iteration via yield
Async Function	async function fn() {}	Handles async code with await

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

Answer: In JavaScript, functions can be defined in multiple ways. Two of the most common forms are **Function Declarations** and **Function Expressions**. While both can be used to create reusable blocks of code, there are key differences in their syntax, behavior, and how the JavaScript engine processes them, especially in terms of **hoisting**, **naming**, and **when they are initialized**.

1. Function Declaration

A **Function Declaration** (also known as a function statement) defines a named function using the function keyword at the beginning of the statement.

Syntax:

```
function functionName(parameters) {
  // function body
}
```

Example:

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

Key Features:

- The function **must have a name**.
- **Hoisted**: Function declarations are hoisted to the top of their scope. This means they can be called **before** their definition in the code.

- Useful for defining functions that are used multiple times throughout the code.

Usage Example with Hoisting:

```
sayHello(); // Works fine

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

---

2. Function Expression

A **Function Expression** defines a function as part of an expression. It can be **named** or **anonymous**, and is often assigned to a variable or passed as an argument.

Syntax:

```
const functionName = function(parameters) {
  // function body
};
```

Example (Anonymous Function Expression):

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

Key Features:

- Can be **anonymous** (no function name) or named.
- **Not hoisted**: The function expression is created **only when the script execution reaches that line**.
- Often used as callbacks, inline functions, or to assign functions to variables.

Usage Example:

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

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

3. Differences Between Function Declaration and Function Expression

Feature	Function Declaration	Function Expression
Syntax	function name() {}	const name = function() {}
Function Name	Mandatory	Optional (can be anonymous)
Hoisting	Yes – can be called before it's defined	No – only defined when interpreter reaches it
When Defined	During parsing phase	During execution phase
Use in Conditional Blocks	Not ideal or reliable in older JS versions	More flexible
Common Usage	Top-level reusable logic	Callbacks, closures, event handlers

4. Named Function Expression Example

```
const factorial = function fact(n) {
  return n <= 1 ? 1 : n * fact(n - 1);
};
```

- Even though the variable is factorial, the internal function name fact is used for recursion.

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

Answer: In JavaScript, **parameters** and **return values** are fundamental concepts used in functions to pass data in and out. They help functions interact with the outside world and make them reusable and dynamic.

1. Parameters

Parameters are placeholders or variables listed inside the parentheses of a function declaration or expression. They act as inputs that the function can use to perform operations.

Syntax:

```
function functionName(parameter1, parameter2) {
  // Use parameters inside the function
}
```

Example:

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

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

- In this example, name is a parameter.
- When the function is called with "Alice" as an argument, that value is assigned to the parameter.

---

2. Arguments vs. Parameters

- **Parameters** are variables declared in the function definition.
- **Arguments** are the actual values passed to the function when it is invoked.

```
function add(x, y) { // x and y are parameters
  return x + y;
}

add(3, 5); // 3 and 5 are arguments
```

---

3. Default Parameters (ES6+)

JavaScript allows you to assign **default values** to parameters. These values are used if no argument or undefined is passed.

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

greet(); // Output: Welcome, Guest
```

---

4. Rest Parameters (ES6+)

Used to accept an **indefinite number of arguments** as an array.

```
function sumAll(...numbers) {
  return numbers.reduce((total, num) => total + num, 0);
}

sumAll(1, 2, 3, 4); // Output: 10
```

---

5. Return Values

A function can optionally return a value using the return statement. This value can then be stored in a variable or used in an expression.

Syntax:

```
function functionName(parameters) {
  // code
  return value;
}
```

Example:

```
function square(n) {
  return n * n;
}

const result = square(5); // result = 25
```

- Once the return statement is executed, the function terminates, and control is returned to the calling code.
- If no return is used, the function implicitly returns undefined.

---

6. Returning Multiple Values

JavaScript functions can only return **one value**, but that value can be a **collection**, such as an array or an object.

Using Array:

```
function getCoordinates() {
  return [10, 20];
}

const [x, y] = getCoordinates(); // x = 10, y = 20
```

Using Object:

```
function getUser() {
  return { name: "Alice", age: 30 };
}

const user = getUser(); // user.name = "Alice"
```

---

7. Void Functions (No Return)

Some functions perform operations but do not return any value.

```
function logMessage(message) {
  console.log("Log:", message);
}

const result = logMessage("Test");

// result is undefined
```

Arrays

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

Answer: In JavaScript, an **array** is a special type of object used to store **multiple values** in a single variable. Instead of declaring separate variables for each item (e.g., let a = 1; let b = 2;), an array can hold all values in an **ordered list** and allow access via numeric indices.

