**What is a callback function?**

A callback function is a function that is passed as an argument to another function and is executed after some operation has been completed.

**Why Use Callback Functions?**

**Asynchronous Programming:** Callbacks allow us to continue executing code. Whenever callback function task completed then executed and other callback codes are not waiting during the callback function code executions, thereby improving efficiency.

**Code Reusability:** Passing functions around allows for more modular and reusable code.

**Event Handling:** They are essential in handling events like user interactions (clicks, keypresses).

**Passing Functions as Arguments:** Functions in JavaScript are first-class citizens, meaning they can be treated like any other variable. You can assign them to variables, pass them as arguments to other functions, and return them from functions.

|  |  |
| --- | --- |
| function greet(name) {  console.log(`Hello, ${name}!`);  }  function processUserInput(callback) {  const name = "Alice";  callback(name);  }  processUserInput(greet);  // Output: "Hello, Alice!" | function fetchData(callback) {  setTimeout(() => {  const data = { id: 1, name: 'John Doe' };  callback(data);  }, 2000);  }  function processData(data) {  console.log('Data received:', data);  }  fetchData(processData); |

|  |
| --- |
| document.getElementById("button").addEventListener("click", () => {  console.log("Button clicked!");  }); |

**Q.Can we use async and await only using callback function not using promise with example.**

The async and await keywords in JavaScript are designed to work with Promises, not directly with callback functions. Callbacks are part of an earlier pattern for handling asynchronous operations, and async/await are built to provide a more straightforward syntax for working with Promises.

|  |  |
| --- | --- |
| function **readFileCallback**(filePath, callback) {  setTimeout(() => {  if (filePath === "valid.txt") {  callback(null, "File content");  } else {  callback(new Error("File not found"), null);  }  }, 1000);  }  **However, you can convert callback-based functions into Promise-based ones using a function like util.promisify in Node.js, and then use async and await with them.** | function **readFileAsync**(filePath) {  return new Promise((resolve, reject) => {  **readFileCallback(**filePath, (err, data) => {  if (err) {  reject(err);  } else {  resolve(data);  } }); }); }  async function processFile() {  try {  const content = await readFileAsync("valid.txt");  console.log("File content:", content);  } catch (err) {  console.error("Error:", err.message); } }  processFile(); |

|  |  |
| --- | --- |
| const fs = require('fs');  function readFileCallback(filePath, callback) {  fs.readFile(filePath, 'utf8', (err, data) => {  if (err) {  return callback(err);  }  callback(null, data);  }); } | const { promisify } = require('util');  const readFilePromise = promisify(readFileCallback);  async function readFileAsync(filePath) { try { const data = await readFilePromise(filePath);  console.log('File data:', data); } catch (err) {  console.error('Error reading file:', err);  }}  readFileAsync('./example.txt'); |

Q.**what happen when we use async with normal function with example.**

When we use the async keyword with a normal function, it transforms the function into an asynchronous function. This means the functions will always return a promise, even if it doesn’t explicitly return one. The await keyword can then be used inside this function to pause its execution until a promise is resolved.

|  |  |
| --- | --- |
| **// Normal function**  Function normalFunction(){  Return “Hello, World!”;}  **//Async function**  Async function asyncFunction(){  Return “Hello,Async World!”;}  //using the async function  asyncFunction().then(result=>  console.log(result)); | **//Async function with await**  Async function fetchData(){  let promise = new Promise((resolve,reject)=>  {setTimeout()=>resolve(“Data fetched”),2000;});  let result = await promise;  console.log(result);  fetched!}  fetchData(); |

|  |
| --- |
| async function throwErrorFunction() {  throw new Error("Something went wrong!");  }  // Handling the error  throwErrorFunction()  .then(() => console.log("This will not run"))  .catch(err => console.error(err.message)); // Outputs: Something went wrong! |

**Execution Context**

1.Everything in JS happens inside the execution context.

2.In the container the first component is memory component and the 2nd one is code component

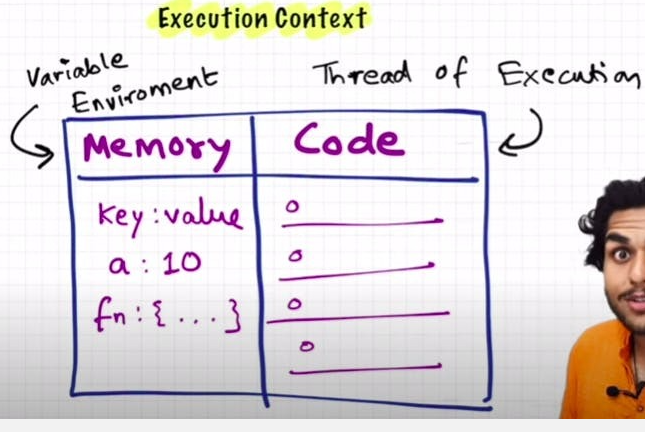
3.Memory component has all the variables and functions in key value pairs. It is also called Variable environment.

