**JavaScript Interview Question:**

**Immutibility of JavaScript ?**

Immutability is a fundamental concept in programming. An immutable value is a value that can not be changed after it has been created.

This concept is important to functional programming and state management. It's a valuable concept, especially when dealing with concurrency and large, complex codebases.

While JavaScript **objects and arrays** are mutable by default, adopting an immutable approach to handling them can lead to cleaner, more reliable, and easier-to-maintain code.

**Primitive DataTypes** are Immutable because:

To prove that primitive data types are immutable in JavaScript, you can demonstrate that once a primitive value is created, it cannot be changed directly. Instead, you always work with new values when performing operations. Here’s a step-by-step approach to illustrate this concept:

**1. Reassigning Variables**

Assigning a new value to a variable holding a primitive type does not alter the original value; it simply updates the reference to the new value.

**Example:**

let num = 10; // `num` holds the value 10

console.log(num); // Outputs: 10

num = 20; // `num` is now reassigned to 20

console.log(num); // Outputs: 20

* **Explanation:** The original value 10 is not changed; num is simply updated to refer to a new value 20. The value 10 is now not referenced by num and can be garbage collected if not referenced elsewhere.

**How to archeive it :**

**2. Immutability for Objects and Arrays**

**Objects and arrays are mutable by default. To achieve immutability, you need to use techniques that avoid modifying the original structures.**

1. **Using Spread Operator**

**Objects:**

const person = { name: 'Alice', age: 25 };

const updatedPerson = { ...person, age: 26 }; // Create a new object with updated age

console.log(person); // Outputs: { name: 'Alice', age: 25 }

console.log(updatedPerson); // Outputs: { name: 'Alice', age: 26 }

**Arrays:**

const numbers = [1, 2, 3];

const updatedNumbers = [...numbers, 4]; // Create a new array with an additional number

console.log(numbers); // Outputs: [1, 2, 3]

console.log(updatedNumbers); // Outputs: [1, 2, 3, 4]

**Using Object.assign**

**Objects:**

const person = { name: 'Alice', age: 25 };

const updatedPerson = Object.assign({}, person, { age: 26 }); // Create a new object

**console.log(person); // Outputs: { name: 'Alice', age: 25 }**

**console.log(updatedPerson); // Outputs: { name: 'Alice', age: 26 }**

**Arrays:**

**const numbers = [1, 2, 3];**

**const updatedNumbers = numbers.concat(4); // Create a new array**

**console.log(numbers); // Outputs: [1, 2, 3]**

**console.log(updatedNumbers); // Outputs: [1, 2, 3, 4]**

**Using Object.freeze**

Object.freeze can be used to make objects immutable. This is a shallow freeze, meaning nested objects remain mutable.

**Example:**

const obj = Object.freeze({ name: "Alice" });

console.log(obj.name); // Outputs: Alice

// Attempting to modify the object (frozen object cannot be changed)

obj.name = "Bob";

console.log(obj.name); // Outputs: Alice (unchanged)

// Comparison with primitive values

let value = 10;

value = 20;

console.log(value); // Outputs: 20 (the original value 10 is not modified)

**Benefits of Immutabilities ?**

First of all, immutable data structures are more stable and predictable. They're immune to unexpected alterations, rendering the code more deterministic and less prone to unforeseen bugs or side effects, which is very useful in large-scale applications.

**What are the different data types present in javascript?**

To know the type of a JavaScript variable, we can use the **typeof**operator.

**1. Primitive types**

**String**- It represents a series of characters and is written with quotes. A string can be represented using a single or a double quote.

Example :

**var** str = "Vivek Singh Bisht"; //using double quotes

**var** str2 = 'John Doe'; //using single quotes

* **Number**- It represents a number and can be written with or without decimals.

Example :

**var** x = 3; //without decimal

**var** y = 3.6; //with decimal

* **BigInt**- This data type is used to store numbers which are above the limitation of the Number data type. It can store large integers and is represented by adding “n” to an integer literal.

Example :

**var** bigInteger = 234567890123456789012345678901234567890;

* **Boolean**- It represents a logical entity and can have only two values : true or false. Booleans are generally used for conditional testing.

Example :

**var** a = 2;

**var** b = 3;

**var** c = 2;

(a == b) // returns false

(a == c) //returns true

* **Undefined**- When a variable is declared but not assigned, it has the value of undefined and it’s type is also undefined.

Example :

**var** x; // value of x is undefined

**var** y = undefined; // we can also set the value of a variable as undefined

* **Null**- It represents a non-existent or a invalid value.

Example :

**var** z = null;

* **Symbol**- It is a new data type introduced in the ES6 version of javascript. It is used to store an anonymous and unique value.

Example :

**var** symbol1 = Symbol('symbol');

* typeof **of primitive types**:

**typeof** "John Doe" // Returns "string"

**typeof** 3.14 // Returns "number"

**typeof** true // Returns "boolean"

**typeof** 234567890123456789012345678901234567890n // Returns bigint

**typeof** undefined // Returns "undefined"

**typeof** null // Returns "object" (kind of a bug in JavaScript)

**typeof** Symbol('symbol') // Returns Symbol

**2. Non-primitive types**

* Primitive data types can store only a single value. To store multiple and complex values, non-primitive data types are used.
* Object - Used to store collection of data.
* Example:

// Collection of data in key-value pairs

**var** obj1 = {

x: 43,

y: "Hello world!",

z: **function**(){

**return** this.x;

}

}

// Collection of data as an ordered list

**var** array1 = [5, "Hello", true, 4.1];

### 2. Difference between “ == “ and “ === “ operators.

Both are comparison operators. The difference between both the operators is that “==” is used to compare values whereas, “ === “ is used to compare both values and types.

**Example:**

**var** x = 2;

**var** y = "2";

(x == y) // Returns true since the value of both x and y is the same

(x === y) // Returns false since the typeof x is "number" and typeof y is "string"

### 3. Difference between var and let keyword in javascript.

### Variables are container for storing data.

Some differences are

1. From the very beginning, the 'var' keyword was used in JavaScript programming **whereas the keyword**'let' was just added in 2015.
2. The keyword 'Var' has a function scope. Anywhere in the function, the variable specified using var is accessible but in ‘let’ the scope of a variable declared with the 'let' keyword is limited to the block in which it is declared. Let's start with a Block Scope.

3.Var allow hoisting but let doesn’t allow hoisting

4. let variable cannot be redeclare but if redeclare present in different scope

Here redeclaration will work because but declaration is present in **different** scope but if it had redeclared in **same** function or same block scope then it would have been error.

Function func() {

let x = 10; //Function scope

console.log(x);

{

let x = 2; //Block Scope

console.log(x);

}

}

**4. Explain Implicit Type Coercion in javascript.**

Implicit type coercion in javascript is the automatic conversion of value from one data type to another. It takes place when the operands of an expression are of different data types.

* **String coercion**

String coercion takes place while using the ‘ + ‘ operator. When a number is added to a string, the number type is always converted to the string type.

Example 1:

**var** x = 3;

**var** y = "3";

x + y // Returns "33"

Example 2:

**var** x = 24;

**var** y = "Hello";

x + y // Returns "24Hello";

**Let’s understand both the examples where we have added a number to a string,**

**When JavaScript sees that the operands of the expression x + y are of different types ( one being a number type and the other being a string type ), it converts the number type to the string type and then performs the operation. Since after conversion, both the variables are of string type, the ‘ + ‘ operator outputs the concatenated string “33” in the first example and “24Hello” in the second example.**

#### **Note - Type coercion also takes place when using the ‘ - ‘ operator, but the difference while using ‘ - ‘ operator is that, a string is converted to a number and then subtraction takes place.**

**var** x = 3;

Var y = "3";

x - y //Returns 0 since the variable y (string type) is converted to a number type

* **Equality Coercion**

Equality coercion takes place when using ‘ == ‘ operator. As we have stated before  
  
**The ‘ == ‘ operator compares values and not types.**  
  
While the above statement is a simple way to explain == operator, it’s not completely true  
  
The reality is that while using the ‘==’ operator, coercion takes place.  
  
The ‘==’ operator, converts both the operands to the same type and then compares them.  
  
Example:

**var** a = 12;

**var** b = "12";

a == b // Returns true because both 'a' and 'b' are converted to the same type and then compared. Hence the operands are equal.

Coercion does not take place when using the ‘===’ operator. Both operands are not converted to the same type in the case of ‘===’ operator.

Example:

**var** a = 226;

**var** b = "226";

a === b // Returns false because coercion does not take place and the operands are of different types. Hence they are not equal.

### 5. What is NaN property in JavaScript?

NaN property represents the **“Not-a-Number”**value. It indicates a value that is not a legal number.

**typeof**of NaN will return a **Number**.

To check if a value is NaN, we use the **isNaN()**function.

**6. Explain passed by value and passed by reference.**

**Pass by Value:**

* **Definition:** In pass by value, a copy of the actual value (primitive data types like numbers, strings, booleans) is passed to the function.
* **Behavior:**
  + Changes made to the parameter inside the function do not affect the original variable outside the function.
  + Variables are independent of each other, and modifications inside the function do not propagate back to the caller.
* **Example:**

function incrementValue(x) {

x = x + 1;

console.log('Inside function:', x); // Output: Incremented value

}

let num = 5;

incrementValue(num);

console.log('Outside function:', num); // Output: Original value (5)

**Pass by Reference:**

* **Definition:** In pass by reference, a reference (memory address) to the original variable is passed to the function.
* **Behavior:**
  + Changes made to the parameter inside the function affect the original variable outside the function.
  + Both the original variable and the parameter inside the function refer to the same memory location.
* **Example:**

function addToArray(arr) {

arr.push('new item');

console.log('Inside function:', arr); // Output: Modified array

}

let myArray = ['item1', 'item2'];

addToArray(myArray);

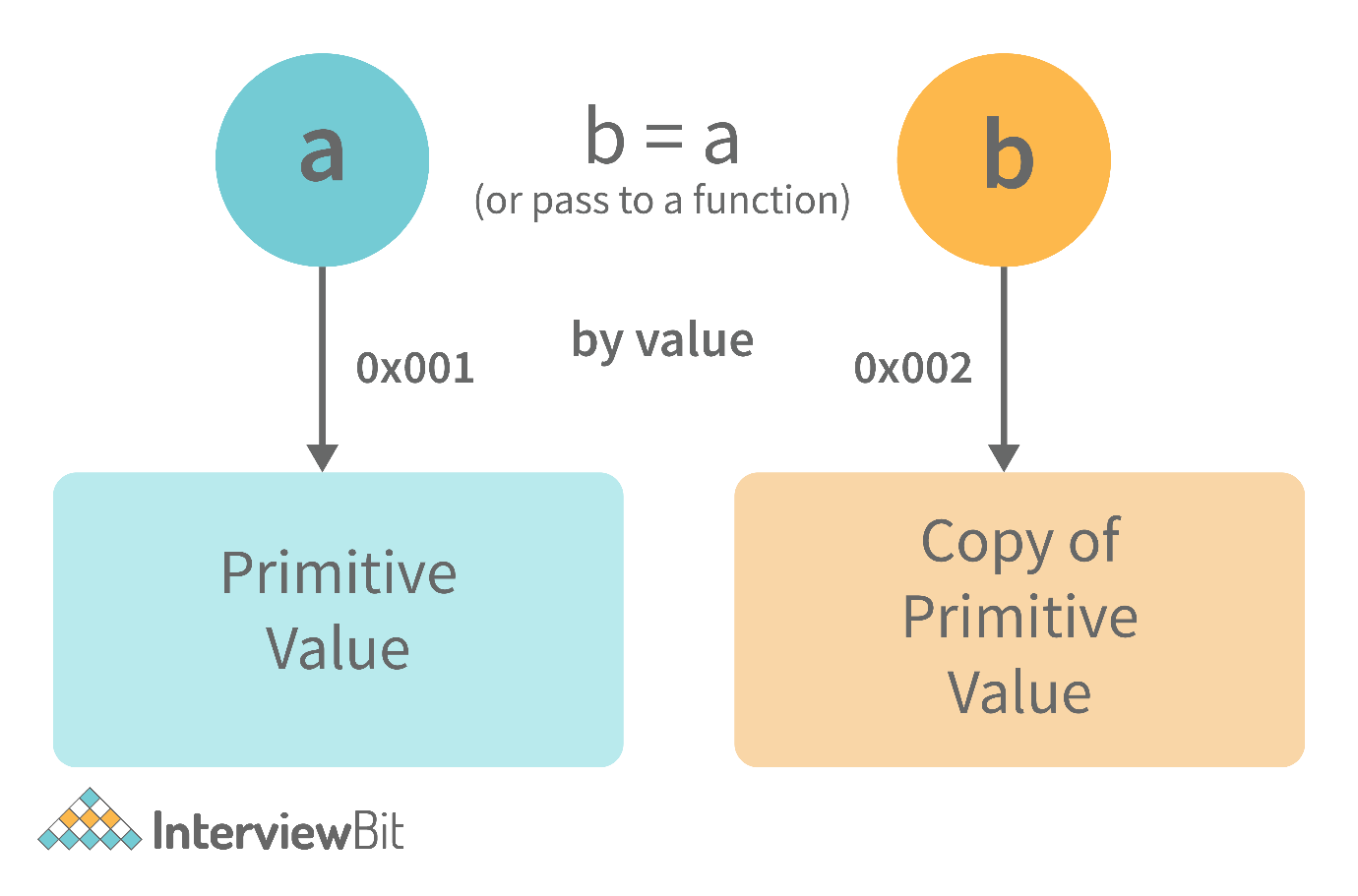
console.log('Outside function:', myArray);

**In JavaScript, primitive data types are passed by value and non-primitive data types are passed by reference.**  
  
For understanding passed by value and passed by reference, we need to understand what happens when we create a variable and assign a value to it,

**var** x = 2;

In the above example, we created a variable x and assigned it a value of “2”. In the background, the “=” (assign operator) allocates some space in the memory, stores the value “2” and returns the location of the allocated memory space. Therefore, the variable x in the above code points to the location of the memory space instead of pointing to the value 2 directly.