Arrays are a fundamental part of the language and are widely used in data handling, looping, iteration, and storage of structured or unstructured data.

What Makes Arrays Special in JavaScript?

- JavaScript arrays are **dynamic**, meaning they can grow and shrink in size at runtime.
- Arrays can store **any type of data**: numbers, strings, booleans, objects, functions, or even other arrays.
- Array elements are **indexed**, starting from 0 (zero-based indexing).
- Arrays come with a powerful set of **built-in methods** for manipulation (e.g., push(), pop(), map(), filter()).

Declaring and Initializing Arrays

Arrays in JavaScript can be created in several ways. Below are the most commonly used techniques with detailed explanations.

1. Using Array Literals (Recommended Method)

The array literal syntax is the most used method due to its simplicity and readability.

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

- This creates an array named fruits with 3 elements.
- Each value is stored in the array at a specific index:
  - `fruits[0] returns "Apple"`
  - `fruits[1] returns "Banana"`
  - `fruits[2] returns "Cherry"`

Why it is preferred:

- Cleaner and shorter syntax.
- Easy to read and write.
- No ambiguity in behavior.

2. Using the new Array() Constructor

This method uses the built-in Array constructor to create a new array.

```
let numbers = new Array(1, 2, 3, 4);
```

- This creates an array with 4 numeric elements.
- Accessing numbers[2] returns 3.

However, **this method can behave differently** when passed a single number:

```
let arr = new Array(5);
```

- Instead of creating an array with one element 5, it creates an **empty array with a length of 5 slots**, each uninitialized.
- These slots are **"empty"** (not undefined, but genuinely not set).

Caution: Because of this behavior, this constructor is rarely used unless the size of the array needs to be predefined for performance or structural reasons.

---

3. Declaring an Empty Array and Populating It Later

You can start with an empty array and add elements to it dynamically:

```
let cities = [];
cities.push("Delhi");
cities.push("Mumbai");
cities.push("Chennai");
```

- The push() method adds elements to the **end** of the array.
- cities now contains: ["Delhi", "Mumbai", "Chennai"]

---

4. Arrays with Mixed Data Types

JavaScript arrays can store different types of data in the same array:

```
let mixed = [42, "Hello", true, null, { name: "Alice" }, [1, 2]];
```

- This array contains:
    - A number
    - A string
    - A boolean
    - A null value
    - An object
    - Another array (nested array)

This flexibility is unique to JavaScript and makes arrays very powerful for complex data handling.

Accessing and Modifying Array Elements

You can access any element using its index:

```
let colors = ["Red", "Green", "Blue"];
console.log(colors[0]);  // Output: Red
console.log(colors[2]);  // Output: Blue
```

You can also modify values directly:

```
colors[1] = "Yellow";    // Changes "Green" to "Yellow"
```

Array Length Property

Every array has a length property that returns the number of elements in the array.

```
let nums = [10, 20, 30];
console.log(nums.length); // Output: 3
```

This is useful in loops or when dynamically adding/removing elements.

Multidimensional (Nested) Arrays

Arrays can contain other arrays, allowing you to build **2D or multi-dimensional** structures:

```
let matrix = [
  [1, 2],
  [3, 4],
  [5, 6]
];

console.log(matrix[1][0]); // Output: 3
```

- matrix[1] refers to [3, 4]
- matrix[1][0] refers to 3

Built-in Array Methods (Quick Reference)

Method	Description
push()	Adds an item to the end of the array
pop()	Removes the last item
shift()	Removes the first item
unshift()	Adds an item to the beginning
splice()	Adds/removes elements at a specific index
slice()	Returns a portion of the array
map()	Applies a function to each element
filter()	Filters array based on condition

forEach()	Executes a function for each element

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

Answer: JavaScript provides a set of powerful **array manipulation methods** that allow developers to add or remove elements from the beginning or end of an array. Among these, the most used methods are:

- push() – Add to the end
- pop() – Remove from the end
- unshift() – Add to the beginning
- shift() – Remove from the beginning

These methods help in building dynamic data structures such as stacks and queues and simplify working with ordered lists.

1. push() Method

Purpose:
Adds one or more elements to the **end** of an array and returns the **new length** of the array.