4.Code component is the place where code is executed one line at a time. It is also called the Thread of Execution.

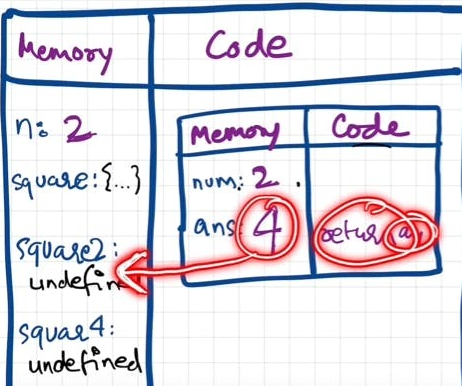
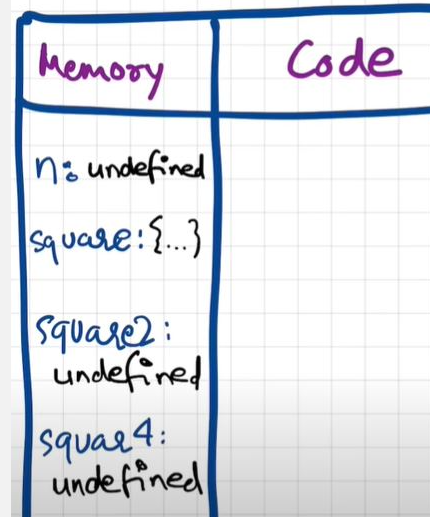
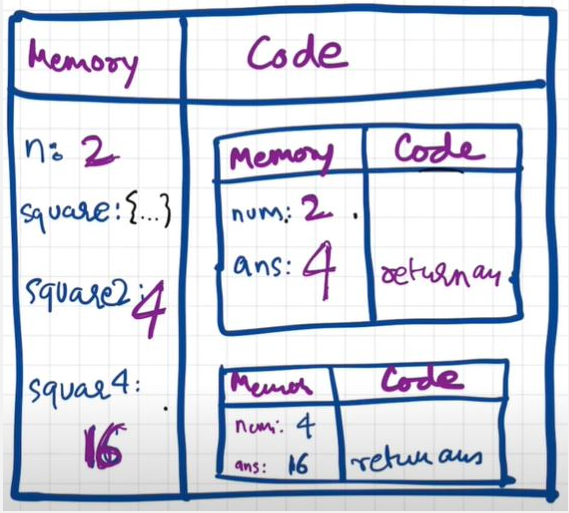
5.JS is a synchronous, single-threaded language

6.Synchronous:- In a specific synchronous order.

Single-threaded:- One command at a time.



|  |  |
| --- | --- |
| var n = 2;  function square(num) {  var ans = num \* num;  return ans;  }  var square2 = square(n);  var square4 = square(4); |  |

**Hoisting in JavaScript**

Hoisting is a concept which enables us to extract values of variables and functions even before initialising/assigning value without getting error and this is happening due to the 1st phase (memory creation phase) of the Execution Context.

|  |  |
| --- | --- |
| getName(); // Namaste Javascript, undefined  console.log(x); // undefined  var x = 7;  function getName() {  console.log("Namaste Javascript",x);  } | getName(); // Namaste JavaScript  console.log(x); // Uncaught Reference: x is not defined.  console.log(getName); // f getName(){ console.log("Namaste JavaScript); }  function getName() {  console.log("Namaste JavaScript");  } |

|  |  |
| --- | --- |
| getName(); // Namaste JavaScript  console.log(x); // Uncaught Reference: x is not defined.  console.log(getName); // f getName(){ console.log("Namaste JavaScript); }  function getName() {  console.log("Namaste JavaScript");  }  } | getName(); // Uncaught TypeError: getName is not a function  console.log(getName);  var getName = function () {  console.log("Namaste JavaScript");  }; |

**Functions and Variable Environments**

|  |  |
| --- | --- |
| var x = 1;  a();  b(); // we are calling the functions before defining them. This will work properly, as seen in Hoisting.  console.log(x); | function a() {  var x = 10; // local scope because of separate execution context  console.log(x);  }  function b() {  var x = 100;  console.log(x);  }  **Outputs:10 100 1** |

|  |  |
| --- | --- |
|  | **Shortest program**  **var x = 10;**  **console.log(x); // 10**  **console.log(this.x); // 10**  **console.log(window.x); // 10** |