Assign operator behaves differently when dealing with primitive and non-primitive data types,  
  
**Assign operator dealing with primitive types:**



**var** y = 234;

**var** z = y;

In the above example, the assign operator knows that the value assigned to y is a primitive type (number type in this case), so when the second line code executes, where the value of y is assigned to z, the assign operator takes the value of y (234) and allocates a new space in the memory and returns the address. Therefore, variable z is not pointing to the location of variable y, instead, it is pointing to a new location in the memory.

**var** y = #8454; // y pointing to address of the value 234

**var** z = y;

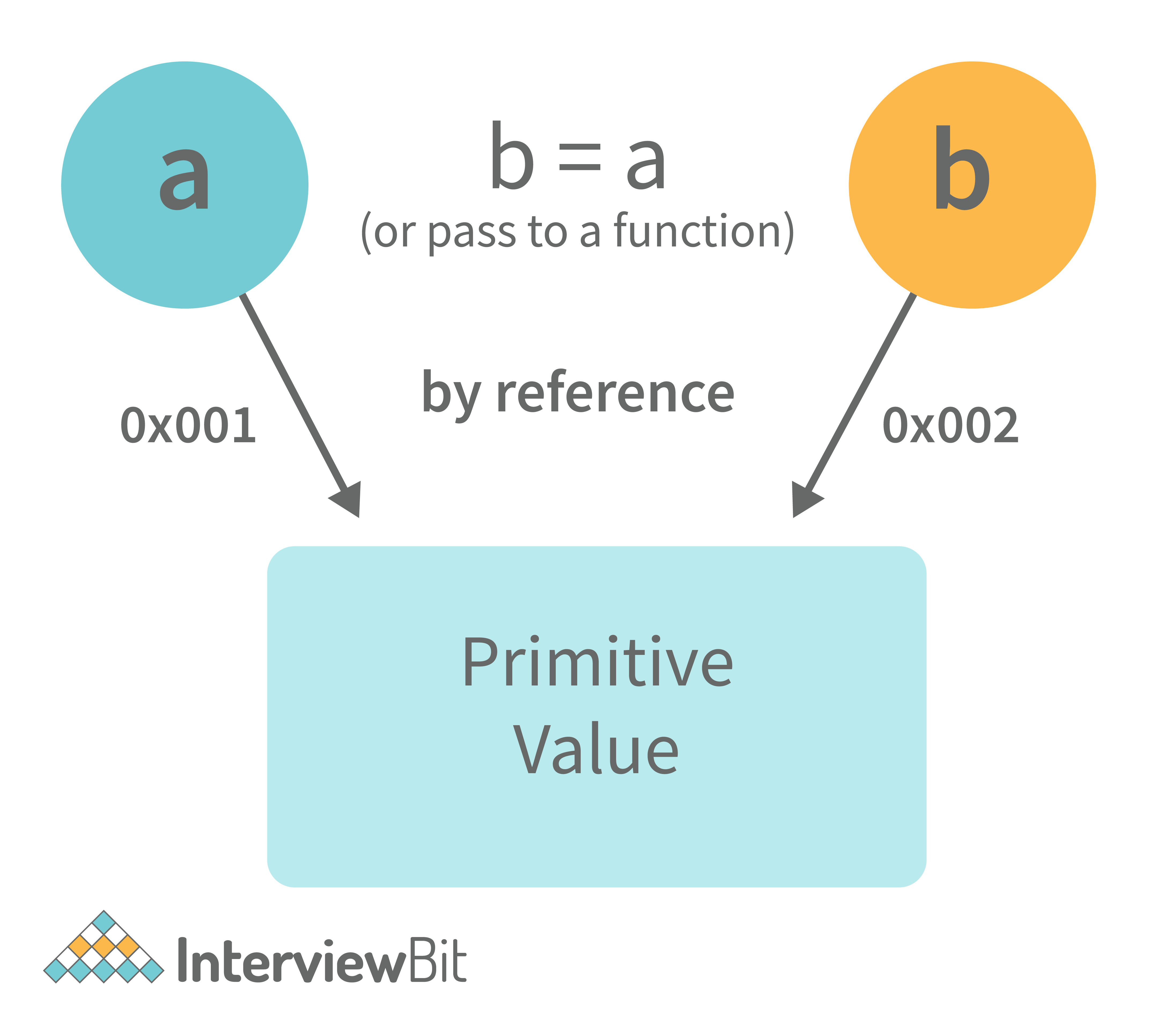
**var** z = #5411; // z pointing to a completely new address of the value 234

// Changing the value of y

y = 23;

console.log(z); // Returns 234, since z points to a new address in the memory so changes in y will not effect z

From the above example, we can see that primitive data types when passed to another variable, are passed by value. Instead of just assigning the same address to another variable, the value is passed and new space of memory is created.  
  
**Assign operator dealing with non-primitive types:**



**var** obj = { name: "Vivek", surname: "Bisht" };

**var** obj2 = obj;

In the above example, the assign operator directly passes the location of the variable obj to the variable obj2. In other words, the reference of the variable obj is passed to the variable obj2.

**var** obj = #8711; // obj pointing to address of { name: "Vivek", surname: "Bisht" }

**var** obj2 = obj;

**var** obj2 = #8711; // obj2 pointing to the same address

// changing the value of obj1

obj.name = "Akki";

console.log(obj2);

// Returns {name:"Akki", surname:"Bisht"} since both the variables are pointing to the same address.

From the above example, we can see that while passing non-primitive data types, the assigned operator directly passes the address (reference).  
  
Therefore, non-primitive data types are always **passed by reference.**

### 7. Explain Higher Order Functions in javascript.

**Functions that operate on other functions, either by taking them as arguments or by returning them, are called higher-order functions.**  
  
Higher-order functions are a result of functions being **first-class citizens**in javascript.

Examples of higher-order functions:

**function** **higherOrder**(fn) {

fn();

}

higherOrder(**function**() { console.log("Hello world") });

**function** **higherOrder2**() {

**return** **function**() {

**return** "Do something";

}

}

**var** x = higherOrder2();

**8. Explain call(), apply() and, bind() methods.**

The this keyword in JavaScript is a **special keyword** that refers to the **execution context** in which a function or block of code is running. The value of this depends on how and where it is used.

In JavaScript, this keyword refers to the current object in context. Its value depends on how it’s used: in methods, it refers to the object; in global scope, it refers to the global object and Browser it refers to Windows.

In JavaScript, call, apply, and bind are methods that allow you to explicitly set the value of this and invoke functions with different contexts.

**call()**, **apply()**, and **bind()** all allow you to bind this to the owner object.

**call()** and **apply()** invoke the function immediately, with differences in how arguments are passed.

**bind()** creates a new function that you can call later, with this permanently bound to the specified object.

You can pass arguments to bind() right away, and additional arguments can be passed when you later call the new function.

**The Method borrowing is the process of borrowing the function of one object and use in another object.**

**1. call():**

function greet(greeting, punctuation) {

    console.log(greeting + ', ' + this.name + punctuation);

  }

  const person = { name: 'Alice' };

  greet.call(person, 'Hello', '!'); // Output: Hello, Alice!

The **call** method is used to invoke a function immediately by explicitly setting the value of the this keyword and passing the arguments individually.

* The **call()** method calls a function with a given this value and arguments provided individually. This method invokes a method (function) by specifying the owner object.
* call() method allows an object to use the method (function) of another object.

const person1 = {

    firstName: "Anup",

    lastName: "Chakra",

    roll: 24,

    func: function(hometown, city){

        return this.firstName+ " " + this.lastName+ " " + hometown + " " + city;

    }

}

const person2 = {

    firstName: "Sachin",

    lastName: "Tendulkar",

}

let final = person1.func.bind(person2, "FilmCity", "Mumbai");

console.log(final());

console.log( person1.func.call(person2, "Patia", "Bhubaneswar"));

console.log(person1.func.apply(person2,["FilmCity", "Mumbai"]));

**apply()**  
  
The apply method is similar to the call() method. The only difference is that,  
  
**call() method takes arguments separately whereas, apply() method takes arguments as an array.**

**Difference between the call() and apply() methods:** The only difference is **call()** method takes the arguments separated by a comma while **apply()** method takes the array of arguments.

**2. bind():**

* The bind method doesnot invoke the function immediately but instead it creates new function where the owner object is passed and the arguments are passed individually.
* The bind() method does not invoke the function immediately. Instead, it creates a new function where this is bound to the specified owner object.and you can pass arguments individually (either immediately when binding or later when invoking the new function).
* This method returns a new function, where the value of **“this”**keyword will be bound to the owner object, which is provided as a parameter.

Example with arguments: The bind() method returns a new [function](https://www.javascripttutorial.net/javascript-function/), when invoked, has its [this](https://www.javascripttutorial.net/javascript-this/) sets to a specific value.

**9) Function Borrowing**

* The bind() method allows an object to borrow a method from another object without making a copy of that method. This is known as function borrowing in JavaScript.

const person1 = {

    firstName: "Anup",

    lastName: "Chakra",

    roll: 24,

    func: function(hometown, city){

        return this.firstName+ " " + this.lastName+ " " + hometown + " " + city;

    }

}

const person2 = {

    firstName: "Sachin",

    lastName: "Tendulkar",

}

let final = person1.func.bind(person2, "FilmCity", "Mumbai");

console.log(final());

let car = {

brand: 'Honda',

getBrand: function () {

return this.brand;

}

}

console.log(car.getBrand());

Since a method is a property of an object which is a value, you can store it in a variable.

let brand = car.getBrand

And then call the method via the variable

console.log(brand()); *// undefined*

You get undefined instead of "Honda" because when you call a method without specifying its object, JavaScript sets this to the global object in non-strict mode and undefined in the strict mode.

To fix this issue, you use the [bind()](https://www.javascripttutorial.net/javascript-bind/) method of the Function.prototype object. The bind() method creates a new function whose the this keyword is set to a specified value.

let brand = car.getBrand.bind(car);

console.log(brand()); *// Honda*

Explain the concept of "this" keyword in JavaScript. How does its value change based on context?

## **Explicit Binding**

When we explicitly bind this keyword using the [call()](https://www.geeksforgeeks.org/explain-call-and-apply-methods-in-javascript/), [bind()](https://www.geeksforgeeks.org/javascript-function-prototype-bind-method/), or [apply()](https://www.geeksforgeeks.org/explain-call-and-apply-methods-in-javascript/) method then this keyword default reference is changed to the object called using the above-specified methods.

**Example:**In this example, we will see the explicit binding of this keyword.

|  |
| --- |
| **function** ageVerify(){  **if**(**this**.age> 18){          console.log("Yes you can drive");      } **else** {          console.log("No you cannot drive");      }  }    const per1 = {age: 21};  const per2 = {age: 16};  ageVerify.call(per1);  ageVerify.call(per2); |

## **Arrow Function Binding**

When this is used in the arrow function then this has lexical scope so without the function keyword this is uanble to refer to the object in the outer scope.

**Example:**This example shows the arrow function binding.

|  |
| --- |
| const person = {      name: "ram",      age: 22,      greet : () =>{  **return** `Hello , you are ${**this**.age} years old`      }  }  console.log(person.greet()); |

**Output**

Hello , you are undefined years old

Method Context:

When a function is called as a method of an object, this refers to the object itself.

this is dynamically bound to the object on which the method is called.

CODE :-

const person = {

name: 'John',

greet: function() {

console.log('Hello, ' + this.name);

}

};

person.greet(); // this refers to the person object

Global Context:

In the global context, the this references the  Global Object, which is the window object on the web browser or global object on Node.js.console.log(this === window); // true in a browser environment.

Constructor Context:

When a function is used as a constructor with the new keyword, this refers to the newly created instance of the object.

Constructors initialize properties and methods on the newly created object using this.

function Person(name) {

this.name = name;

}

const john = new Person('John');

console.log(john.name); // 'John'

10) What is currying in JavaScript?

**Currying is an advanced technique to transform a function of arguments n, to n functions of one or fewer arguments.**

Currying in JavaScript transforms a function with multiple arguments into a nested series of functions, each taking a single argument. Currying helps you avoid passing the same variable multiple times, and it helps you create a higher order function

Example of a curried function:

**function** **add** (a) {

**return** **function**(b){

**return** a + b;

}

}

add(3)(4)

For Example, if we have a function **f(a,b)**, then the function after currying, will be transformed to **f(a)(b).**  
  
By using the currying technique, we do not change the functionality of a function, we just change the way it is invoked.  
  
Let’s see currying in action:

**function** **multiply**(a,b){

**return** a\*b;

}

**function** **currying**(fn){

**return** **function**(a){

**return** **function**(b){

**return** fn(a,b);

}

}

}

**var** curriedMultiply = currying(multiply);

multiply(4, 3); // Returns 12

curriedMultiply(4)(3); // Also returns 12

As one can see in the code above, we have transformed the function **multiply(a,b)**to a function **curriedMultiply**, which takes in one parameter at a time.