Syntax:

```
array.push(element1, element2, ..., elementN);
```

Example:

```
let colors = ["Red", "Green"];
let newLength = colors.push("Blue", "Yellow");

console.log(colors);     // ["Red", "Green", "Blue", "Yellow"
console.log(newLength);  // 4
```

Use Case:
Useful when you want to **append items** to the array dynamically, such as during list building or data fetching.

---

2. pop() Method

Purpose:
Removes the **last** element from an array and returns that element. The length of the array is reduced by one.

Syntax:

```
array.pop();
```

Example:

```
let fruits = ["Apple", "Banana", "Cherry"];
let removed = fruits.pop();

console.log(fruits);   // ["Apple", "Banana"]
console.log(removed);  // "Cherry"
```

Use Case:
Often used to implement **stack-like** behavior (LIFO – Last In First Out) where elements are removed in reverse order of insertion.

---

3. shift() Method

Purpose:
Removes the **first** element from an array and returns that element. All remaining elements are **shifted one position to the left**, and the array length decreases by one.

Syntax:

```
array.shift();
```

Example:

```
let numbers = [10, 20, 30];
let removed = numbers.shift();

console.log(numbers);  // [20, 30]
console.log(removed);  // 10
```

Use Case:
Commonly used when removing the **first item** in queue-like operations (FIFO – First In First Out).

---

4. unshift() Method

Purpose:
Adds one or more elements to the **beginning** of an array and returns the **new length** of the array. Existing elements are **moved to the right** to make space.

Syntax:

```
array.unshift(element1, element2, ..., elementN);
```

Example:

```
let animals = ["Cat", "Dog"];
let newLength = animals.unshift("Elephant", "Tiger");

console.log(animals);    // ["Elephant", "Tiger", "Cat", "Dog"]
console.log(newLength);  // 4
```

Use Case:
Useful when elements need to be added to the **front of a list**, such as implementing a queue.

Comparison Table

Method	Adds/Removes	Position	Returns

push()	Adds	End	New array length
pop()	Removes	End	Removed element
unshift()	Adds	Beginning	New array length
shift()	Removes	Beginning	Removed element

Objects

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

Answer: In JavaScript, an **object** is a complex and powerful data structure that allows developers to store, access, and manipulate **collections of data** using **key-value pairs**. Objects are one of the most fundamental elements of the language and are used extensively in real-world applications such as data modeling, DOM manipulation, API interactions, and more.

An object can represent almost anything: a person, a car, a book, or even a complex system. It can contain **data (properties)** and **behavior (methods)**.

Syntax for Declaring an Object

1. Object Literal Syntax (most used):

```
let person = {
  name: "Alice",
  age: 25,
  isStudent: true,
  greet: function () {
    return "Hello, my name is " + this.name;
  }
};
```

- name, age, isStudent are **properties** (keys with associated values).
- greet() is a **method** (a function defined inside the object).
- this refers to the object itself.

2. Using the new Object() Constructor:

```
let person = new Object();
person.name = "Alice";
person.age = 25;
person.isStudent = true;
```

While valid, this approach is less concise and less common in modern JavaScript.

Accessing Object Properties

- **Dot notation**:

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

- **Bracket notation**:

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

Bracket notation is especially useful when property names are dynamic or contain special characters.

Modifying Object Properties

- **Update a property**:

  ```
  person.age = 26;
  ```

- **Add a new property**:

  ```
  person.city = "Mumbai";
  ```

- **Delete a property**:

  ```
  delete person.isStudent;
  ```

Objects are **mutable**, meaning their properties can be updated or removed at any time.

Objects Can Contain Any Data Type

JavaScript objects can store:

- Primitive values (strings, numbers, booleans)
- Arrays
- Functions (as methods)
- Other objects (nested objects)

Example of a Nested Object:

```
let employee = {
  name: "Rahul",
  department: {
    name: "IT",
    floor: 3
  }
};
```

Differences Between Arrays and Objects

Feature	Objects	Arrays
Basic Structure	Collection of key-value pairs	Ordered list of elements
Key Type	Keys are **strings** (or symbols)	Keys are **numerical indices** starting from 0
Use Case	Ideal for representing entities with named attributes (e.g., a user profile)	Best suited for **ordered collections** of items (e.g., list of names)
Data Representation	Unordered data	Ordered data

Property Access	Access using property names (object.property or object["property"])	Access using indices (array[index])
Iteration Methods	for...in, Object.keys(), Object.entries(), Object.values()	for, forEach(), map(), filter(), reduce()
Mutability	Mutable – properties can be added, updated, or removed	Mutable – elements can be added or removed using methods like push() etc.
Can Hold Mixed Types?	Yes (values can be of any type including functions and nested objects)	Yes, but typically used for similar types for consistency
Support for Methods	Can have methods (functions as values)	Not designed to store methods
Examples	{name: "Alice", age: 25}	["Alice", "Bob", "Charlie"]

Real-Life Analogy

- **Object**: Like a **dictionary** or **identity card** – each entry (or field) has a label or name (key) and corresponding value.
- **Array**: Like a **shopping list** – items are listed in a specific order and accessed by their position.