**undefined vs not defined in JS**

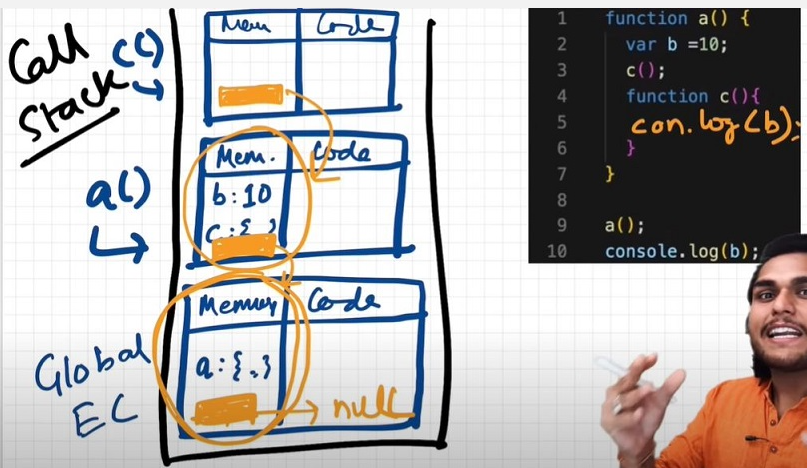
1.In first phase (memory allocation) JS assigns each variable a placeholder called undefined.

2.undefined is when memory is allocated for the variable, but no value is assigned yet.

3.If an object/variable is not even declared/found in memory allocation phase, and tried to access it then it is Not defined

|  |  |
| --- | --- |
| console.log(x); // undefined  var x = 25;  console.log(x); // 25  console.log(a); // Uncaught ReferenceError: a is not defined | Not Defined !== Undefined |

**The Scope Chain, Scope & Lexical Environment**



|  |  |
| --- | --- |
| function a() {  function c() {  // logic here  }  c(); // c is lexically inside a  } // a is lexically inside global execution | // CASE 1  function a() {  console.log(b); // 10  // Instead of printing undefined it prints 10, So somehow this a function could access the variable b outside the function scope.  }  var b = 10; a(); |

1.Scope in Javascript is directly related to Lexical Environment

2.So, Lexical Environment = local memory + lexical env of its parent. Hence, Lexical Environement is the local memory along with the lexical environment of its parent

3.Lexical: In hierarchy, In order

4.Whenever an Execution Context is created, a Lexical environment(LE) is also created and is referenced in the local Execution Context(in memory space).

5.The process of going one by one to parent and checking for values is called scope chain or Lexcial environment chain.

6.Lexical or Static scope refers to the accessibility of variables, functions and object based on physical location in source code.

|  |  |
| --- | --- |
| // CASE 1  function a() {  console.log(b); // 10  // Instead of printing undefined it prints 10, So somehow this a function could access the variable b outside the function scope.  }  var b = 10;  a(); | // CASE 2  function a() {  c();  function c() {  console.log(b); // 10  }  }  var b = 10;  a(); |
| // CASE 3  function a() {  c();  function c() {  var b = 100;  console.log(b); // 100  }  }  var b = 10;  a(); | // CASE 4  function a() {  var b = 10;  c();  function c() {  console.log(b); // 10  }  }  a();  console.log(b); // Error, Not Defined |

**let & const in JS, Temporal Dead Zone**

|  |
| --- |
| console.log(a); // ReferenceError: Cannot access 'a' before initialization  console.log(b); // prints undefined as expected  let a = 10;  console.log(a); // 10  var b = 15;  console.log(window.a); // undefined  console.log(window.b); // 15 |

1.It looks like let isn't hoisted, but it is, let's understand

2.Both a and b are actually initialized as undefined in hoisting stage. But var b is inside the storage space of GLOBAL, and a is in a separate memory object called script, where it can be accessed only after assigning some value to it first ie. one can access 'a' only if it is assigned. Thus, it throws error.

1.Temporal Dead Zone : Time since when the let variable was hoisted until it is initialized some value.

2. So any line till before "let a = 10" is the TDZ for a

Since a is not accessible on global, its not accessible in window/this also. window.b or this.b -> 15; But window.a or this.a ->undefined, just like window.x->undefined (x isn't declared anywhere)

Reference Error are thrown when variables are in temporal dead zone.

3.Syntax Error doesn't even let us run single line of code.

|  |  |
| --- | --- |
| let a = 10;  let a = 100; //this code is rejected upfront as SyntaxError. (duplicate declaration)  ------------------  let a = 10;  var a = 100; // this code also rejected upfront as SyntaxError. (can't use same name in same scope)  Let is a stricter version of var. Now, const is even more stricter than let. | let a;  a = 10;  console.log(a) // 10. Note declaration and assigning of a is in different lines.  ------------------  const b;  b = 10;  console.log(b); // SyntaxError: Missing initializer in const declaration. (This type of declaration won't work with const. const b = 10 only will work)  ------------------  const b = 100;  b = 1000; //this gives us TypeError: Assignment to constant variable. |
|  |  |

* Types of **Error**: Syntax, Reference, and Type.
  + Uncaught ReferenceError: x is not defined at ...
    - This Error signifies that x has never been in the scope of the program. This literally means that x was never defined/declared and is being tried to be accesed.
  + Uncaught ReferenceError: cannot access 'a' before initialization
    - This Error signifies that 'a' cannot be accessed because it is declared as 'let' and since it is not assigned a value, it is its Temporal Dead Zone. Thus, this error occurs.
  + Uncaught SyntaxError: Identifier 'a' has already been declared
    - This Error signifies that we are redeclaring a variable that is 'let' declared. No execution will take place.
  + Uncaught SyntaxError: Missing initializer in const declaration
    - This Error signifies that we haven't initialized or assigned value to a const declaration.
  + Uncaught TypeError: Assignment to constant variable
    - This Error signifies that we are reassigning to a const variable.

