Interview Questions - 1

1. What are the differences between var, let, and const?

Answer:

- var is **function-scoped** and can be re-declared and re-assigned. It is hoisted and initialized with undefined.
- let is **block-scoped**, cannot be re-declared in the same scope, and is also hoisted but not initialized (leads to a "temporal dead zone").
- const is also **block-scoped**, but it must be **initialized during declaration** and cannot be re-assigned. However, if it points to an object or array, that object or array can still be modified.

---

2. What are JavaScript's primitive data types and how are they different from reference types?

Answer:

JavaScript has the following **primitive types**:

- string, number, bigint, boolean, undefined, symbol, and null.

These are **immutable** and stored directly in the variable. When assigned or passed, they create **copies**.

Reference types (like arrays, objects, functions) store a **reference to memory**. When passed or assigned, the **reference** is copied, so changes affect all references to the same object.

---

3. How does == differ from === in JavaScript?

Answer:

- == is the **loose equality operator**. It allows **type coercion** before comparing values.
- === is the **strict equality operator**. It checks both **type and value** without converting data types.

Example:

```
0 == '0'    // true (string is converted to number)

0 === '0'   // false (number vs string)
```

Using === is recommended to avoid unexpected type conversions.

---

4. What are the spread (...) and rest (...) operators in JavaScript?

Answer:

Both use the same ... syntax but serve different purposes:

- **Spread Operator:** Expands elements from an array or object.
- const nums = [1, 2, 3];
- const newNums = [...nums, 4]; // [1, 2, 3, 4]
-
- **Rest Operator:** Collects multiple values into a single array parameter.
- function sum(...args) {
- return args.reduce((a, b) => a + b, 0);
- }
- sum(1, 2, 3); // 6
-

Spread unpacks, rest packs.

---

5. What does Array.prototype.map() do, and how is it different from forEach()?

Answer:

- map() creates a **new array** by applying a function to each element. It returns the transformed array.
- forEach() simply **executes a function** for each element but **does not return** anything useful (it returns undefined).

Example:

const nums = [1, 2, 3];

const doubled = nums.map(n => n * 2); // [2, 4, 6]

nums.forEach(n => console.log(n)); // Logs 1, 2, 3 to console

Use map() when you want to transform data and create a new array. Use forEach() for side effects like logging or updating external values.

6. What is the difference between null and undefined in JavaScript?

Answer:

- undefined means a variable has been declared but **has not been assigned a value** yet.

Example:

let x;

console.log(x); // undefined

- null is an **explicit assignment** indicating the variable is empty or has "no value".

Example:

let y = null;

console.log(y); // null

In short, undefined is JavaScript's default; null is the developer's intent.

---

7. What is the use of the typeof operator in JavaScript?

Answer:

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

Examples:

typeof 123 // "number"

typeof "hello" // "string"

typeof true // "boolean"

typeof undefined // "undefined"

typeof null // "object" (a known quirk in JavaScript)

typeof {} // "object"

typeof [] // "object"

It's useful for basic type checking.

---

8. What is the purpose of isNaN() in JavaScript?

Answer:

isNaN() checks whether a value is **NaN (Not a Number)** after trying to convert it to a number.

Examples:

isNaN(123) // false

isNaN("123") // false

isNaN("hello") // true (can't convert to number)

isNaN(NaN) // true

Note: isNaN() performs type coercion. To avoid that, use Number.isNaN() for stricter checking.

---

9. How do you check if a value is an array in JavaScript?

Answer:

The most reliable way is using Array.isArray():

```
Array.isArray([1, 2, 3]); // true

Array.isArray("hello");   // false

Array.isArray({});        // false
```

Avoid using typeof for this because arrays return "object".

---

10. What does the slice() method do in arrays?

Answer:

The slice() method **returns a shallow copy** of a portion of an array into a new array, **without modifying** the original array.