Question 2: Explain how to access and update object properties using dot notation and bracket notation.

Answer: In JavaScript, objects store data as **key-value pairs**. Each key is a **property name**, and its corresponding value can be a primitive, array, function, or even another object. To work with this data, JavaScript provides two main syntaxes:

- **Dot Notation**
- **Bracket Notation**

Both can be used to **read (access)** or **change (update)** the properties of an object, but they have different use cases.

---

1. Dot Notation

Dot notation is the **most commonly used** and most readable way to access or update an object's properties. It uses the syntax:

```
objectName.propertyName
```

Accessing Property with Dot Notation

```
let student = {
  name: "Ravi",
  age: 21
};

console.log(student.name);  // Output: Ravi
console.log(student.age);   // Output: 21
```

Updating Property with Dot Notation

```
student.age = 22;
console.log(student.age);   // Output: 22
```

Adding a New Property

```
student.course = "Computer Science";
console.log(student.course);  // Output: Computer Science
```

Note: Dot notation can **only be used** when the property name is a **valid JavaScript identifier** (i.e., it has no spaces, hyphens, or special characters, and does not start with a number).

---

2. Bracket Notation

Bracket notation allows more **flexibility**, especially when the property name is **dynamic** or **not a valid identifier**. It uses the syntax:

```
objectName["propertyName"]
```

Accessing Property with Bracket Notation

```
let employee = {
  "first-name": "Priya",
  department: "Sales"
};

console.log(employee["first-name"]);  // Output: Priya
console.log(employee["department"]);  // Output: Sales
```

Accessing with Dynamic Property Names

```
let prop = "department";
console.log(employee[prop]);   // Output: Sales
```

Updating Property with Bracket Notation

```
employee["department"] = "Marketing";
console.log(employee["department"]);  // Output: Marketing
```

Adding a New Property

```
employee["experience"] = 5;
console.log(employee["experience"]);  // Output: 5
```

Note: Bracket notation must be used when:

- Property name has **spaces** or **special characters**
- Property name is **stored in a variable**
- You want to **compute** the property name dynamically at runtime

---

Key Differences Between Dot and Bracket Notation

Feature	Dot Notation	Bracket Notation
Syntax	object.property	object["property"]
When to Use	When the property name is a valid identifier	When the property name is dynamic or invalid identifier
Support for Special Characters	Not supported	Supported
Variable-based Access	Cannot use variables directly	Can access properties stored in variables
Example (static)	person.name	person["name"]
Example (dynamic)	person.prop won’t work if prop is a variable	person[prop] works if prop = "name"

JavaScript Events

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

Answer: In JavaScript, **events** are actions or occurrences that happen in the system you are programming, which the browser or user triggers. These events allow developers to create interactive web pages by responding to user actions or changes in the environment.

Examples of events include:

- User interactions: clicking a button, typing on the keyboard, moving the mouse, submitting a form
- Browser actions: page loading, resizing, scrolling
- Media events: playing or pausing a video or audio

How Events Work

When an event occurs, the browser generates an **event object** that contains information about that event (e.g., the type of event, the target element where it occurred, and other relevant data). JavaScript programs can respond to these events by executing **event handler functions**.

Event Types

JavaScript supports many event types, some of the most common are:

- **Mouse events:** click, dblclick, mouseover, mouseout, mousemove
- **Keyboard events:** keydown, keyup, keypress
- **Form events:** submit, change, focus, blur
- **Window events:** load, resize, scroll

What is an Event Listener?

An **event listener** (or event handler) is a **function** or **callback** that waits for a specific event to occur on a particular DOM element and executes code in response to that event.

Event listeners are essential because they allow you to **attach behavior to elements dynamically** without modifying the HTML directly.

Role and Importance of Event Listeners

- **Listen and respond:** Event listeners monitor for user interactions or system events.