**Block Scope & Shadowing in JS**

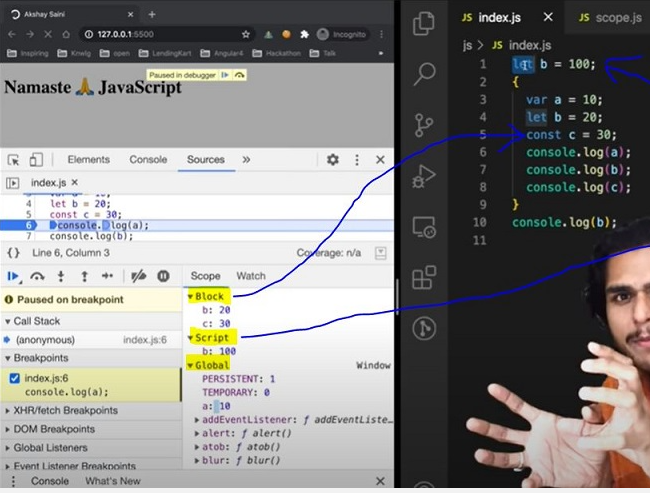
What is a Block?

Block aka compound statement is used to group JS statements together into 1 group. We group them within {...}

|  |  |
| --- | --- |
| **{**  **var a = 10;**  **let b = 20;**  **const c = 30;**  **// Here let and const are hoisted in Block scope,**  **// While, var is hoisted in Global scope.**  **}** | **{**  **var a = 10;**  **let b = 20;**  **const c = 30;**  **}**  **console.log(a); // 10**  **console.log(b); // Uncaught ReferenceError: b is not defined** |

|  |
| --- |
| **\* Reason?**  **\* In the BLOCK SCOPE; we get b and c inside it initialized as \*undefined\* as a part of hoisting (in a seperate memory space called \*\*block\*\*)**  **\* While, a is stored inside a GLOBAL scope.**  **\* Thus we say, \*let\* and \*const\* are BLOCK SCOPED. They are stored in a separate mem space which is reserved for this block. Also, they can't be accessed outside this block. But var a can be accessed anywhere as it is in global scope. Thus, we can't access them outside the Block.** |

|  |  |
| --- | --- |
| **What is Shadowing?**  var a = 100;  {  var a = 10; // same name as global var  let b = 20;  const c = 30;  console.log(a); // 10  console.log(b); // 20  console.log(c); // 30  }  console.log(a); // 10, instead of the 100 we were expecting. So block "a" modified val of global "a" as well. In console, only b and c are in block space. a initially is in global space(a = 100), and when a = 10 line is run, a is not created in block space, but replaces 100 with 10 in global space itself. | **let b = 100;**  **{**  **var a = 10;**  **let b = 20;**  **const c = 30;**  **console.log(b); // 20**  **}**  **console.log(b); // 100, Both b's are in separate spaces (one in Block(20) and one in Script(another arbitrary mem space)(100)). Same is also true for \*const\* declarations.** |



|  |  |
| --- | --- |
| **Same logic is true even for functions**  const c = 100;  function x() {  const c = 10;  console.log(c); // 10  }  x();  console.log(c); // 100 | **What is Illegal Shadowing?**  let a = 20;  {  var a = 20;  }  // Uncaught SyntaxError: Identifier 'a' has already been declared  We cannot shadow let with var. But it is valid to shadow a let using a let. However, we can shadow var with let.  All scope rules that work in function are same in arrow functions too.  Since var is function scoped, it is not a problem with the code below.  let a = 20;  function x() {  var a = 20;  } |

**Closures in JS**

Function bundled along with it's lexical scope is closure.

JavaScript has a lexcial scope environment. If a function needs to access a variable, it first goes to its local memory. When it does not find it there, it goes to the memory of its lexical parent. See Below code, Over here function y along with its lexical scope i.e. (function x) would be called a closure.

|  |  |
| --- | --- |
| function x() {  var a = 7;  function y() {  console.log(a);  }  return y;  }  var z = x();  console.log(z); // value of z is entire code of function y. | In above code, When y is returned, not only is the function returned but the entire closure (fun y + its lexical scope) is returned and put inside z. So when z is used somewhere else in program, it still remembers var a inside x().  function z() {  var b = 900;  function x() {  var a = 7;  function y() {  console.log(a, b);  }  y();  }  x();  }  z(); // 7 900 |

