JAVA SCRIPT
INTERVIEW QUESTION

PW SKILLS

EASY

Q1.Explain the role of the V8 engine in JavaScript.

Answer:
The V8 JavaScript engine is a critical component of the JavaScript ecosystem, playing a pivotal role in executing JavaScript code efficiently and enabling its widespread adoption across web browsers and server-side applications. It is an open-source high-performance JavaScript and WebAssembly engine written in C++ and developed by Google.

Core Functionalities of V8

- V8 is responsible for interpreting and executing JavaScript code, converting the human-readable JavaScript syntax into machine-executable instructions that the computer can understand.
- V8 manages the memory allocation and usage for JavaScript objects and data structures, ensuring efficient memory utilization and preventing memory leaks.
- V8 automatically reclaims unused memory by identifying and removing objects that are no longer referenced, preventing memory bloat and improving overall performance.
- V8 employs various optimization techniques, such as just-in-time (JIT) compilation and type inference, to enhance the execution speed of JavaScript code.
- V8 supports WebAssembly, a low-level binary format for executing code in the browser, enabling developers to write high-performance applications using languages like C and C++

Q2.What is the Document Object Model (DOM) in JavaScript, and how does it relate to web pages?

Answer:
The Document Object Model (DOM) is a cross-platform and language-independent interface that treats an HTML, XHTML, or XML document as a tree structure wherein each node is an object representing a part of the document. The DOM is a W3C (World Wide Web Consortium) standard that defines the way a document is accessed and manipulated by programming languages.
In JavaScript, the DOM plays a crucial role in creating dynamic and interactive web pages. It allows JavaScript programmers to access, modify, and manipulate the elements and content of a web page, enabling them to create rich user experiences and responsive web applications

Q3.What are the different types of data types in JavaScript, Differentiate between Primitive and Non-primitive data types in JavaScript.

Answer:
JavaScript has two main types of data types: primitive and non-primitive.
Primitive data types -
Primitive data types are the basic building blocks of JavaScript. They are immutable, meaning their values cannot be changed once created. There are seven primitive data types in JavaScript:

1. **Number:** Represents numeric values, both integers and decimals.
2. **String:** Represents textual data, enclosed in single or double quotes.
3. **Boolean:** Represents logical values, either true or false.
4. **Null**: Represents an intentional absence of value.
5. **Undefined:** Represents an uninitialized or unknown value.
6. **Symbol:** Represents a unique identifier for a property or method.
7. **BigInt:** Represents integer values with arbitrary precision

Non-Primitive Data Types
Non-primitive data types are complex data structures that can hold multiple values. They are mutable, meaning their values can be changed after creation. There are two main non-primitive data types in JavaScript:

1. **Object**: Represents a collection of key-value pairs, where keys are strings and values can be any type of data.
2. **Array:** Represents an ordered collection of values, where each value can be any type of data.

Differentiating between Primitive and Non-Primitive Data Types

Feature	Primitive Data Types	Non-Primitive Data Types
Immutability	Immutable	Mutable
Composition	Cannot hold other data types	Can hold other data types, including primitive and non-primitive data types
Reference Comparison	Compared by value	Compared by reference
Memory Management	Automatically managed by the garbage collector	Manual memory management may be required

Q4. What is an object in JavaScript, and how is it different from an array?

Answer:
In JavaScript, objects and arrays are both data structures used to store and organize data. However, there are some key differences between the two.

Objects
An object is a collection of key-value pairs, where each key is a unique identifier for a value. Objects are used to represent real-world entities, such as people, products, or events.

Arrays
An array is an ordered collection of values, where each value is identified by an index number. Arrays are used to store groups of related data, such as a list of numbers, a collection of names, or a sequence of tasks.

