1.Compare the differences between let, var, and const when declaring variables in JavaScript.

In JavaScript, `let`, `var`, and `const` are used to declare variables, but they have some differences in terms of scoping, hoisting, and mutability. Let's explore each one separately:

1. `var`:

```
// Example 1
var x = 10;
if (true) {
  var x = 20;
  console.log(x); // Outputs 20
}
console.log(x); // Outputs 20

// Example 2
console.log(y); // Outputs undefined
var y = 5;
console.log(y); // Outputs 5
```

- **Scope:** `var` is function-scoped. It means that the variable is visible throughout the entire function in which it is declared.
- **Hoisting:** Variables declared with `var` are hoisted to the top of their scope. This means you can use the variable before it's declared, but it will have an initial value of `undefined`.

- **Reassignment:** Variables declared with `var` can be redeclared and reassigned.

2. `let`:

```
// Example 1
let a = 10;
if (true) {
  let a = 20;
  console.log(a); // Outputs 20
}
console.log(a); // Outputs 10

// Example 2
// ReferenceError: Cannot access 'b' before initialization
console.log(b);
let b = 5;
console.log(b); // Outputs 5
```

- **Scope:** `let` is block-scoped. It means that the variable is only visible within the block, statement, or expression where it is defined.

- **Hoisting:** Variables declared with `let` are hoisted to the top of their block, but they are not initialized. Accessing the variable before its declaration results in a `ReferenceError` as they get in an area called Temporal Dead Zone

- **Reassignment:** Variables declared with `let` can be reassigned, but not redeclared in the same scope.

3. `const`:

```
// Example 1
const PI = 3.14;
// SyntaxError: Assignment to constant variable
PI = 3.14159;
```

- **Scope:** `const` is also block-scoped.
- **Hoisting:** Like `let`, variables declared with `const` are hoisted to the top of their block, but they are not initialized. Accessing the variable before its declaration results in a `ReferenceError` as they get in an area called Temporal Dead Zone
- **Reassignment:** Variables declared with `const` cannot be reassigned. They are constants. However, if the variable is an object or an array, the properties or elements of the object/array can be modified.

In summary:

- Use `var` when you need function-scoping (though it's less common nowadays).
- Use `let` when you need block-scoping and plan to reassign the variable.
- Use `const` when you need block-scoping and the variable should not be reassigned.

2.What is Hoisting in JS?

Hoisting is a behavior in JavaScript where variable and function declarations are moved to the top of their containing scope during the compilation phase. This means that you can use a variable or a function before it's declared in your code. However, it's important to note that only the declarations are hoisted, not the initializations.

Let's go through some examples to illustrate hoisting in JavaScript:

Example 1: Variable Hoisting

```
console.log(x); // Output: undefined
var x = 5;
```

```
console.log(x); // Output: 5
```

In this example, the variable `x` is hoisted to the top of its scope during the compilation phase. The first `console.log` outputs `undefined` because the declaration is hoisted, but the initialization (`x = 5`) is not hoisted. The second `console.log` outputs `5` after the variable has been initialized.

Example 2: Function Hoisting

```
sayHello(); // Output: "Hello, World!"

function sayHello() {
  console.log("Hello, World!");
}
```

In this example, the function `sayHello` is hoisted to the top of its scope. That's why we can call the function before its declaration. The output is "Hello, World!" as expected.

Example 3: Hoisting with Function Expressions

```
greet(); // TypeError: greet is not a function

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

In this case, the variable `greet` is hoisted, but since it's assigned a function expression (not a function declaration), the assignment is not hoisted. Therefore, when we try to call `greet()` before the assignment, we get a `TypeError` because `greet` is `undefined` at that point.

Example 4: Hoisting Inside Functions

```
function example() {
  console.log(a); // Output: undefined
  var a = 10;
  console.log(a); // Output: 10
}

example();
```

Even within functions, variables are hoisted to the top of their scope. In this example, the variable `a` is hoisted within the function `example`.

It's important to understand hoisting in JavaScript, but it's generally considered good practice to declare and initialize variables at the beginning of their scope to avoid confusion and potential issues.

3.What is Temporal dead zone in JS?

The "temporal dead zone" in JavaScript refers to the period between entering scope and the actual declaration of a variable with `let` or `const`, during which accessing the variable results in a `ReferenceError`. Let me break it down with some code examples:

```
console.log(x); // ReferenceError: Cannot access 'x' before initiali
let x = 10;
```

In this example, even though `x` is declared later in the code, trying to access it before the declaration causes a `ReferenceError`. This is because `x` is in the temporal dead zone until the `let` initialization is encountered.

Same thing Happens with const as well!

4.What is a lexical Scope and Scope Chain in JS?

Let's illustrate lexical scoping and the scope chain with an example:

```
function outerFunction() {
  var outerVar = "I'm outer";

  function innerFunction() {
    var innerVar = "I'm inner";
    console.log(outerVar); // Output: I'm outer
  }

  innerFunction();
}

outerFunction();
```

In this example, `innerFunction` has access to the variable `outerVar` declared in its outer scope (`outerFunction`), but not vice versa. This is because of lexical scoping. The inner function has access to variables declared in its parent functions due to the lexical environment in which it is defined.

The scope chain in this example can be visualized as follows:

1. Global Scope
    - Contains built-in objects and functions like `console`.
2. `outerFunction` Scope
    - Contains `outerVar`.
    - Outer scope of `innerFunction`.
3. `innerFunction` Scope
    - Contains `innerVar`.
    - Inner scope of `outerFunction`.

When `innerFunction` tries to access `outerVar`, JavaScript first looks for `outerVar` in its local scope. If it doesn't find it there, it looks in the

scope of its parent function, which is `outerFunction`'s scope. If `outerVar` is still not found there, it continues up the scope chain until it reaches the global scope. So basically a chain is formed to search for the value from innermost function to the outer functions that are up in the hierarchy but the vice versa is not possible , and this chain is known as Scope Chaining.

So, in this case, `outerVar` is found in `outerFunction`'s scope, and that's where the value is retrieved from , if the value is not found anywhere up in the hierrarchy then an error is returned.

5 What are Closures in JS and how they are formed?

In JavaScript, closure is a fundamental concept that arises from the combination of lexical scope and function definitions.

Lexical scope refers to the idea that a function's scope is determined by its surrounding code at the time of its definition, not at the time of its execution. This means that a function "remembers" the variables that were in scope at the time it was defined, even if it is executed in a different scope.

The scope chain is the mechanism by which JavaScript resolves variable names. When a variable is referenced inside a function, JavaScript first looks for that variable within the function's own scope. If it's not found there, it looks in the next outer scope, and so on, until it reaches the global scope.

Now, when a function is defined within another function, it forms a closure. This means that the inner function has access to the outer function's variables, even after the outer function has finished executing. This is possible because the inner function maintains a

reference to the variables in the outer function's scope, creating a closure over those variables.

Here's an example to illustrate:

```
function outer() {
  var outerVar = "I'm in the outer function";


  function inner() {
    console.log(outerVar); // inner function has access to outerVar
  }

  return inner;
}

var closureFunc = outer(); // outer function has finished executing,
closureFunc(); // Outputs: "I'm in the outer function"
```

In this example, `inner()` forms a closure over the variable `outerVar`, even though `outer()` has already finished executing. This is possible because `inner()` maintains a reference to `outerVar` through the closure, allowing it to access the variable's value when it's called later on.

So, in summary, closure in JavaScript is the combination of lexical scope and function definitions, allowing inner functions to maintain access to variables in their outer scope even after the outer function has finished executing.

6 What are Higher Order Functions and what are the advantages of using Higher order functions?

In JavaScript, a higher-order function is a function that either takes one or more functions as arguments or returns a function as its result, or both. Essentially, it treats functions as first-class citizens, allowing them to be manipulated and passed around as data. Here's a simple example:

```
// Higher-order function example
function applyOperation(operation, x, y) {
  return operation(x, y);
}

function add(x, y) {
  return x + y;
}

function subtract(x, y) {
  return x - y;
}

console.log(applyOperation(add, 5, 3)); // Output: 8
console.log(applyOperation(subtract, 5, 3)); // Output: 2
```

In this example, `applyOperation` is a higher-order function because it takes another function (`operation`) as an argument.

Advantages of using higher-order functions:

1. **Code Reusability**: Higher-order functions promote code reuse by allowing you to abstract common functionality into separate functions. This can lead to more concise and maintainable code.

2. **Abstraction**: They allow you to abstract over actions, promoting a more declarative and expressive coding style.

3. **Encapsulation**: Higher-order functions can encapsulate complex behavior, making it easier to understand and reason about the code.

4. **Functional Composition**: They facilitate functional composition, enabling you to build complex functions by composing simpler ones together.

5. **Flexibility and Extensibility**: Higher-order functions make your code more flexible and extensible by enabling you to pass behavior as arguments or return it as a result.

Overall, higher-order functions are a powerful feature of JavaScript that can lead to more modular, flexible, and expressive code.

7 Explain Map , Filter and Reduce in JS?

In JavaScript, `map`, `filter`, and `reduce` are powerful array methods used for manipulating arrays and performing operations on their elements.

1. **Map**: The `map` method creates a new array by applying a function to each element of the original array. It does not modify the original array.

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

// Example 1: Doubling each number
const doubled = numbers.map((num) => num * 2);
console.log(doubled); // Output: [2, 4, 6, 8, 10]

// Example 2: Converting each number to a string
const stringNumbers = numbers.map((num) => num.toString());
console.log(stringNumbers); // Output: ['1', '2', '3', '4', '5']
```

2. **Filter**: The `filter` method creates a new array with all elements that pass the test implemented by the provided function basically on a condition

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

```
// Example 1: Filtering even numbers
const evenNumbers = numbers.filter((num) => num % 2 === 0);
console.log(evenNumbers); // Output: [2, 4]

// Example 2: Filtering numbers greater than 3
const greaterThanThree = numbers.filter((num) => num > 3);
console.log(greaterThanThree); // Output: [4, 5]
```

3. **Reduce**: The `reduce` method applies a function against an accumulator and each element in the array (from left to right) to reduce it to a single value.

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

// Example 1: Summing all numbers
const sum = numbers.reduce(
  (accumulator, currentValue) => accumulator + currentValue,
  0
);
console.log(sum); // Output: 15

// Example 2: Concatenating all numbers as a string
const concatenatedString = numbers.reduce(
  (accumulator, currentValue) => accumulator + currentValue.toString
  ""
);
console.log(concatenatedString); // Output: '12345'
```

In each example:

- `map` transforms each element of the array according to the function provided.
- `filter` selectively includes elements from the original array based on a condition defined by the provided function.
- `reduce` iterates over each element of the array, accumulating a final result based on the logic defined in the provided function.

8 Write a Polyfill for Array.map() method?

```
// Polyfill for Array.map()

Array.prototype.myMap = function (callback, thisArg) {
  if (this == null) {
    throw new TypeError("Array.prototype.map called on null or unde
  }

  if (typeof callback !== "function") {
    throw new TypeError(callback + " is not a function");
  }

  const newArray = [];
  for (let i = 0; i < this.length; i++) {
    if (i in this) {
      newArray[i] = callback.call(thisArg, this[i], i, this);
    }
  }
  return newArray;
};
```

Now let's break down how this polyfill works:

1. It checks if `Array.prototype.map` does not exist. If it doesn't, it means the browser doesn't support the `map()` method, so we need to provide our own implementation.

2. Inside the polyfill function, it first checks if the `this` value is `null` or `undefined`, as using `map()` on `null` or `undefined` would throw an error in a native implementation.

3. It checks if the `callback` provided is actually a function. If not, it throws a `TypeError`.

4. It creates a new array `newArray` to store the results of applying the callback to each element of the original array.

5. It iterates over each element of the array using a for loop, checking if the current index `i` is present in the array using `if (i in this)` to skip over holes in sparse arrays.

6. For each element, it calls the `callback` function with three arguments: the current element, the current index, and the original array. It uses `call()` to ensure that the `this` value within the callback function is set to `thisArg`, if provided.

7. It stores the result of the callback in the corresponding index of the `newArray`.

8. Finally, it returns the `newArray` containing the results of applying the callback to each element of the original array.

This polyfill provides a basic implementation of `Array.map()` that should work in older browsers or environments lacking support for it.

9 Write a Polyfill for Array.filter() method?

It is extremely common question in JavaScript interviews where it is asked to implement polyfill for the filter() method.

Array.filter polyfill should have these three functionalities.

- The filter() function should take an callback function as an argument.
- Current element, its index, and the context should be passed as an argument to the callback function.
- All the elements which pass text implemented in this callback function should be returned in a new array.