**11) What happens when you call regular function inside the setTimeout in the object?**setTimeout is arrow function so this doesnot have own this keyword so when the it refers to this keyword in surrounding

setTimeout is a global function and when you pass a regular function (the one inside setTimeout) as its callback, that regular function is executed in the global context.

In regular functions, when they are called in the global context, this refers to the global object (window in browsers).

const person = {

name: "Anup",

sayHi: function () {

setTimeout(function () {

console.log("Hi, I am " + this.name); // 'this' refers to the global object (window), not 'person'

}, 1000);

}

};

person.sayHi(); // Output: Hi, I am undefined

**In JavaScript, this refers to the context in which a function is invoked (called). When a function is called as a method of an object, this inside that method refers to the object the method was called on.**

**12) What is the Arrow function and difference between the regular and arrow function.?**

Arrow functions, introduced in ECMAScript 6 (ES6), are a **concise** way to write functions. They provide a shorter syntax compared to regular functions, but also behave a bit differently in terms of this, arguments, and other features.

Difference between the regular function and arrow functions?

**Regular function :**

In a **regular function**, this is not automatically bound to the surrounding context where the function was written. Instead, this is determined by **how the function is invoked**, meaning the value of this depends on the **calling context**.

**Arrow Function:**

* Arrow functions **do not have their own this**. Instead, **they inherit this** from the surrounding context in which they are defined. This is known as **lexical scoping**.

This means that an arrow function **takes this from its enclosing context**, rather than determining this based on how it is called.

const person = {

name: "Suman",

greet: function () {

console.log("Inside greet: " + this.name); // 'this' refers to 'person'

const sayHello = () => {

console.log("Inside sayHello (Arrow): " + this.name); // 'this' refers to 'person' because of lexical scoping

};

sayHello();

}

};

NOTE: Here in the above example this of arrow function is taken from surrounding like regular function

person.greet(); // Output: Inside greet: Suman, Inside sayHello (Arrow): Suman

* If the function body contains a single expression, the result of that expression is **automatically returned**, so you don't need the return keyword.

| **Feature** | **Regular Function** | **Arrow Function** |
| --- | --- | --- |
| 🔁 this binding | this is **dynamic**, depends on how the function is called | this is **lexically bound**, taken from the enclosing context |
| 🏗️ Can be used as constructor | ✅ Yes, can use new to create objects | ❌ No, cannot be used with new, throws error |
| 💼 Arguments object | ✅ Has its own arguments object | ❌ No arguments object, must use rest parameters (...args) |
| 📦 Implicit return (one-line) | ❌ Must use return keyword for return values | ✅ Supports implicit return if using one-line syntax |
| 🧠 Scope behavior | Has its own this, super, arguments, and new.target | Inherits all of these from the parent (lexical) scope |
| 💬 Syntax | function myFunc() {} | const myFunc = () => {} |
| 📍 Best used for | Object methods, event handlers (when dynamic this needed), constructor functions | Callbacks, array methods (map, filter), or inner functions needing parent this |

**13. Explain Scope and Scope Chain in javascript.**

Scope in JS determines the accessibility of variables and functions at various parts of one’s code.  
  
In general terms, the scope will let us know at a given part of code, what are variables and functions we can or cannot access.  
  
There are three types of scopes in JS:

* Global Scope
* Local or Function Scope
* Block Scope

**Global Scope:**Variables or functions declared in the global namespace have global scope, which means all the variables and functions having global scope can be accessed from anywhere inside the code.

**var** globalVariable = "Hello world";

**function** **sendMessage**(){

**return** globalVariable; // can access globalVariable since it's written in global space

}

**function** **sendMessage2**(){

**return** sendMessage(); // Can access sendMessage function since it's written in global space

}

sendMessage2(); // Returns “Hello world”

**Function Scope:**Any variables or functions declared inside a function have local/function scope, which means that all the variables and functions declared inside a function, can be accessed from within the function and not outside of it.

**function** **awesomeFunction**(){

**var** a = 2;

**var** multiplyBy2 = **function**(){

console.log(a\*2); // Can access variable "a" since a and multiplyBy2 both are written inside the same function

}

}

console.log(a); // Throws reference error since a is written in local scope and cannot be accessed outside

multiplyBy2(); // Throws reference error since multiplyBy2 is written in local scope

**Block Scope:**Block scope is related to the variables declared using let and const. Variables declared with var do not have block scope. Block scope tells us that any variable declared inside a block { }, can be accessed only inside that block and cannot be accessed outside of it.

{

**let** x = 45;

}

console.log(x); // Gives reference error since x cannot be accessed outside of the block

**for**(**let** i=0; i<2; i++){

// do something

}

console.log(i); // Gives reference error since i cannot be accessed outside of the for loop block

**Scope Chain:**

"In JavaScript, a **scope chain** determines how variables are resolved. When a variable is referenced inside a function, the engine looks for it in the current scope. If it's not found, it moves up to the parent scope, and continues up the chain until it reaches the global scope. This process of moving outward through the nested scopes is what we call the *scope chain*."

**var** y = 24;

**function** **favFunction**(){

**var** x = 667;

**var** anotherFavFunction = **function**(){

console.log(x); // Does not find x inside anotherFavFunction, so looks for variable inside favFunction, outputs 667

}

**var** yetAnotherFavFunction = **function**(){

console.log(y); // Does not find y inside yetAnotherFavFunction, so looks for variable inside favFunction and does not find it, so looks for variable in global scope, finds it and outputs 24

}

anotherFavFunction();

yetAnotherFavFunction();

}

favFunction();

**As you can see in the code above, if the javascript engine does not find the variable in local scope, it tries to check for the variable in the outer scope. If the variable does not exist in the outer scope, it tries to find the variable in the global scope.**

If the variable is not found in the global space as well, a reference error is thrown.

 14)What are the features of JavaScript?

These are the features of JavaScript:

* Lightweight, interpreted programming language
* Dynamically typed language
* Cross-platform compatible
* Open-source
* Object-oriented
* Integration with other backend and frontend technologies
* Used especially for the development of network-based applications

## 15) What is the Difference between Java and JavaScript?

Java is an OOP programming language, and it helps to create applications that function in a virtual machine or browser, while JavaScript is an OOP scripting language. Also, the JavaScript code runs on a browser only.

#### 16)Explain rest parameter and spread operator.

The ES6 version of Javascript introduced the rest parameter and spread operator.

**Rest Parameter**

The use of three dots (...) before any parameter shows a rest parameter. This improves the handling of function parameters. With the help of the rest parameter, we can create such functions that can take a different number of arguments. By using the rest parameter, all these arguments will be converted into arrays. The rest parameter can also be used to extract any or all parts of an argument.

1. 16) **Debounce Function:** Implement a debounce function that delays invoking a given function until after n milliseconds have elapsed since the last time the debounced function was invoked.
2. **Async/Promise Handling:** Write a function that takes an array of promises and resolves them sequentially, one after another, without using any external libraries like **async** or **Promise.all**.
3. **Memoization:** Implement a memoization function that caches the results of expensive function calls and returns the cached result when the same inputs occur again.

Closures and Scope:

**17) CURRYING FUNCTION**

It is a technique in functional programming, that transforms the function of multiple arguments into several functions of a single argument in sequence.

The translation of function happens something like this,

function simpleFunction(param1, param2, param3, .....) => function

curriedFunction(param1)(param2)(param3)(....

We simply wrap a function inside a function, which means we are going to return a function from another function to obtain this kind of translation. The parent function takes the first provided argument and returns the function that takes the next argument and this keeps on repeating till the number of arguments ends. Hopefully, the function that receives the last argument returns the expected result.

**function** calculateVolume(length) {

**return** **function** (breadth) {

**return** **function** (height) {

**return** length \* breadth \* height;

        }

    }

}

console.log(calculateVolume(4)(5)(6));

**18) Discuss lexical scope and how it affects the behavior of closures.**

* Lexical Scoping is the scope of variable and function based on where it is defined in source code.
* It determines the accessibility of variable and function with a program.
* A function scope’s ability to access variables from the parent scope is known as lexical scope. We refer to the parent function’s lexical binding of the child function as “lexically binding.
* The inner function has access to its own scope (local variables), the scope of its outer function (enclosing variables), and the global scope.

If we are defining the function inside another function then the inner function can access to all it’s outer function variables this is due to lexical scoping.

Ex :- function outerFunction() {

Let var = “Hello”;

Function innerFunction() {  
clog(“function B”);

}

}

Here we can say that the innerFunction has access to all variable declared in outerFunction()’s lexical scope.

|  |
| --- |
| let name = "John"; // Global variable    **function** sayHello() {        console.log("Hello " + name);  }    sayHello(); // Output: "Hello John" |

The variable name is declared in the global scope, outside of any function. Therefore, it is accessible to all functions defined within the global scope.

The function sayHello() is defined within the same lexical scope as the variable name. This means that sayHello() has access to the variable name due to lexical scoping rules.

When sayHello() is invoked, it references the variable name directly. JavaScript looks up the variable name in its lexical scope chain, finds it in the global scope, and outputs "Hello John" to the console.

**19) What is Closure?**

**closure** is a feature that allows inner functions to access the outer scope of a function

When a function is defined within another function, the inner function has access to the outer function's variables and parameters, even after the outer function has finished executing.

In this Code We are using Closure

If we use the innerfunction inside the outerfunction then in order to access the innerfunction we have to create another variable (inner) outside the outer funciton and call the outer function and call the variable( inner() ). This is the way to access the innerFuncition.

This is the technique of closure.

The outerFunction defines a variable outVar and declares an inner function innerfunction.

The outerFunction then returns innerfunction.

The returned innerfunction is assigned to the variable inner.

Finally, the inner function is invoked.

In this scenario, a closure is created when the innerfunction is returned from outerFunction. The closure retains a reference to the outVar variable, even though outerFunction has finished executing. This means that when inner is called later, it still has access to the outVar variable, which is part of the outer lexical scope.

When innerfunction is defined inside outerFunction, it captures and retains a reference to the outVar variable from the lexical scope of outerFunction.

Even though outerFunction has finished executing and its execution context is no longer active, the innerfunction still maintains access to the outVar variable through the closure.

Therefore, when inner is called later, it can still access and use the outVar variable, demonstrating the concept of closure in action.

function outerFunction(){

    let outVar = "The Example of outer var";

    console.log("Outer function is working");

    function innerfunction() {

        console.log("This is inner function");

    }

    return innerfunction;

}

let inner = outerFunction();

inner();

But if I want to access the two nested function of outer Function then We can do it this way:

CASE 2:

Here we are returning a object so if we use previous technique then it will throw the error that ibnner is not a function as it is a object.

function outerFunction(){

    let outVar = "The Example of outer var";

    console.log("Outer function is working");

    function innerfunction() {

        console.log(outVar)

        console.log("This is inner function");

    }

    function secondFunction() {

        console.log("This is second function");

    }

    return {innerfunction, secondFunction};

}

let inner = outerFunction();

inner.innerfunction();

inner.secondFunction();

CASE 3:

Here We have used inner function outside the outer function so it will run

Directly innerFunction when we call outer function.

function outerFunction(){

    let outVar = "The Outer variable is working";

    innerFunction();

}

function innerFunction(){

    console.log("The inner Function is working");

}

outerFunction();

**20) Describe how you handle multiple asynchronous operations in parallel and sequentially using promises and async/await.**

Solution :- By using Promise.all and async Await and .then

**Event Loop and Concurrency Model:**

**Explain the event loop in JavaScript and how it handles asynchronous operations.**

**Discuss the concurrency model of JavaScript and how it differs from other programming languages.**

**Promises and Async/Await:**

**20) Prototype and Inheritance:**

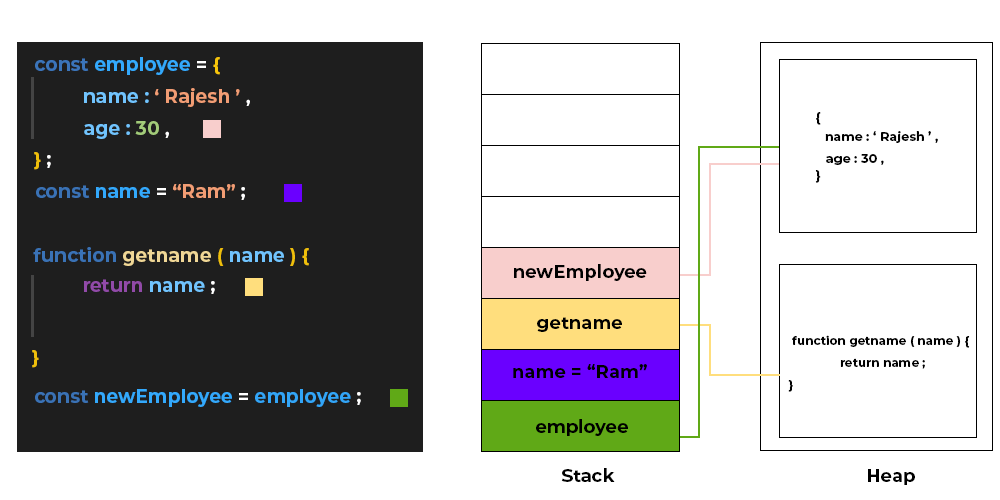
JavaScript is a prototype based language, so, whenever we create a function using JavaScript, JavaScript engine adds a prototype property inside a function, **Prototype property** is basically an object (also known as Prototype object), where we can attach methods and properties in a prototype object, which enables all the other objects to inherit these methods and properties.

