MODULE 8) JAVASCRIPT

JavaScript Introduction

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

Ans. JavaScript is a high-level, interpreted programming (or scripting) language that runs in web browsers. Role of JavaScript in Web Development

1. Adds Interactivity & Dynamic Content • JavaScript enables live updates and interactions—example features include interactive buttons, form validation, dynamic image galleries, animated effects, and more. • It connects with Web APIs like the Canvas API, Audio/Video APIs, and Geolocation to do advanced multimedia and graphics work.

2. Manipulates the DOM • JavaScript interacts with the Document Object Model (DOM) to modify page structure, content, and styles on the fly, giving users a responsive experience without page reloads.

3. Enables Asynchronous Behavior (AJAX, Fetch, etc.)

Techniques like AJAX (Asynchronous JavaScript and XML) let pages fetch data from the server behind the scenes—updating content dynamically without a full page reload.

• JavaScript supports asynchronous programming (callbacks, promises, async/await), helping web apps stay smooth and non-blocking.

4. Accesses Browser APIs

• Through JavaScript, developers can tap into browser capabilities such as history manipulation, device hardware (camera, microphone), and geolocation to build rich features.

5. Extensive Ecosystem: Libraries & Frameworks

• JavaScript offers a rich ecosystem with libraries and frameworks like jQuery, React, Angular, Vue.js, and many more. These tools simplify development, handle UI efficiently, and are widely adopted in modern.

6. Goes Beyond the Browser: Server-Side JavaScript

• Thanks to environments like Node.js, JavaScript isn't limited to the browser—it can also run on servers, enabling full-stack development using a single language.

• Node.js offers modules for networking, file I/O, and more, facilitating scalable, event-driven back-end systems.

7. Universal Adoption & Continuous Evolution

• JavaScript is virtually ubiquitous: 99% of websites use it on the client side, and over 80% integrate third-party libraries/frameworks.

Despite early quirks and criticisms, such as odd type conversions or naming mishaps, the language has matured significantly. Modern JS runtimes like the V8 engine (used in Chrome and Node.js) offer high-performance execution.

• Its development is officially guided by TC39 under Ecma International—ensuring it's evolving with transparency and community input.

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

Ans. JavaScript vs. Java

• Typing & Execution

o Java is statically typed: variable types are declared at compile time and enforced, leading to early detection of type errors. It’s compiled into bytecode and executed on the Java Virtual Machine (JVM).

o JavaScript is dynamically and loosely typed: variable types are determined at runtime, with frequent implicit type coercions. It’s generally interpreted (by browsers or

JS engines), though modern engines often use JIT compitions.

• Object Model

o Java uses class-based, classical inheritance. Every piece of code must reside within a class.

JavaScript relies on prototype-based inheritance, where objects can inherit directly from other objects and share behavior dynamic.

• Concurrency o Java supports multi-threading through threads.

o JavaScript is inherently single-threaded, relying on an event-driven, asynchronous model for concurrency (e.g., callbacks, promises, async/await.

• Typical Use Cases

o Java is often used for large-scale enterprise apps, Android development, and backend systems. It is designed for robustness and long-term maintainability.

o JavaScript was created for interactive web pages and runs on both client (browser) and server (Node.js) . JavaScript vs. Python

• Typing & Syntax o Both languages are dynamically typed and executed as scripts at runtime.

o Python uses clean, indentation-based syntax. o JavaScript uses curly braces {} with semicolons to structure code.

Ecosystem & Domains o Python excels in domains like data science, machine learning, scientific computing, and automation. It has an extensive standard library and rich third-party packages (e.g., NumPy, Pandas).

o JavaScript dominates web development and has matured into full-stack use via robust frameworks and infrastructures (React, Angular, Node.js.

• Inheritance Model

o Python uses a class-based model for inheritance.

o JavaScript operates with prototype-based inheritance, which affords flexible object sharing.

• Language Strengths & Tooling

o Python shines in readability, clarity, and ease of learning, making it great for educational use and quick prototyping.

o JavaScript's flexible typing and event-driven nature can both accelerate development and introduce subtle runtime issues.