```
Array.prototype.myfilter = function (callback) {
  //Store the new array
```

```
  const result = [];
  for (let i = 0; i < this.length; i++) {
    //call the callback with the current element, index, and context
    //if it passes the text then add the element in the new array.
    if (callback(this[i], i, this)) {
      result.push(this[i]);
    }
  }

  //return the array
  return result;
};
```

10 Write a Polyfill for Array.reduce() method?

`Array.reduce()` is a powerful method in JavaScript used to reduce an array into a single value. If you want to create a polyfill for it, you can follow these steps:

Here's a polyfill for `Array.reduce()`:

```
// Define Array.reduce() if it doesn't exist
Array.prototype.reduce = function (callback, initialValue) {
  // Check if the array is empty and there is no initial value
  if (this.length === 0 && arguments.length < 2) {
    throw new TypeError("Reduce of empty array with no initial valu
  }

  // Set initial accumulator value
  let accumulator = initialValue !== undefined ? initialValue : th

  // Start iteration from the second element if initialValue is not
  let startIndex = initialValue !== undefined ? 0 : 1;

  // Iterate through the array
  for (let i = startIndex; i < this.length; i++) {
    // Invoke callback function with accumulator and current element
    accumulator = callback(accumulator, this[i], i, this);
```

```
    accumulator = callback(accumulator, this[i], i, this);
  }

  // Return the final accumulator value
  return accumulator;
};
```

Now, let's break down this polyfill:

- We first check if `Array.prototype.reduce` already exists.
- If it doesn't exist, we define `Array.prototype.reduce` by assigning it a function.
- This function takes two parameters: a callback function and an optional initialValue.
- Inside the function, we handle the case where the array is empty and no initialValue is provided.
- We set the initial accumulator value based on whether an initialValue is provided or not.
- We start iteration from the second element if an initialValue is not provided.
- We iterate through the array, invoking the callback function with the accumulator, current element, index, and the array itself.
- Finally, we return the final accumulator value.

This is the polyfill for reduce method.

11 What are pure and impure functions in JS?

In JavaScript, pure functions and impure functions serve different purposes and have distinct characteristics:

1. **Pure Functions**:

- A pure function is a function where the return value is determined only by its input values, without observable side effects.
- It doesn't modify variables outside of its scope or perform any I/O operations.
- Given the same input, a pure function will always return the same output.
- Pure functions are predictable and easier to test.

Here's an example of a pure function:

```
function add(a, b) {
  return a + b;
}

console.log(add(2, 3)); // Output: 5
```

In this example, the `add` function takes two parameters and returns their sum. It doesn't modify any variables outside its scope, and given the same inputs, it always produces the same output.

2. **Impure Functions**:
   - An impure function is a function that may produce side effects or depends on external factors.
   - It can modify variables outside of its scope, perform I/O operations, or rely on mutable data.
   - Impure functions may not always produce the same output for the same input, making them less predictable.

Example of an impure function:

```
let result = 0;

function impureAdd(a) {
  result += a; // Modifying external variable 'result'
  return result;
}
```

```
console.log(impureAdd(2)); // Output: 2
console.log(impureAdd(3)); // Output: 5
```

In this example, `impureAdd` modifies the external variable `result` each time it's called. The output of the function depends not only on its input but also on the current state of `result`, making it impure.

In summary, pure functions are predictable and have no side effects, while impure functions may have side effects and rely on external factors.

12 What are Imperative and declarative way of writing code?

let's compare imperative and declarative styles of writing code in JavaScript, particularly focusing on functions.

Imperative Programming: In imperative programming, you define the exact steps to achieve a desired result. This often involves explicitly stating how to do something, step by step.

Example:

```
// Imperative approach to finding the sum of an array
function sumArray(arr) {
  let sum = 0;
  for (let i = 0; i < arr.length; i++) {
    sum += arr[i];
  }
  return sum;
}

console.log(sumArray([1, 2, 3, 4, 5])); // Output: 15
```

In this imperative style, we explicitly loop through the array, keeping track of the sum by accumulating it with each element.

Declarative Programming: In declarative programming, you focus more on what you want to achieve rather than how to achieve it. You describe the desired result without specifying the exact steps.

Example:

```
// Declarative approach to finding the sum of an array using Array.r
function sumArray(arr) {
  return arr.reduce((acc, curr) => acc + curr, 0);
}

console.log(sumArray([1, 2, 3, 4, 5])); // Output: 15
```

In this declarative style, we use the `reduce()` method, which takes a callback function and an initial value. It abstracts away the looping process and accumulates the sum for us.

Comparison:

- **Imperative**: Focuses on the detailed steps of how to achieve a task. It often involves mutable state and explicit control flow (loops, conditionals).
- **Declarative**: Focuses on the desired result or outcome. It often involves higher-order functions and functional composition, abstracting away implementation details.

while imperative programming emphasizes the “how,” declarative programming emphasizes the “what.” Declarative code tends to be more concise and easier to understand once you’re familiar with the paradigms and functions being used.

13 What is Destructuring in JS?

Destructuring in JavaScript is a concise way to extract values from arrays or properties from objects and assign them to variables. It

allows you to unpack values from arrays or objects into distinct variables, making your code cleaner and more readable.

Here's how destructuring works with code examples:

Array Destructuring:

```
// Example array
const myArray = [1, 2, 3, 4, 5];

// Destructuring assignment
const [first, second, ...rest] = myArray;

console.log(first); // Output: 1
console.log(second); // Output: 2
console.log(rest); // Output: [3, 4, 5]
```

In this example, `first` and `second` variables capture the first two elements of the array `myArray`, and the `rest` variable captures the rest of the elements using the rest syntax (`...`).

Object Destructuring:

```
// Example object
const myObject = {
  name: "John",
  age: 30,
  country: "USA",
};

// Destructuring assignment
const { name, age, country } = myObject;

console.log(name); // Output: John
console.log(age); // Output: 30
console.log(country); // Output: USA
```

In this example, `name`, `age`, and `country` variables capture the corresponding properties of the `myObject`.

Nested Destructuring:

```
// Nested object
const person = {
  name: "Alice",
  age: 25,
  address: {
    city: "New York",
    country: "USA",
  },
};

// Destructuring assignment with nested objects
const {
  name,
  age,
  address: { city, country },
} = person;

console.log(name); // Output: Alice
console.log(age); // Output: 25
console.log(city); // Output: New York
console.log(country); // Output: USA
```

In this example, `city` and `country` variables are extracted from the nested `address` object within the `person` object.

Destructuring provides a cleaner syntax for extracting values from arrays and objects, improving code readability.

14 How does the this keyword works in JS?

It seems like you're referring to the `this` keyword in JavaScript. The `this` keyword refers to the context in which a function is executed. Its value depends on how a function is called.

In JavaScript, the value of `this` is determined by the invocation context of the function and can vary based on how a function is called:

1. **Global Context**: When used in the global scope (outside of any function), `this` refers to the global object, which is `window` in a web browser and `global` in Node.js.

