# **Disclaimer:**

COPYRIGHT NOTICE

©All content, materials, and intellectual property contained within this document, including but not limited to text, images, diagrams, graphics, code, and any other original works, are protected by copyright laws and international treaties. This intellectual property is the sole and exclusive property of SRIKANTH TEKUMUDI

Unauthorized use, reproduction, distribution, modification, or transmission of any portion of this document, in any form or by any means, electronic or mechanical, without the prior written permission of SRIKANTH TEKUMUDI, is strictly prohibited and may result in severe civil and criminal penalties.

By accessing, viewing, or using this document, you acknowledge and agree to abide by all applicable copyright laws and the terms and conditions outlined herein. This copyright notice serves as a legal warning and reminder that the unauthorized use of this intellectual property is strictly prohibited and will be vigorously enforced.

Thank you for respecting the intellectual property rights of SRIKANTH TEKUMUDI

# **1)what is closure**

| A closure is a feature in JavaScript where an inner function has access to its own scope and the parent lexical environment in which it was defined, even after the outer function has returned. This means that the inner function "closes over" its surrounding state and can still access it, even if the outer function has finished executing.  In simpler terms, a closure is a way to access variables from an outer function inside an inner function, even after the outer function has completed execution   function outer(x,y){  function inner(){  console.log(x,y)  }   return inner  }   let x=outer(2,3)  x()  In the provided code example, the outer() function takes in two arguments x and y, and defines an inner function inner(). When inner() is returned from outer(), it still has access to the x and y variables from outer() due to the closure created. Finally, the let x=outer(2,3) line assigns the returned inner() function to the variable x, which is then called with x(). This results in the values of x and y being logged to the console. |
| --- |

# **2)what is function binding,call,apply**

| Function binding, call, and apply are all concepts in JavaScript that allow you to manipulate the context of a function. The context refers to what this keyword points to within a function.  Here's a brief explanation of each:  Function Binding Function binding allows you to explicitly set the value of this for a function. You can do this using the bind() method of a function, which returns a new function with the this value set to the first argument passed to bind().  In the code snippet you provided, greet.sayHello.bind(greet) creates a new function that has its value set to the greet object. When you call this new function, the this keyword inside the sayHello() function will point to the greet object.  Call The call() method allows you to invoke a function and explicitly set its this value to the first argument passed to call(). Additional arguments can be passed to the function as well. In the code snippet, sayHello.call(greet, 25) invokes the sayHello() function with this set to the greet object and the age argument set to 25.  Apply The apply() method is similar to call(), but it takes an array of arguments instead of listing them individually. In the code snippet, sayHello.apply(greet, [25]) is equivalent to sayHello.call(greet, 25).  Here's the updated code snippet with comments:  // Function binding var greet = {  fname: "srikanth",  lname: 'tekumudi',   sayHello: function() {  console.log(this.fname, this.lname);  } };  greet.sayHello(); *// Output: "srikanth tekumudi" - works fine*  *// Using function binding to fix "undefined" error* let myGreet = greet.sayHello; myGreet(); *// Output: "undefined undefined"*  *// Use bind() to set the context of the function* myGreet = greet.sayHello.bind(greet); myGreet(); *// Output: "srikanth tekumudi"*  *// Using call()* function sayHello(age) {  console.log(this.fname, this.lname, age); }  sayHello.call(greet, 25); *// Output: "srikanth tekumudi 25"*  *// Using apply()* sayHello.apply(greet, [25]); *// Output: "srikanth tekumudi 25"* |
| --- |

# **3)Call vs Bind vs Apply**

| In JavaScript, functions can be called in different ways, and each method has its own way of handling the "this" keyword and passing arguments. Here is a brief explanation of the differences between call, bind, and apply:  Bind: The bind method returns a new function with a specific "this" value and any arguments you provide. It does not call the original function, but instead returns a new function that you can call later. This is useful when you want to create a function with a specific "this" value that you can call later.  Call: The call method allows you to call a function with a specific "this" value and arguments passed in one by one. The first argument of the call method is the value of "this" inside the function, and the rest of the arguments are the arguments that will be passed to the function.  Apply: The apply method is similar to the call method, but instead of passing arguments one by one, you pass an array of arguments. The first argument of the apply method is the value of "this" inside the function, and the second argument is an array of arguments that will be passed to the function. |
| --- |

# **4)what is prototypal inheritance**

| prototypal inheritance refers to ability to access object properties from another object means sharing of properties, methods across objects can be done by prototypal inheritance  Prototypal inheritance is a type of object-oriented programming inheritance in which an object inherits properties and methods from another object, known as its prototype. In this model, objects are created by cloning or copying an existing object, which serves as the prototype.  In prototypal inheritance, each object in a program has a special internal property called [[Prototype]] (also sometimes called \_\_proto\_\_ in JavaScript). This property holds a reference to another object, known as its prototype.  prototype is a property of a constructor function in JavaScript, while \_\_proto\_\_ is a property of every object.  When you create a constructor function in JavaScript, it automatically gets a prototype property, which is an object. This object is used as the prototype for all objects created using the constructor function as a constructor with the new keyword. For example:  Prototypal inheritance is used in many programming languages, including JavaScript, which uses prototypal inheritance as its primary mechanism for object-oriented programming. |
| --- |