- **Separation of concerns:** They allow JavaScript to handle logic separately from HTML markup.
- **Reusability:** One event listener can be attached to many elements or triggered multiple times.
- **Dynamic interactivity:** They enable real-time changes to UI based on user actions (e.g., opening a modal on button click).
- **Event propagation control:** They can control event flow through methods like stopPropagation() and preventDefault().

How to Attach Event Listeners

There are multiple ways to attach event listeners in JavaScript:

1. Using the onclick or other inline event properties (not recommended for complex apps):

```
button.onclick = function() {
  alert("Button clicked!");
};
```

2. Using addEventListener() method (recommended approach):

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

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

- addEventListener allows adding multiple listeners to the same element without overwriting existing ones.
- It supports **event capturing** and **bubbling** phases (advanced event flow concepts).

Event Object

When an event listener is triggered, it receives an **event object** parameter that contains details such as:

- event.target — the element that triggered the event
- event.type — the event type (e.g., "click")
- event.preventDefault() — method to stop default browser behavior (like following a link)
- event.stopPropagation() — method to prevent the event from bubbling up to parent elements

Question 2: How does the addEventListener() method work in JavaScript? Provide an example.

Answer: The addEventListener() method is a fundamental part of JavaScript's event handling mechanism. It allows you to **attach one or more event handlers (listeners)** to a specified DOM element for a particular event type, such as clicks, keyboard presses, mouse movements, and more.

How addEventListener() Works

- The method takes at least **two arguments**:
    1. **Event type (string)**: The name of the event to listen for (e.g., "click", "keydown", "submit").
    2. **Callback function**: The function to execute when the event occurs on the target element.
- Optionally, a **third argument** can be provided for advanced options, such as specifying whether to use event capturing or bubbling.
- Multiple event listeners can be attached to the same element for the same event type without overwriting each other, which is a significant advantage over older inline event handlers (like onclick).
- It provides **better control** over event propagation and default behaviors.

Syntax

```
element.addEventListener(eventType, callbackFunction, useCapture);
```

- eventType (string): The event name (e.g., "click", "mouseover", "keyup").
- callbackFunction (function): The function executed when the event occurs.
- useCapture (optional boolean): Defines whether the event is captured during the capturing phase (true) or bubbling phase (false). Default is false (bubbling).

Example

```html
<!DOCTYPE html>
<html>
<head>
  <title>addEventListener Example</title>
</head>
<body>

<button id="myButton">Click Me</button>

<script>
  // Select the button element by its ID
  let button = document.getElementById("myButton");

  // Attach a 'click' event listener to the button
  button.addEventListener("click", function(event) {
    alert("Button was clicked!");

    // Log details from the event object
    console.log("Event type:", event.type);
    console.log("Clicked element:", event.target);
  });
</script>

</body>
</html>
```

Explanation

- We first select the button element using document.getElementById().
- We use addEventListener() to listen for the "click" event.
- When the button is clicked, the callback function executes, showing an alert and logging event information to the console.
- The event object is passed automatically to the callback function, providing useful context about the event.

Advantages of Using addEventListener()

- **Multiple listeners:** Attach more than one listener to the same event and element without overwriting.

- **Separation of concerns:** Keeps HTML clean by separating JavaScript from markup.
- **Advanced options:** Control event propagation using the third parameter or additional options (like once, passive).
- **Cross-browser compatibility:** Supported in all modern browsers.

DOM Manipulation

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

Answer: The **Document Object Model (DOM)** is a programming interface (API) provided by browsers that represents the structure of an HTML or XML document as a **tree of objects**. It allows programming languages, primarily JavaScript, to access, manipulate, and update the content, structure, and styles of a web page dynamically.

Key Characteristics of the DOM:

- The DOM treats the entire web page as a **hierarchical tree** of nodes.
- Each element, attribute, and piece of text in the HTML document becomes a **node** in this tree.
- The root of this tree is the document object, representing the entire HTML document.
- The DOM is language-agnostic but JavaScript is the most commonly used language to interact with it in web browsers.

Structure of the DOM

- **Document Node:** Represents the entire document (document object).
- **Element Nodes:** Represent HTML tags like , <p>, <a>, etc.
- **Text Nodes:** Contain the text inside elements.
- **Attribute Nodes:** Represent attributes like class, id, src, etc.

How Does JavaScript Interact with the DOM?