```
console.log(this); //Window in a browser
console.log(this); // Global in Node.js
```

2. **Function Context**: Inside a function, the value of `this` depends on how the function is called:

    a. **Regular Function**: In non-strict mode, `this` inside a regular function refers to the global object, but in strict mode, it defaults to `undefined`.

```
function myFunction() {
  return this;
}
console.log(myFunction() === window); // true in a browser
```

    b. **Method**: When a function is called as a method of an object, `this` refers to the object itself.

```
const obj = {
  method() {
    return this;
  },
};
console.log(obj.method() === obj); // true
```

    c. **Constructor Function**: When a function is used as a constructor with the `new` keyword, `this` refers to the newly created object.

```
function MyClass() {
  this.property = "value";
}
const instance = new MyClass();
console.log(instance.property); // 'value'
```

3. **Event Handlers**: In event handler functions, `this` usually refers to the element that triggered the event.

```
<button onclick="console.log(this)">Click me</button>
```

4. **Explicit Binding**: You can explicitly set the value of `this` using `call()`, `apply()`, or `bind()` methods.

```
function greet() {
  return `Hello, ${this.name}!`;
}

const person = { name: "Alice" };

const greetPerson = greet.bind(person);
console.log(greetPerson()); // Hello, Alice!
```

Here the this keyword will now start pointing to the person object.

15 What is a Constructor function in JS ?

Constructor functions in JavaScript are a way to create objects with a blueprint or template. They are used to instantiate multiple objects with similar properties and methods.

Here's how constructor functions work:

1. **Definition**: You define a constructor function using the `function` keyword. By convention, constructor functions are named with an initial capital letter to distinguish them from regular functions.

2. **Properties and Methods**: Inside the constructor function, you can define properties and methods using the `this` keyword. These properties and methods will be assigned to each instance of the object created using the constructor function.

3. **Instantiation**: To create an instance of an object using a constructor function, you use the `new` keyword followed by the name of the constructor function.

Here's an example to illustrate this:

```
// Constructor function for creating Car objects
function Car(make, model, year) {
  this.make = make;
  this.model = model;
  this.year = year;

  // Method to display car information
  this.displayInfo = function () {
    return `This is a ${this.year} ${this.make} ${this.model}`;
  };
}

// Creating instances of Car objects
const car1 = new Car("Toyota", "Camry", 2020);
const car2 = new Car("Honda", "Civic", 2018);

// Accessing properties and methods of Car objects
console.log(car1.displayInfo()); // Output: This is a 2020 Toyota Ca
console.log(car2.displayInfo()); // Output: This is a 2018 Honda Civ
```

In this example:

- `Car` is a constructor function that defines the blueprint for creating `Car` objects.
- `this.make`, `this.model`, and `this.year` are properties of the `Car` objects.

- `displayInfo()` is a method of the `Car` objects that returns a string containing information about the car.
- `car1` and `car2` are instances of the `Car` objects created using the `new` keyword.

Using constructor functions allows for code reusability and organization, as you can easily create multiple instances of objects with similar properties and methods.

16 What are Call Apply Bind Methods and how they work ?

1. `call` Method:

The `call` method allows you to call a function with a given `this` value and arguments provided individually.

```
const person1 = {
  fullName: function () {
    return this.firstName + " " + this.lastName;
  },
};

const person2 = {
  firstName: "John",
  lastName: "Doe",
};

// Using call to invoke the method with a specific context
const result = person1.fullName.call(person2); // "John Doe"
console.log(result);
```

In this example, `call` invokes the `fullName` method of `person1` with the `this` value set to `person2`.

2. `apply` Method:

The `apply` method is similar to `call`, but it accepts arguments as an array.

```
const person1 = {
  fullName: function (city, country) {
    return this.firstName + " " + this.lastName + ", " + city + ",
  },
};

const person2 = {
  firstName: "John",
  lastName: "Doe",
};

const args = ["New York", "USA"];

// Using apply to invoke the method with a specific context and argu
const result = person1.fullName.apply(person2, args); // "John Doe,
console.log(result);
```

Here, `apply` invokes the `fullName` method of `person1` with the `this` value set to `person2` and the arguments provided in the array.

3. `bind` Method:

The `bind` method creates a new function with a specified `this` value and initial arguments.

```
const person1 = {
  fullName: function (city, country) {
    return this.firstName + " " + this.lastName + ", " + city + ",
  },
};

const person2 = {
  firstName: "John",
  lastName: "Doe",
```

```
};

// Using bind to create a new function with a specific context
const boundFunction = person1.fullName.bind(person2, "New York", "US

// Invoking the bound function
const result = boundFunction(); // "John Doe, New York, USA"
console.log(result);
```

In this example, `bind` creates a new function with `this` set to `person2` and `city` and `country` set to `"New York"` and `"USA"` respectively. The new function `boundFunction` can be invoked later.

17 What is Prototypal Inheritance in JS ?

Prototypal inheritance is a fundamental concept in JavaScript where objects inherit properties and methods from other objects. Every object in JavaScript has a prototype property, which references another object. When you access a property or method on an object, JavaScript first looks for it on that object. If it doesn't find it, it looks at the prototype of that object, and so on, forming a chain known as the prototype chain.

Here's an example to illustrate prototypal inheritance:

```
// Define a constructor function
function Animal(name) {
  this.name = name;
}

// Add a method to the prototype of Animal
Animal.prototype.sayName = function () {
  console.log("My name is " + this.name);
};

// Create a new object using the Animal constructor
var cat = new Animal("Whiskers");
```

```
// Call the method inherited from Animal's prototype
cat.sayName(); // Output: My name is Whiskers
```

In this example:

1. We define a constructor function `Animal` that initializes an object with a `name` property.
2. We add a method `sayName` to the `prototype` of `Animal`, which all instances created using the `Animal` constructor will inherit.
3. We create a new object `cat` using the `Animal` constructor.
4. We call the `sayName` method on the `cat` object, which is inherited from the `Animal` prototype.

Additionally, you can create a prototype chain by inheriting from another object. Here's how you can do it:

```
// Define a constructor function
function Dog(name, breed) {
  Animal.call(this, name); // Call the Animal constructor with 'thi
  this.breed = breed;
}

// Inherit from Animal prototype
Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog; // Reset constructor

// Add a method specific to Dog
Dog.prototype.bark = function () {
  console.log("Woof! I'm a " + this.breed);
};

// Create a new Dog object
var myDog = new Dog("Buddy", "Golden Retriever");