A closure is a function that has access to its outer function scope even after the function has returned. Meaning, A closure can remember and access variables and arguments reference of its outer function even after the function has returned.\*

Advantages of Closure:

Certainly! Let's explore examples for each of the advantages you've mentioned:

1. **Module Design Pattern**:
   * The module design pattern allows us to encapsulate related functionality into a single module or file. It helps organize code, prevent global namespace pollution, and promotes reusability.
   * Example: Suppose we're building a web application, and we want to create a module for handling user authentication. We can create a auth.js module that exports functions like login, logout, and getUserInfo.

|  |  |
| --- | --- |
| // auth.js  const authModule = (function () {  let loggedInUser = null;  function login(username, password) {  // Authenticate user logic...  loggedInUser = username;  }  function logout() {  loggedInUser = null;  } | function getUserInfo() {  return loggedInUser;  }  return {  login,  logout,  getUserInfo,  };  })();  // Usage  authModule.login('john\_doe', 'secret');  console.log(authModule.getUserInfo()); // 'john\_doe' |

1. **Currying:**

Currying is a technique where a function that takes multiple arguments is transformed into a series of functions that take one argument each. It enables partial function application and enhances code flexibility.

Example: Let's create a curried function to calculate the total price of items with tax.

|  |  |
| --- | --- |
| const calculateTotalPrice = (taxRate) => (price) => price + price \* (taxRate / 100);  const calculateSalesTax = calculateTotalPrice(8); // 8% sales tax  const totalPrice = calculateSalesTax(100); // Price with tax  console.log(totalPrice); // 108 |  |

1. **Memoization:**

Memoization optimizes expensive function calls by caching their results. It's useful for recursive or repetitive computations.

Example: Implement a memoized Fibonacci function.

|  |
| --- |
| function fibonacci(n, memo = {}) {  if (n in memo) return memo[n];  if (n <= 1) return n;  memo[n] = fibonacci(n - 1, memo) + fibonacci(n - 2, memo);  return memo[n];  }  console.log(fibonacci(10)); // 55 |

1. **Data Hiding and Encapsulation:**

Encapsulation hides the internal details of an object and exposes only necessary methods and properties. It improves code maintainability and security.

Example: Create a Person class with private properties.

|  |  |
| --- | --- |
| class Person {  #name; // Private field  constructor(name) {  this.#name = name;  }  getName() {  return this.#name;  }  } | const person = new Person('Alice');  console.log(person.getName()); // 'Alice'  // console.log(person.#name); // Error: Private field '#name' must be declared in an enclosing class |

1. **setTimeouts:**

setTimeout allows scheduling a function to run after a specified delay. It's commonly used for asynchronous tasks, animations, and event handling.

Example: Delayed message display.

|  |
| --- |
| function showMessage(message, delay) {  setTimeout(() => {  console.log(message);  }, delay);  }  showMessage('Hello, world!', 2000); // Display after 2 seconds |

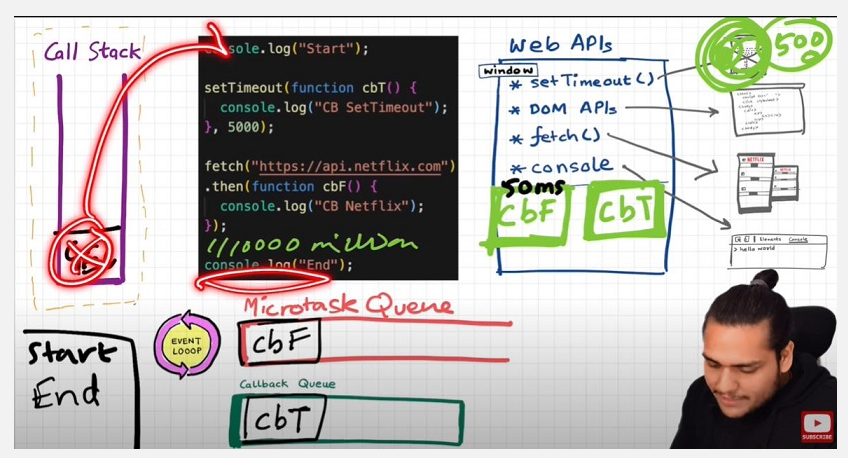
**setTimeout + Closures Interview Question**

|  |  |
| --- | --- |
| function x() {  for (var i = 1; i <= 5; i++) {  setTimeout(function () {  console.log(i);  }, i \* 1000);  }  console.log("Namaste Javascript");  }  x();  // Output:  // Namaste Javascript  // 6  // 6  // 6  // 6  // 6 | function x() {  for (var i = 1; i <= 5; i++) {  function close(i) {  setTimeout(function () {  console.log(i);  }, i \* 1000);  // put the setT function inside new function close()  }  close(i); // everytime you call close(i) it creates new copy of i. Only this time, it is with var itself!  }  console.log("Namaste Javascript");  }  x(); |