**21) Explain how memory is managed in JavaScript, including concepts like garbage collection and memory leaks?**

## **JavaScript engines have two places to store data**

* **Stack:**It is a data structure used to store static data. Static data refers to data whose size is known by the engine during compile time. In JavaScript, static data includes primitive values like strings, numbers, boolean, null, and undefined. References that point to objects and functions are also included. A fixed amount of memory is allocated for static data. This process is known as **static memory allocation.**
* **Heap:**It is used to store objects and functions in JavaScript. The engine doesn’t allocate a fixed amount of memory. Instead, it allocates more space as required.

| **Stack** | **Heap** |
| --- | --- |
| Primitive data types and references | Objects and functions |
| Size is known at compile time | Size is known at run time |
| Fixed memory allocated | No limit for object memory |



***Memory Management in JavaScript***

## **Garbage Collection**

Garbage collectors are used in releasing memory. Once the engine recognizes that a variable, object, or function is not needed anymore, it releases the memory it occupied. The main issue here is that it is very difficult to predict accurately whether a particular variable, object, or function is needed anymore or not. Some algorithms help to find the moment when they become obsolete with great precision.

 **Mark-and-Sweep**: Most widely used. The engine "marks" all reachable objects, and anything unmarked is collected.

 **Reference Counting** (less common now): Objects with a reference count of zero are collected. Prone to issues with circular references.

Discuss strategies for optimizing JavaScript performance, such as minimizing DOM manipulation, reducing reflows and repaints, and optimizing algorithms.

**1. Minimize DOM Manipulation**

DOM operations are expensive because they often cause layout recalculations and repaints.

* I try to **batch DOM updates** using techniques like DocumentFragment.
* I avoid manipulating the DOM inside loops.

const fragment = document.createDocumentFragment();

for (let i = 0; i < 1000; i++) {

const div = document.createElement('div');

fragment.appendChild(div);

}

document.body.appendChild(fragment);

**Optimize Algorithms**

Efficient logic is crucial, especially in loops or data-heavy applications.

* I prefer using Set, Map, and built-in array methods like filter or reduce.
* I avoid nested loops and aim for **O(n)** complexity when possible.

**Error Handling and Debugging:**

Error handling and debugging are essential for building reliable applications. In JavaScript, I use a mix of **structured error handling** (like try...catch) and **debugging tools** to identify and fix issues efficiently.

**Using Browser Debugging Tools**

I heavily use:

* **Chrome DevTools** for stepping through code
* **Breakpoints** to pause execution
* **Call stacks** to trace the error origin
* **Network tab** for inspecting API requests

Error handling and Debugging annd

**Describe different types of errors in JavaScript and how they can be handled using try-catch blocks.**

ReferenceError

TypeError

**SyntaxError**

RangeError

22) **Explain the difference between null and undefined in JavaScript.**

Null is an assigned value. It means nothing. undefined means a variable has been declared but not defined yet.

23) Èxplain the Concat() :-

The concat() method in JavaScript is used to concatenate (join together) two or more arrays or values to create a new array. It doesn't modify the original arrays; instead, it returns a new array containing the concatenated elements.

REVERSE THE ARRAY ……………………………..

let arr = [4,5,6,7,8];

let reverseArray = arr.reduce((acc, curr) =>

[curr].concat(acc), []

)

console.log(reverseArray);

FLATERRING THE ARRAY……………………………….

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

let flattenedArray = [].concat(...arrayOfArrays);

console.log(flattenedArray);

**24) {} [] parenthesis difference and use of {} is reduce and not in reduce?**

Brackets []: Allow you to access an object's property using a variable. This is necessary when the property name is not known until runtime.

acc[curr] = (acc[curr] || 0) + 1;

•In this line, curr is a variable that holds the name of the current fruit. By using acc[curr], you dynamically access the property of the acc object that corresponds to the value of curr.

•Parentheses (): Used to invoke a function. If you try to use acc(curr), JavaScript interprets this as an attempt to call acc as a function, which leads to the error because acc is not a function but an object.

**Using reduce to Transform into an Object:**

let maxValues = items.reduce((acc, curr) => { ... }, {});

reduce iterates over each object in the items array.

The callback function (acc, curr) => { ... } processes each object.

acc (accumulator): An object that will store the maximum values for each category.

curr (currentValue): The current object being processed.

**25) Type of Accumulator in Reduce methods ?**

Different scenario has different type of accumulator.

The type of the accumulator in the reduce function is determined by the initial value you provide and how you manipulate it in the callback function. Here are some common types:

* Number: For operations involving numerical calculations.
* Object: For counting occurrences, grouping by properties, or other structured data manipulations.
* Array: For accumulating arrays or flattening nested arrays.
* Set: For creating collections of unique values.
* String: For concatenating strings or building complex strings.

The reduce method in JavaScript can accumulate results into various types of initial values. Here’s how it works with both objects {} and arrays []:

**26) 9 Use Cases for Array.reduce()**:

Use Case 1: Summing Numbers

One of the most straightforward use cases for reduce() is summing up a bunch of numbers. Let's say you have an array of integers and you want to find the total sum.

const numbers = [1, 2, 3, 4, 5];

const sum: number = numbers.reduce((acc, curr) => acc + curr, 0);

console.log(sum); // Output: 15

Boom! With just one line of code, you've calculated the sum of all elements in the array. The initial value of the accumulator is set to 0, and in each iteration, we add the current element to the accumulator.

Use Case 2: Flattening Arrays

Have you ever found yourself with an array of arrays and thought, "I wish I could flatten this into a single array"?

const nestedArray: number[][] = [[1, 2], [3, 4], [5, 6]];

const flattenedArray: number[] = nestedArray.reduce((acc, curr) => acc.concat(curr), []);

console.log(flattenedArray); // Output: [1, 2, 3, 4, 5, 6]

In this example, we start with an empty array as the initial.

Use Case 3: Counting Occurrences of Elements in an Array.

In this example, we initialize an empty object as the accumulator. For each fruit in the array, we check if it already exists as a property in the accumulator object. If it does, we increment its count by 1; otherwise, we initialize it to 1. The result is an object that tells us how many times each fruit appears in the array.

// ......................maximum number of occurence.....................

const fruits = ["apple", "orange", "apple", "orange", "mango", "mango",  "mango", "watermelon", "papaya"];

let container = fruits.reduce((acc, curr) => {

    acc[curr] = (acc[curr] || 0) + 1;

    return acc;

},{}

)

console.log(container);

Use Case 4: Removing the Duplicate

// Removing the Duplicate

const duplicate = numbers.reduce((acc,curr) => {

if(!acc.includes(curr)){

    acc.push(curr);

}

return acc;

},[]

)

console.log(duplicate);

Use Case 5:

To create a lookup map using reduce(), you can transform an array of objects into an object where each key is a unique identifier (such as an ID) and the corresponding value is the object itself. This allows for constant time complexity when accessing elements by their unique identifier.

const products = [

    { id: 1, name: 'Laptop', price: 999 },

    { id: 2, name: 'Phone', price: 699 },

    { id: 3, name: 'Tablet', price: 499 }

  ];

  const productMap = products.reduce((acc, curr) => {

acc[curr.id] = curr;

return acc;

  },{}

)

const laptop = productMap[3];

console.log(laptop);

Use Case 6:

Caculating the Total cost of item in the shopping list

const cart = [

    { item: 'apple', price: 1.0, quantity: 3 },

    { item: 'banana', price: 0.5, quantity: 5 },

    { item: 'orange', price: 1.25, quantity: 2 }

  ];

  const cartCount = cart.reduce((acc, curr) => acc + curr.price \* curr.quantity,0);

  console.log(cartCount);

**27) What is Debouncing and throttling ?**

**Debouncing and Throttling**

Debouncing and throttling are techniques used to control the rate at which a function is executed, particularly in response to events like user inputs or window resizing. They help improve performance and enhance user experience by preventing excessive or unnecessary function calls.

**Debouncing:**

Debouncing ensures that a function is only called once after a specified period of inactivity. This means that if an event keeps occurring, the function will only be executed after the event stops for the specified wait time.

Debounce is a programming technique used to ensure that a function is only executed once after a specified period of inactivity. When multiple events are triggered in quick succession, debouncing delays the execution of the function until a certain amount of time has passed without any new events occurring.

In the context of debouncing, a **"specified period of inactivity"** refers to a designated amount of time during which no new events occur. Only after this period of inactivity has passed without any new events, the debounced function is executed.

**Use Case:**

Search Bar Suggestions: When a user types in a search bar, an API request should only be sent after the user has stopped typing for a short period (e.g., 300 milliseconds). This prevents a flood of API requests.

**Analogy: Answering the Phone:**

Imagine you're at home, and the phone starts ringing. You don't want to run to the phone immediately because sometimes people hang up quickly or it might be a wrong number. Instead, you decide to wait a few seconds to see if the ringing continues. If the phone stops ringing before the few seconds are up, you don't answer it. If it keeps ringing beyond those few seconds, you then decide to pick up and answer the call.

**In the Context of Debouncing:**

Ringing Phone: The continuous ringing is like a series of events (e.g., keystrokes in a search bar).

Waiting a Few Seconds: This waiting period is the debounce time.

Answering the Phone: You only answer the phone if it rings continuously for those few seconds, similar to how a debounced function only executes after the events stop for a specified time.

**Real-Life Use Case: Typing in a Search Bar**

When you type in a search bar, you want to fetch suggestions from a server. Without debouncing, every single keystroke would send a request to the server. This can be inefficient and overwhelming, especially if someone types quickly.

**Explanation:**

**Problem Without Debouncing**

Typing "debounce" without Debouncing:

User types "d" → Request sent to server.

User types "e" → Request sent to server.

User types "b" → Request sent to server.

User types "o" → Request sent to server.

User types "u" → Request sent to server.

User types "n" → Request sent to server.

User types "c" → Request sent to server.

User types "e" → Request sent to server.

This results in 8 separate requests to the server for a single query, which is inefficient and can lead to performance issues.

**Solution With Debouncing**

With Debouncing:

You start typing "debounce".

Each keystroke restarts the debounce timer.

User types "d" → Timer starts for 300 milliseconds.

User types "e" within 300 milliseconds → Timer restarts.

User types "b" within 300 milliseconds → Timer restarts.

User continues typing "o", "u", "n", "c", "e", each time restarting the timer.

Only after you stop typing for a set amount of time (e.g., 300 milliseconds), the function sends a single request to fetch suggestions.

Outcome:

The user finishes typing "debounce" and stops.

After 300 milliseconds of no typing activity, the function sends a single request to the server to fetch suggestions.

**Throttling:**

Throttling ensures that a function is called at most once in a specified period. This means that if an event keeps occurring, the function will be executed at regular intervals, no matter how many times the event occurs.

Throttling in Simple Words

Throttling is a technique to control the frequency at which a function is executed. It ensures that a function is called at most once in a specified time period, no matter how many times an event occurs.

Real-Life Analogy: Checking Your Phone

Imagine you have a rule to check your phone for messages only once every 10 minutes, no matter how many messages you receive. Even if you receive multiple notifications within those 10 minutes, you will only check your phone once in that period.

Use Case: Scrolling a Webpage

When you scroll a webpage, certain actions might need to occur, like updating the navigation bar or loading more content. Without throttling, these actions could be triggered excessively, causing performance issues.

Throttling Example: Scroll Event Handling

Scenario:

You want to update something on the webpage as the user scrolls, but you only want to do it at most once every 100 milliseconds to avoid performance issues.

**29) What is Memorization ?**

Memoization is an optimization technique used in computing to speed up function calls by caching the results of expensive function calls and returning the cached result when the same inputs occur again.

You can then use this memoize function to wrap other functions that you want to memoize. For example:

function expensiveOperation(n) {

console.log("Computing expensiveOperation for", n);

return n \* 2;

}

const memoizedExpensiveOperation = memoize(expensiveOperation);

console.log(memoizedExpensiveOperation(5)); // Calls expensiveOperation(5) and returns 10

console.log(memoizedExpensiveOperation(5)); // Returns the cached result 10 without calling expensiveOperation again

In this example, expensiveOperation is a function that performs a costly computation. By wrapping it with memoize, the result of the function call for a particular input is stored in the cache the first time it's called with that input. Subsequent calls with the same input retrieve the result from the cache instead of re-computing it. This can significantly improve the performance of the function, especially if it's called multiple times with the same arguments.

30) **What is Spread Operator ?**

When I write the Spread operator inside the curly bracket then it is copying the property into the object .

If I write it in square bracket then It is copying to array.

Object

const newObj = {...obj, d:4};

Array

const mergedArray = [...array1, ...array2];

**31)Spread Operator use Case :-**

1................................

**Converting String to Array.**

const str = "anup";

const strArr = [...str];

console.log(strArr);

2...............................

**original array is copied to duplicate**

const original = [1,2,3,4,5];

const duplicate = [...original];

console.log(duplicate);

3................................

**merging the two array**

const array1 = [1, 2, 3];

const array2 = [4, 5, 6];

const mergedArray = [...array1, ...array2];

console.log(mergedArray); // Output: [1, 2, 3, 4, 5, 6]

4……………………………………………………………….