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**

1. Embedding JavaScript Directly in HTML (Inline Scripts)

You can write JavaScript code inside your HTML like this:

```
<script>
  alert("Hello from inline JavaScript!");
</script>
```

But this approach mixes markup with logic and is best reserved for very small scripts or quick testing.

2. Linking an External JavaScript File

To separate code from markup and improve maintainability, you use the src attribute:

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

Variables and Data Type

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

Ans. A variable in JavaScript is like a named container or box that holds a piece of data—such as a number, string, object, or other values. You can store, retrieve, and manipulate data using variables.
1.var

• Scope: var is function-scoped (or global if declared outside a function) and ignores blocks like {}.

• Hoisting: The declaration is hoisted to the top of its scope and initialized with undefined, even if declared later in the code.

• Re-declaration & Re-assignment: You can redeclare (var x; var x;) and reassign variables freely within the same scope.

2. let

• Scope: Block-scoped—only accessible within the enclosing {} block.

• Hoisting: Hoisted, but not initialized. Accessing the variable before its declaration results in a Reference Error (known as the Temporal Dead Zone).

• Re-declaration & Re-assignment: You can reassign (change the value) but you cannot redeclare a variable in the same scope. 3. const

• Scope: Also block-scoped.

• Hoisting: Behaves like let in being uninitialized initially— access before declaration leads to a Reference Error.

• Re-declaration & Re-assignment: Cannot be redeclared or reassigned. Once initialized, the variable binding is constant. However, if the value is an object or an array, its contents can still be mutated.

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

Ans. 1. Primitive Data Types These are immutable values (can't be changed once created), and they represent single, simple data pieces. According to MDN, JavaScript defines seven primitive types: string, number, bigint, boolean, undefined, symbol, and null.

• String Represents text data. Example:

• let greeting = "Hello, world!"; Strings are immutable, but JavaScript allows you to call methods like .length or .toUpperCase() via a temporary wrapper object.

• Number For all numerical values—integers, floating points, special values like Infinity, -Infinity, or NaN:

• let score = 42;

• let price = 49.99;

• let result = NaN; JavaScript follows the IEEE-754 standard and has limits on safe integers.

• BigInt Used for integers larger than the safe limit of Number:

• let huge = 1234567890123456789012345678 90n;

• let same = BigInt("9007199254740992"); BigInt doesn’t mix with Number in arithmetic directly.

• Boolean Two possible values: true and false:

• let isActive = true;

• let hasLicense = false; Helpful in conditionals and logical operations.

• Undefined Indicates a variable that hasn’t been assigned a value yet:

• let x;

• console.log(x); // undefined Occurs when accessing nonexistent properties or missing function returns.

• Null Represents an explicit “no value” or “empty object”:

• let user = null;

Despite being its own type, typeof null returns "object" — a quirk in JavaScript.

• Symbol A unique and immutable identifier, often used as object property keys:

• let id1 = Symbol("id");

• let id2 = Symbol("id");

• console.log(id1 === id2); // false Symbols are perfect for creating non-colliding keys.

2. Object (Non-Primitive) Types Anything that isn't a primitive is an object—mutable and capable of storing many values or behaviors. MDN lists the following built-in object categories: arrays, functions, dates, maps, sets, typed arrays, promises, and more. Example object types:

• Object (plain object)

• let person = { name: "Alice", age: 28 };

• Array

• let fruits = ["apple", "banana", "cherry"]; Arrays are special objects with indexed elements and useful methods like .push(), .length, etc.

• Date Represent dates and times:

• let today = new Date();

• Map / Set Collections with unique capabilities:

• let map = new Map([["key", "value"]]);

• let set = new Set([1, 2, 3]);

• Typed Arrays (e.g., Uint8Array) Used to handle binary data efficiently:

• let buffer = new Array Buffer(8);

• let view = new Uint8Array(buffer); Particularly useful in contexts like WebGL, audio/video processing, or networking.

• Function Functions themselves are objects, and they're callable.