# **5)different ways to create object**

| In JavaScript, there are several ways to create objects. Here are four common ways:  Object literals: This is the simplest and most common way to create an object in JavaScript. You can create an object by defining its properties and values within curly braces. Example:  javascript Copy code let person = {  name: "srikath",  age: 20 }; Constructor objects: You can create objects using constructor functions. A constructor function is a special type of function that is used to create an object. Inside the constructor function, you can define the properties and methods of the object using the "this" keyword. Example:  javascript Copy code function Person(name, age) {  this.name = name;  this.age = age; } let p = new Person("srikanth", 20); console.log(p instanceof Person); *// true* Class-based objects: In ES6, you can create objects using classes. Classes are a syntactic sugar for constructor functions. You can define the properties and methods of the object using the "constructor" and "method" keywords. Example:  javascript Copy code class Person {  constructor(name, age) {  this.name = name;  this.age = age;  }  greet() {  console.log(this.name, this.age);  } } let p = new Person("srikanth", 10); Object.create() method: You can create an object by using the Object.create() method. This method creates a new object and sets its prototype to the object you pass as the first argument. You can also add properties and methods to the new object by passing an object as the second argument. Example:  javascript Copy code let person = {  name: "srikath",  age: 20 }; let specialPerson = Object.create(person, {sex: 'male', lname: "tekumudi"}); |
| --- |

# **5)what is event bubbling and event delegation**

| Event bubbling and event delegation are two methods of event propagation in the HTML DOM when an event occurs in an element inside another element, and both elements have registered a handler for that event.  Event bubbling means that the event is first captured and handled by the innermost element and then propagates to the outer element. In contrast, event capturing means that the event is first captured by the outermost element and then propagates to inner elements.  To control the event propagation mode, the addEventListener method can be used with the useCapture parameter. If useCapture is true, the event uses the capturing propagation mode. If useCapture is false (or not specified), the event uses the bubbling propagation mode.  Here's an example of event bubbling:  <!DOCTYPE html> <html>  <head>  <title>Event Bubbling Example</title>  </head>  <body>  <div id="outer">  <div id="inner">  <button id="button">Click me!</button>  </div>  </div>  <script>  const outer = document.querySelector("#outer");  const inner = document.querySelector("#inner");  const button = document.querySelector("#button");   outer.addEventListener("click", () => {  console.log("Outer clicked");  });   inner.addEventListener("click", () => {  console.log("Inner clicked");  });   button.addEventListener("click", () => {  console.log("Button clicked");  });  </script>  </body> </html> In this example, when the button is clicked, the event first propagates to the inner element, then to the outer element. Therefore, the output will be:  Button clicked Inner clicked Outer clicked Event delegation is a technique that involves attaching a single event listener to a parent element, rather than to each child element. When the event occurs on a child element, the event bubbles up to the parent element, which can then handle the event based on the target element. This can improve performance and simplify code in cases where multiple child elements need the same event listener.  Here's an example of event delegation:  php Copy code <!DOCTYPE html> <html>  <head>  <title>Event Delegation Example</title>  </head>  <body>  <ul id="list">  <li>Item 1</li>  <li>Item 2</li>  <li>Item 3</li>  <li>Item 4</li>  </ul>  <script>  const list = document.querySelector("#list");   list.addEventListener("click", (event) => {  if (event.target.tagName === "LI") {  console.log(`Clicked ${event.target.textContent}`);  }  });  </script>  </body> </html> In this example, a single event listener is attached to the ul element. When a li element is clicked, the event bubbles up to the ul element, which then handles the event based on the target element. Therefore, the output will be:  Clicked Item 1 Clicked Item 2 Clicked Item 3 Clicked Item 4 |
| --- |

# **6)what is hoisting**

| What is hoisting in JavaScript? Hoisting is a JavaScript mechanism that moves variables and function declarations to the top of their respective scopes during the creation phase of the execution context. This means that variables and functions can be used before they are declared. However, it is important to note that only declarations are hoisted, not the assignments.  What is variable hoisting? Variable hoisting is the mechanism by which all variables declared with the var keyword are initialized with a value of undefined and moved to the top of their respective scope during the creation phase of the execution context.  Can let and const variables be hoisted? Yes, let and const variables are hoisted, but they are not initialized with a value. Instead, they are in a "temporal dead zone" until they are declared in the code. Accessing them before their declaration results in a ReferenceError.  What is function hoisting? Function hoisting is the mechanism by which all function declarations, whether named or anonymous, are moved to the top of their respective scope during the creation phase of the execution context. This means that functions can be called before they are declared in the code.  Are function expressions and arrow functions hoisted? No, function expressions and arrow functions are not hoisted. They must be declared before they can be used, just like variables declared with the let and const keywords. If you try to call a function expression or arrow function before it is declared, you will get a TypeError. |
| --- |