Merging the two object

 Combines multiple objects into one.

const object1 = { a: 1, b: 2 };

const object2 = { c: 3, d: 4 };

const mergedObject = { ...object1, ...object2 };

console.log(mergedObject); // Output: { a: 1, b: 2, c: 3, d: 4 }

5………………………………………………. Adding Properties to Objects

Adds new properties to an object while copying existing properties.

javascript

Copy code

const originalObject = { a: 1, b: 2 };

const newObject = { ...originalObject, c: 3 };

console.log(newObject); // Output: { a: 1, b: 2, c: 3 }

**32) What is Shallow Copy and Deep Copy ?**

A shallow copy means that the top-level elements of the array (or object) are copied, but nested elements (such as arrays or objects within the array) are not copied by value; instead, references to these nested elements are copied. This means that changes to nested elements in the copied array will affect the original array and vice versa.

Shallow Copy can be archived by Object.assign and spread operator in dono case me top level property copy hoti hai lekin nested property copy nahi hoti.lekin ye refenced hai.

const obj = {a:2, b:4, c:5, d:{d1:1}};

const Copy1 = {...obj};

// Copy1.d.d1 = 2;

Copy1.c = 2;

console.log(obj);

console.log(Copy1);

Output:

{ a: 2, b: 4, c: 5, d: { d1: 1 } }

{ a: 2, b: 4, c: 2, d: { d1: 1 } }

Here I changed the top level Property (c) so that’s why it change the property c in copy1 not in original But if I would do changes in d1 then it will reflect in the original obj also. So this is Shallow copy.

Example of Array :

const arr = [1,2,3,4,5,6,[7,8]];

const Copy2 = [...arr];

 Copy2[6][0] = 10;

 console.log(arr);

 console.log(Copy2);

**Methods to Create a Shallow Copy:**

**Using Object.assign for objects:**

 const original = { a: 1, b: { c: 2 } };

const shallowCopy = Object.assign({}, original);

**Using Spread syntax for objects:**

 const shallowCopy = { ...original };

**Using Array.prototype.slice:**

 const originalArray = [1, 2, [3, 4]];

 const shallowCopyArray = originalArray.slice();

**Using Spread syntax for Arrays:**

const shallowCopy = [ ...original ];

In the context of shallow and deep copying, a "reference" refers to the way JavaScript handles objects and arrays in memory. When an object or an array is copied by reference, it means that both the original and the copied variables point to the same underlying data in memory. Changes made to the nested objects or arrays will be reflected in both the original and the copy because they share the same reference.

**Deep Copy:**

A deep copy of an object or array copies all properties, including nested objects and arrays. This means that the nested objects and arrays are recursively copied, ensuring that the new object is entirely independent of the original.

const arr = [1,2,3,4,5,6,[7,8]];

const Copy2 = JSON.parse(JSON.stringify(arr));

 Copy2[6][0] = 10;

 console.log(arr);

 console.log(Copy2);

Methods of Deep Copy:

Using JSON.parse and JSON.stringify:

 const original = { a: 1, b: { c: 2 } };

 const deepCopy = JSON.parse(JSON.stringify(original));

The JSON.stringify and JSON.parse methods work together to create a deep copy of an array or object because they serialize and then deserialize the entire structure, including nested objects and arrays. Here’s why this approach works:

How JSON.stringify and JSON.parse Create a Deep Copy:

**Serialization:**

JSON.stringify converts the entire structure (array or object) into a JSON string.

This process traverses the entire structure, including nested objects and arrays, and represents them as a JSON string.

**Deserialization:**

JSON.parse takes the JSON string and reconstructs a new array or object from it.

The result is a completely new structure with no references to the original nested objects or arrays.

**Why It Works**

Complete Traversal: JSON.stringify goes through the entire structure, so all nested objects and arrays are included in the JSON string.

Reconstruction: JSON.parse creates a new object or array based on the JSON string, ensuring that all nested elements are new copies and not references to the original

**33) What is Reference Data Type ?**

In programming, a reference data type is a type of data where the variable doesn't store the actual data directly. Instead, it stores a reference (or memory address) to the location where the actual data is stored. This means that when you work with a reference data type, you're manipulating the reference to the data, not the data itself.

Example in JavaScript:

Consider the following example with objects, which are reference data types in JavaScript:

let person1 = {

    name: "Alice",

    age: 25

  };

  let person2 = person1; // person2 is now a reference to the same object as person1

  // Modifying the object using person2

  person2.age = 30;

  console.log(person1.age); // Output: 30

  console.log(person2.age); // Output: 30

Sure, let's delve into the concept of memory management and how reference data types work in the context of memory in programming.

**34)** **Memory Management in Programming**

In programming, memory management is crucial for the efficient allocation, use, and release of memory during the execution of programs. Memory is divided into two main areas:

Stack: The stack is used for static memory allocation. It stores primitive data types (like integers, floats, characters) and references to objects and arrays. The stack operates in a last-in, first-out (LIFO) manner, making it very fast for managing memory.

Heap: The heap is used for dynamic memory allocation. It stores objects and arrays themselves. Memory in the heap can be allocated and freed in an arbitrary order, which makes it flexible but also requires more management to avoid fragmentation and memory leaks.

**Reference Data Types and Memory**

Creating an Object in Memory

When you create an object in a programming language like JavaScript, the following steps occur:

**Allocation on the Heap**: The object is created in the heap because its size may not be known until runtime, and it can grow dynamically.

**Reference in the Stack:** A reference (or pointer) to the location of the object in the heap is stored in the stack.

 let person = {

    name: "Alice",

    age: 25

  };

Heap Memory: The actual object { name: "Alice", age: 25 } is stored in the heap.

Stack Memory: The variable person stores a reference to the object's location in the heap.

Copying References

When you copy a reference type, you copy the reference (memory address), not the actual object. This means both variables point to the same location in the heap.

Example: Copying an Object Reference

let person1 = {

    name: "Alice",

    age: 25

  };

  let person2 = person1; // person2 now references the same object as person1

Heap Memory: Only one object { name: "Alice", age: 25 } exists.

Stack Memory: Both person1 and person2 hold the same reference to the object in the heap.

**Deep Copying**

**To create a truly independent copy of a reference data type, a deep copy is needed. A deep copy involves creating a new object in the heap with the same properties and values as the original**.

Example: Deep Copy

 let person1 = {

    name: "Alice",

    age: 25

  };

  let person2 = JSON.parse(JSON.stringify(person1)); // Deep copy of person1

  person2.age = 30;

  console.log(person1.age); // Output: 25

  console.log(person2.age); // Output: 30

Heap Memory: Two separate objects now exist. person1 is { name: "Alice", age: 25 } and person2 is { name: "Alice", age: 30 }.

Stack Memory: person1 and person2 reference different objects, so changes to one do not affect the other.

Summary

Primitive Types: Stored directly in the stack. Copying creates independent values.

Reference Types: Stored in the heap, with references in the stack. Copying creates new references pointing to the same object.

Deep Copy: Involves creating a completely new object in the heap, ensuring independence from the original.

### 35) Is there any difference between the postfix and prefix increment/decrement positions?

**Interview Response:** Prefix increments the counter and returns the new value. Postfix increments the counter but returns the old value before being incremented.

let counterOne = 5;

let a = ++counterOne; // (\*)

console.log(a); // console.logs 6

// Postfix Position:

let counterTwo = 5;

let b = counterTwo++; // (\*) changed ++counter to counter++

console.log(b); // console.logs 5

**35)What is the comma Operator? And Practical Use Case?**

When multiple expressions are separated by commas within a single statement in JavaScript, all the expressions are evaluated from left to right, and the value of the entire comma-separated expression is the value of the last expression. This is known as the comma operator.

Detailed Explanation

Evaluation Order: Each expression is evaluated in the order they appear, from left to right.

Side Effects: Any side effects from the evaluation of these expressions (like modifying a variable) will occur.

Return Value: The result of the whole comma-separated expression is the value of the last expression.

let a = 1;

let b = 2;

let c;

c = (a += 1, b += 2, a + b);

console.log(c); // Outputs: 6

for (let i = 0, j = 10; i < j; i++, j--) {

    console.log(i, j);

}

**Practical Use Case in a for Loop**

A common scenario where the comma operator is used is in the update expression of a for loop, allowing you to update multiple variables:

In this loop:

The initialization expression let i = 0, j = 10 initializes two variables, i and j.

The condition i < j is checked before each loop iteration.

The update expression i++, j-- increments i and decrements j after each iteration.

**34) What is NaN?**

In JavaScript, dividing zero by zero results in NaN, which stands for "Not-a-Number". This special value represents an undefined or unrepresentable mathematical operation.

Code Example:

Here's a code example that demonstrates what happens when you divide zero by zero in JavaScript:

let result = 0 / 0;

console.log(result);  // Output: NaN

// Checking the type of the result

console.log(typeof result);  // Output: "number"

// Checking if the result is NaN using Number.isNaN

console.log(Number.isNaN(result));  // Output: true

### 35) What happens when you divide zero by zero in JavaScript?

Nan

**36) How does JavaScript compare strings to see if their greater or less than another?**

Interview Response: In JavaScript, strings are compared lexicographically, which means that the characters in the strings are compared one by one in order until a difference is found. The comparison is based on the Unicode values of the characters.

console.log('Z' > 'A'); // true

console.log('Glow' > 'Glee'); // true

console.log('Bee' > 'Be'); // true

// Unicode Values

let myLetter = 'Hello';

console.log(myLetter.charCodeAt(0)); // returns Unicode value 72

console.log(myLetter.charCodeAt(1)); // returns Unicode value 101

When comparing values of different types, does JavaScript convert the values to numbers?

Interview Response: Yes, when comparing values of different types, it converts the values to numbers. For Boolean values, true becomes 1 and false becomes 0.

Code Example:

console.log('2' > 1); // true, string '2' becomes a number 2

console.log('01' == 1); // true, the string '01' becomes a number 1

### 37) Hoisting in function :

The process in which the declaration is moved to the top level of the code.

### Function Declaration

Function declarations are hoisted, meaning you can call the function before it is defined in the code. Here's an example:

javascript

Copy code

sayHello(); // This works because function declarations are hoisted

function sayHello() {

console.log("Hello, world!");

}

In the above example, sayHello can be called before its declaration because function declarations are hoisted to the top of their scope.

### Function Expression

Function expressions are not hoisted. This means if you try to call the function before the expression is evaluated, you will get an error. Here's an example:

try {

sayHello(); // This will throw an error

} catch (error) {

console.error(error); // ReferenceError: Cannot access 'sayHello' before initialization

}

const sayHello = function() {

console.log("Hello, world!");

};

In the above example, calling sayHello before it is assigned to a function expression will result in a ReferenceError.

### Arrow Function

Arrow functions, like function expressions, are not hoisted. Here's an example:

javascript

Copy code

try {

sayHello(); // This will throw an error

} catch (error) {

console.error(error); // ReferenceError: Cannot access 'sayHello' before initialization

}

const sayHello = () => {

console.log("Hello, world!");

};

In the above example, calling sayHello before it is assigned to an arrow function will result in a ReferenceError.

**38) Difference between function expression and function declaration ?**

* **Function Expressions**:
  + Not hoisted, must be defined before use.
  + Can be anonymous or named.
  + Useful for conditional function definitions and passing functions as arguments.
* **Function Declarations**:
  + Hoisted, can be used before they are defined.
  + Always have a name.
  + Useful for defining functions that need to be accessible throughout their scope.

**39) Event Delegation :**

**Event Delegation:**

* **Definition:** Event delegation refers to the technique of attaching event listeners to a parent element rather than to the individual child elements.
* **How it works:** When an event occurs on a child element (e.g., a button inside a <div>), the event bubbles up or propagates to its parent elements.
* **Advantages:**
  + Simplifies event handling, especially for dynamically added elements.
  + Reduces memory consumption by having fewer event listeners.
  + Allows handling of events on elements that are added later (dynamically) without needing to add new event listeners.
* **Example:** Suppose you have a list (<ul>) with list items (<li>). Instead of adding a click event to each <li>, you can add a single event listener to the <ul> and determine which <li> was clicked based on the event target.

**40) What is Event propagation ?**

Event Propagation determines in which order the elements receive the event. There are two ways to handle this event propagation order of HTML DOM is Event Bubbling and Event Capturing.

**Event Bubbling Explained:**

When you click on an element in a web page, like a button, the click event doesn't just happen on that button alone. It also triggers on all of its parent elements up through the DOM tree, all the way to the <html> element.

**Example:**

Imagine you have a button inside a few nested <div> elements:

html

Copy code

<div id="outer">

Outer Div

<div id="middle">

Middle Div

<div id="inner">

Inner Div with Button

<button id="myButton">Click me!</button>

</div>

</div>

</div>

If you click the button, the click event will trigger in this order:

1. **Button (<button id="myButton">)**: This is where the click event starts.
2. **Inner Div (<div id="inner">)**: The event then bubbles up to the parent of the button.
3. **Middle Div (<div id="middle">)**: It continues bubbling up to the next parent.
4. **Outer Div (<div id="outer">)**: Finally, it reaches the outermost parent element.

**Why Event Bubbling Matters:**

Understanding event bubbling is crucial because it allows you to handle events more efficiently. Instead of attaching event listeners to each individual element, you can attach them to a common ancestor (like #outer in this example) and use JavaScript to determine which specific child element triggered the event. This technique is known as **event delegation** and is very useful for dynamically generated content or when you have many similar elements.

**What is Event capturing ?**

**Capturing**

It is the process in which the event starts from parent to targeted element.

It is the opposite of bubbling. The event handler is first on its parent component and then on the component where it was actually wanted to fire that event handler. In short, it means that the event is first captured by the outermost element and propagated to the inner elements. It is also called trickle down.