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

Ans. undefined

• Meaning & Origin: Represents a variable that has been declared but not assigned a value. In other cases—like accessing a non-existent object property, or when a function doesn't explicitly return anything—JavaScript will also implicitly return undefined.

Type: A primitive type. When you use type of, it reports "undefined".

• Default Behavior: Automatically used by JavaScript during runtime. It's essentially the absence of a value. null

• Meaning & Origin: Indicates the intentional absence of any object value. It's typically assigned by developers to show that a variable should explicitly hold "no value."

• Type Quirk: Although null is a primitive, typeof null returns "object"—an idiosyncrasy in JavaScript maintained for backward compatibility.

JavaScript Operator

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

Ans. Types of JavaScript Operators JavaScript offers a variety of operators to manipulate values and control logic flow. Here's a breakdown of the primary categories:

1. Arithmetic Operators Used for mathematical operations on numbers:
   • + (addition or string concatenation)
   • - (subtraction)
   • * (multiplication)
   • / (division)
   • % (modulus/remainder)
   • ** (exponentiation)
   • ++ (increment), -- (decrement)
   Example:
   let a = 5 + 3;
   // 8 let b = 10 % 3;
   // 1 let c = 2 ** 3;
   // 8 let d = ++a;
   // a = 9, so d = 9
2. Assignment Operators Assign values to variables, with compound forms for inline operations:
   • = • +=, -=, *=, /=, %=
   • **= (exponentiation assignment)
   Example:
   let x = 10;
   x += 5; // x = 15
   x *= 2; // x = 30
3. Comparison Operators Compare values and return true or false:

• ==, != (loose equality)

• ===, !== (strict equality/type + value)

• >, =, <= Example: 5 == '5'; // true (type coercion) 5 === '5'; // false (different types) 10 > 5; // true

4. Logical Operators Use boolean logic to combine or invert expressions:

• && (AND)

• || (OR)

• ! (NOT)

Example:

true && false;

// false false || true;

// true !true;

// false

5. Bitwise Operators Operate on binary representations of integers:
   • &, |, ^, ~, <>, >>>
   Example: 5 & 1;
   // 1 (binary AND)
6. String Operators The + operator also serves to concatenate strings; += appends.
   Example: let str = "Hello" + " " + "World!";
   // "Hello World!" str += " JS";
   // "Hello World! JS"
7. Ternary (Conditional) Operator A concise inline conditional expression: condition ? exprIfTrue : exprIfFalse
   Example: let status = (age >= 18) ? "Adult" : "Minor";

8. Type Operators & Miscellaneous Includes operators for type checking, membership, deletion, etc.:
   • typeof, instanceof
   • delete, void, in
   • , (comma operator), spread ..., logical assignment (&&=, ||=, ??=)

   Examples:

   Type of 42; // "n

   [] instance of Array; // true

   let obj = {a: 1};

   delete obj .a;

   // removes property a const val = undefined ?? 'default';
9. Unary Operators Operate on a single operand:

• !, +, -, ~, ++, -- (prefix or postfix)

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

Ans. Overview: == vs. === == (Loose Equality)

• Also known as the abstract equality operator, == performs type coercion—meaning JavaScript may convert one or both operands to a common type before comparison.

• This can lead to unexpected matches: o '5' == 5 → true (string is coerced to number) o null == undefined → true (special case in JavaScript) === (Strict Equality)

• The strict equality operator checks both value and type, with no type conversion. If types differ, it returns false.

• Examples:

o '5' === 5 → false (string vs number)

o null === undefined → false (different types)

Control Flow (If-Else, Switch)

Question 1: What is control flow in JavaScript? Explain how ifelse statements work with an example?

Ans. Control flow refers to the order in which JavaScript executes statements in your program. if...else is one of the key tools to control this flow. It allows your script to branch based on conditions—executing one block of code if a condition is true, or another if it's false.

Syntax Basics:

```
if (condition)
{ // Executes when condition is truthy
}
Else {
// Executes when condition is falsy
}
```

A more advanced version using multiple conditions:

```
if (condition1)
```

```
{ // when condition1 is true
}else if (condition2) {
 // when condition1 is false but condition2 is true
 } else {
 // when none of the above are true
 }
```

This construct allows for branching logic—only the first true condition’s block will run.