JavaScript interacts with the DOM to **read**, **modify**, **add**, or **remove** elements and their content, enabling dynamic web pages. This interaction allows web applications to respond to user inputs, update content without refreshing the page, and create complex interfaces.

Common DOM Manipulation Operations in JavaScript:

1. **Accessing Elements**
    - Using methods like:
        - document.getElementById("id")

- document.getElementsByClassName("className")
- document.getElementsByTagName("tagName")
- document.querySelector(selector)
- document.querySelectorAll(selector)

2. **Changing Content**
   - Modify text or HTML inside an element:
     - `heading.textContent = "New Heading Text";`
     - `heading.innerHTML = "<span>Updated Text</span>";`

3. **Changing Attributes and Styles**
   - Update element attributes:
     - `let link = document.querySelector("a");`
     - `link.setAttribute("href", "https://example.com");`
   - Change styles:
     - heading.style.color = "blue";

4. **Adding or Removing Elements**
   - Create new elements:
     - `let newDiv = document.createElement("div");`
     - `newDiv.textContent = "Hello!";`
     - `document.body.appendChild(newDiv);`
   - Remove elements:
     - `let oldElement = document.getElementById("old");`
     - `oldElement.parentNode.removeChild(oldElement);`

5. **Responding to Events**
   - Attach event listeners to elements to respond to user interactions:
     - ```
       button.addEventListener("click", function() {
           alert("Button clicked!");
       });
       ```

Why is the DOM Important?

- It acts as a **bridge** between web pages (HTML) and scripts (JavaScript).
- It allows pages to be **dynamic and interactive** rather than static.
       ```

- Without the DOM, JavaScript would not be able to manipulate web content in real-time.
- It forms the foundation for modern web frameworks and libraries like React, Angular, and Vue.

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

Answer: Selecting elements from the DOM is one of the most fundamental tasks in JavaScript for manipulating web pages. The DOM API provides several methods to access elements based on their identifiers, class names, or CSS selectors. Among the most used methods are:

- getElementById()
- getElementsByClassName()
- querySelector()

Each method has its specific behavior and use cases.

---

1. getElementById()

- **Purpose:** Selects a **single element** based on its unique id attribute.
- **Returns:** A single DOM element object or null if no element with the specified id exists.
- **Usage:** Since id attributes should be unique within a page, this method is the most direct way to get one specific element.

Syntax:

```
let element = document.getElementById("elementId");
```

Example:

```
<div id="header">Welcome to the site</div>

<script>
  let header = document.getElementById("header");
  console.log(header.textContent); // Output: Welcome to the site
</script>
```

Notes:

- This method is **fast and efficient** due to the uniqueness of the id.
- If no matching element is found, it returns null.

---

2. getElementsByClassName()

- **Purpose:** Selects **all elements** that have a specified class name.

- **Returns:** A **live HTMLCollection** of elements (not an array, but can be iterated over).
- **Usage:** Useful when you want to select multiple elements sharing the same class.

Syntax:

```
let elements = document.getElementsByClassName("className");
```

Example:

```
<ul>
  <li class="item">Item 1</li>
  <li class="item">Item 2</li>
  <li class="item">Item 3</li>
</ul>

<script>
  let items = document.getElementsByClassName("item");
  console.log(items.length); // Output: 3
  console.log(items[0].textContent); // Output: Item 1
</script>
```

Notes:

- The returned HTMLCollection is **live**, meaning if elements with that class are added or removed from the DOM, the collection updates automatically.
- It only accepts a **single class name** (no CSS selectors).

---

3. querySelector()

- **Purpose:** Selects the **first element** in the document that matches a **CSS selector**.
- **Returns:** The first matching DOM element or null if no matches are found.
- **Usage:** Very versatile as it supports any valid CSS selector including element types, classes, IDs, attribute selectors, and pseudo-classes.

Syntax:

```
let element = document.querySelector("CSS-selector");
```

Example:

```
<div class="container">
  <p class="text">Paragraph 1</p>
  <p class="text highlight">Paragraph 2</p>
</div>

<script>
  let firstHighlighted = document.querySelector(".highlight");
  console.log(firstHighlighted.textContent); // Output: Paragraph 2
</script>
```

Notes:

- Supports **complex selectors** like #id, .class, tag, [attribute=value], and combinators like div > p.
- Only returns the **first match**. To get all matching elements, use querySelectorAll().