Syntax:

```
array.slice(start, end)
```

Example:

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

const result = fruits.slice(1, 3); // ["banana", "cherry"]
```

- The original array remains unchanged.
- The end index is **not included**.

11. What is the difference between push() and pop() methods in JavaScript arrays?

Answer:

- push() adds one or more elements **to the end** of an array.
- pop() removes the **last element** from an array and returns it.

Example:

```
let arr = [1, 2, 3];
```

```
arr.push(4);      // [1, 2, 3, 4]

arr.pop();        // returns 4, array becomes [1, 2, 3]
```

---

12. What does the join() method do in an array?

Answer:

The join() method joins all elements of an array into a **single string**, separated by a specified delimiter.

Example:

```
let words = ['hello', 'world'];

let result = words.join(' ');  // "hello world"
```

If no separator is provided, the default is a comma.

Hoisting, Lexical Scoping and Closure

Hoisting in JavaScript: Comprehensive Explanation

What is Hoisting?

Hoisting is a mechanism in JavaScript where variable and function declarations are moved to the top of their containing scope during the **creation phase** of the execution context. This means you can reference variables or functions before their declaration in code, though their behavior depends on how they are declared (var, let, const, or as functions).

How Hoisting Works?

When JavaScript code is executed, two phases occur:

1. **Creation Phase**:
    - Memory is allocated for variables, functions, and classes.
    - Declarations are hoisted to the top of their scope.
    - Initialization is done differently based on the type (undefined for var, no initialization for let and const, and full hoisting for function declarations).
2. **Execution Phase**:
    - Code is executed line by line.
    - Variables and functions are assigned values if their initialization exists in the code.

Behavior of Hoisting for Variables

1. **var**:
    - Hoisted to the top of the scope and initialized with undefined.
    - Can be accessed before its declaration, but the value will be undefined.
2. console.log(a); // undefined
3. var a = 10;
4. console.log(a); // 10
5.
6. **let and const**:
    - Hoisted but **not initialized**.
    - Exist in the **Temporal Dead Zone (TDZ)** until the code reaches their declaration.
    - Accessing them before declaration results in a ReferenceError.
7. console.log(b); // ReferenceError: Cannot access 'b' before initialization
8. let b = 20;
9.

Temporal Dead Zone (TDZ)

The **TDZ** is the period between the start of the block and the point where the variable is declared. Variables in the TDZ are hoisted but cannot be accessed until the code execution reaches their declaration.

Example:

```
{
  console.log(x); // ReferenceError
  let x = 5;     // x is hoisted but in TDZ
}
```

Behavior of Hoisting for Functions

1. **Function Declarations**:
    - Fully hoisted, including their body.
    - Can be called before their declaration.
2. greet(); // "Hello!"

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

```

- **Function Expressions**:
  - Behave like variables.
  - Their hoisting depends on how they are declared (var, let, or const).
  - Not initialized during hoisting.

```
console.log(greet); // undefined (because `var` is hoisted as undefined)
var greet = function () {
    console.log("Hi!");
};

console.log(greet); // Function is now defined

```

- **Arrow Functions**:
  - Also behave like variables and depend on how they are declared (let, const, or var).

Behavior of Hoisting for Classes

Classes are hoisted but not initialized, meaning they behave like let and const:

- Accessing a class before its declaration results in a ReferenceError.
- Methods and static properties are not hoisted individually; the class must be fully declared before it can be used.

Example:

```
const obj = new MyClass(); // ReferenceError
class MyClass {
   constructor() {
      this.name = "Example";
   }
}
```

Hoisting in Function Parameters

Function parameters are implicitly hoisted, similar to let. However, they are automatically initialized with the value passed during the function call or undefined if no value is provided.