Example of if...else if...else Here’s a practical illustration using temperature values:

```
const temp = 25;
 if (temp > 30) {
 console.log("It's hot.");
 } else if (temp >= 20) {
 console.log("It's warm.");
} else {
 console.log("It's cold.");
 } // Output: "It's warm."
```

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

Ans. A switch statement lets you compare one expression against multiple possible values (cases), running the matching block and stopping at a break—or continuing if you intentionally omit break for fall-through. It's often cleaner than chaining many if...else if...else checks.

• The expression is evaluated once, then compared (using strict equality, i.e., ===) against each case.

• If a matching case is found, execution begins there and continues until a break is reached or the switch ends.

• If none match, the default block runs (if provided).

Example: const fruit = "Papayas";

```
switch (fruit) {
case "Oranges":
console.log("Oranges are $0.59 a pound.");
break;
case "Mangoes":
case "Papayas":
console.log("Mangoes and papayas are $2.79 a pound.");
break;
default:
console.log(`Sorry, we are out of ${fruit}.`);
}
```

This will output: "Mangoes and papayas are $2.79 a pound."

When to Use switch vs if...else

Here's when a switch statement might be a better choice over if...else and vice versa: Use switch When:

• Checking a single value against many potential exact matches (like multiple strings or numbers).

• Multiple related cases share the same behavior, enabling an elegant fall-through.

• You want your code to be easier to read and maintain— especially for long sets of value comparisons.

• Performance matters in large branch sets: JS engines may optimize switch with jump tables, potentially making it faster than sequential if...else chains. Use if...else When:

• You need to evaluate complex or range-based conditions, not just value equality. For instance, conditions like x > 10 && y < 5 can't be handled by switch.

• There are only a few conditions—if...else is simpler and more concise for just two or three branches. Perspectives from the Community:

Some developers advise minimizing both long switch and if...else chains, suggesting that using lookup tables or objects can often lead to cleaner and more maintainable solution.

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.

Ans. In JavaScript, loops are used to execute a block of code repeatedly until a specified condition is met. There are several types of loops, with the most common being for, while, and do...while loops. Each type has its own use cases and syntax.

1. For Loop The for loop is used when you know in advance how many times you want to execute a statement or a block of statements. It consists of three parts: initialization, condition, and increment/decrement. Syntax for (initialization; condition; increment/decrement)
   { // code to be executed }
2. While Loop The while loop is used when you want to execute a block of code as long as a specified condition is true. The condition is evaluated before the execution of
   // code to be executed }
3. Do-While Loop The do...while loop is similar to the while loop, but it guarantees that the block of code will be executed at least once, as the condition is evaluated after the execution of the loop's body.
   Syntax do {
   // code to be executed } while (condition);
   } while (condition);

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

Ans. while loop (entry-controlled):

• Evaluates the condition before executing the loop body.

• If the condition is false initially, the loop may not execute at all.

• It can run zero or more times.

```
let i = 0;
 while (i > 0) {
 console.log("This will NOT run");
 }
```

do...while loop (exit-controlled):

• Executes the loop body first, then checks the condition.

• Guarantees that the code inside the loop runs at least once, regardless of the condition.

```
 let i = 0;
do { console.log("This WILL run once");
 } while (i > 0);
```

Functions

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

Ans. Functions in JavaScript are reusable blocks of code that perform a specific task. They allow you to encapsulate logic, making

your code more organized, modular, and easier to maintain. Functions can take inputs (known as parameters), perform operations, and return outputs (known as return values). Syntax for Declaring a Function There are several ways to declare a function in JavaScript, but the most common methods are:

1. Function Declaration

2. Function Expression

3. Arrow Function

1. Function Declaration A function declaration defines a named function that can be called later in the code.

Syntax: function functionName (parameters) {

// code to be executed return value;

// optional

2. Function Expression A function expression defines a function that can be assigned to a variable. This function can be anonymous (without a name) or named.