# **7)what is use of this keyword**

| In JavaScript, this is a keyword that refers to the current object or context in which code is being executed.   The value of this is determined by the way a function is called  Global context: When this is used in the global scope, it refers to the global object. In a browser environment, this is the window object.  console.log(this); // "window" object in a browser environment, or "global" object in Node.js  Object context: When a function is called as a method of an object, this refers to the object itself. For example:  const person = {  name: 'John',  sayHello() {  console.log(`Hello, my name is ${this.name}`);  } };  person.sayHello(); *// "Hello, my name is John"*  Constructor context: When a function is called with the new keyword to create a new object, this refers to the newly created object. For example:  function Person(name) {  this.name = name; }  const john = new Person('John'); console.log(john.name); *// "John"*  Function context: When a function is called as a standalone function, this refers to the global object in non-strict mode, or undefined in strict mode. For example:  function sayHello() {  console.log(`Hello, my name is ${this.name}`); }  sayHello(); *// "Hello, my name is undefined" in strict mode, or "Hello, my name is [window.name]" in non-strict mode*    1) in most cases this is determined by how the function is called. It can't be set during execution phase  2)this alway refers to object   es5 introduces bind method to set values of functions this regardless how function invoked  es2015 introduces arrow functions which don't have it's own this(it retails this of enclosing context)   global context:  in global context this refers to global object whether in strict mode or not  in strict-mode this is undefined  function context:  in this , this refers to the method called..   function f1(){  return this  }  f1()==> window ==>in browser  f1==>globalthis ==>in node |
| --- |

# **8)differences between classes and function**

| Functions and classes are both important concepts in JavaScript for creating reusable code, but they have some key differences.  Syntax Functions can be declared using function declarations, function expressions, arrow functions, and more. Classes, on the other hand, have a specific syntax that was introduced in ES6.  Hoisting Function declarations are hoisted to the top of their scope, which means they can be called before they are declared. However, class declarations are not hoisted.  Use of this When using this keyword inside a function, it refers to the object that called the function. However, when using this inside a class, it refers to the class instance that is created using the new keyword.  new keyword The new keyword can be used to create instances of both functions and classes. When a function is called using the new keyword, it creates a new object instance and sets this to point to that instance. Classes also use the new keyword to create new instances.  Strict mode Classes are always in strict mode, which means they have stricter syntax rules and some behaviors are changed. Functions can be used with or without strict mode.  Inheritance Classes in JavaScript support inheritance through the use of the extends keyword, which allows one class to inherit properties and methods from another class. Functions can be used to create object prototypes for inheritance, but the syntax is not as straightforward as using classes. |
| --- |

# **9)what is difference between proto and \_\_proto**

| In JavaScript, every object has a special property called \_\_proto\_\_ (pronounced dunder proto or underscore underscore proto) which refers to the object's prototype. On the other hand, prototype is a property that is only defined on functions. Here's an example to illustrate the difference between prototype and \_\_proto\_\_: javascriptCopy code function Animal(name) {  this.name = name; }  Animal.prototype.sayName = function() {  console.log(`My name is ${this.name}`); }  let cat = new Animal("Fluffy");   In the example above, Animal is a constructor function that takes a name parameter and sets it as an instance variable. The sayName method is added to the Animal constructor's prototype object. When we create a new instance of Animal using the new keyword, the \_\_proto\_\_ property of cat is set to Animal.prototype. This means that cat inherits any properties and methods defined on Animal.prototype, including sayName. javascriptCopy code console.log(cat.\_\_proto\_\_ === Animal.prototype); // true cat.sayName(); // "My name is Fluffy"   The prototype property is used when creating new objects with the new keyword. When we create a new object using new Animal(), the new object's \_\_proto\_\_ property is set to Animal.prototype. javascriptCopy code console.log(Animal.prototype.constructor === Animal); // true   We can also add properties and methods to the Animal constructor's prototype object after it's been defined, and they will be available to all instances of Animal. javascriptCopy code Animal.prototype.eat = function(food) {  console.log(`${this.name} is eating ${food}`); }  cat.eat("fish"); *// "Fluffy is eating fish"*   the prototype property is a property on functions that allows you to add properties and methods to instances created from that function, while the \_\_proto\_\_ property is a property on objects that refers to the object's prototype |
| --- |

# **10)What is a class in javascript?**

| In JavaScript, a class is a special type of object that provides a template for creating objects. It is introduced in ECMAScript 2015 (ES6) and is used to create objects using the class keyword. A class defines a set of properties and methods that are common to all instances of the class.  Here's an example of a class in JavaScript:  javascript Copy code class User {  constructor(name) {  this.name = name;  }   sayHi() {  console.log(this.name);  } }  let user = new User("Srikath"); user.sayHi(); In this example, the User class has a constructor method that takes a name parameter and initializes the name property of the instance. It also has a sayHi method that logs the name property to the console.  To create an instance of the User class, you use the new keyword followed by the name of the class and any arguments required by the constructor. In this case, we create a new User instance with the name "Srikath" and call the sayHi method on it.  It's worth noting that classes in JavaScript are just syntactic sugar over the existing prototype-based inheritance model. The code you wrote using a function constructor and prototype would achieve the same result as the class example. However, classes make it easier to define and manage object-oriented code in JavaScript. |
| --- |

# **11)Explain about Symbol**

| Symbols are a new primitive type introduced in ES6. Symbols are completely unique identifiers. Just like their primitive counterparts (Number, String, Boolean), they can be created using the factory function Symbol() which returns a Symbol.  Every symbol returned by Symbol() is unique, so every symbol has its own identity:  > typeof symbol 'symbol' |
| --- |