|  |  |
| --- | --- |
| **Will the below code still forms a closure?**  function outer() {  function inner() {  console.log(a); }  var a = 10;  return inner;  }  outer()(); // 10  Ans: Yes, because inner function forms a closure with its outer environment so sequence doesn't matter. | **Q: What is Function statement?**  Below way of creating function are function statement.  function a() {  console.log("Hello");  }  a(); // Hello |

|  |  |
| --- | --- |
| **Q: What is Function Expression?**  Assigning a function to a variable. Function acts like a value.  var b = function () {  console.log("Hello"); }; b();  **Q: What is Function Declaration?**  Other name for function statement.  **Q: What is Anonymous Function?**  A function without a name.  function () {}  // this is going to throw Syntax Error - Function Statement requires function name.  **They** don't have their own identity. So an anonymous function without code inside it results in an error.  **Anonymous** functions are used when functions are used as values eg. the code sample for function expression above. | **Q: Difference between function statement and expression**  The major difference between these two lies in Hoisting.  a(); // "Hello A"  b(); // TypeError  function a() {  console.log("Hello A");  }  var b = function () {  console.log("Hello B");  };  // Why? During mem creation phase a is created in memory and function assigned to a. But b is created like a variable (b:undefined) and until code reaches the function() part, it is still undefined. So it cannot be called. |
| **Q: What is Named Function Expression?**  Same as Function Expression but function has a name instead of being anonymous.  var b = function xyz() {  console.log("b called");  };  b(); // "b called"  xyz(); // Throws ReferenceError:xyz is not defined.  // xyz function is not created in global scope. So it can't be called. | **Q: Parameters vs Arguments?**  var b = function (param1, param2) {  // labels/identifiers are parameters  console.log("b called");  };  b(arg1, arg2); // arguments - values passed inside function call |

|  |  |
| --- | --- |
| **Q: What is First Class Function aka First Class Citizens?**  We can pass functions inside a function as arguments and /or return a function(HOF). These ability are altogether known as First class function. It is programming concept available in some other languages too.  var b = function (param1) {  console.log(param1); // prints " f() {} "  };  b(function () {}); | **// Other way of doing the same thing:**  var b = function (param1) {  console.log(param1);  };  function xyz() {}  b(xyz); // same thing as prev code  **// we can return a function from a function:**  var b = function (param1) {  return function () {};  };  console.log(b()); //we log the entire fun within b. |



|  |
| --- |
| **1. call()**  The call() method invokes a function with a specified this context and individual arguments.  const person = {  fullName: function(city, country) {  return `${this.firstName} ${this.lastName} from ${city}, ${country}`;  }  };  const person1 = { firstName: "John", lastName: "Doe" };  console.log(person.fullName.call(person1, "New York", "USA"));  // Output: John Doe from New York, USA |

|  |
| --- |
| **2. apply()**  The apply() method is similar to call(), but it takes arguments as an array (or array-like object).  const person = {  fullName: function(city, country) {  return `${this.firstName} ${this.lastName} from ${city}, ${country}`;  }  };  const person1 = { firstName: "Jane", lastName: "Smith" };  console.log(person.fullName.apply(person1, ["Los Angeles", "USA"]));  // Output: Jane Smith from Los Angeles, USA |

|  |
| --- |
| **3. bind()**  The bind() method creates a new function that, when called, has a specific this value and optional arguments pre-set. It doesn’t invoke the function immediately but returns a new bound function.  const person = {  fullName: function() {  return `${this.firstName} ${this.lastName}`;  }  };  const person1 = { firstName: "Alice", lastName: "Brown" };  const getFullName = person.fullName.bind(person1);  console.log(getFullName());  // Output: Alice Brown |

**Difference between primitive and non-primitive datatypes in JavaScript**

|  |  |
| --- | --- |
| **1. Primitive Data Types**  Primitive types are immutable (cannot be changed), and they are directly stored in memory by value.  **Characteristics:**  **Immutable:**  The value of a primitive type cannot be changed once created. Any operation that seems to modify a primitive actually creates a new value.  **Example:**  let str = "hello";  str[0] = "H"; // This does nothing because strings are immutable.  console.log(str); // Output: "hello” | **Stored by Value:**  When a primitive is assigned to a variable, a copy of the value is stored. Modifying one variable does not affect others.  **Example:**  let a = 5;  let b = a; // Copy of `a` is stored in `b`  b = 10;  console.log(a); // Output: 5  console.log(b); // Output: 10 |

**Six Types:** string ,number ,bigint, Boolean ,undefined

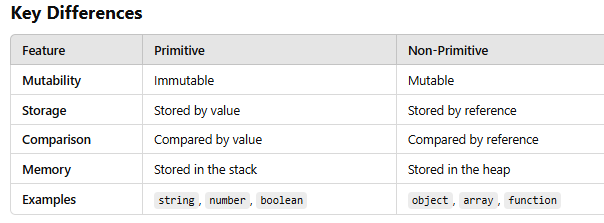
|  |  |
| --- | --- |
| **Symbol Comparison**:  Compared by their values.  Example:  let x = 10;  let y = 10;  console.log(x === y); // Output: true | **Common Operations:**  Although primitives are immutable, operations like concatenation or arithmetic create new values.  let greeting = "Hello";  greeting += " World"; // Creates a new string  console.log(greeting); // Output: "Hello World" |