Syntax: const functionName = function(parameters)

{ // code to be executed return value;

// optional };

3.Arrow Function Arrow functions provide a more concise syntax for writing function expressions. They are particularly useful for writing short functions.

Syntax: const functionName = (parameters) => {

// code to be executed return value; // optional };

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

Ans. Function Declaration

• Involves using the function keyword followed by a name and a body.

• Fully hoisted: Both the function's name and the body are hoisted to the top of their scope, allowing the function to be called before its definition in the code. Example:

• console.log(sum(2, 3)); // 5, works because of hoisting

• function sum(a, b) {

• return a + b;

• }

• Standalone constructs—not assigned to a variable—and can't be conditionally declared inside blocks without special syntax considerations.

• Offers better readability, especially for named and recursive functions. Function Expression

• Involves defining a function (possibly anonymous) and assigning it to a variable. The definition is treated as an expression.

• Not hoisted in the same way: While the variable name is hoisted (if declared with var, as undefined), the function itself isn't initialized

until that line executes. Calling it before will result in a runtime error. Example:

• console.log(square(2)); // ReferenceError or TypeError

•

• const square = function(x) {

• return x * x;

• };

• Can be anonymous, and is often used for callbacks, immediately invoked function expressions (IIFEs), or dynamic assignment.

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

• Ans. Parameters are named placeholders listed in a function's definition; they represent inputs that the function can accept.

• function greet(poonam) {

• return `Hello, ${poonam}!`;

• }

• JavaScript doesn't enforce parameter types or arity:

o If a function is called with fewer arguments than declared parameters, the missing ones default to undefined.

o Using default parameter values (introduced in ES6) remedies this: o function add(x, y = 10) {

o return x + y; o }

o console.log(add(5)); // 15

• The rest parameter syntax ...params allows gathering all extra arguments into an array:

• function sum(...nums) {

• return nums.reduce((a, b) => a + b, 0);

• }

• console.log(sum(1, 2, 3, 4)); // 10

• JavaScript also provides the arguments object (array-like) within traditional functions to access all passed arguments—even those beyond declared parameters.

• Passing behavior:

o Primitives (like numbers, strings) are passed by value—modifying the parameter inside the function does not affect the original variable.

Return values in Functions

• A function's return value is the value it produces when it completes execution. You define it using the return statement.

• If no return statement is used, the function implicitly returns undefined

• The value returned by a function can be assigned, used in expressions, or passed to other functions:

• function random(max) {

• return Math.floor(Math.random() * max);

• }

• const x = random(100);

• console.log(x); // e.g., 57

• Functions can only directly return one value, but you can bundle multiple values into an object or array:

• function areaPerimeter(w, h) {

• return { area: w * h, perimeter: 2 * (w + h) };

• }

• const { area, perimeter } = areaPerimeter(5, 3);

• console.log(area, perimeter); // 15, 16 Developer Perspective A Reddit user puts it simply: “Think of a function as a black box. The

black box can accept input → the parameters. It also can produce output → the return value.”

This analogy captures the essence: parameters feed data into the function; the return value is what comes out of it.

Arrays

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

Ans.An array in JavaScript is a special type of object that stores an ordered collection of values—called elements—under a single

variable name. These values can be of any type, including numbers, strings, objects, or even other arrays. Elements are zero-indexed, meaning the first element is at index 0, then 1, and so on. JavaScript arrays are also dynamic (they can grow or shrink) and heterogeneous (can mix different data types in one array).

Arrays are optimized for operations like iteration, indexing, and various data transformations via built-in methods like .push(), .pop(), .length, .forEach(), and many others.

How to Declare and Initialize an Array

1. Array Literal (Recommended)
   The most common and readable approach is using array literals with square brackets []:
   const colors = ["red", "green", "blue"];
   This creates an array colors containing three string elements. You can also start with an empty array and add elements later: const numbers = [];
   numbers[0] = 10;
   numbers.push(20);
   This method is concise, clear, and widely preferred.
2. Array Constructor (Less Common) You can also use the Array() constructor syntax:
   const arr = new Array("apple", "banana", "cherry");
   However, be cautious—passing a single number to the constructor creates a sparse array with that length, not filled elements:
   const arr = new Array(3);
   console.log(arr.length); // 3

console.log(arr[0]);

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

Ans. 1. push()

• Purpose: Adds one or more elements to the end of an array.