# **12)Explain about iterator**

| In JavaScript, an iterator is an object that provides a way to access the elements of a collection one at a time,  To use an iterator, you first create an instance of the iterator by calling the Symbol.iterator method on the collection.  This returns an object that has a next method, which you can call repeatedly to get the next element of the collection. Each time you call next, the iterator returns an object with two properties: value, which is the next element of the collection, and done,  which is a boolean indicating whether you have reached the end of the collection.  const arr = [1, 2, 3]; const iterator = arr[Symbol.iterator]();  console.log(iterator.next()); *// { value: 1, done: false }* console.log(iterator.next()); *// { value: 2, done: false }* console.log(iterator.next()); *// { value: 3, done: false }* console.log(iterator.next()); *// { value: undefined, done: true }* |
| --- |

# **13)Explain about Generator**

| In JavaScript, a generator is a special type of function that can be paused and resumed. When a generator is called, it doesn't run all the way through like a normal function; instead, it can be paused at any point using the yield keyword, and then resumed later from where it left off  infinite loop using generator  function\* counter() {  let i = 0;  while (true) {  yield i;  i++;  } }  const gen = counter();  console.log(gen.next().value); *// 0* console.log(gen.next().value); *// 1* console.log(gen.next().value); *// 2*  use cases of   Lazy evaluation: Generators are often used for lazy evaluation of data. Instead of generating all of the data at once, you can generate it on-demand as it is needed.  Asynchronous programming: Generators can be used with promises to simplify asynchronous programming. You can use a generator function to pause and resume execution while waiting for promises to resolve.  Iterating over large datasets: Generators can be used to iterate over large datasets that don't fit into memory. Instead of loading the entire dataset into memory, you can generate the data on-the-fly and process it one chunk at a time.  Customizing iteration: Generators provide a way to customize the iteration process. For example, you can skip or filter certain elements, or you can generate elements in a specific order.  Generating sequences: Generators can be used to generate sequences of data, such as Fibonacci numbers or prime numbers.  Infinite loops: Generators can be used to create infinite loops  Here's an example of how to use a generator function to iterate over an array one chunk at a time: function\* chunkArray(array, chunkSize) {  for (let i = 0; i < array.length; i += chunkSize) {  yield array.slice(i, i + chunkSize);  } }  *// Example usage:* const myArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];  for (const chunk of chunkArray(myArray, 3)) {  console.log(chunk); }  *// Output:* *// [1, 2, 3]* *// [4, 5, 6]* *// [7, 8, 9]* *// [10]* |
| --- |

# **14)What is the purpose of the new keyword?**

t

| he purpose of new keyword is to invoke a function with this point to empty object link that object to other object return this  In JavaScript, the new keyword is used to create a new instance of an object created by a constructor function or a class. When used with a constructor function, it performs the following steps:  Creates a new empty object. Sets this keyword to point to the newly created object. Calls the constructor function with the provided arguments. sets instance of object \_\_proto to its class function Returns the newly created object, unless the constructor function explicitly returns a non-primitive value (in which case, that value is returned instead). |
| --- |

# **15)explain async, await, when these promises already exist.**

| Asynchronous programming in JavaScript is typically done using Promises, which provide a way to handle the results of an asynchronous operation once it completes. However, the use of Promises can sometimes result in complex, difficult-to-read code, particularly when you have to chain multiple Promises together.  This is where async/await comes in. Async/await is a syntax for handling asynchronous operations that makes the code look more like synchronous code, which can be easier to read and understand.  In JavaScript, you can use the async keyword to define a function that returns a Promise. Within an async function, you can use the await keyword to pause the execution of the function until a Promise is resolved. Here's an example:  async function fetchData() {  const response = await fetch('<https://example.com/data>');  const data = await response.json();  return data; }  In this example, the fetchData function is declared as async. Within the function, the await keyword is used to pause the execution until the fetch Promise resolves. Once the response is received, the await keyword is used again to pause the execution until the JSON is parsed. The parsed JSON is then returned as the result of the function.  Async/await is particularly useful when you need to perform multiple asynchronous operations in a specific order. With Promises, this often requires nested callbacks or chained then calls, which can be difficult to read and reason about. Async/await simplifies this by allowing you to write asynchronous code in a more synchronous style. |
| --- |

# **16)promise (resolve, reject)**

| In JavaScript, a Promise is an object that represents a value that may not be available yet, but will be resolved at some point in the future. Promises are commonly used for asynchronous programming, where a task may take some time to complete and you don't want to block the rest of the code from executing.   - a promise is used to handle asynchronous results of operation. But javascript is synchronous language  -with promises we can defer execution of code until async request is completed  states of promises;  peinding: initial state before promise is succeed or fail  resolved: promise completed  rejected: failed promise'  To use a Promise, you can attach then and catch methods to it, which will be called when the Promise is resolved or rejected, respectively. For example:   ex: const mypromise-new promise((resove, reject)=>{  let condition=true  if(condition){  resolve("promise resolve")  }  else{  reject("promise reject")  }  })   using promises:  mypromise. Then((res)=>{  console. Log(res)  }). Catch((err)=>console. Log(err))   promises act as placeholder hoping data to get back from web browser  promises contain following properties  {  value: , response  onfulfillment: [getdata] *//it holds function and runt with value when task completes*  }   functions display(data){  console. Log(data)  }   const futuredata=fetch('https: //') *//so when data return futuredata. Value='hi'*   futuredata. Then(display)*//future. Onfullfillment=[display]*  promise. All([p1, p2, p3]). Then(console. Log(results)) const allsettled = promises => promise. All(promises. Map(promise => promise . Then(value => ({ status: 'fulfilled', value: value })) . Catch(reason => ({ status: 'rejected', reason: reason })) ));  Promise.all() : takes an array of promises as input, and returns a new promise that resolves with an array of resolved  values from all the input promises, in the same order as the input array.  If any of the input promises are rejected, the entire promise chain will  be rejected with the first error encountered  Promise.allSettled():  is similar to Promise.all(), but it returns a new promise that resolves with  an array of objects representing the final state(either fulfilled or rejected) of  all the input promises, in the same order as the input array.  This method is useful when you need to handle all the input promises,  regardless of whether they were fulfilled or rejected. |
| --- |