// Call inherited method
myDog.sayName(); // Output: My name is Buddy



// Call the method specific to Dog
myDog.bark(); // Output: Woof! I'm a Golden Retriever
```

In this example:

1. We define a constructor function `Dog` that takes `name` and `breed` as parameters.
2. We call the `Animal` constructor inside the `Dog` constructor using `Animal.call(this, name)`.
3. We set `Dog.prototype` to a new object created with `Object.create(Animal.prototype)`, establishing a prototype chain.
4. We define a new method `bark` on `Dog.prototype`.
5. We create a new `Dog` object `myDog` and demonstrate calling both inherited (`sayName`) and specific (`bark`) methods.

This demonstrates how you can achieve inheritance in JavaScript using prototypes.

18 What is Astnchronous Programming in JS an how it works ?

Asynchronous JavaScript refers to the ability of JavaScript to execute multiple tasks concurrently without blocking the execution of other tasks. It allows JavaScript code to perform operations such as fetching data from a server, reading files, or executing time-consuming tasks without halting the entire program's execution.

Asynchronous operations are crucial in web development because they enable better responsiveness and efficiency. Without them, tasks that require waiting for resources, like fetching data from an API or reading files, would cause the entire program to freeze, leading to a poor user experience.

Here's a simple example to illustrate asynchronous JavaScript using the `setTimeout()` function:

```
console.log("Start");

setTimeout(() => {
  console.log("Inside setTimeout");
}, 2000); // Wait for 2 seconds

console.log("End");
```

In this example, “Start” and “End” will be logged immediately, while “Inside setTimeout” will be logged after a 2-second delay. This delay doesn’t block the execution of subsequent code, allowing other operations to continue while waiting for the timeout to finish.

Under the hood, asynchronous JavaScript relies on concepts like event loops and callback functions. When an asynchronous operation is initiated, it’s scheduled to be executed later, and meanwhile, the JavaScript engine continues to execute other tasks. Once the asynchronous operation completes, a callback function associated with that operation is placed in the event queue. The event loop continuously checks the event queue and executes the callback functions when the execution stack is empty.

Here’s an example using a callback function with `setTimeout()`:

```
console.log("Start");

function callback() {
  console.log("Inside setTimeout callback");
}

setTimeout(callback, 2000); // Wait for 2 seconds

console.log("End");
```

In this example, the `callback()` function is executed after the timeout, maintaining the asynchronous behavior.

Asynchronous programming in JavaScript has evolved with the introduction of Promises and async/await syntax, providing cleaner

and more structured ways to handle asynchronous operations, but the underlying principles remain the same.

19 How Asynchronous Programming in JS works ?

Asynchronous programming is a programming paradigm that allows tasks to be executed concurrently, enabling non-blocking behavior in applications. In JavaScript, it's commonly used for tasks such as fetching data from a server, handling user input, or performing I/O operations.

Here's a breakdown of how asynchronous programming works in JavaScript, involving the call stack, event loop, and different queues:

1. **Call Stack**: The call stack is a mechanism used by JavaScript to keep track of function calls. When a function is called, it's added to the top of the call stack. When a function finishes executing, it's removed from the stack.

2. **Event Loop**: The event loop is a crucial part of asynchronous programming in JavaScript. It continuously checks the call stack and the task queue to determine if there's any work to be done. If the call stack is empty, it takes the first task from the task queue and pushes it onto the call stack for execution.

3. **Task Queue**: The task queue (also known as the callback queue or message queue) holds tasks that are ready to be executed once the call stack is empty. Tasks in the queue are processed in the order they were added.

Now, let's see how these components work together with some code examples:

```
console.log("Start");

// Asynchronous function with setTimeout
setTimeout(() => {
  console.log("Inside setTimeout callback");
}, 0);

// Synchronous function
console.log("End");
```

When this code runs, here's what happens:

1. "Start" is logged to the console.
2. `setTimeout` is encountered. It schedules a task to be executed asynchronously after a minimum of 0 milliseconds.
3. "End" is logged to the console.
4. Since there's no other synchronous code to execute, the event loop checks the call stack. It's empty, so it moves to the task queue.
5. The task from `setTimeout` is in the task queue. The event loop picks it up and pushes it onto the call stack.
6. "Inside setTimeout callback" is logged to the console.

This demonstrates how asynchronous code (the callback inside `setTimeout`) is executed after the synchronous code (logging "Start" and "End") has finished, thanks to the event loop.

In more complex scenarios involving asynchronous operations like network requests or file I/O, callbacks or promises are typically used to handle the asynchronous results. These callbacks or promises are added to the task queue once the asynchronous operation completes, and they're executed when the call stack is empty.

20 What are all the stages of a promise and how to use a promise ?

A Promise in JavaScript represents the eventual completion (or failure) of an asynchronous operation and its resulting value. It has three states:

1. **Pending**: Initial state, neither fulfilled nor rejected.
2. **Fulfilled**: The operation completed successfully.
3. **Rejected**: The operation failed.

Here's how you can use a Promise with code examples:

```
// Example 1: Creating a Promise
const myPromise = new Promise((resolve, reject) => {
  // Simulating an asynchronous operation (e.g., fetching data)
  setTimeout(() => {
    const data = { message: "Promise resolved successfully!" };
    // Resolve the promise with data
    resolve(data);
    // or reject the promise if an error occurs
    // reject(new Error("Promise rejected!"));
  }, 2000);
});

// Example 2: Consuming a Promise
myPromise
  .then((result) => {
    // Promise fulfilled, handle the result
    console.log("Fulfilled:", result);
  })
  .catch((error) => {
    // Promise rejected, handle the error
    console.error("Rejected:", error);
  })
  .finally(() => {
    // Optional: Perform cleanup or final tasks
    console.log("Promise completed.");
  });
```

In this example:

1. **Creating a Promise**:

- A Promise is created with the `new Promise()` constructor, which takes a function (executor) as an argument.
- The executor function receives two parameters: `resolve` and `reject`. These are functions provided by JavaScript to change the state of the Promise.
- Inside the executor function, you perform the asynchronous operation. Once the operation is completed, you call `resolve` with the result if successful, or `reject` with an error if it fails.

2. **Consuming a Promise**:

   - You use the `then()` method to handle the fulfillment of the Promise. It takes a callback function as an argument, which is called when the Promise is resolved.
   - You use the `catch()` method to handle the rejection of the Promise. It takes a callback function as an argument, which is called when the Promise is rejected.
   - You can also use the `finally()` method, which takes a callback function as an argument and is called regardless of whether the Promise is fulfilled or rejected. This is often used for cleanup or final tasks.

3. **Promise States**:

   - If the asynchronous operation inside the Promise executor succeeds, `resolve` is called, transitioning the Promise to the fulfilled state.
   - If the operation fails, `reject` is called, transitioning the Promise to the rejected state.
   - While the Promise is in the pending state, it may transition to either fulfilled or rejected, depending on the outcome of the asynchronous operation.

Promises provide a more structured way to handle asynchronous operations compared to callbacks, making code easier to read and

maintain, especially when dealing with multiple asynchronous tasks.

21 What is async await how it works

Sure! Asynchronous programming in JavaScript traditionally relied heavily on callbacks, which could lead to callback hell and hard-to-read code. To mitigate this issue, JavaScript introduced the `async/await` syntax as part of ES2017 (ES8). `async/await` makes asynchronous code look and behave more like synchronous code, which can greatly improve readability and maintainability.

Here's how `async/await` works:

1. **Async Function Declaration (`async`):**
    - You declare a function as `async` by prefixing it with the `async` keyword. This tells JavaScript that the function will operate asynchronously and may contain `await` expressions inside it.