**3 Types / Phases of Event Propagation**

| **Phase** | **Description** |
| --- | --- |
| **1. Capturing** | Event starts at the window and travels **down** to the target element |
| **2. Target** | The event reaches the **target element** (e.g., button clicked) |
| **3. Bubbling** | Event then **bubbles back up** from the target to the window |

**What is Event Pooling?**

* **Definition**: Event pooling refers to React's practice of reusing synthetic event objects rather than creating new ones for each event. This is achieved by pooling event objects in memory and reusing them to reduce the overhead of object creation and garbage collection.

**How It Works**

1. **Synthetic Events**: When an event occurs, React creates a synthetic event object that wraps the native DOM event. This object contains normalized properties for consistent behavior across different browsers.
2. **Event Pooling**: After the event handler finishes executing, React does not immediately destroy the event object. Instead, it returns the event object to a pool for reuse in future events.
3. **Event Persistency**: If you need to access the event object asynchronously (e.g., in a setTimeout or Promise), you must call event.persist() to prevent React from reusing or resetting the event object.

**Benefits**

* **Performance**: Reduces the overhead associated with creating and disposing of event objects, leading to better performance and lower memory usage.
* **Efficiency**: Helps manage large numbers of events more efficiently, especially in applications with many interactive components.

**41) Local Storage and Session Storage and Cookies ?**

| **Local Storage** | **Session Storage** | **Cookies** |
| --- | --- | --- |
| The storage capacity of local storage is 5MB/10MB | The storage capacity of session storage is 5MB | The storage capacity of Cookies is 4KB |
| As it is not session-based, it must be deleted via javascript or manually | It’s session-based and works per window or tab. This means that data is stored only for the duration of a session, i.e., until the browser (or tab) is closed | Cookies expire based on the setting and working per tab and window |
| The client  can read and write local storage | The client can read and write session storage | Both clients and servers can read and write the cookies |
| There is no transfer of data to the server | There is no transfer of data to the server | Data transfer to the server is exist |
| Supported by all browsers, including older ones. | Supported by all browsers, including older ones | It is supported by all the browser including older browser |

**Difference between Object and keys ?**

* Map:
  + Supports any data type for keys.
  + Maintains the insertion order of keys.
  + Provides built-in methods for iteration and size.
  + Generally better performance for dynamic operations.
* Plain Object:
  + Keys are always strings or symbols.
  + Key order may not be consistent (though modern engines typically maintain insertion order).
  + Requires additional methods to compute size and iterate over keys.
  + Suitable for simpler or static key-value scenarios.

**Asyc and Awaits ?**

These are used to handle the asynchrounous behaviuor of the code.

** async:** This keyword is used to define a function that will always return a promise. It allows you to write asynchronous code in a way that looks synchronous.

 **await:** This keyword is used inside async functions to pause the execution of the function until the promise is resolved. It makes the code wait for the promise to complete and returns the result.

**What is Factory Function ?**

A **factory function** is a function that returns a new object each time it is called. Unlike constructor functions or classes, factory functions do not use the new keyword and are often used to create multiple instances of an object with a shared structure.

It’s an alternative to using class or constructor functions to create multiple objects with similar structure or behavior.

**Key Characteristics:**

* **Encapsulation**: Factory functions encapsulate the creation logic and provide a way to create multiple instances with similar structure and behavior.
* **No new Keyword**: Unlike classes or constructor functions, factory functions do not require the new keyword to create instances.
* **Flexibility**: They can include more complex logic for initializing objects and can easily handle different configurations.
* // Function creating new objects
* // without use of 'new' keyword
* **function** createRobot(name) {
* **return** {
* name: name,
* talk: **function** () {
* console.log('My name is '
* + name + ', the robot.');
* }
* };
* }
* //Create a robot with name Chitti
* const robo1 = createRobot('Chitti');
* robo1.talk();

* // Create a robot with name Chitti 2.O Upgraded
* const robo2 = createRobot('Chitti 2.O Upgraded');
* robo2.talk();

**Strings:**

**how can these methods used to format user input ?**

**1. Trimming Whitespace**

**Purpose:** Remove any leading or trailing whitespace from the input.

**Method:** trim()

**Example:**

let userInput = ' Hello World ';

userInput = userInput.trim();

console.log(userInput); // Outputs: 'Hello World'

**2.Formatting Phone Numbers**

**Purpose:** Format phone numbers into a standardized format.

**Example:** Using replace() to format a number

3. **Splitting and Joining Input**

**Purpose:** Convert user input into an array and then back into a formatted string.

**Example:** Using split() and join() for formatting a CSV input

**What is Generator Functions ?**

In JavaScript, generator functions are a special kind of function that allows you to control the execution flow and pause/resume it at certain points. Generator functions are defined using function\* the syntax and use the yeild keyword to produce a sequence of values.

When a generator function is called, it returns an iterator called a generator.

Next() is used to resume the generator object.

NOTE: Generator functions can pause execution at each yield statement and resume from that point when the next() method is called on the generator object.

Example :

function\* counter() {

  let count = 0;

  while (true) {

    yield count++;

  }

}

const counterGen = counter();

console.log(counterGen.next().value); // 0

console.log(counterGen.next().value); // 1

console.log(counterGen.next().value); // 2

EXPLANATION:

function\* counter() {

    let count = 0;

    while (true) {

      yield count++;

    }

  }

 **Generator Function Declaration**: function\* counter() defines a generator function. The \* syntax after function indicates that this is a generator function.

 **Variable Initialization**: Inside the generator function, let count = 0; initializes a local variable count with the value 0.

 **Infinite Loop**: while (true) creates an infinite loop. This loop will continue indefinitely, yielding values as long as the generator is active.

 **Yield Statement**: yield count++; is the key operation here. The yield keyword pauses the function’s execution and returns the current value of count. After yielding, count is incremented by 1 due to the post-increment operator ++.

**Generator Object and Iteration**

const counterGen = counter();

* **Create Generator Object**: Calling counter() returns a generator object assigned to counterGen. This object is an iterator that will produce values when its next() method is called.

**Using the Generator**

console.log(counterGen.next().value); // 0

console.log(counterGen.next().value); // 1

console.log(counterGen.next().value); // 2

1. **First Call to next()**:
   * counterGen.next() starts the generator function and runs it until it hits the first yield statement.
   * At this point, count is 0, so yield count++ yields 0 and then increments count to 1.
   * counterGen.next().value returns 0, which is logged to the console.
2. **Second Call to next()**:
   * counterGen.next() resumes execution from where it was paused, just after the first yield.
   * count is now 1, so yield count++ yields 1 and then increments count to 2.
   * counterGen.next().value returns 1, which is logged to the console.
3. **Third Call to next()**:
   * counterGen.next() resumes execution again, with count now being 2.
   * yield count++ yields 2 and then increments count to 3.
   * counterGen.next().value returns 2, which is logged to the console.

**What is Temporal Dead Zone ?**

The Temporal Dead Zone is a phenomenon in JavaScript associated with the use of the let and const keywords, unlike the var keyword. In ECMAScript 6, attempting to access a let or const variable before it is declared within its scope results in a ReferenceError. The term "temporal dead zone" refers to the timeframe during which this occurs, spanning from the creation of the variable's binding to its actual declaration. Let's illustrate this behaviour with an example:

When you declare a variable using let or const, it is **hoisted** to the top of its block scope (like any variable declaration), but it is **not initialized** until the code execution reaches the line where the variable is actually defined.

Until that point (the "temporal dead zone"), accessing the variable will result in a **ReferenceError**. The variable exists in the **TDZ** and is considered in an "uninitialized" state. Once the execution reaches the line where the variable is initialized, the TDZ ends, and the variable can be safely accessed.

IMPORTANT:

when a variable is declared using let or const, it enters the Temporal Dead Zone (TDZ) from the start of its block scope until the variable is initialized. During this TDZ period, the variable is in scope but cannot be accessed. Accessing it before initialization results in a ReferenceError.

function exampleMethod()

{

console.log(value1);

console.log(value2);

var value1 = 1;

let value2 = 2;

}

// Outputs: undefined

// Throws a ReferenceError.

In this example, attempting to access VALUE 2 before its declaration causes a ReferenceError due to the temporal dead zone, while accessing VALUE 1 results in an output of undefined.

**What is Local Storage and Session Storage ?**

Both are key-value storage mechanisms but have different use cases and characteristics.

**Local storage** is the most recent mechanism. It allows for larger amounts of data to be stored, but the data is not deleted when the browser is closed. Local storage is useful for storing data that the user will need to access later, such as offline data.

Local Storage provides about 5MB of storage space per origin (i.e., per domain). This limit can vary slightly between different browsers.

**Session storage** is similar to cookies, but the data is only stored for the current session. This means that the data will be deleted when the user closes the browser. Session storage is useful for storing data that is sensitive, such as login credentials.

Like Local Storage, Session Storage also provides around 5MB of storage space per origin, but the data is only accessible within the current tab or window.

**What is This Keyword ?**

The this keyword in JavaScript is a **special keyword** that refers to the **execution context** in which a function or block of code is running. The value of this depends on how and where it is used.

In JavaScript, this keyword refers to the current object in context. Its value depends on how it’s used: in methods, it refers to the object; in global scope, it refers to the global object and Browser it refers to Windows.

In a JavaScript function, the behavior of the this keyword varies depending on how the function is invoked.

* **Global Context**:  
  If this is used **outside of any function** or **block**, it refers to the **global object**. In a **browser environment**, this refers to the window object, while in **Node.js**, this refers to the global object.

console.log(this); // In a browser, this refers to the window object

* **Inside an Object Method**:  
  If this is used inside an **object method**, it refers to the **object itself**—the object on which the method was called.

const person = {

name: "Alice",

greet: function() {

console.log(this.name); // Here, `this` refers to the `person` object

}

};

person.greet(); // Output: Alice

* **Inside a Function (Global Function)**:  
  If this is used inside a regular function, **the value of this depends on how the function is called**.
  + In **non-strict mode**, it refers to the **global object** (window in browsers, global in Node.js).
  + In **strict mode**, this will be undefined if it's used inside a function.

function regularFunction() {

console.log(this); // In non-strict mode, `this` refers to the global object (window in browsers)

}

regularFunction();

* **InsideArrowFunctions**:  
  Arrow functions behave differently. They do **not** have their own this. Instead, they **inherit** the value of this from their **lexical context**—the surrounding function or block where the arrow function is defined.
* **Explicit Binding (call, apply, bind)**

**How object can be accessed ?**

You can access the properties of an object in JavaScript in 3 ways:

1. Dot property accessor: object.property
2. Square brackets property accessor: object['property']
3. Object destructuring: const { property } = object

**Use of [ ] for accessing the object ?**

**1. Accessing Object Properties**

You can use square brackets to access properties of an object, especially when the property name is stored in a variable or is not a valid identifier.

**Example: Using a Variable**

javascript

Copy code

const person = {

name: 'Alice',

age: 30

};

const property = 'name';

console.log(person[property]); // Output: Alice

* **Explanation**: Here, property is a variable containing the string 'name'. person[property] accesses the name property of the person object.

**Example: Using Non-Standard Property Names**

const obj = {

'first name': 'John',

'last name': 'Doe'

};

console.log(obj['first name']); // Output: John

* **Explanation**: Properties with spaces or special characters need to be accessed using square brackets.

**2. Adding or Modifying Object Properties**

You can use square brackets to dynamically add or modify properties of an object.

**Example: Adding Properties**

javascript

Copy code

const car = {};

car['make'] = 'Toyota';

car['model'] = 'Corolla';

console.log(car); // Output: { make: 'Toyota', model: 'Corolla' }

* **Explanation**: Properties are added to the car object using square brackets and assignment.

**Example: Modifying Properties**

javascript

Copy code

const car = {

make: 'Toyota',

model: 'Corolla'

};

car['model'] = 'Camry';

console.log(car); // Output: { make: 'Toyota', model: 'Camry' }

* **Explanation**: The model property is updated using square brackets.

**3. Using Expressions for Property Access**

Square brackets can also be used with expressions to determine the property name at runtime.

**Example: Dynamic Property Names**

javascript

Copy code

const person = {

firstName: 'John',

lastName: 'Doe'

};

const propName = 'firstName';

console.log(person[propName]); // Output: John

* **Explanation**: The property name is determined by the value of propName.

**Advance topics :**

1. **Definition of a Closure:**

A closure is a function that retains access to its lexical scope, even after the function that created the lexical scope has finished executing. In other words, closures allow functions to capture and remember the environment in which they were defined.

**2.How Closures Maintain State:**

Closures maintain state by preserving access to variables from their outer (enclosing) function, allowing those variables to persist between function calls.

**3.What is Memory Leak ?**

Memory leaks in JavaScript occur when objects are no longer needed but continue to occupy memory because there are still references to them. Closures can contribute to memory leaks by holding onto references of variables from their outer scope even after they are no longer needed. This prevents those variables from being garbage collected, leading to increased memory usage.

**4.Pitfalls of Closure ?**

* 1. **Memory leak:**

When a closure retains references to variables from its outer function, those variables persist in memory for as long as the closure itself is accessible. If not managed properly, this can lead to memory leaks.

Here’s a more detailed explanation with examples:

### ****Example of Memory Leak with Closures****

Consider the following code where a closure keeps a reference to a large object:

function createLargeObject() {

1. let largeObject = new Array(1000000).fill('Large Data'); // Large object
2. return function() {
3. console.log(largeObject.length); // Closure accesses the large object
4. };
5. }
6. const closure = createLargeObject(); // The largeObject is referenced by closure
7. // Even if `createLargeObject` finishes executing, `largeObject` stays in memory

**Explanation:**

* createLargeObject Function: Creates a large object and returns a closure that accesses this object.
* Closure: Holds a reference to largeObject, preventing it from being garbage collected.
* Memory Leak: largeObject remains in memory as long as the closure exists, even though createLargeObject has finished execution.