# **17)async await vs promises**

| Async/await is a syntactic sugar built on top of promises in JavaScript. It provides a more readable and straightforward way to write asynchronous code that uses promises.  Promises are a powerful tool for handling asynchronous operations in JavaScript. They allow you to write code that can run in parallel and respond to the results of those operations when they complete.  However, working with promises can be challenging, especially when it comes to handling errors or chaining multiple promises together.  Promise chaining can lead to deeply nested code: Promises can be chained together using the then() method, but this can quickly become difficult to read and maintain when multiple promises are chained together  ex: getUserData()  .then(userData => {  getRelatedData(userData.id)  .then(relatedData => {  *// do something with relatedData*  })  .catch(error => {  console.error(error);  });  })  .catch(error => {  console.error(error);  });  Error handling can be tricky: Promises have their own way of handling errors using the catch() method, but it can be easy to forget to add error handling to a chain of promises, which can lead to unhandled rejections  getUserData()  .then(userData => {  *// do something with userData*  return getRelatedData(userData.id);  })  .then(relatedData => {  *// do something with relatedData*  return getMoreData(relatedData.id);  })  .then(moreData => {  *// do something with moreData*  })  .catch(error => {  console.error(error);  });  If any of the promises in the chain are rejected, the error will propagate down the chain and be caught by the final catch block. However, it can be difficult to determine which promise caused the error. |
| --- |

# **18)What is callback hell? explain with example in simple terms**

| it happens when a function requires the result of another function, which in turn requires the result of yet another function, and so on. This can create a pyramid-like structure of nested functions, which can make the code hard to follow and debug.  function foo(callback) {  asyncCall1(function(result1) {  asyncCall2(result1, function(result2) {  asyncCall3(result2, function(result3) {  *// and so on...*  asyncCallN(resultN, function(resultN) {  callback(resultN);  });  });  });  }); } |
| --- |

# **19)what are higherorder function, callback**

| -functions are first class objects in javascript. Because they can be send as parameter and function can be returned from function  functions can be assign to variable  -a function which is send to other function as parameter is called callback  -a function which accepts other function as parameter is called higher order function |
| --- |

# **20)what is currying**

| currying is transformation of functions which is of the form f(a, b, c) => f(a)(b)(c)  ex:  function curry(f){  return function (a){  return function (b){  return function (c){  return f(a, b, c)  }  }   }   function sum(a, b, c){  return a+b+c  }   let currisum=curry(sum)   print(currisum(2)(3)(4)) |
| --- |

# **21)What is polyfill and transpiler (babel)?**

| A polyfill is a piece of code that enables modern functionality in older browsers that do not support it. For example, if a new JavaScript function is introduced that is not supported by an older browser, a polyfill can be used to provide an equivalent function using the available features in the older browser.  A transpiler, such as Babel, is a piece of software that translates source code to another source code. It parses modern code and rewrites it in an older syntax that can be understood by older browsers. This allows developers to write modern code without worrying about browser compatibility issues. For example, if a developer writes code using the latest syntax and features of JavaScript, a transpiler can be used to convert that code into an older syntax that can be understood by older browsers.  For instance, consider the following code snippet:  const height = height ?? 100; This code uses the nullish coalescing operator, which is a new feature introduced in ECMAScript 2020. If a user uses an outdated browser that does not support this feature, the code may not work as expected. However, by using a transpiler like Babel, the code can be converted into an older syntax that can be understood by older browsers:  const height = (height !== undefined && height !== null) ? height : 100; Polyfills are typically used for new language features that are not just new syntax or operators but also include new built-in functions or methods. For example, if a new built-in function is introduced, such as the Math.trunc function that truncates the decimal part of a number, a polyfill can be used to provide an equivalent function for older browsers that do not support it:  if (!Math.trunc) {  Math.trunc = function(n) {  return n < 0 ? Math.ceil(n) : Math.floor(n);  }; } |
| --- |