Example:

```
function myFunc(param1, param2) {
  console.log(param1); // 10
  console.log(param2); // undefined (no value passed)
}
myFunc(10);
```

Key Details:

1. Parameters cannot be redeclared inside the function using let or const.
2. They can be shadowed inside nested blocks.

Scopes in JavaScript

1. **Global Scope**:
    - Variables declared outside of functions or blocks.
    - Accessible throughout the program.
2. **Function Scope**:
    - Variables declared with var inside a function are accessible only within that function.
3. **Block Scope**:
    - Variables declared with let or const are accessible only within the block they are defined in.

Why Hoisting Happens?

The purpose of hoisting is to allow flexible ordering of code. By hoisting declarations to the top of the scope, JavaScript ensures that the interpreter recognizes variables, functions, and classes even if they are referenced before being explicitly declared.

Key Differences in Hoisting

Entity	Hoisted?	Initialized?	Access Before Declaration?
var	Yes	undefined	Yes, but value is undefined.
let	Yes	No (TDZ)	No, ReferenceError.
const	Yes	No (TDZ)	No, ReferenceError.
Function Declaration	Yes	Fully hoisted with body	Yes, can call before declaration.
Function Expression (var)	Yes	undefined	No, behaves like var.
Function Expression (let or const)	Yes	No (TDZ)	No, ReferenceError.
Class	Yes	No (TDZ)	No, ReferenceError.
Function Parameters	Yes	Initialized with argument value	Yes, using the passed argument value.

Practical Examples

1. **var Hoisting**:
2. console.log(x); // undefined
3. var x = 10;
4. console.log(x); // 10
5.
6. **let and TDZ**:
7. console.log(y); // ReferenceError
8. let y = 20;
9.
10. **Function Declaration**:
11. greet(); // "Hello!"
12. function greet() {
13. console.log("Hello!");
14. }
15.
16. **Class Hoisting**:
17. const obj = new MyClass(); // ReferenceError

```
class MyClass {
    constructor() {
        this.name = "Example";
    }
}
```

Function Parameters:

```
function myFunc(param1 = 5) {
    console.log(param1); // 10
}
myFunc(10);
```

By understanding hoisting and scoping, you can write more predictable and bug-free JavaScript code.

Lexical Scoping in JavaScript: Simple Explanation

What is Lexical Scoping?

Imagine your code is like a set of **nested boxes** (scopes), where each box contains variables. **Lexical scoping** means that a function can look into the boxes it is written inside (outer scopes) to find variables, but it cannot see inside smaller boxes (inner scopes).

Key Points About Lexical Scoping

1. **Where Variables Can Be Accessed**:
    - A function can use variables from its own box (scope) and any boxes it is written inside (outer scopes).
    - A variable outside a function cannot access variables inside the function.
2. **Why "Lexical"?**:
    - It's called **lexical** because it depends on **where the function is written** in the code, not where it is called.

Simple Example

```
function outerFunction() {
```

```
  let outerVariable = "Hello from the outer scope";

  function innerFunction() {
    console.log(outerVariable); // Can access outerVariable
  }

  innerFunction();
}

outerFunction();
// Output: "Hello from the outer scope"
```

What's Happening?

- outerFunction creates a variable called outerVariable.
- innerFunction is written **inside** outerFunction, so it can access outerVariable.

Scope: The Box Example

Think of this:

1. **Global Scope** (biggest box):
    - Variables here are available everywhere.
2. **Function Scope** (smaller box inside global):
    - Variables here are only available inside the function.
3. **Block Scope** (even smaller box inside a function):
    - Variables declared with let or const inside {} are only available inside that block.

Scope Chain

When a variable is used, JavaScript looks for it in the current box (scope). If it doesn't find it, it looks into the outer box. This is called the **scope chain**.

Example:

```
function outerFunction() {
  let outerVariable = "Outer";
```

```
    function innerFunction() {
        console.log(outerVariable); // Looks in outerFunction's box
    }

    innerFunction();
}

outerFunction();
// Output: "Outer"
```

Why Can’t Outer Scopes See Inside Inner Scopes?

Just like a smaller box is closed inside a bigger one, the bigger box cannot see what’s inside the smaller box.