Comparison Table

Method	Return Type	Selects	Selector Type	Returns Multiple Elements?	Notes
getElementById()	Single Element or null	Element with specific id	id (string)	No	Fastest, unique id only
getElementsByClassName()	Live HTMLCollection	All elements with a class	Single class name (string)	Yes	Live collection, only class name
querySelector()	Single Element or null	First element matching CSS selector	Any valid CSS selector	No	Flexible and powerful, returns first match

Summary

- **getElementById()** is ideal for selecting a unique element by its id.
- **getElementsByClassName()** is used to retrieve all elements sharing the same class, returned as a live collection.
- **querySelector()** is the most flexible, allowing any CSS selector to get the first matching element.

JavaScript Timing Events (setTimeout, setInterval)

Question 1: Explain the setTimeout() and setInterval() functions in JavaScript. How are they used for timing events?

Answer: JavaScript provides built-in functions to handle **timing events** — actions that should happen after a delay or repeatedly at fixed intervals. The two primary functions for these purposes are:

- **setTimeout()** — Executes a function once after a specified delay.
- **setInterval()** — Executes a function repeatedly at fixed time intervals.

Both are part of the Web APIs available in browsers and are essential for creating timed behaviors in web applications, such as animations, delays, or periodic updates.

1. setTimeout()

Purpose:

- Schedules a function to be executed **once** after a specified delay (in milliseconds).

Syntax:

```
let timeoutID = setTimeout(functionToExecute, delayInMilliseconds, arg1, arg2, ...);
```

- functionToExecute: The function (or code snippet) to run after the delay.
- delayInMilliseconds: Time delay before execution, in milliseconds (1000 ms = 1 second).
- Additional arguments (optional): Parameters passed to the function.

Example:

```
function greet() {
  console.log("Hello after 3 seconds!");
}


// Call greet after 3000 milliseconds (3 seconds)
setTimeout(greet, 3000);
```

Explanation:
The greet function will be executed once after 3 seconds.

Stopping a Timeout

If needed, you can **cancel** the scheduled timeout using clearTimeout() and the timeout ID returned by setTimeout():

```
let timeoutID = setTimeout(greet, 3000);
clearTimeout(timeoutID);  // cancels the scheduled execution
```

---

2. setInterval()

Purpose:

- Executes a function **repeatedly** at fixed time intervals (in milliseconds) until stopped.

Syntax:

```
let intervalID =
setInterval(functionToExecute,intervalInMilliseconds, arg1, arg2,
...);
```

- functionToExecute: The function to run at every interval.
- intervalInMilliseconds: Time delay between each execution.
- Additional arguments (optional): Parameters passed to the function.

Example:

```
function showTime() {
  console.log("Current time: " + new Date().toLocaleTimeString());
}


// Call showTime every 2 seconds
let intervalID = setInterval(showTime, 2000);
```

Explanation:

The showTime function will log the current time to the console every 2 seconds indefinitely.

Stopping an Interval

To stop the repeated execution, use clearInterval() with the interval ID:

```
clearInterval(intervalID);  // stops the repeated execution
```

Key Differences Between setTimeout() and setInterval()

Feature	setTimeout()	setInterval()
Execution	Executes **once** after the delay	Executes **repeatedly** at intervals
Use Case	Delayed action, one-time event	Periodic tasks, repeated updates
Returns	Timeout ID (for cancelling)	Interval ID (for cancelling)
Cancel Method	clearTimeout(timeoutID)	clearInterval(intervalID)

Advanced Usage

- **Anonymous functions:**
  You can use anonymous or arrow functions directly:

```
setTimeout(() => {
  console.log("Executed after delay");
}, 1500);
```

- **Passing parameters to callback:**

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

setTimeout(greet, 2000, "Alice"); //Outputs: Hello, Alice
after 2 seconds
```

- **Avoiding overlaps in setInterval():** If the callback takes longer than the interval time, executions can overlap. Using recursive setTimeout() calls is often recommended for precise timing control.

Use Cases for Timing Events

- **setTimeout():**
  - Delaying animations or effects.
  - Showing notifications after some time.
  - Deferring execution to wait for other processes.
- **setInterval():**
  - Updating clocks or timers.
  - Polling data from a server periodically.
  - Running repeated animations or auto-sliders.

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

Answer: Here is a clear example demonstrating how to use setTimeout() to delay an action by 2 seconds:

```
// Function to be executed after the delay
function delayedAction() {
  console.log("This message appears after a 2-second delay.");
}