• Returns: The new length of the array after insertion. Example:

const colors = ["red", "green"]; const newLength = colors.push("blue", "yellow");

// colors is now ["red", "green", "blue", "yellow"]

// newLength is 4

2. pop()

• Purpose: Removes the last element from an array.

• Returns: The element that was removed. If the array is empty, returns undefined. Example:

const fruits = ["apple", "banana", "cherry"];

const last = fruits.pop();

// last is "cherry"

// fruits is now ["apple", "banana"

3. shift()

• Purpose: Removes the first element from an array and shifts all subsequent elements down by one index.

• Returns: The element that was removed. If the array is empty, returns undefined.

Example:

const queue = ["first", "second", "third"]; const first = queue.shift();

// first is "first"

// queue is now ["second", "third"]

4. unshift()

• Purpose: Adds one or more elements to the beginning of an array and shifts existing elements rightward.

• Returns: The new length of the array.

Example:

```
const letters = ["b", "c", "d"];
const newLength = letters.unshift("a");
// letters is now ["a", "b", "c", "d"]
// newLength is 4
```

Performance Considerations

• push() and pop() are highly efficient, with a time complexity of O(1)—adding or removing at the end doesn't shift other elements.

• shift() and unshift() are less efficient, with potential time complexity of O(n) because they involve re-indexing all remaining elements.

Objects

Question 1: What is an object in JavaScript? How are objects different from arrays? Ans. What Is an Object in JavaScript? A JavaScript object is a data structure used to represent a collection of key-value pairs, where keys (also known as property names) are typically strings (or symbols) and each key is mapped to a corresponding value. Objects are mutable entities that can hold diverse types—including primitives, functions, arrays, or even nested objects—and allow you to dynamically add, modify,

or remove properties at runtime. This behavior is rooted in JavaScript's prototypal inheritance model.

You commonly create objects using object literals:

const person = { name: "Alice", age: 30, greet() {
console.log(` Hello, I'm ${this.name}`); }
[p}; Or via constructors and prototypes—especially in more complex or reusable patterns.

How Are Objects Different from Arrays?

Though arrays in JavaScript are technically a specialized type of objects, they behave quite differently.

Here's how they compare:

1. Access Patterns & Key Types
   - Objects use named keys, so you access data by object.key or object["key"].
   - Arrays use numeric, zero-based indexing: array[0], array[1], etc.
2. Order Semantics
   Arrays maintain a specific order—it's important to preserve the sequence of elements.
   - Objects are inherently unordered; although modern JavaScript engines preserve insertion order in most cases, objects are not designed for ordered data traversal. 3. Use Cases
   - Use arrays when you need an ordered list or collection you can index through or iterate over sequentially.
   - Use objects when you want to model entities with distinct, named properties—e.g., a user object with properties like id, name, and email.

3. Iteration Patterns
   • Arrays are typically iterated with for, for...of, or .forEach()—which operate based on element order.
   • Objects use for...in, Object.keys(), or Object.entries() to iterate through properties.
4. Technical Classification
   • Arrays are a subtype of objects—i.e., typeof [] returns "object" in JavaScript. But arrays have special behavior, like optimized indexing and built-in methods (push(), pop(), etc.), while objects serve general-purpose mapping.

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

Ans Dot Notation

• Syntax: object.propertyName
• Use when the property name is a valid JavaScript identifier (no spaces, special characters, not starting with a number).
• Clean, concise, and easy to read.
• const person = { name: "Alice", age: 30 };
• console.log(person.name); // Alice
• person.age = 31; // Updates age Bracket Notation
• Syntax: object[expression]
• Use when:
o The property name is dynamic, stored in a variable, or composed via an expression.
o The property name contains invalid identifier characters, like spaces or hyphens.
o You access numeric or unusual property names.

• const key = "name";
• console.log(person[key]); // Evaluates to person["name"] → "Alice".
• person["favorite color"] = "blue"; // Works even with spaces in key.