Example:

```
function outerFunction() {
    function innerFunction() {
        let innerVariable = "Inner";
    }

    console.log(innerVariable); // Error: innerVariable is not defined
}

outerFunction();
```

Here, innerVariable is inside the innerFunction box, so outerFunction cannot see it.

Lexical Scoping in Real Life

Lexical scoping enables **closures**, which let a function remember the variables from where it was written.

Example:

```
function outerFunction() {
   let counter = 0;

   return function increment() {
      counter++;
      console.log(counter);
   };
}

const add = outerFunction();
add(); // 1
add(); // 2
add(); // 3
```

Here's what happens:

- outerFunction creates a counter.
- increment remembers counter because it was written inside outerFunction.
- Even after outerFunction finishes, increment can still access counter.

Simple Takeaways

1. A function can always see variables in the box (scope) where it was **written**.
2. JavaScript looks for variables in the current box and keeps looking in outer boxes until it finds the variable or reaches the global box.
3. This is called **lexical scoping** because it depends on where things are written in the code.

What is a Closure in Simple Terms?

A **closure** happens when a function **remembers** the variables from its outer function **even after the outer function has finished running**.

Think of it like this:

A function is holding on to a **"backpack"** of variables that were in scope when it was created, so it can use them later, no matter where it's called.

How Closures Work

1. A function is **nested** inside another function.
2. The inner function can **access** variables from its outer function.
3. Even after the outer function is done, the inner function keeps access to those variables because it "remembers" its environment.

A Simple Example

```
function outerFunction() {
  let count = 0; // A variable in the outer function

  function innerFunction() {
    count++; // Inner function can access `count`
    console.log(count);
  }

  return innerFunction; // Return the inner function
}

const increment = outerFunction(); // Call outerFunction, get `innerFunction`
increment(); // 1
increment(); // 2
increment(); // 3
```

What's Happening?

1. When outerFunction runs:
    - It creates the count variable.
    - It defines innerFunction and returns it.
2. When you call increment():
    - innerFunction is executed.
    - It **remembers** the count variable, even though outerFunction is finished.

This happens because **closures let a function remember its outer variables**.

Why Closures Are Useful

Closures are useful for many real-life coding scenarios, such as:

1. **Remembering a Value (Encapsulation)**:
    - You can use closures to create private variables that can't be accessed directly.

Example:

```
function createCounter() {
  let count = 0;

  return function increment() {
    count++;
    return count;
  };
}

const counter = createCounter();
console.log(counter()); // 1
console.log(counter()); // 2
console.log(counter()); // 3
```

Here, count is private. It's only accessible to the increment function.

---

1. **Creating Functions with Specific Data**

Example:

```
function createMultiplier(multiplier) {
  return function multiply(number) {
    return number * multiplier;
  };
}

const double = createMultiplier(2);
```

```
const triple = createMultiplier(3);


console.log(double(5)); // 10
console.log(triple(5)); // 15
```

What's Happening?

- double remembers multiplier = 2.
- triple remembers multiplier = 3.

---

1. **Event Listeners or Timers**

Closures are often used in asynchronous code, like timers or event listeners.

Example:

```
function delayedGreeting(name) {
  setTimeout(function () {
    console.log(`Hello, ${name}!`);
  }, 2000);
}


delayedGreeting("Sohan"); // After 2 seconds: "Hello, Sohan!"
```

What's Happening?

- The function inside setTimeout remembers the name variable.

How Closures Use Memory

Closures keep variables alive in memory as long as the inner function exists.

That's why closures can lead to **memory usage** if not used carefully.

Key Takeaways

1. **Closures = Function + Variables from Outer Scope**:
   - A function **remembers** variables from where it was created.
2. **Closures Are Everywhere**:

- Whenever you have a function inside another, a closure is created.

3. **Real-Life Uses**:
    - Maintaining private variables.
    - Customizing functions (like the multiplier example).
    - Handling asynchronous code (like timers).