# **22)what are feature in es6/es 2015**

| New JavaScript features:  - Arrow Functions: shorthand syntax for writing functions, with implicit binding and concise syntax.  - Classes: new syntax for defining classes with support for inheritance and advanced features.  - Template Literals: new way to create strings with variable interpolation and multi-line strings.  - let and const: new variable declaration keywords with more precise scoping  rules than var.  - Destructuring: new syntax for extracting values from arrays and objects into separate variables.  - Spread and Rest Operators: new operators that allow spreading an array into separate arguments or combining multiple arguments into an array.  - Promises: new way to handle asynchronous operations with more readable and maintainable code.  - Generators: new type of function that allows pausing and resuming execution, useful for asynchronous and iterative code.  - Modules: new syntax for defining and importing reusable code, with support for exporting and importing specific functions and objects.  - Default Parameter Values: new syntax for specifying default values for  function parameters. - Object Literal Enhancements: new syntax for defining objects with concise property values and computed property names.  - Symbols: new primitive type for creating unique identifiers that can be used as object keys.  - Map and Set Data Structures: new built-in data structures for storing data collections with advanced features. |
| --- |

/

# **23)What is ecmascript?**

| Ecmascript (european computer manufacturer association) is standard for scripting languages like javascript, jscript in ecma2020 following are added  1)static( infront variable), private field (static keyword),  2)dynamic import import ('. /math. '). Then((math)=> math. Add()) 3)big int |
| --- |

# **24)what is difference between function expression, function statement, anonymous function, arrow functions**

| function f(){  *//function statement*  }   let f=function (){  *//anonymus function , function expression*  }  let f=()=>{  *//arrow function*  } |
| --- |

# **25)how this behaves in arrow function**

| - arrow functions doesn't have it's own binding this or super  -not suitable for call, apply, bind methods which generally relay on scope  -arrow funcitons cannot be used as constructors and throw error  -arrow functions don't have prototype  In arrow functions, this keyword behaves differently than in regular functions.  In a regular function, the value of this is determined based on how the function is called. If the function is called as a method of an object, this will refer to the object that the method was called on. If the function is called without a context, this will refer to the global object (in non-strict mode) or be undefined (in strict mode).  However, in an arrow function, the value of this is determined lexically, based on where the function is defined. Specifically, this will always refer to the value of this in the surrounding lexical scope. In other words, arrow functions do not have their own this binding; instead, they inherit the this value from the surrounding context  const obj = {  name: 'Alice',  greetRegular: function() {  console.log('Hello, my name is ' + this.name);  },  greetArrow: () => {  console.log('Hello, my name is ' + this.name);  } };  obj.greetRegular(); // "Hello, my name is Alice" obj.greetArrow(); // "Hello, my name is undefined" |
| --- |

# **26)map, reduce, filter custom implementations**

| array. Prototype. Mymap = function (f) {  let res=[]  let arr=this  arr. Foreach(x => res. Push(f(x)))  return res;  };   array. Prototype. Myfilter = function (f) {  let arr = this  let res = []   arr. Foreach(x => f(x)==true&& res. Push(x))  return res;   }   array. Prototype. Myreduce = function (cb, init) {  let res = []  let arr = this;  let accumlator=init   for (let i = 0; i < arr. Length; i++){  if (accumlator ! = undefined) {  accumlator = cb. Call(undefined, accumlator, arr[i], i, this)  console. Log(accumlator)  }  else {  accumlator=arr[i]  }  }   return accumlator  }  list=[1,3,34,45,43]  list. Map(x=>x+10) list. Mymap(x=>x+10) |
| --- |

# **27)difference between null and undefined**

| whenever a variable is not assigned to any value.. It is assign to undefined by javascript engine  developers explicitly assign variables to null to avoid this.  null=>it is intentional absence of value  undefined=>it means value not exist in compiler |
| --- |

# **28) why use minified code in production**

| it is used to reduce load times and bandwidth usage on websites it removes spaces, comments and minified code is 48% smalle |
| --- |

# **29)difference between prototype and proto**

| prototype is the property of function object\\   prototypes are only present as properties of functions.   proto is an object with in every object that points to prototype that has been set for that object  \_\_proto\_\_ is an actual object that is used in lookup chain to resolve methods   ex;  function a(name){  this. Name  }   javascript will add prototype property to hai   prototype property is an object with two properties:  1)constructor: a  2)\_\_proto\_\_: object it points to object   var b=new a('sri')   it create empty object  it sets b. \_\_proto\_\_=a. Prototype  it executes a. Prototype. Constructor (which is a definition)  it return newly created object   now if we made it. Prototype. Car="bmw"  so car property will add in prototype object  and call  b. Car (it will search car property in on but doesn't found and got to \_\_proto\_\_ => which is pointed to  a. Prototype and fine there  inheritance in javascript |
| --- |

# **30)context and scope in javascript:**

| scope deals with visibility of variables.  it has 2 scope with var:  1)function scope  2)global scope'   3 scopes with let, const:  1)global, function, block   context is related to object:  1)context refers to this  context is determined how a function invoked  this will default to global or window objects   var obj = {  foo: function() {  return this;  }  };   obj. Foo() === obj; trye   function foo() {  alert(this);  }   foo() *// window*  new foo() *// foo* |
| --- |

# **31)What is method chaining?**

| Calling one method after another on same object, in one continuous one line of  code. Ex: add(3). Mul(4). Divide(5)  class arthemetic{ constructor(){ this. Value=0 } add(n){ this. Value=this. Value+n return this; } mul(n){ this. Value=this. Value\*n return this; } divide(n){ this. Value=this. Value. N return this } } let a=new arthemetic() a. Add(4). Mul(4). Divide(4) |
| --- |