**How to Prevent it ?**

**Explicitly Nullify References:**

* If a closure is no longer needed, set its references to null to help the garbage collector clean up memory.

function createLargeObject() {

let largeObject = new Array(1000000).fill('Large Data');

return function() {

console.log(largeObject.length);

};

}

let closure = createLargeObject();

closure = null; // Help garbage collector by removing the reference

1. **What is the problem with closures inside loops?**

The problem is that when we use closure in a loop then it only capture the last/final value not the value of each iterations. This can be solved by creating new function inside a loop which will create a new scope of each iterations. This will create new closure which captures the value not just the variable itself.

When you define a function inside another function (e.g., a callback for setTimeout), it forms a closure. The inner function has access to the variables of its outer function.

Why is that happening ?

 The closures created inside the loop do not capture the value of i at each iteration. Instead, they capture a reference to the i variable itself.

Since all the closures share the same reference, they all end up logging the final value of i after the loop finishes.

 When using a loop to create multiple closures (like inside a for loop), all closures created during the loop share the same outer function scope.

 If you use var to declare the loop variable, all closures will reference the same variable. Since var is function-scoped, not block-scoped, the loop variable retains its value after each iteration, and all closures will reference the final value of the loop variable.

**Pitfall of Closure in a for loop with setTimeout:**

* **Issue**: When using setTimeout inside a for loop with var, all the setTimeout callbacks share the same reference to the i variable.
* **Why It Happens**:
  + setTimeout is asynchronous, so the loop finishes executing quickly, and the callbacks are executed after the delay (3000ms).
  + The i variable is updated during the loop, and by the time the callbacks execute, i has already reached the final value (5).
  + The closures inside each setTimeout don’t capture i at each iteration. They all reference the same i.
* **Result**: Instead of logging 0, 1, 2, 3, 4, all callbacks log 5 because i has the final value after the loop finishes.

**It is solved Using IIFE and LET**

**What is the Output ?**

for (var i = 0; i < 5; i++) {

    (function(i){setTimeout(function() {

        console.log(i);

    },3000);

})(i);

}

The reason you see [0, 1, 2, 3, 4] is because each IIFE captures and retains the value of i for its specific iteration. setTimeout callbacks use these captured values due to the closures created by the IIFE. This avoids the common problem with var where all asynchronous operations might share the same variable value if var is used without such closures.

**5) Use Cases of Closure:**

**1) Data Encapsulation and Privacy**

Closures can be used to create private data that is not accessible from outside the function. This is useful for encapsulating and protecting internal state.

**Example**

function createCounter() {

let count = 0;

return {

increment() {

count += 1;

console.log(count);

},

getCount() {

return count;

}

};

}

const counter = createCounter();

counter.increment(); // Outputs: 1

counter.increment(); // Outputs: 2

console.log(counter.getCount()); // Outputs: 2

Here, count is a private variable that can only be accessed through the increment and getCount methods.

**2. Function Factories**

Closures allow you to create function factories, which are functions that generate other functions with customized behavior.

A function factory is a higher-order function that returns a new function based on the parameters it receives. It's called a "factory" because it "manufactures" functions dynamically.

**Example**

function multiplyBy(factor) {

return function(number) {

return number \* factor;

};

}

const multiplyByTwo = multiplyBy(2);

const multiplyByThree = multiplyBy(3);

console.log(multiplyByTwo(5)); // Outputs: 10

console.log(multiplyByThree(5)); // Outputs: 15

In this example, multiplyBy is a factory function that creates functions to multiply by a specific factor.

**3. Maintaining State in Asynchronous Code**

Closures can be used to maintain state in asynchronous code, such as in callbacks or promises.

**Example**

function delayedGreeting(name) {

setTimeout(function() {

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

}, 1000);

}

delayedGreeting('Alice'); // Outputs: Hello, Alice! (after 1 second)

The inner function retains access to the name parameter even though it is executed asynchronously.

**4)Currying**

Closures facilitate currying, a technique where a function returns another function with some of its arguments pre-set.

**Example**

function power(base) {

return function(exponent) {

return Math.pow(base, exponent);

};

}

const square = power(2);

const cube = power(3);

console.log(square(4)); // Outputs: 16

console.log(cube(3)); // Outputs: 27

In this example, power returns a function that calculates the base raised to the exponent.

**How to handle parallel Api requestion in js ?**

 **Promise.all:** Waits for all promises to resolve or any to reject.

 **Promise.allSettled:** Waits for all promises to settle (resolve or reject).

 **Promise.race:** Resolves or rejects as soon as the first promise resolves or rejects.

 **async/await with Promise.all**: Allows parallel execution with cleaner syntax.

async function fetchData() {

const urls = [

'https://api.example.com/data1',

'https://api.example.com/data2',

'https://api.example.com/data3'

];

try {

const responses = await Promise.all(urls.map(url => fetch(url)));

const data = await Promise.all(responses.map(response => response.json()));

console.log(data); // Array of results from all the API requests

} catch (error) {

console.error('Error fetching data:', error);

}

}

**fetchData();**

**2 ND Method :**

**Using Promise.race**

**Promise.race returns a promise that resolves or rejects as soon as one of the promises in the iterable resolves or rejects. It’s useful if you want to get the result of the first completed request.**

**Example:**

async function fetchData() {

const urls = [

'https://api.example.com/data1',

'https://api.example.com/data2',

'https://api.example.com/data3'

];

try {

const firstResolved = await Promise.race(urls.map(url => fetch(url)));

const data = await firstResolved.json();

console.log('First resolved data:', data);

} catch (error) {

console.error('Error fetching data:', error);

}

}

fetchData();

**Explain how the async and await keywords work in JavaScript. How do they compare to using Promises directly, and what are some potential pitfalls?"**

The async and await keywords in JavaScript provide a more readable and intuitive way to work with asynchronous operations compared to using Promises directly. Here’s a breakdown of how they work and how they compare to Promises:

**How async and await Work**

1. **async Keyword**:
   * **Definition**: The async keyword is used to declare a function as asynchronous. An async function automatically returns a Promise. If the function returns a value, that value is wrapped in a Promise. If the function throws an error, the error is also wrapped in a Promise.
   * **Syntax**:

javascript

Copy code

async function example() {

return "Hello, World!";

}

// Equivalent to:

function example() {

return Promise.resolve("Hello, World!");

}

1. **await Keyword**:
   * **Definition**: The await keyword can only be used inside an async function. It pauses the execution of the function and waits for the Promise to resolve or reject. Once the Promise is resolved, await returns the resolved value. If the Promise is rejected, await throws the rejected value as an exception.
   * **Syntax**:

async function fetchData() {

let data = await fetch('https://api.example.com/data');

let json = await data.json();

return json;

}

**Comparison to Promises**

1. **Readability**:
   * **Promises**:

javascript

Copy code

fetch('https://api.example.com/data')

.then(response => response.json())

.then(data => console.log(data))

.catch(error => console.error(error));

* + **async/await**:

javascript

Copy code

async function fetchData() {

try {

let response = await fetch('https://api.example.com/data');

let data = await response.json();

console.log(data);

} catch (error) {

console.error(error);

}

}

fetchData();

* + **Advantage**: async/await generally makes code easier to read and maintain, especially with multiple asynchronous operations.

**What are Common Pitfalls of Async and Await ?**

**1. Blocking Behavior**

* **Issue**: The await keyword pauses the execution of the async function until the Promise is resolved or rejected. This can block the event loop and cause performance issues if used improperly.
* **Example**:

async function fetchData() {

let data1 = await fetch(url1);

let data2 = await fetch(url2); // This will not start until data1 is resolved

// Process data1 and data2

}

* **Solution**: Use Promise.all() to handle multiple promises concurrently if they are independent of each other:

javascript

Copy code

async function fetchData() {

let [data1, data2] = await Promise.all([fetch(url1), fetch(url2)]);

// Process data1 and data2

}

**2. Error Handling**

* **Issue**: If you don’t handle errors properly with try/catch, unhandled promise rejections can occur, leading to unexpected behavior.
* **Example**:

async function fetchData() {

let data = await fetch(url); // If fetch fails, no error handling

// Process data

}

* **Solution**: Always wrap await calls in a try/catch block to handle potential errors:

async function fetchData() {

try {

let data = await fetch(url);

// Process data

} catch (error) {

console.error(error);

}

}

**3. Unintentional Sequential Execution**

* **Issue**: Using await inside loops or multiple await calls in sequence may lead to unintentional serial execution instead of concurrent execution.
* **Example**:

async function fetchData() {

for (let i = 0; i < urls.length; i++) {

let data = await fetch(urls[i]); // Each fetch waits for the previous one

// Process data

}

}

* **Solution**: Use Promise.all() for parallel execution of independent operations:

async function fetchData() {

let promises = urls.map(url => fetch(url));

let results = await Promise.all(promises);

// Process results

}

**How would you approach debugging a JavaScript application where a specific function intermittently fails without any obvious error message? Describe your process?**

**Check for Error Handling**

* **Add Error Handling**: Ensure that all potential errors are caught and logged. Sometimes, failures don’t produce visible errors due to missing or inadequate error handling.

try {

// Code that might fail

} catch (error) {

console.error('Error occurred:', error);

}

* **Improve Logging**: Enhance logging within the function and surrounding code to capture more context about the failure. Include inputs, state, and any relevant information.

console.log('Function inputs:', inputs);

console.log('Current state:', state);

**Use Debugging Tools**

* **Browser Developer Tools**: Utilize the built-in debugging tools in browsers (e.g., Chrome DevTools) to set breakpoints, step through the code, and inspect variables and the call stack.
  + Set breakpoints on the function entry point and step through the code to observe its behavior.
  + Monitor the Network tab for asynchronous requests and responses.
* **Debugging Libraries**: Consider using libraries like debug or loglevel to manage and format logs more effectively.

### ****Is Javascript a case-sensitive language?****

Yes, Javascript is a case-sensitive language. This means that the keywords,function-name, variable name all should be typed with a consistent capitalization of letters. The word let and LET will have different meanings in JS.So while using any variable or keyword, we should keep the case of the letters in mind.

**What is the role of the V8 engine in JavaScript execution? How does it impact the performance of a JavaScript application?**

The V8 engine is a critical component in JavaScript execution, particularly in environments like Google Chrome and Node.js. Here’s a detailed look at its role and impact on performance:

**Role of the V8 Engine**

1. **JavaScript Execution**:
   * **Definition**: V8 is an open-source JavaScript engine developed by Google. It is designed to execute JavaScript code efficiently by converting it into machine code that can be executed directly by the CPU.
2. **Just-In-Time (JIT) Compilation**:
   * **Compilation**: V8 compiles JavaScript code into machine code at runtime rather than interpreting it line-by-line. This process is known as Just-In-Time (JIT) compilation.
   * **Optimizations**: During JIT compilation, V8 applies various optimizations to improve performance. For example, it can inline functions, optimize loops, and apply inline caching to speed up repeated operations.
3. **Garbage Collection**:
   * **Memory Management**: V8 includes a garbage collector that automatically manages memory by reclaiming unused memory. This helps prevent memory leaks and ensures efficient memory usage.
   * **Generational Collection**: V8 uses a generational garbage collection strategy, dividing memory into different generations (e.g., young and old generations) to optimize collection based on the lifespan of objects.
4. **Execution Contexts**:
   * **Context Management**: V8 manages different execution contexts for JavaScript code, including global and function contexts. It maintains scope, variable access, and function calls within these contexts.

**What is Cross-Site Scripting?**

Cross-Site Scripting (XSS) is a type of security vulnerability commonly found in web applications. It occurs when an attacker injects malicious scripts into web pages that are then executed by other users' browsers.

**DOM:**

**1. What is DOMContentLoaded?**

The DOMContentLoaded event in JavaScript is a crucial part of handling the loading of web pages. It plays a significant role in ensuring that your JavaScript code executes only after the HTML document has been fully loaded and parsed, but before all external resources like images and stylesheets are fully loaded. Here's a detailed look at its role and how it can be used:

* **Event Description:** The DOMContentLoaded event is fired when the initial HTML document has been completely loaded and parsed, without waiting for stylesheets, images, and subframes to finish loading. This means that the DOM is ready to be manipulated.
* **Event Trigger:** It does not wait for stylesheets, images, and subframes to be loaded.

**2. Why Use DOMContentLoaded?**

* **Early Access to DOM Elements:** It allows you to safely interact with and manipulate DOM elements as soon as the DOM tree is constructed, without waiting for all external resources to load.
* **Faster Script Execution:** By using DOMContentLoaded, you can start running your JavaScript code earlier, which can improve the responsiveness of your web page.

Code:

document.addEventListener('DOMContentLoaded', (event) => { // Code to run when DOM is fully loaded console.log('DOM fully loaded and parsed'); });

**2. load Event**

* **Trigger Time:** Fires when the entire page, including all dependent resources such as images, stylesheets, and iframes, has fully loaded.
* **Purpose:** Ensures that all resources are loaded and available before running scripts, which is useful if your script depends on external resources being fully loaded.
* **Use Case:** Useful for scripts that need to operate on or after the full rendering of the page, including external resources.

What is THIS Keyword?