Real-Life Use Cases of Closures

Closures are everywhere in JavaScript! Here are some practical implementations to help you understand where and how closures are used in real-world scenarios:

---

1. Data Privacy (Encapsulation)

Closures are often used to create private variables in JavaScript. This is especially useful when you want to hide certain data or logic from outside access.

Example: Creating a Private Counter

```
function createCounter() {
  let count = 0; // Private variable


  return {
    increment: function () {
      count++;
      return count;
    },
    decrement: function () {
      count--;
      return count;
    },
    getCount: function () {
      return count;
    },
  };
}


const counter = createCounter();
```

```
console.log(counter.increment()); // 1
console.log(counter.increment()); // 2
console.log(counter.decrement()); // 1
console.log(counter.getCount());  // 1
```

Why Useful?

- count is private and cannot be modified directly from outside the createCounter function.

---

2. Function Factories

Closures can generate functions dynamically based on input parameters. This is useful for customizing behavior.

Example: Multipliers

```
function createMultiplier(multiplier) {
   return function (number) {
      return number * multiplier;
   };
}

const double = createMultiplier(2);
const triple = createMultiplier(3);

console.log(double(4)); // 8
console.log(triple(4)); // 12
```

Why Useful?

- You can create reusable functions without repeating code.

---

3. Event Listeners

Closures are used in event listeners to "remember" values from the outer scope.

Example: Button Click Tracker

```
function setupButton() {
```

```
    let clickCount = 0;

    document.getElementById("myButton").addEventListener("click", function () {
        clickCount++;
        console.log(`Button clicked ${clickCount} times`);
    });
}

setupButton();
```

Why Useful?

- The clickCount variable persists across multiple button clicks because of the closure.

---

4. Timer and Async Operations

Closures are crucial in asynchronous code like timers or callbacks to remember the state.

Example: Delayed Messages

```
function delayedMessage(message, delay) {
    setTimeout(function () {
        console.log(message);
    }, delay);
}

delayedMessage("Hello after 2 seconds", 2000); // Prints after 2 seconds
```

Why Useful?

- The function inside setTimeout remembers the message variable due to the closure.

---

5. Maintaining State in Iterative Functions

Closures can help when creating functions inside loops to remember the correct value.

Problem: Variable Reference in Loops

```
function createButtons() {
```

```
  for (let i = 1; i <= 3; i++) {
    document.getElementById(`button${i}`).addEventListener("click", function () {
      console.log(`Button ${i} clicked`);
    });
  }
}

createButtons();
// Clicking Button 1 logs: "Button 1 clicked"
// Clicking Button 2 logs: "Button 2 clicked"
```

Here, the block-scoped let in the loop works due to closures.

---

6. Customizable Event Handlers

Closures allow you to create event handlers with pre-configured data.

Example: Logging Custom Data

```
function logOnClick(data) {
  return function () {
    console.log(`You clicked: ${data}`);
  };
}

const button1 = document.getElementById("button1");
button1.addEventListener("click", logOnClick("Button 1"));

const button2 = document.getElementById("button2");
button2.addEventListener("click", logOnClick("Button 2"));
```

Why Useful?

- You can "pre-load" specific data for each button click without creating global variables.

---

7. Memoization

Closures are commonly used to store results of expensive function calls and reuse them when the same inputs are provided.

Example: Fibonacci Memoization

```
function memoize(fn) {
   const cache = {};

   return function (num) {
      if (cache[num] !== undefined) {
         return cache[num];
      }
      const result = fn(num);
      cache[num] = result;
      return result;
   };
}

function fibonacci(n) {
   if (n <= 1) return n;
   return fibonacci(n - 1) + fibonacci(n - 2);
}

const memoizedFibonacci = memoize(fibonacci);
console.log(memoizedFibonacci(40)); // Much faster than plain recursion
```

Why Useful?

- Reduces computation time by caching results.