# **32)is javascript compiled or interpreted language**

javascript is interpreted language

# **33)object. Freeze vs object. Seal?**

| object. Freeze makes object immutable means we can't change or add existing properties   let person = {  name: 'srika',  age: 20,  sex: 'male'  }   person. Height = 34.53   object. Freeze(person)   person. Name='satish'  person. Salary='3999'   console. Log(person) //{  name: 'srika',  age: 20,  sex: 'male',  height = 34.53  }   object will not change   object. Seal makes object inextensible but can change value of properties   let person = {  name: 'srika',  age: 20,  sex: 'male'  }   person. Height = 34.53   object. Seal(person)   person. Salary='3999'  person. Name='satish'   console. Log(person) //{  name: 'satish',  age: 20,  sex: 'male',  height = 34.53  }  //existing properties can be changed  The reason for this behavior is that const only prevents reassignment of the variable itself, not the contents of the variable. When an object or an array is assigned to a const variable, the variable holds a reference to the object or array in memory. While the reference cannot be changed, the contents of the object or array can be modified because they are stored elsewhere in memory |
| --- |

# **34)why const values can be mutated if value is array or object**

| The reason for this behavior is that const only prevents reassignment of the variable itself, not the contents of the variable. When an object or an array is assigned to a const variable, the variable holds a reference to the object or array in memory. While the reference cannot be changed, the contents of the object or array can be modified because they are stored elsewhere in memory |
| --- |

# **35)Difference between object. Create and {}**

| The Object.create() method takes an existing object as its prototype and creates a new object that inherits from that prototype. The syntax for Object.create() is as follows:  On the other hand, the object literal syntax {} creates a new object with no prototype. The syntax for {} is as follows  new test():   create new object() obj  set obj. \_\_proto\_\_ to test. Prototype  return test. Call(obj) || obj;  *// normally obj is returned but constructors in js can return a value*  object. Create( test. Prototype )   create new object() obj  set obj. \_\_proto\_\_ to test. Prototype  return obj;  so basically object. Create doesn't execute the constructor.   if an object is made from literal, if change is made to an object , it affects the object in the entire script.  if an object is made from an object. Create() the change is made to object, does not affect object throughout  the script  object. Create() function creates a new object with specified prototype object and properties   let name={name: 'sri', age:20}  name {name: 'sri', age: 20}   let person=object. Create(name)  person {} //no properties  but we can use  person. Name 'sri' |
| --- |

# **36)What is the difference between window and document?**

| window: javascript window is global objects which holds function, history,  function, screen, location   document: document is property of window which hold dom object |
| --- |

# **37)What is the difference between innerhtml and innertext?**

| Innerhtml processes the html tag if found in a string . Where as innertext don't document. Queryselector('p'). Innerhtml='<di> hi srik<div>' it process the  inner html  document. Queryselector('p'). Innertext='<di> hi srik<div>' it doesn't process html |
| --- |

# **38)What is the difference between html collection and nodelist?**

| In the nodelist type we can use arr functions. Queryselectorall() return nodelist  in which foreach can be used directly  in htmlcollection type we cannot use foreach. Getelementsbyclassname() or getelementsbytagname() returns htmlcollection.  we need to convert it to array by array. From(elename) and traversal is possible using foreach |
| --- |

# **39)What is tree shaking?**

| Tree shaking is a technique used to eliminate unused code (or "dead code") from a codebase, resulting in a smaller bundle size and faster loading times. The term "tree shaking" comes from the idea of shaking a tree to remove dead leaves and branches.  In JavaScript, tree shaking typically involves analyzing the code to determine which modules and functions are actually used, and then removing any code that is not used. This is achieved using a tool called a module bundler, such as Webpack, Rollup, or Parcel.  There are several ways to perform tree shaking:  ES6 modules: Using the ES6 module syntax allows the bundler to determine which modules are actually used, as opposed to the CommonJS syntax, which does not allow for static analysis.  Static analysis: Static analysis tools, such as Babel and TypeScript, can help the bundler determine which functions and variables are actually used.  Minification: Minification tools, such as UglifyJS, can remove unused code as part of the minification process.  Optimization plugins: Many bundlers offer optimization plugins that can perform tree shaking automatically, such as the "terser-webpack-plugin" for Webpack. |
| --- |

# **40)difference between set and weakset**

| References to Values: A Set can store references to any type of value, including primitives and objects, and does not remove values until they are explicitly deleted or the Set object is cleared.   A WeakSet, on the other hand, can only store references to objects and removes values automatically when the corresponding object is garbage-collected.  -weakset hold collection of objects  -weak set only holds objects within its collection. (can't store primitive values)   -reference to the objects are held weak. This means whenever there is no other reference towards object  it will be garbage collected   ex: let weakset=new weakset();  obj={}  weakset. Add(obj)  console. Log(weakset. Has(obj)) //return true  obj=5 //reference removed  console. Log(weakset. Has(obj)) //return false |
| --- |