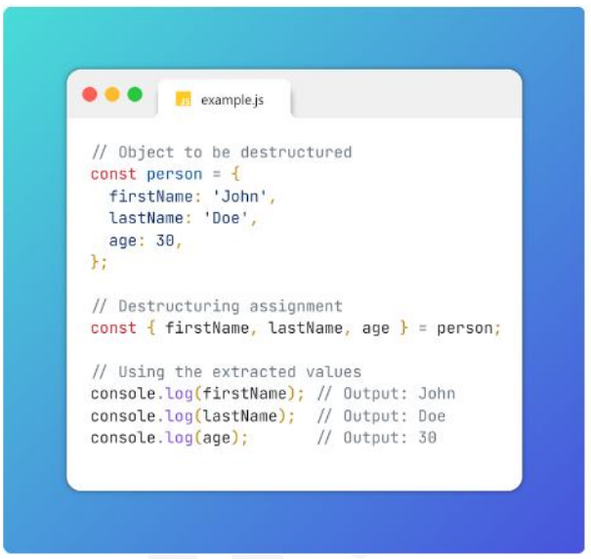
Feature	Object	Arrays
Data Structure	Key-value pairs	Ordered collection of values
Access	Access by key name	Access by index number
Mutation	Mutable	Mutable

Q5. What is destructuring in JavaScript, How do you destructure an object in JavaScript and Why is it useful?

Answer:
Destructuring in JavaScript is a concise way to extract data from objects and arrays. It allows you to unpack values from objects and arrays into distinct variables, making your code more readable and maintainable.

Destructuring Objects
Object destructuring involves extracting values from an object and assigning them to variables with the same name as the object's property. Here's a basic

Example:

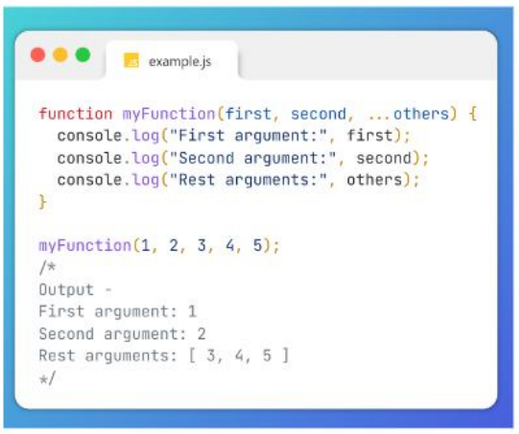


Q6. Explain the difference between object and array destructuring in JavaScript.

Feature	Object Destructuring	Array Destructuring
Data Structure	Objects	Arrays
Access	Access by key name	Access by index number
Order	Variables can be	Variables must be
	assigned in any order	assigned in the order they appear in the array
Default Values	Default values can be specified for missing properties	Default values cannot be specified for missing elements
Nested Destructuring	Can be nested to extract data from nested objects	Can be nested to extract data from nested arrays

Q7.Explain the concept of the Rest parameter and illustrate its usage in function parameters

Answer:
The rest parameter is a feature introduced in ECMAScript 2015 (ES6) that allows you to collect an indefinite number of arguments into an array. It is represented by three dots (...) at the end of a function parameter list. The rest parameter collects all the remaining arguments after the preceding parameters into a single array.
Example below -



Q8.What is a prototype in JavaScript, and why is it significant for object-oriented programming?

Answer:
In JavaScript, a prototype is the object that serves as the template for creating new objects. It plays a crucial role in object-oriented programming (OOP) by providing a mechanism for inheritance and code sharing among objects.

Significance of Prototypes in OOP

Prototypes are fundamental to JavaScript's object-oriented programming capabilities. They provide several key benefits:

1. Prototypes allow for code reuse by defining properties and methods on a prototype, making them available to all objects derived from that prototype.
2. Prototypes enable prototypal inheritance, allowing new objects to inherit properties and methods from existing objects, creating class-like hierarchies.
3. Prototypes provide a flexible approach to object-oriented programming, allowing for dynamic creation and modification of object structures.
4. Prototypes promote efficient memory usage by sharing properties and methods among objects, reducing the need for redundant code duplication.

Q9.Describe the different types of built-in Errors in JavaScript, and illustrate with an example each.

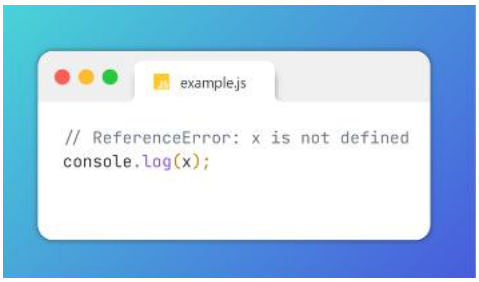
Answer:
JavaScript has seven built-in error types that represent different categories of errors that can occur during program execution.
These error types are:

1. **SyntaxError:** This error occurs when the code violates the syntax rules of the JavaScript language. It typically indicates a typo, missing closing bracket, or an incorrect use of keywords or operators.