```
async function fetchData() {
  // Asynchronous operations
}
```

2. **Await Expression (`await`):**
    - Inside an `async` function, you can use the `await` keyword before an expression that returns a Promise. The `await` keyword pauses the execution of the async function until the promise is resolved, and then it resumes the execution with the resolved value.

```
async function fetchData() {
  const result = await someAsyncOperation();
  // code here executes after someAsyncOperation() resolves
}
```

3. **Promises:**

    - If the expression after `await` is not a Promise, JavaScript automatically wraps it into a resolved Promise. This simplifies error handling, as any errors thrown in the awaited expression will be caught and propagated as rejected Promises.

4. **Event Loop Integration:**

    - When an `async` function is called, it returns a Promise immediately, even if the asynchronous operations inside it haven't completed yet.
    - When encountering an `await` expression, the JavaScript engine pauses execution of the `async` function and continues executing the rest of the program until the awaited promise settles (either resolves or rejects).
    - Once the awaited Promise settles, the event loop places the function back into the call stack, allowing it to resume execution with the resolved value (or propagate the rejection).
    - This asynchronous behavior enables non-blocking execution, allowing other code to run while waiting for I/O operations to complete, which improves performance and responsiveness.

By utilizing `async/await`, you can write asynchronous code in a more readable and maintainable manner, avoiding the callback pyramid of doom and making error handling more straightforward.

Certainly! Here are more advanced JavaScript interview questions focused on various key concepts, complete with problem descriptions, sample answers, and code examples.

Focusing on the advanced topics you've mentioned, here are detailed questions along with their explanations and code examples.

22. Prototypal Inheritance: Explain how prototypal inheritance works in JavaScript.

Problem Description: JavaScript's object model is based on prototypal inheritance. Explain how this differs from classical inheritance and provide an example.

Sample Answer: Prototypal inheritance allows JavaScript objects to inherit properties and methods from other objects. Unlike classical inheritance, where classes inherit from other classes, JavaScript uses prototypes—a special type of object from which other objects can inherit. Every JavaScript object has a prototype property that makes this inheritance possible. When you try to access a property on an object, JavaScript will first search on the object itself, and if it doesn't find it, it will continue searching on the object's prototype. This chain continues until the property is found or the end of the prototype chain is reached.

Example:

```
function Animal(name) {
  this.name = name;
}
Animal.prototype.speak = function() {
  console.log(`${this.name} makes a noise.`);
};

function Dog(name) {
  Animal.call(this, name);
}
Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;
Dog.prototype.speak = function() {
  console.log(`${this.name} barks.`);
};
```

```
let dog = new Dog('Rex');
dog.speak(); // Rex barks.
```

23. Flatten an Array: Implement a function to flatten an array.

Problem Description: Write a function that flattens a nested array into a single array.

Sample Answer: A common approach to flatten an array in JavaScript is to use recursion. The function checks each element to see if it is an array. If it is, the function is called recursively on that array. This process continues until all levels of the array are flattened.

Example:

```
function flattenArray(arr) {
  return arr.reduce((acc, val) => Array.isArray(val) ? acc.concat(f
}
```

```
let nestedArray = [1, [2, 3], [4, [5, 6]]];
console.log(flattenArray(nestedArray)); // [1, 2, 3, 4, 5, 6]
```

24. Flatten an Object: Write a function to flatten a nested object.

Problem Description: Given a nested object, write a function that converts it into a flat object where the keys are paths to the values in the nested structure.

Sample Answer: Flattening an object involves iterating over its properties and, for each property that is an object itself, recursively

flatten it. The keys in the flattened object represent the path to the corresponding value in the nested structure, joined by dots.

Example:

```
function flattenObject(obj, prefix = '', res = {}) {
  for (let [key, value] of Object.entries(obj)) {
    const flatKey = prefix ? `${prefix}.${key}` : key;
    if (typeof value === 'object' && value !== null && !Array.isArr
      flattenObject(value, flatKey, res);
    } else {
      res[flatKey] = value;
    }
  }
  return res;
}

let nestedObject = { a: 1, b: { c: 2, d: { e: 3 } } };
console.log(flattenObject(nestedObject));
// Output: { 'a': 1, 'b.c': 2, 'b.d.e': 3 }
```

25. Deep Copy vs. Shallow Copy: Explain the difference with examples.

Problem Description: Describe the difference between a deep copy and a shallow copy of an object. Provide examples.

Sample Answer: A shallow copy of an object copies the top-level properties, but the nested objects are shared between the original and the copy. A deep copy, on the other hand, recursively copies every level, making the copy completely independent of the original.

Example:

```
// Shallow Copy Example
let original = { a: 1, b: { c: 2 } };
let shallowCopy = { ...original };
shallowCopy.b.c = 3;
console.log(original.b.c); // 3 - original is affected
```

```
// Deep Copy Example
let deepCopy = JSON.parse(JSON.stringify(original));
deepCopy.b.c = 4;
console.log(original.b.c); // 3 - original is not affected
```

26. Currying: Implement a curried function.

Problem Description: Explain currying in JavaScript and provide an example of a curried function.

Sample Answer: Currying is a technique of evaluating functions with multiple arguments, transforming them into a sequence of functions with a single argument. In essence, when a function is called with fewer arguments than it expects, it returns another function that takes the remainder of

the arguments.

Example:

```
function curry(f) { // curry(f) does the currying transform
  return function(a) {
    return function(b) {
      return f(a, b);
    };
  };
}

// Usage
function sum(a, b) {
  return a + b;
}

let curriedSum = curry(sum);
console.log(curriedSum(1)(2)); // 3
```

27. Closures and the `this` Keyword: Explain with an example.

Problem Description: Explain how closures work in JavaScript, especially in relation to the `this` keyword. Provide an example.

Sample Answer: A closure is a function that has access to its outer scope variables even after the outer function has returned. The `this` keyword in closures can be tricky because it depends on how the function is called, not where it is defined. In the global context, `this` refers to the global object. In the context of an object method, `this` refers to the object.

Example:

```
function outer() {
  let count = 0;
  return function inner() {
    count++;
    console.log(count, this);
  };
}

const counter = outer();
counter(); // 1, Window {...} or global object in node.js
```

In this example, `inner` is a closure that accesses `count` from its outer function `outer`. The `this` keyword inside `inner` refers to the global object because it's not called as a method of an object but as a standalone function.

28. Promises vs. Async/Await: Differences and when to use each.

Problem Description: Compare Promises and Async/Await in JavaScript. Explain the differences and provide scenarios where one

might be preferred over the other.

Sample Answer: Promises and async/await are both used to handle asynchronous operations in JavaScript. Promises provide a `.then()` method for chaining asynchronous operations in a more manageable way than callbacks. Async/await, introduced in ES2017, makes working with Promises more straightforward by allowing asynchronous code to be written in a synchronous style using the `async` function declaration and the `await` keyword.

Example:

```
// Using Promises
function fetchUserData() {
  fetch("https://api.example.com/user")
    .then((response) => response.json())
    .then((data) => console.log(data))
    .catch((error) => console.error(error));
}

// Using async/await
async function fetchUserDataAsync() {
  try {
    const response = await fetch("https://api.example.com/user");
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error(error);
  }
}
```

Async/await provides a cleaner, more readable syntax for handling Promises, especially when dealing with complex chains of asynchronous operations. However, both approaches are still widely used and often a matter of personal or project-based preference.

29. Module Patterns in JavaScript: Explain different ways to create modules.

Problem Description: Modules are an essential part of building maintainable, scalable, and encapsulated applications. Describe different module patterns in JavaScript and their benefits.

Sample Answer: JavaScript supports several module patterns, including the Module pattern, Revealing Module pattern, ES6 Modules, and CommonJS modules. The Module pattern uses closures to create private and public properties and methods. The Revealing Module pattern is a variation that explicitly returns an object literal with pointers to private functionality. ES6 Modules are the standard in modern JavaScript, allowing you to export and import modules across different files. CommonJS modules are used in Node.js for server-side development.

Example:

```
// Revealing Module Pattern
const myModule = (function () {
  let privateVar = "I am private";
  function privateMethod() {
    console.log(privateVar);
  }

  return {
    publicMethod: function () {
      privateMethod();
    },
  };
})();

myModule.publicMethod(); // Accesses the private method

// ES6 Module (in a separate file, e.g., module.js)
export const publicMethod = () => console.log("I am public");
// Importing in another file
import { publicMethod } from "./module.js";
```

Certainly, I'll enhance the explanations and include HTML examples alongside the JavaScript code to offer a more

comprehensive understanding and demonstrate practical applications.

30. Explain the difference between `document.getElementById()` and `document.querySelector()`. When would you use each?

Explanation:

`document.getElementById()` selects an HTML element based on its unique ID. It's a fast and straightforward way to access an element when you know its ID.

`document.querySelector()`, conversely, is more versatile, allowing you to select the first element that matches a specified CSS selector, such as an ID, class, or any complex selector.

Example:

HTML:

```
<div id="uniqueElement"></div>
<div class="someClass"></div>
```

JavaScript:

```
// Selecting by ID
const elementById = document.getElementById('uniqueElement');

// Selecting by class (or any CSS selector)
const elementByQuery = document.querySelector('.someClass');
```

31. What is Event Bubbling in the DOM and how can you stop it?

Explanation:

Event bubbling is a process where an event on a child element propagates up through its ancestors in the DOM tree. You can stop this propagation using `event.stopPropagation()` in the event handler.

Example:

HTML:

```
<div id="parent">Parent
  <div id="child">Child</div>
</div>
```

JavaScript:

```
document.getElementById('child').addEventListener('click', function
  event.stopPropagation(); // Stops the event from bubbling up
  console.log('Child element clicked!');
});

document.getElementById('parent').addEventListener('click', functio
  // This won't be called if the child is clicked, due to stopPropag
  console.log('Parent element clicked!');
});
```

32. Create a function to toggle a class on an element when it is clicked. Show the event listener attachment.

Problem Statement:

Write a JavaScript function to toggle a class `active` on an element when it is clicked.

Solution Code:

HTML:

```
<div id="toggleElement">Click me</div>
```

JavaScript:

```
function toggleActiveClass(event) {
  event.target.classList.toggle('active');
}

document.getElementById('toggleElement').addEventListener('click', t
```

Explanation:

This function listens for a click event on the specified element. Upon clicking, it toggles the `active` class on the element, adding it if it's not present, and removing it if it is.

33. Describe the difference between `innerText` and `textContent`, with an example.

Explanation:

`innerText` returns the visible text of an element and its descendants. It respects CSS styles, which means it does not include text hidden by CSS.

`textContent` gives the full text content of an element and its descendants, including text in `<script>` and `<style>` elements,

ignoring CSS styling.

Example:

HTML:

```
<div id="example">Hello <span style="display:none;">World</span><
```

JavaScript:

```
// innerText ignores the text within <span> due to display:none
console.log(document.getElementById('example').innerText); // Output

// textContent includes all text, regardless of styling
console.log(document.getElementById('example').textContent); // Outp
```

34. Explain how to create and append a new element to the DOM.

Explanation:

To add a new element to the DOM, you create the element using `document.createElement()` and then append it to an existing element with `appendChild()`.

Example:

HTML:

```
<div id="container"></div>
```

JavaScript:

```
const newElement = document.createElement('div');
newElement.innerText = 'I am new here!';
```

```
document.getElementById('container').appendChild(newElement);
```

Creating machine coding problems that focus on the JavaScript DOM (Document Object Model) can be a great way to assess a candidate's understanding of web technologies and their ability to interact with web page content dynamically. Below are five coding problems that range from basic to intermediate complexity. Each problem includes the problem statement, necessary requirements, a solution approach, and the complete solution with code.

Problem 35: Dynamic List Creation

Problem Statement: Create a function that dynamically adds items to an unordered list in the HTML document when invoked. The function should accept a string, which will be the content of the list item to be added.

Necessary Requirements: - HTML document with an empty unordered list (`<ul>`). - JavaScript function that accepts a string parameter.

Solution Approach: - Select the unordered list element using `document.querySelector()` or `document.getElementById()`. - Create a new list item (`<li>`) element using `document.createElement()`. - Set the text content of the new list item to the string parameter using `textContent`. - Append the new list item to the unordered list using `appendChild()`.

Complete Solution:

HTML:

```
<ul id="myList"></ul>
```

JavaScript:

```
function addItemToList(itemText) {
  const list = document.getElementById("myList");
  const listItem = document.createElement("li");
  listItem.textContent = itemText;
  list.appendChild(listItem);
}
```

Problem 36: Toggle Element Visibility

Problem Statement: Implement a function that toggles the visibility of a specified HTML element on the page each time the function is called.

Necessary Requirements: - An HTML element with a unique identifier. - JavaScript function that toggles visibility.

Solution Approach: - Select the target element using `document.getElementById()` or `document.querySelector()`. - Check the current value of the element's `style.display` property. - If the display property is `"none"`, set it to `""` or a specific display value (e.g., `"block"`); otherwise, set it to `"none"`.

Complete Solution:

HTML:

```
<div id="toggleElement">Toggle me!</div>
```

JavaScript:

```
function toggleVisibility(elementId) {
  const element = document.getElementById(elementId);
  if (element.style.display === "none") {
    element.style.display = "";
  } else {
    element.style.display = "none";
  }
}
```

Problem 37: Form Input Validation

Problem Statement: Create a function that validates the input of a text field in a form. The input should not be empty and must contain at least 8 characters. Display an error message next to the input field if the validation fails.

Necessary Requirements: - A form with a text input field and a submission button. - An area to display the error message.

Solution Approach: - Select the form, input field, and area for the error message using appropriate selectors. - Add an event listener to the form's `submit` event. - In the event listener, prevent the form's default submission with `preventDefault()` if the validation fails. - Check the length of the input field's value; if it doesn't meet the criteria, display the error message.

Complete Solution:

HTML:

```
<form id="myForm">
  <input type="text" id="myInput">
  <button type="submit">Submit</button>
  <div id="errorMessage" style="color: red;"></div>
</form>
```

JavaScript:

```
document.getElementById("myForm").addEventListener("submit", functi
  const inputField = document.getElementById("myInput");
  const errorMessage = document.getElementById("errorMessage");

  if (inputField.value.length < 8) {
    event.preventDefault();
    errorMessage.textContent = "Input must be at least 8 characters
  } else {
    errorMessage.textContent = "";
  }
});
```

Problem 38: Dynamic Table Population

Problem Statement: Write a JavaScript function that populates a table with data from a given array of objects. Each object in the array represents a row in the table, and each property of the object represents a cell.

Necessary Requirements: - A `<table>` element in the HTML document. - An array of objects with consistent properties.

Solution Approach: - Select the table element. - Iterate over the array of objects. - For each object, create a new table row (`<tr>`). - For each property in the object, create a table cell (`<td>`) and fill it with the property's value. - Append the cells to the row, and then append the row to the table.

Complete Solution:

HTML:

```
<table id="myTable">
  <thead>
    <tr>
      <th>Name</th>
      <th>Age</th>
    </tr>
  </
```

```
thead>
  <tbody>
  </tbody>
</table>
```

JavaScript:

```
const data = [
  { name: "John Doe", age: 30 },
```

```
  { name: "Jane Doe", age: 25 }
];

function populateTable(data) {
  const tableBody = document.querySelector("#myTable tbody");

  data.forEach(item => {
    const row = document.createElement("tr");
    Object.values(item).forEach(value => {
      const cell = document.createElement("td");
      cell.textContent = value;
      row.appendChild(cell);
    });
    tableBody.appendChild(row);
  });
}

populateTable(data);
```

Problem 39: Real-time Search Filter

Problem Statement: Implement a function that filters a list of elements based on the user's input in real-time, hiding elements that do not match the search query.

Necessary Requirements: - An input field for the search query. - A list of items (e.g., `<ul>` with multiple `<li>` elements) to be filtered.

Solution Approach: - Add an event listener to the input field for the `input` event. - On each input event, iterate over the list items. - If the item's text content does not include the input value, hide it using `style.display = "none"`. - Otherwise, ensure the item is visible.

Complete Solution:

HTML:

```
<input type="text" id="searchInput">
<ul id="itemsList">
```

```
  <li>Apple</li>
  <li>Banana</li>
  <li>Cherry</li>
</ul>
```

JavaScript:

```
document.getElementById("searchInput").addEventListener("input", fur
  const searchValue = this.value.toLowerCase();
  const items = document.querySelectorAll("#itemsList li");

  items.forEach(item => {
    if (!item.textContent.toLowerCase().includes(searchValue)) {
      item.style.display = "none";
    } else {
      item.style.display = "";
    }
  });
});
```

Problem 40: Star Rating Component

Problem Statement: Develop a JavaScript function to create a dynamic star rating component that allows users to rate on a scale of 1 to 5 stars. When a user clicks on a star, all stars up to that point should be highlighted. The component should also display the current rating value in text next to the stars. Ensure that the component is reusable, allowing multiple instances on the same page without interference.

Necessary Requirements: - An HTML container for the star rating component. - Each star can be an `<i>` element with classes for filled and unfilled states (e.g., using Font Awesome icons) or simple `<span>` elements styled with CSS. - A text element to display the current rating value dynamically. - The component should be initialized through a JavaScript function that can be applied to any container, making it reusable.

Solution Approach: - Write a JavaScript function that accepts a container element as an argument. - Within the function, generate five star elements and append them to the container. Attach a click event listener to each star for the rating functionality. - When a star is clicked, determine its index (rating value) and update the appearance of all stars to reflect the current rating. - Display the current rating value in a text element next to the stars.

Complete Solution:

HTML (Container example):

```
<div class="star-rating" id="rating1"></div>
```

CSS (Basic styles for stars):

```
.star-rating .star {
  cursor: pointer;
  color: #ccc; /* Unfilled star color */
  font-size: 24px;
  padding: 5px;
}

.star-rating .star.filled {
  color: gold; /* Filled star color */
}
```

JavaScript:

```
function initializeStarRating(containerId) {
  const container = document.getElementById(containerId);
  const ratingValue = document.createElement("span");
  ratingValue.textContent = "Rating: 0";

  for (let i = 1; i <= 5; i++) {
    const star = document.createElement("i");
    star.classList.add("star");
    star.textContent = "☆"; // Use "★" for filled star in 'updateRa
    star.addEventListener("click", () => updateRating(i));
    container.appendChild(star);
  }
```

```
  container.appendChild(ratingValue);

  function updateRating(rating) {
    const stars = container.querySelectorAll(".star");
    ratingValue.textContent = `Rating: ${rating}`;
    stars.forEach((star, index) => {
      if (index < rating) {
        star.classList.add("filled");
        star.textContent = "★";
      } else {
        star.classList.remove("filled");
        star.textContent = "☆";
      }
    });
  }
}

// Initialize for each container
initializeStarRating("rating1");
```

More questions will be added