---

8. Module Pattern

Closures allow you to emulate modules, keeping variables private while exposing only necessary functions.

Example: Module

```
const CounterModule = (function () {
  let count = 0; // Private variable

  return {
    increment: function () {
      count++;
      return count;
    },
    reset: function () {
      count = 0;
    },
  };
})();

console.log(CounterModule.increment()); // 1
console.log(CounterModule.increment()); // 2
CounterModule.reset();
console.log(CounterModule.increment()); // 1
```

Why Useful?

- Organizes code and protects private data.

---

9. Dynamic UI Components

Closures are helpful in creating components with isolated state for frameworks like React or vanilla JavaScript.

Example: Accordion

```
function createAccordion(section) {
  let isOpen = false;

  section.addEventListener("click", function () {
    isOpen = !isOpen;
```

```
        console.log(`Section is now ${isOpen ? "open" : "closed"}`);
    });
}

const section1 = document.getElementById("section1");
createAccordion(section1);
```

---

10. Function Debouncing or Throttling

In performance optimization, closures help manage state across rapid function calls.

Example: Debouncing

```
function debounce(func, delay) {
    let timeout;

    return function () {
        clearTimeout(timeout);
        timeout = setTimeout(func, delay);
    };
}

const logMessage = debounce(() => console.log("Debounced!"), 300);
logMessage(); // Waits 300ms before executing
```

Why Useful?

- Prevents a function from running too frequently.

Summary

Closures are incredibly versatile. They allow:

1. **Data Privacy**: Keeping variables safe from outside interference.
2. **State Management**: Storing values for dynamic behavior.
3. **Reusable Functions**: Generating customized functions (e.g., multipliers).

4. **Performance**: Optimizing expensive computations with memoization.
5. **Dynamic Applications**: Building interactive and event-driven programs.

In short, closures are a core concept that powers modern JavaScript applications.

Lexical Scoping and Closure

Lexical Scoping (also called *Static Scoping*)

Definition:

In JavaScript, *lexical scope* means that the **scope of a variable is defined by its position in the source code**. Nested functions have access to variables declared in their outer (parent) functions.

How it works:

The function's scope is **determined at the time of writing the code**, not when the function is executed.

Example:

```
function outer() {
  let name = "Ujjwal";

  function inner() {
    console.log(name); // inner can access name from outer
  }

  inner();
}

outer();
```

Output:

Ujjwal

Here, inner() can access name because it is defined inside outer(), and JavaScript uses **lexical scoping** to look up variables.

Closure

Definition:

A **closure** is created when a **function "remembers" its lexical scope** even when it is executed **outside** of that scope.

In short:

Closure = Function + Lexical Environment (variables it had access to when it was created)

Example:

```
function outer() {
  let counter = 0;

  return function inner() {
    counter++;
    console.log(counter);
  };
}

const count = outer(); // outer is called, inner is returned and assigned to count
count(); // 1
count(); // 2
count(); // 3
```

Here, count is a closure. It **remembers** the variable counter even though outer() has finished executing.

Why Closures Are Useful

- **Data encapsulation:** Keep variables private
- **Function factories:** Return customized functions
- **Callback functions & event handlers:** Hold on to parent scope

Example: Data Privacy

```
function secretHolder() {
  let secret = "JavaScript is awesome";

  return {
    getSecret: () => secret,
```

```
    setSecret: (newSecret) => { secret = newSecret; }
  };
}

const mySecret = secretHolder();
console.log(mySecret.getSecret()); // JavaScript is awesome
mySecret.setSecret("Closures are powerful");
console.log(mySecret.getSecret()); // Closures are powerful
```

Summary

Concept	Description
Lexical Scope	Scope is determined by where variables/functions are written (static scope)
Closure	Function "remembers" variables from its lexical scope even after it exits
Concept	Description
Lexical Scope	Scope is determined by where variables/functions are written (static scope)
Closure	Function "remembers" variables from its lexical scope even after it exits