**What are template literals in JavaScript?**

**Answer:**  
Template literals are enclosed by backticks ( ` ` ) and allow embedded expressions using the ${expression} syntax. They are used for multi-line strings and string interpolation.

**Example:**

js

CopyEdit

let name = "Ahsan";

console.log(`Hello, my name is ${name}.`);

**2. What is hoisting? Provide an example.**

**Answer:**  
Hoisting is JavaScript's behavior of moving variable and function declarations to the top of their scope before code execution.

* var variables are hoisted and initialized with undefined.
* let and const are hoisted but not initialized, resulting in a ReferenceError if accessed before declaration.

**Example:**

js

CopyEdit

console.log(a); // undefined

var a = 5;

console.log(b); // ReferenceError

let b = 10;

**3. Difference between let, var, and const.**

**Answer:**

* var: Function-scoped, hoisted and initialized as undefined.
* let: Block-scoped, hoisted but not initialized.
* const: Block-scoped, must be initialized at the time of declaration and cannot be reassigned.

**4. Data types in JavaScript.**

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

* **Primitive Types:** String, Number, Boolean, Null, Undefined, Symbol, BigInt.
* **Non-Primitive Types (Reference Types):** Object, Array, Function.

**5. What is an array, and how to access its elements?**

**Answer:**  
An array is a collection of elements stored in a single variable. Elements are accessed using indices (starting from 0).

**Example:**

js

CopyEdit

let arr = [10, 20, 30];

console.log(arr[1]); // 20

**6. Difference between == and ===.**

**Answer:**

* ==: Compares values after type coercion.
* ===: Compares both value and data type.

**Example:**

js

CopyEdit

console.log(5 == "5"); // true

console.log(5 === "5"); // false

**7. Purpose of the isNaN() function.**

**Answer:**  
isNaN() checks whether a value is NaN (Not-a-Number). It returns true if the value is not a number.

**Example:**

js

CopyEdit

console.log(isNaN("abc")); // true

console.log(isNaN(123)); // false

**8. What is null vs undefined?**

**Answer:**

* undefined: A variable is declared but not assigned a value.
* null: Explicitly assigned to a variable as an empty value.

**Example:**

js

CopyEdit

let a;

console.log(a); // undefined

let b = null;

console.log(b); // null

**9. Use of the typeof operator.**

**Answer:**  
typeof is used to determine the data type of a variable or value.

**Example:**

js

CopyEdit

let name = "Ahsan";

console.log(typeof name); // string

let num = 42;

console.log(typeof num); // number

Difference between Event Bubbling and capturing:

**hort Description:**

1. **Event Bubbling**:
   * This is the default behavior where an event starts from the **target element** (the element that triggered the event) and **bubbles up** to its ancestors (parent, grandparent, etc.).
   * **Example**: If you click a button inside a div, the click event will first fire on the button and then propagate to the div, then to its parent, and so on.
2. **Event Capturing**:
   * This is the opposite of bubbling. The event starts from the **root** of the DOM tree and **catches** the event as it travels down to the target element.
   * **Example**: If you click a button inside a div, the event will first fire on the document or body, then travel down to the div, and finally reach the button.

**Why We Use Bubbling and Capturing:**

1. **Event Delegation (Bubbling)**:
   * **Delegation** means handling events at a **higher level** in the DOM rather than attaching event listeners to every individual element.
   * **Why we use it**: It's efficient for handling events on dynamically created elements. Instead of adding listeners to each individual element, we can add one listener to a parent element and let events bubble up.
   * **Example**: Imagine a list of items that may change over time. Instead of adding click listeners to each item, you can add a click listener on the parent <ul> that listens for clicks on any <li> inside it.
2. **Event Interception (Capturing)**:
   * **Interception** means catching events as they **descend** the DOM (before reaching the target).
   * **Why we use it**: We use event capturing when we need to **intercept** events early, before they reach their intended target.
   * **Example**: If you want to stop an event before it gets to a button (for example, blocking a form submission), you might use capturing to stop the event right at the document or body level.

**Is Bubbling Automatic?**

Yes, **event bubbling happens automatically** when an event is triggered. The event bubbles from the target element up to its ancestors unless you stop it using methods like event.stopPropagation().

**Summary:**

* **Event Bubbling**: Used for delegation (efficient event handling).
* **Event Capturing**: Used for interception (catching the event early).
* **Bubbling happens by default**, but you can choose to use capturing by setting the capture flag in event listeners.

Top of Form

Closure in js:  
**Closure in JavaScript:**

A **closure** is a function that **remembers** its lexical scope, even when the function is executed outside that scope.

**Explanation:**

* When a function is defined inside another function, it has access to the outer function's variables, even after the outer function has finished execution. This is called a closure.
* A closure allows the inner function to "remember" the variables from its outer scope, even if the outer function has already returned.

**Example:**

javascript

CopyEdit

function outerFunction() {

let outerVariable = "I'm from outer scope";

// Inner function has access to outerFunction's variables

function innerFunction() {

console.log(outerVariable); // Accesses outerVariable from the outer scope

}

return innerFunction; // Returning inner function as a closure

}

const closure = outerFunction();

closure(); // Output: I'm from outer scope

**Key Points:**

* **Closures** are created whenever a function is created inside another function.
* The inner function **remembers** the variables from its outer function, even if the outer function has already executed and returned.

Top of Form

Bottom of Form

Event Delegation:  
**Event Delegation:**

**Event delegation** is a technique in JavaScript where you attach a single event listener to a parent element, and use it to manage events for child elements. Instead of adding individual event listeners to each child element, you use the parent element to "delegate" the event handling to the children.

**How it works:**

* The event listener is added to a parent element.
* When an event occurs on a child element, the event bubbles up to the parent.
* The parent then checks if the event originated from a specific child element and handles it accordingly.

**Why use it?**

* **Improves performance**: Less memory usage because only one event listener is needed.
* **Dynamic elements**: Works even for dynamically added child elements.

**Example:**

javascript

CopyEdit

document.getElementById("parent").addEventListener("click", function(event) {

if (event.target && event.target.matches("button.className")) {

console.log("Button clicked");

}

});

In this example, even if new buttons are added to the parent element, the event listener will still work for them.

Top of Form

Event loop in js:

The **event loop** in JavaScript is a mechanism that allows asynchronous code (like setTimeout, API calls, or event listeners) to be executed after the synchronous code (like regular function calls) has finished running.

**How it works:**

1. **Call Stack**: JavaScript executes functions in the call stack one by one. If a function is synchronous, it gets executed immediately.
2. **Event Queue**: If there’s asynchronous code (like setTimeout), the callback is moved to the event queue after the task completes.
3. **Event Loop**: The event loop constantly checks if the call stack is empty. If it is, it moves the first task from the event queue to the call stack for execution.

**Process flow:**

1. Synchronous code runs and fills the call stack.
2. Asynchronous code (e.g., setTimeout) is placed in the event queue.
3. Once the call stack is clear, the event loop picks up the tasks from the event queue and pushes them to the call stack for execution.

**Example:**

javascript

CopyEdit

console.log("Start");

setTimeout(() => {

console.log("Inside setTimeout");

}, 0);

console.log("End");

**Output**:

sql

CopyEdit

Start

End

Inside setTimeout

Here, even though the timeout is 0ms, the event loop ensures that "Inside setTimeout" is executed **after** the synchronous code (i.e., "Start" and "End") because the event queue waits for the call stack to clear.

Top of Form

Bottom of Form

Bottom of Form

Bottom of Form