**What is the purpose of the bind() method, call( ) and apply( ) in JavaScript?**

Describe the principles of functional programming and provide examples of pure functions, immutability, and higher-order functions.

ES6 Features:

Discuss the features introduced in ES6 (ECMAScript 2015) such as arrow functions, template literals, destructuring, and classes.

Provide examples of how these features improve JavaScript code readability and maintainability.

Arrow function in notes

**Execution Context:**

It is a container where the js code memory is created and code executed.

In JavaScript, the execution context is essentially a container for the code execution environment where memory is allocated and where the code is actually executed

It is created for every function call.

In js memory will be created for all variable and function before code gets executed.

**Created for Every Function Call**:

* **Function Execution Context**: Whenever a function is called, a new execution context is created for that function. This context includes local variables, arguments, and the this value.
* **Call Stack Management**: Each function call pushes a new execution context onto the call stack, and once the function execution completes, the context is popped off the stack.

**How is Execution Context created and how is the program run?**

When JavaScript code runs **Global Execution Context (GEC)** is created. Execution of code is happening in two phases:

* **Phase 1 (Creation Phase)** is allocating memory to the variables and functions. For all variables value stored in memory is **undefined** and functions are copied in memory.
* In **Phase 2 (Execution Phase)** JavaScript code is executed line by line and it is defining variables in memory here. Every function invoked creates its own local execution context, which is put in call stack (explained in the next chapter).

**What is a WeakMap in JavaScript?**

A WeakMap is a collection of key/value pairs where the keys are objects and the values can be arbitrary values. The primary feature of a WeakMap is that it holds “weak” references to the keys, meaning that if there are no other references to the key object, it can be garbage collected.

**How do you create a WeakMap?**

You can create a WeakMap using the WeakMap constructor.

**What are the main differences between Map and WeakMap?**

* **Key Type:** Map keys can be of any type, while WeakMap keys must be objects.

That means the map can contains the primitive and non primitive but weaMap only contains the object type.

* **Garbage Collection:** WeakMap holds weak references to keys, allowing them to be garbage collected, whereas Map holds strong references.

Example: if we make the keys of map to null still it exist in map but not in case of weakMap.

* **Iterability:** Map is iterable, meaning you can loop through its entries. WeakMap is not iterable, and you cannot get a list of its keys or values.

### 8) What is the behavior of comparison operators when comparing values with different types

When comparing values with different types using comparison operators in JavaScript, the behavior depends on the operator being used: loose equality (==), strict equality (===), or relational operators (<, <=, >, >=). Here's a breakdown:

**1. Loose Equality Operator (==)**

The loose equality operator attempts type coercion if the operands have different types before making the comparison. If both operands are of different types, JavaScript will attempt to convert them to a common type. Here are the general rules:

If one of the operands is a number and the other is a string, the string is converted to a number.

If one of the operands is a boolean, the boolean is converted to a number (true becomes 1, false becomes 0).

If one of the operands is an object and the other is a primitive value, the object is converted to a primitive value using the valueOf or toString methods.

For example:

console.log(5 == '5'); // true (string '5' is coerced to number 5)

console.log('10' == true); // true (boolean true is coerced to number 1)

**2. Strict Equality Operator (===)**

The strict equality operator does not perform type coercion. It returns true only if both operands are of the same type and have the same value.

For example:

console.log(5 === '5'); // false (different types)

console.log('10' === true); // false (different types)

**3. Relational Operators (<, <=, >, >=)**

Relational operators work differently compared to equality operators. When comparing different types with relational operators, JavaScript converts both operands to numbers and then performs the comparison. If one of the operands cannot be converted to a number, it will return NaN, which is considered neither greater nor less than any other number.

**For example:**

console.log('10' > '5'); // true (string comparison, '10' is greater than '5')

console.log('hello' > 5); // false (NaN is considered less than any number)

**Summary**

Loose Equality (==): Performs type coercion before comparison.

Strict Equality (===): Does not perform type coercion, requires both type and value to be equal.

Relational Operators: Convert operands to numbers before comparison. If one operand cannot be converted to a number, it returns NaN.

**9) Does Relational Operator always convert all types to Number ?**

Yes, that's correct. When comparing different types with relational operators (<, <=, >, >=) in JavaScript, the operands are always converted to numbers before the comparison is made. This ensures consistent behavior and allows for meaningful comparisons across different types.

### 10) How do JavaScript comparison operators handle objects?

JavaScript comparison operators compare object references, not the contents. They check if the compared objects refer to the same memory location, rather than comparing their properties or values.

 let obj1 = { name: "Alice" };

let obj2 = { name: "Alice" };

let obj3 = obj1;

console.log(obj1 === obj2);  // Output: false (Different object references)

console.log(obj1 === obj3);  // Output: true (Same object reference)

**11) Explain in comparision string converted to number or vise versa?**

In JavaScript, when comparing values of different types, especially involving strings and numbers, type coercion can occur. Type coercion refers to the automatic conversion of values from one data type to another in order to perform the comparison.

Here's how JavaScript handles comparisons involving strings and numbers:

**String to Number Conversion**

When comparing a string to a number, JavaScript will attempt to convert the string to a number before making the comparison. This typically happens when using relational operators (<, <=, >, >=) or arithmetic operators (+, -, \*, /, %, \*\*).

**Example:**

console.log('10' > 5); // true (string '10' is converted to number 10)

**Number to String Conversion**

When comparing a number to a string, JavaScript will also attempt to convert the number to a string before making the comparison. This typically happens when using concatenation (+) or relational operators with strings.

**Example:**

console.log(5 + '10'); // '510' (number 5 is converted to string '5')

**Other Comparisons**

For other comparisons, such as using equality operators (==, ===, !=, !==), JavaScript may perform type coercion depending on the operator used. For example, the loose equality operator (==) performs type coercion, while the strict equality operator (===) does not.

**Example:**

console.log('5' == 5); // true (string '5' is coerced to number 5)

console.log('5' === 5); // false (different types, no coercion)

**Summary**

When comparing a string to a number or vice versa, JavaScript may perform type coercion.

JavaScript will attempt to convert the operands to a common type before making the comparison.

It's important to be aware of type coercion when writing comparisons in JavaScript, as it can sometimes lead to unexpected results. Using strict equality (===) is often recommended to avoid unintended type coercion.

**12) Explain Truthy and falsy Value?**

In JavaScript, truthy and falsy values are concepts related to boolean evaluation. Every value in JavaScript has an inherent boolean “truthiness” or “falsiness,” which means they can be implicitly evaluated to true or false in boolean contexts, such as in conditional statements or logical operations.

**Truthy Values:** A value is considered truthy if, when coerced to a boolean, it evaluates to true. Examples of truthy values include non-empty strings, numbers other than 0, arrays, objects, and functions.

**Falsy Values:** Values that are not true are considered False values. 0, null, undefined, NaN, false(Boolean value), and an empty string ("").

**13) What type of conversion does the JavaScript “if” statement use?**

The JavaScript "if" statement uses implicit type coercion, converting the condition to a boolean value. This is done through the Abstract Equality Comparison algorithm, often called "loose" comparison.

**What has the highest operator precedence over all the logical operators?**

Interview Response: The logical NOT (!) operator has the highest precedence among all logical operators in JavaScript. It always executes first.

Technical Response: The precedence of NOT ! is the highest of all logical operators, so it always executes first, before && (AND) or || (OR).

Diffrence between Null and undefined ?

 undefined is its own type.

 null is of type "object" (historical quirk).

 undefined is used by JavaScript to represent the absence of a value for variables, object properties, and function arguments.

 null is used by programmers to explicitly indicate the absence of an object or value.

null is used in JavaScript to explicitly indicate the intentional absence of any object value. Unlike undefined, which is a default value for uninitialized variables or missing properties, null is explicitly assigned.

**Arrow functions and the differences from regular functions?**

1. Arrow functions cannot use the arguments object.
2. They have a different syntax.
3. Arrow functions do not have their own this context. When referencing this, an arrow function takes the context from the surrounding scope.
4. Arrow functions cannot be used as constructor functions. In other words, they cannot be invoked with the new keyword.

### What is Function declaration and Function expression ?

| **Function Declaration** | **Function Expression** |
| --- | --- |
| A function declaration must have a function name. | A function expression is similar to a function declaration without the function name. |
| Function declaration does not require a variable assignment. | Function expressions can be stored in a variable assignment. |
| These are executed before any other code. | Function expressions load and execute only when the program interpreter reaches the line of code. |
| The function in function declaration can be accessed before and after the function definition. | The function in function expression can be accessed only after the function definition. |
| Function declarations are hoisted | Function expressions are not hoisted |
| **Syntax:** function geeksforGeeks(paramA, paramB) { // Set of statements } | **Syntax:** var geeksforGeeks= function(paramA, paramB) { // Set of statements } |

### Can you name at least three types of functions in JavaScript?

Interview Response: In JavaScript, the three most common types of functions include: named functions, anonymous, and arrow functions.

Code Example: Here are examples of each type of function.

Regular function:

function greeting(name) {

    return `Hello, ${name}!`;

}

console.log(greeting("John")); // Outputs: Hello, John!

Arrow function:

const greeting = (name) => `Hello, ${name}!`;

console.log(greeting("John")); // Outputs: Hello, John!

Anonymous Function:

setTimeout(function() {

    console.log("This is an anonymous function!");

}, 1000);

1. **What is Function ?**

Functions are one of the fundamental building blocks in JavaScript. A function is a JavaScript procedure — a set of statements that performs a task or calculates a value. To use a function, you must define it somewhere in the scope from which you wish to call it.

A JavaScript function is a reusable block of code designed to perform a specific task, taking input as arguments, processing the data, and returning a result.

### What is one of the primary purposes of JavaScript functions?

Interview Response: The primary purpose of JS functions is to avoid code duplication.

Technical Response: The primary purpose of functions is to avoid code duplication. If we ever need to change the message or the way it displays, it is enough to modify the code in one place based on the function which outputs it.

function showMessage(name) {

    console.log('Hello, ' + name);

  }

  showMessage('John'); // Hello, John

  showMessage('Jane'); // Hello, Jane

1. **Why Error is coming ?**

function outerFunc() {

    let outerVar = "Anup";  // outerVar is in the local scope of outerFunc

    function innerFunc() {

        let innerVar = "Chakra";  // innerVar is in the local scope of innerFunc

    }

    innerFunc();  // Call innerFunc

}

outerFunc();  // Call outerFunc

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

The error in the given code occurs because of the concept of scope in JavaScript. Specifically, outerVar is defined in the local scope of outerFunc, and therefore it is not accessible outside of outerFunc.

### Can functions access variables outside the function body?

Yes, functions can access variables outside their body, as they have access to the outer scope, allowing them to use variables declared in a containing scope.

let userName = 'John';

function showMessage() {

  userName = 'Bob'; // (1) changed the outer variable

  let message = 'Hello, ' + userName;

  console.log(message);

}

console.log(userName); // John before the function call

showMessage(); // Hello, Bob modified through invocation

console.log(userName); // Bob, the value was modified by the function

### What is the difference between a parameter and a argument in JavaScript?

In JavaScript, a parameter is a variable named in the function declaration. An argument is the actual value that gets passed into the function when it is invoked.

### What are WeakSet and WeakMap and how do they differ from Map and Set?

The first difference between WeakMap and Map is that the keys in WeakMap must be objects, not primitive values.  
The second difference is in the memory storage of the data structures. The JavaScript engine keeps values in memory as long as they are reachable, meaning they can be used.  
Usually, object properties, array elements, or other data structures are considered reachable and are kept in memory as long as the data structure exists, even if there are no other references to them.  
In the case of WeakMap and WeakSet, it works differently. Once an object becomes unreachable, it is removed from the data structure.

**Why do we need Promises if we can work with asynchronous code using callbacks?**

If we want to asynchronously fetch some data from a server using callback functions, it would result in the following:

func((x) => {

anotherFunc(x, (y) => {

andAnotherFunc(i, (j) => {

// some code

})

})

})

This is called **callback hell**, as each callback is nested inside another, and each inner callback depends on the parent function.

Using Promises, we can rewrite the code above:

func()

.then((x) => {

return anotherFunc(x)

})

.then((y) => {

return andAnotherFunc(y)

})

.then((i) => {

return i

})

With Promises, the execution sequence is clear, making the code more readable.

**Compare the both empty array ?**

let arr1 = [];

let arr2 = [];

console.log(arr1 == arr2); // Output: false

* **Explanation**: arr1 and arr2 are two different objects (even though both are empty arrays). The loose equality operator checks if they refer to the same object, which they don’t. Thus, arr1 == arr2 evaluates to false.
* **Arrays are compared by reference**, not by value. Even if two arrays have the same elements, they are considered different if they don’t refer to the same object in memory.

**Example:**

javascript

Copy code

let arr1 = [];

let arr2 = [];

console.log(arr1 === arr2); // Output: false

* **Explanation**: Even though both arr1 and arr2 are empty arrays, they are two distinct objects in memory. Therefore, arr1 === arr2 evaluates to false.
* Arrays are compared by reference, not by value. Even if two arrays have the same elements, they are considered different if they don’t refer to the same object in memory.

Example:

let arr1 = [];

let arr2 = [];

console.log(arr1 === arr2); // Output: false

**for in and for of:**

For In:

* When working with **objects** to access keys or properties.
* When iterating over an array and you specifically need the **indices**.

For Of :

 Iterating over **values** in arrays, strings, or other iterable objects.

 When you don’t care about the indices or keys.