// Schedule the delayedAction function to run after 2000
milliseconds (2 seconds)

setTimeout(delayedAction, 2000);
```

Explanation:

- The function delayedAction contains the code you want to run after the delay.

- setTimeout(delayedAction, 2000) schedules this function to execute once after 2000 milliseconds (which equals 2 seconds).
- After 2 seconds, the message "This message appears after a 2-second delay." will be logged to the console.

JavaScript Error Handling

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

Answer: In programming, **errors** (also called exceptions) can occur during code execution due to various reasons such as invalid input, unavailable resources, or logic errors. If not handled properly, these errors can cause the program to stop or behave unexpectedly.

Error handling in JavaScript is a mechanism that allows developers to **detect, manage, and respond to runtime errors gracefully** without crashing the entire application. This improves the reliability and user experience of the software.

The try...catch...finally Statement

JavaScript provides a structured way to handle errors using the **try...catch...finally** construct.

1. try Block

- Contains the **code that may throw an error** (exception).
- JavaScript will attempt to execute this code.
- If an error occurs inside the try block, execution immediately stops inside try and control is passed to the catch block.

---

2. catch Block

- This block **handles the error** thrown in the try block.
- It receives the error object as a parameter, which provides information about what went wrong.
- Allows you to define fallback or recovery behavior, such as displaying an error message or logging details.

---

3. finally Block

- The finally block contains code that **always executes**, regardless of whether an error was thrown or caught.
- Useful for cleanup activities like closing files, releasing resources, or resetting states.
- This block is optional.

Syntax

```
try {
  // Code that may throw an error
} catch (error) {
  // Code to handle the error
} finally {
  // Code that runs regardless of error occurrence
}
```

Example

```
function divide(a, b) {
  try {
    if (b === 0) {
      // Manually throw an error if divisor is zero
      throw new Error("Division by zero is not allowed.");
    }

    let result = a / b;
    console.log("Result:", result);

  } catch (error) {
    // Handle the error here
    console.error("An error occurred:", error.message);
  } finally {
    // This block executes regardless of error or no error
    console.log("Division operation completed.");
  }
}

// Test cases
divide(10, 2);  // Valid division
divide(5, 0);   // Error: division by zero
```

Explanation:

- The try block attempts to perform the division.
- When dividing by zero (b === 0), an error is **explicitly thrown** using throw new Error().
- The catch block catches the error and logs a meaningful message to the console.
- The finally block runs after either try or catch blocks, printing a message that the operation is complete.

Benefits of Error Handling

- Prevents the program from crashing unexpectedly.
- Helps identify where and why errors occur.
- Allows graceful recovery or alternative flows.
- Improves debugging and user experience.

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

Answer: Error handling is a crucial aspect of software development, including JavaScript applications. It refers to the process of anticipating, detecting, and managing errors or exceptions that may occur during the execution of a program.

Proper error handling ensures that a JavaScript application behaves predictably, even when unexpected situations arise, and helps maintain a smooth user experience.

Reasons Why Error Handling is Important

1. Prevents Application Crashes

- Without error handling, runtime errors can cause the entire JavaScript application or webpage to stop working abruptly.
- Properly handling errors allows the program to **continue running gracefully**, avoiding crashes that disrupt user activities.

2. Improves User Experience

- Errors can confuse or frustrate users if they are left unhandled or cause the application to freeze.
- Error handling enables the application to display **meaningful messages or fallback content**, guiding users on what went wrong or how to proceed.

3. Helps in Debugging and Maintenance

- When errors are caught and logged, developers get valuable information about what caused the issue.
- This information speeds up **troubleshooting and fixing bugs**, making the application more reliable over time.

4. Supports Robust and Secure Code

- Proper error handling can prevent sensitive information from being exposed to users accidentally.
- It also helps protect the application from **unexpected input or malicious actions** that could otherwise cause failures or security vulnerabilities.

5. Enables Controlled Recovery

- Some errors can be anticipated and handled by executing alternative logic or retrying operations.
- This controlled recovery can **keep the application functional** even in adverse conditions (e.g., network failures, invalid user input).

6. Ensures Proper Resource Management

- The finally block or equivalent cleanup code ensures resources like files, network connections, or memory are properly released even when errors occur.
- This prevents **resource leaks and performance degradation**.

Example Scenario

Imagine a web application that fetches user data from a server:

- If the server is unreachable, the fetch operation might fail.
- Without error handling, the entire application could break or display nothing.
- With proper error handling, the app can show a user-friendly message like “Unable to load data, please try again later” and still keep other parts of the app responsive.