**2. Non-Primitive Data Types**

Non-primitive types are mutable and stored by reference.

|  |  |
| --- | --- |
| Characteristics:  **Mutable:**  The contents of a non-primitive type can be changed after creation.  Example:  let obj = { key: "value" };  obj.key = "newValue"; // Modifies the same object  console.log(obj); // Output: { key: "newValue" } | **Stored by Reference:**  Variables that hold non-primitives store a reference (memory address) to the actual data. Changing the value via one reference affects all references.  Example:  let arr1 = [1, 2, 3];  let arr2 = arr1; // Both refer to the same array  arr2.push(4);  console.log(arr1); // Output: [1, 2, 3, 4]  console.log(arr2); // Output: [1, 2, 3, 4] |

|  |  |
| --- | --- |
| **Compared by reference, not value. Two objects with the same content are considered different unless they reference the same object.**  Example:  let obj1 = { key: "value" };  let obj2 = { key: "value" };  console.log(obj1 === obj2); // Output: false | **Common Mutations:**  Since non-primitives are mutable, you can directly modify their properties or elements.  let person = { name: "John" };  person.age = 30; // Adding a new property  console.log(person); // Output: { name: "John", age: 30 } |



**Practical Implications**

|  |  |
| --- | --- |
| **Copy Behavior:**  Primitives create independent copies.  Non-primitives share references, leading to side effects.  **// Primitive Example**  let a = 10;  let b = a; // Copies value  b = 20;  console.log(a); // Output: 10  **// Non-Primitive Example**  let arr1 = [1, 2];  let arr2 = arr1; // Shares reference  arr2.push(3);  console.log(arr1); // Output: [1, 2, 3] | **Cloning:**  For non-primitives, creating independent copies requires deep cloning.  Example:  let obj1 = { key: "value" };  let obj2 = { ...obj1 }; // Shallow copy  obj2.key = "newValue";  console.log(obj1.key); // Output: "value"  **Memory Efficiency:**  Non-primitives being stored in the heap allows efficient sharing of large structures, but it requires careful management to avoid unintended mutations. |

**1. Shallow Cloning**

Shallow cloning creates a new object or array but only copies the top-level properties. If the original object contains nested objects or arrays, the references to those nested objects are copied rather than the objects themselves.

|  |  |
| --- | --- |
| **Examples of Shallow Cloning**  Using the Spread Operator (...)  let original = { name: "Alice", address: { city: "Wonderland" } };  let shallowClone = { ...original };  shallowClone.name = "Bob";  **// Modifies only the top-level property**  shallowClone.address.city = "Gotham"; **// Modifies the nested object in both**  console.log(original.address.city);  **// Output: "Gotham**" | **Using Object.assign**  let original = { name: "Alice", address: { city: "Wonderland" } };  let shallowClone = Object.assign({}, original);  shallowClone.name = "Bob"; // Top-level change  shallowClone.address.city = "Gotham"; // Affects both because `address` is shared  console.log(original.address.city); // Output: "Gotham |

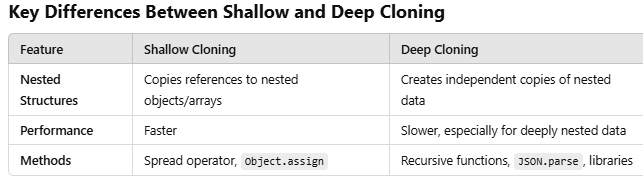
|  |
| --- |
| **Shallow Clone for Arrays**  let arr = [1, 2, [3, 4]];  let shallowClone = [...arr];  shallowClone[0] = 99; // Changes only the first element  shallowClone[2][0] = 33; // Changes the nested array for both  console.log(arr); // Output: [1, 2, [33, 4]] |

**2. Deep Cloning**

Deep cloning creates a completely independent copy, including all nested objects and arrays. Changes to the clone do not affect the original.