# **service worker vs web worker**

| Service Worker:  \*\*A Service Worker\*\* is a script that runs separately from the web page and provides a persistent background processing environment. It is mainly used for caching web pages and resources, handling push notifications, and performing other tasks that can be performed in the background. Service Workers work as intermediaries between web pages and the network, and they can intercept network requests made by web pages.  Example: A good example of using Service Worker is to make a web application work offline. By caching all the required files, including HTML, CSS, JavaScript, and images, the application can still function even if the user loses internet connectivity.  \*\*Web Worker\*\*:  Web Workers, on the other hand, are used to execute scripts in parallel with the main JavaScript thread, enabling the execution of CPU-intensive tasks without blocking the main thread. This can significantly improve the performance and responsiveness of web applications    In summary, the key difference between Service Worker and Web Worker is that Service Workers run separately from the main thread and can perform background processing, while Web Workers run in parallel with the main thread and can improve application performance.   service workers are proxy between browser and network. By intercepting request may by document, service worker  redirects request to cache.   navigator. Serviceworker. Register('serviceworker. Js')   serviceworker. Js  self. Addeventlistener('activate', (event)=>{   })  self. Addeventlistener('fetch', (event)=>{  event. Respondwith(caches. Match(event. Request))  })   web workers:   Web workers are simple means for running scripts in background threads.  the worker thread can perform io tasks without interfering with the user interface.  once created, a worker can send messages to js code that created it by posting message   it has following methods  let worker=new worker('apple. Js')  worker. Postmessage('do work')  worker. Onmessage=function (e)=>{  console. Log(e. Data)  }   apple. Js   onmessage = function (e) {  let final = 0;  for (int i = 0; i < 10 \*\* 10; i++) {  final+=i  }  return final;  postmessage(final)  } |
| --- |

# **41)how to make a few properties immutable?**

| let name={  firstname: 'srikanth',  lastname: 'tekumudi'  }  objcet. Defineproperty(name, 'firstname': {  writable: false;  configurable: false;  })  object. As([], []) //return false object. As('hello', 'hello') return true; object. As(nan,1/0) return true  checks if two params are same or not |
| --- |

# **32)What is the difference between e. Target and e. Currenttarget**

| \*\*e.target\*\* refers to the actual element that triggered the event, i.e., the element on which the event occurred. For example, if a user clicks on a button element, e.target will refer to that button element. If the same event propagates through multiple nested elements, e.target will always refer to the element that was clicked.  On the other hand, \*\*e.currentTarget\*\* refers to the element to which the event handler was attached, i.e., the element on which the event listener was registered.   <div id="outer">  <div id="inner">  <button>Click me</button>  </div> </div>  const outer = document.getElementById('outer'); const inner = document.getElementById('inner'); const button = document.querySelector('button');  outer.addEventListener('click', function(e) {  console.log(e.target); // will log the clicked element (button)  console.log(e.currentTarget); // will log the outer element });  inner.addEventListener('click', function(e) {  console.log(e.target); // will log the clicked element (button)  console.log(e.currentTarget); // will log the inner element });  button.addEventListener('click', function(e) {  console.log(e.target); // will log the button element  console.log(e.currentTarget); // will also log the button element });  In summary, e.target refers to the actual element that triggered the event, while e.currentTarget refers to the element to which the event listener was attached |
| --- |

# **43)what is difference between spread and rest operator**

| let a=[1,2,3,4,5, ]  let res=[... A] *//it is unpacking array it is spread*   packing of elements is called rest operators   function sum(... Arr){  console. Log(arr)  } |
| --- |

# **44)how to deep freeze object**

| function deepfreeze(object) {  let propnames = object. Getownpropertynames(object);  for (let name of propnames) {  let value = object[name];  object[name] = value && typeof value === "object" ?  deepfreeze(value) : value;  }  return object. Freeze(object);  }  let person = {  name: "leonardo",  profession: {  name: "developer"  }  };  deepfreeze(person);  person. Profession. Name = "doctor"; |
| --- |

# **45)explain pipe and compose?**

| The concept of pipe is simple: it combines n functions.  it's a pipe flowing left-to-right, calling each function with the output of the last one.   previously:  reverse(get6characters(uppercase(getname({ name: 'buckethead' }))));  using pipe  pipe(  getname,  uppercase,  get6characters,  reverse  )({ name: 'buckethead' });   pipe = (... Fns) => (x) => fns. Reduce((v, f) => f(v), x);   pipe = (... Functions) => (value) => {  debugger;   return functions. Reduce((currentvalue, currentfunction) => {  debugger;   return currentfunction(currentvalue);  }, value);  };  compose?   so the main difference between compose and pipe is the order of the composition.  compose performs a right-to-left function composition  since the pipe performs a left-to-right composition. So let's write the pipe high-order function:   const compose = (... Functions) => args => functions. Reduceright((arg, fn) => fn(arg), args);   compose is a high order function. It is a function that returns another function.  compose takes multiple functions as arguments and we convert them into an  array of functions using the spread operator: ...  then we simply apply a reduceright on those functions. The first parameter of the callback is the current argument. The second argument is the current  function. Then we call each function with the current argument and the result  is used for the next call. |
| --- |

# **46)how to deeply copy an object without built-in functions?**

| Function deepcopy(obj) { let newobj = {};  for (let key in obj) {  if (obj. Hasownproperty(key)) {  if (typeof obj[key] === 'object') {  newobj[ky] = deepcopy(obj[key]);  } else {  newobj[key] = obj[key];  } } }  let car={ name: 'bmw', model: 'bmw', year:2018 address: { city: 'bangalore', state: 'karnataka' }  } |
| --- |