JavaScript Events

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

Ans. In JavaScript, events are actions or occurrences that happen in the browser, which can be triggered by user interactions or by the browser itself. Events can include actions such as clicking a button, submitting a form, moving the mouse, pressing a key, resizing the window, and many others. JavaScript allows developers to respond to these events and create interactive web applications. Types of Events Some common types of events include:

• Mouse Events: click, dblclick, mouseover, mouseout, mousemove, etc. • Keyboard Events: keydown, keyup, keypress.

• Form Events: submit, change, focus, blur.

• Window Events: load, resize, scroll, unload.

• Touch Events: touchstart, touchmove, touchend (for mobile devices).

Event Listeners

An event listener is a function that waits for a specific event to occur on a particular element. When the event occurs, the event listener executes a callback function, allowing you to define the behavior that should happen in response to the event. How to Use Event Listeners

1. Selecting an Element: First, you need to select the HTML element you want to attach the event listener to.

2. Adding an Event Listener: Use the addEventListener() method to attach the event listener to the selected element. This method takes two arguments: the event type and the callback function to execute when the event occurs.

   Syntax: element.addEventListener(eventType, callbackFunction); Benefits of Using Event Listeners
   • Separation of Concerns: Event listeners allow you to separate your
   JavaScript code from your HTML, making your code cleaner and easier to
   maintain.
   • Multiple Listeners: You can attach multiple event listeners to the same
   element for different events or the same event type.
   • Dynamic Behavior: Event listeners enable you to create dynamic and
   interactive web applications by responding to user actions in real-time.
   Removing Event Listeners

If needed, you can also remove an event listener using the removeEventListener() method. This requires that you reference the same
function that was used when adding the listener.

Question 2: How does the addEventListener() method work in JavaScript? Provide an example.
Ans. The addEventListener() method in JavaScript is used to attach an event handler (also known as an event listener) to a specified element. This method allows you to listen for specific events (like clicks, key presses, mouse movements, etc.) and execute a callback function when that event occurs.
Syntax The syntax for the addEventListener() method is as follows: element.addEventListener(eventType, callbackFunction, useCapture);
• element: The DOM element to which you want to attach the event l
• eventType: A string representing the type of event to listen for (e.g., "click", "keydown", "mouseover").
• callbackFunction: The function that will be called when the event occurs. This function can be defined inline or as a separate function.
• useCapture (optional): A boolean value that indicates whether to use event bubbling or capturing. The default is false, which means the event will be handled in the bubbling phase. Benefits of Using addEventListener()
• Multiple Listeners: You can attach multiple event listeners to the same element for different events or the same event type.

• Separation of Concerns: It helps keep your JavaScript code separate from your HTML, making it easier to maintain.
• Dynamic Behavior: It allows for more dynamic and interactive web applications by responding to user actions in real-time.

DOM Manipulation

Question 1: What is the DOM (Document Object Model) in JavaScript? How does JavaScript interact with the DOM?
Ans. The Document Object Model (DOM) is a programming interface that represents the structure of an HTML or XML document as a tree of objects. Each element, attribute, and piece of text in the document is a node in this free.
How JavaScript Interacts with the DOM:
1. Selecting Elements: Use methods like document.getElementById(), document.querySelector(), etc., to select DOM elements.
2. Modifying Content: Change the content of elements using properties like innerHTML or textContent.
3. Changing Styles: Modify CSS styles using the style property.
4. Creating and Removing Elements: Create new elements with document.createElement() and append them using appendChild(). Remove elements with removeChild() or remove().

5. Event Handling: Respond to user interactions by adding event listeners with addEventListener(). In summary, JavaScript allows dynamic manipulation of the DOM, enabling interactive and responsive web applications.

Question 2: Explain the methods getElementById(), getElementsByClassName(), and querySelector() used to select elements from the DOM. Here's a brief overview of the methods used to select elements from the DOM:

Ans. **1. getElementById()**

- Retrieves a **single element** with a specific id, e.g., document.getElementById("myId").
- Returns the exact element object if found; otherwise, returns null.
- Designed for efficiency—operates in **O(1)** time since IDs are unique and accessed directly.
- Can only be called on document (not on other element nodes).
- Note: IDs must be unique—if duplicates exist, behavior may be unpredictable.