|  |  |
| --- | --- |
| **Examples of Deep Cloning**  Using JSON.parse and JSON.stringify  This method works well for simple objects but fails if the object contains functions, special objects like Date, Map, or Set.  let original = { name: "Alice", address: { city: "Wonderland" } };  let deepClone = JSON.parse(JSON.stringify(original));  deepClone.address.city = "Gotham";  console.log(original.address.city); // Output: "Wonderland" | **Using a Recursive Function**  **For more complex objects, you can write a custom recursive function.**  function deepClone(obj) {  if (obj === null || typeof obj !== "object") return obj;  let copy = Array.isArray(obj) ? [] : {};  for (let key in obj) {  if (obj.hasOwnProperty(key)) {  copy[key] = deepClone(obj[key]);  // Recursively clone nested objects/arrays  } }  return copy; }  let original = { name: "Alice", address: { city: "Wonderland" } };  let deepCloneObj = deepClone(original);  deepCloneObj.address.city = "Gotham";  console.log(original.address.city); // Output: "Wonderland" |

|  |
| --- |
| Using Lodash Library (\_.cloneDeep)  **Lodash provides a reliable utility for deep cloning.**  const \_ = require("lodash");  let original = { name: "Alice", address: { city: "Wonderland" } };  let deepClone = \_.cloneDeep(original);  deepClone.address.city = "Gotham";  console.log(original.address.city);  // Output: "Wonderland" Deep Cloning for Arrays  let original = [1, 2, [3, 4]];  let deepCloneArray = JSON.parse(JSON.stringify(original));  deepCloneArray[2][0] = 99;  console.log(original); // Output: [1, 2, [3, 4]] |



**Arrow functions**

Arrow functions in JavaScript are a concise way to write anonymous functions. They provide a more concise syntax compared to traditional function expressions.

Key differences and features of arrow functions:

**Conciseness:** Arrow functions are often more concise than traditional functions, especially when dealing with simple, single-expression functions.

**Implicit Return:** If the arrow function body consists of a single expression, it is implicitly returned without the need for the return keyword.

**No Binding of this:** Arrow functions do not bind their own this value. Instead, they inherit the this value from the enclosing execution context (lexical scoping).

**No arguments Object:** Arrow functions do not have their own arguments object.

**Concise Syntax:**

|  |  |
| --- | --- |
| Single-line functions return the result implicitly without using the return keyword:  const add = (a, b) => a + b;  console.log(add(2, 3)); // Output: 5 | For multi-line functions, you need curly braces {} and an explicit return:  const subtract = (a, b) => {  return a - b;  }; |

**Lexical this Binding:**

Arrow functions do not have their own this. Instead, they inherit this from the surrounding scope (lexical scoping). This makes them especially useful in callback functions and event handlers.

|  |  |
| --- | --- |
| Example:  function Person(name) {  this.name = name;  setTimeout(() => {  console.log(this.name); // Arrow function inherits 'this' from Person  }, 1000);  }  const john = new Person("John"); // Output: John | **Arrow Funtion**  const obj = {  name: "Alice",  sayName: () => {  console.log(this.name); // 'this' is undefined  } };  obj.sayName(); // Output: undefined  **Normal Function**  const obj = {  name: "Bob",  sayName() {  console.log(this.name);  }  };  obj.sayName(); // Output: Bob |

|  |  |
| --- | --- |
| **Cannot Be Used as Constructors:**  Arrow functions do not have a [[Construct]] method and cannot be used with new:  const Foo = () => {};  // new Foo(); // TypeError: Foo is not a constructor | **Implicit Return:**  If the body of the function is a single expression, it will be returned automatically:  const square = x => x \* x;  console.log(square(4)); // Output: 16 |

**No arguments Object:**

Arrow functions do not have their own arguments object. Use rest parameters instead:

|  |  |
| --- | --- |
| const sum = (...args) => args.reduce((total, num) => total + num, 0);  console.log(sum(1, 2, 3)); // Output: 6 | function sum() {  let total = 0;  // The arguments object holds all arguments passed to the function  for (let i = 0; i < arguments.length; i++) {  total += arguments[i];  }  return total; }  // Calling the function with multiple arguments  console.log(sum(1, 2, 3, 4, 5)); // Output: 15  console.log(sum(10, 20, 30)); // Output: 60  console.log(sum()); // Output: 0 |

|  |  |
| --- | --- |
| **No Prototype:**  Arrow functions do not have a prototype property.  **// Normal function**  function NormalFunction() {}  **// Arrow function**  const ArrowFunction = () => {};  **// Checking for prototype property**  console.log("Normal Function's Prototype:", NormalFunction.prototype); // Output: {}  console.log("Arrow Function's Prototype:", ArrowFunction.prototype); // Output: undefined  **// Adding a method to the prototype of the normal function**  NormalFunction.prototype.sayHello = function() {  console.log("Hello from NormalFunction!");  }; | **// Trying to add a method to the "prototype" of the arrow function**  try {  ArrowFunction.prototype.sayHello = function() {  console.log("Hello from ArrowFunction!");  };  } catch (error) {  console.log("Error when accessing prototype of ArrowFunction:", error.message);  }  **// Instantiating an object using the normal function**  const normalInstance = new NormalFunction();  normalInstance.sayHello(); // Output: "Hello from NormalFunction!"  **// Trying to instantiate an object using the arrow function (this will throw an error)**  try {  const arrowInstance = new ArrowFunction();  } catch (error) {  console.log("Error when using 'new' with ArrowFunction:", error.message);  } |