Example:

```
const header = document.getElementById("header");
if (header) {
  header.style.color = "blue";
```

2. getElements By ClassName()

- Returns an **HTML Collection** of all elements matching the specified class or classes, e.g., document.getElements ByClass Name ("btn primary").

- This collection is **live**—it updates automatically as matching elements are added or removed from the DOM.
- It also operates in **O(1)** time as more elements are added.
- The returned collection is array-like but doesn't support methods like .forEach() directly. Use Array.from() or spread syntax ([...collection]) for array methods.

Example:

```
const items = document.getElementsByClassName("list-item");
for (let i = 0; i < items.length; i++) {
  items[i].style.fontWeight = "bold";
```

3. query Selector()

- Takes any valid **CSS selector** and returns the **first matching element**, e.g., document.querySelector(".nav a.active".
- Offers much greater flexibility than older methods since you can combine selectors, use attribute matches, etc..
- Operates in **O(n)** time (scans elements sequentially).
- Returns a **single element** or null if nothing matches.

JavaScript Timing Events (setTimeout, setInterval)

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

Ans. **1. setTimeout()**

- Schedules execution of a function **once**, after a specified delay in milliseconds.
- Syntax:

let timerId = setTimeout(callbackFunction, delay, arg1, arg2, ...);

Example:

setTimeout(() => console.log("Hello after 2 seconds"), 2000);

2. setInterval()

- Schedules a function to run **repeatedly**, at specified intervals.
- Syntax:
- let intervalId = setInterval(callbackFunction, interval, arg1, arg2, ...);
- Example:

setInterval(() => console.log("Every second"), 1000);

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

Ans. function sayHello() {

console.log("Hello, world! (after 2 seconds)");

}

setTimeout(sayHello, 2000);

JavaScript Error Handling

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

Ans. Error handling in JavaScript is a mechanism that allows developers to manage and respond to runtime errors in a controlled way. This helps prevent the application from crashing and provides a way to gracefully handle unexpected situations.

Key Components of Error Handling

1. try Block: Contains code that may throw an error. If an error occurs, control is transferred to the catch block.

2. catch Block: Contains code that executes if an error is thrown in the try block. It can access the error object to understand what went wrong.

3. finally Block: (Optional) Contains code that will execute after the try and catch blocks, regardless of whether an error occurred or not. It is often used for cleanup actions.

Syntax try {

// Code that may throw an error }

catch (error) {

```
 // Code to handle the error
} finally {
// Code that runs regardless of the outcome
}
```

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

Ans. **1. Prevents Crashes & Preserves Stability**

Without proper error handling, unhandled exceptions can crash your application entirely—disrupting functionality and frustrating users. Catching errors keeps the app running and maintains reliability.

2. Enhances User Experience

Instead of exposing confusing or raw error messages—or worse, blank screens—proper handling means users see clear and graceful feedback.

3. Boosts Debugging Efficiency

Well-managed errors let developers see where and why failures occur, improving trackability and fixing time. Plus, logging aids in diagnosing and resolving recurring bugs.

4. Reduces Security Risks

Raw or uncaught errors may expose internal system details to baddies. Handling and sanitizing errors helps protect your system's vulnerabilities from prying eyes.

5. Improves Performance

Uncontrolled errors—especially repeated ones—can degrade performance by clogging up servers or overloading processes. Proactive error catching avoids such bottlenecks.

6. Encourages Graceful Degradation

Handling errors allows your app to function in a limited or fallback mode instead of failing entirely, thereby providing resilience and better usability.

7. Facilitates Maintenance & Scalability

Effective error management and monitoring (e.g., through tools like Sentry) uncover trends, helping you prioritize fixes and improve long-term maintainability.

8. Maintains Predictable Flow

Exception handling avoids disruptions in code flow—important across browsers and environments—by managing unexpected errors cleanly.