Example

2. **ReferenceError:** This error occurs when a reference is made to a variable or property that has not been declared or initialized. It indicates that the identifier is not recognized by the JavaScript interpreter.

Example



3. **Type:** This error occurs when an operation is attempted on a value of an inappropriate type. It indicates that the value cannot perform the requested operation or that the arguments passed to a function are not of the expected types.

Example

Q10.Explain synchronous and asynchronous operations with examples in JavaScript

Answer:

Synchronous Operations in JavaScript

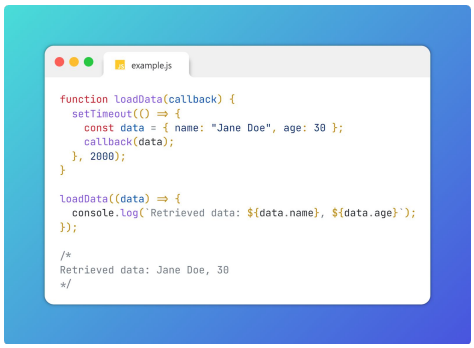
In synchronous programming, the code execution follows a sequential order, where each line of code is executed one at a time before moving on to the next. Synchronous operations block the main thread, meaning the program waits for the completion of one task before starting another.

Example

Asynchronous Operations in JavaScript

In asynchronous programming, code execution is not sequential; instead, multiple tasks can run concurrently without blocking the main thread. Asynchronous operations allow the program to continue running while waiting for the completion of other tasks, improving responsiveness and handling long-running operations efficiently.

Example



Q11.What does the "this" keyword refer to in JavaScript and when is it determined?

Answer:
The "this" keyword is a special keyword in JavaScript that refers to the current executing context of the code. It dynamically changes its value depending on how the code is being invoked. The value of this is determined at runtime, and it can refer to different objects depending on the context.

Some common use of "this"

1. **Inside Methods** - When a method is called on an object, **this** refers to the object itself. This allows the method to access and modify the properties and methods of the object.
2. **Event Handlers** - When an event handler is triggered, **this** refers to the element that triggered the event. This allows the handler to access and manipulate the event-related data and the element itself.
3. **Global context** - When code is executed outside of any function or object, **this** refers to the global object, which is typically the window object in web browsers or the global object in Node.js.
4. **Arrow functions** - Arrow functions have a special lexical **this** binding, which means that their **this** value is not affected by the calling context and remains the same as the enclosing function's **this.**

Determining the Value of this
The value of this is determined at runtime based on the calling context of the code. It is not fixed at compile time, and it can change depending on how the code is invoked. This dynamic nature of this can lead to confusion if not properly understood.

Q12.What is a function in Javascript, what are the different types of functions, and why it is useful?

Answer -
A function in JavaScript is a block of code that is designed to perform a specific task. Functions are essential for code modularity and reusability. They allow you to encapsulate functionality and organize your code into well-defined blocks, making it easier to read, understand, and maintain.

Types of Functions in JavaScript
JavaScript offers various types of functions, each with its own characteristics and usage scenarios -

1. Functions Declaration
2. Function Expressions
3. Arrow Function
4. IIFE (Immediately Invoked Function Expression)
5. Generator Functions
6. Recursive Functions

Benefits of Using Functions

- **Code Reusability:** Functions allow you to reuse code multiple times throughout your program, avoiding duplication and promoting maintainability.

- **Improved Readability:** Functions break down complex tasks into smaller, more manageable units, making your code easier to read and understand.
- **Modular Design:** Functions promote modular design, allowing you to organize your code into well-defined modules, enhancing maintainability and reusability.
- **Encapsulation:** Functions encapsulate code and data, hiding implementation details and promoting data protection.
- **Parameterization:** Functions allow you to pass arguments to customize their behavior, making them more versatile and adaptable.
- **Abstraction:** Functions provide a level of abstraction, allowing you to focus on the high-level functionality without getting bogged down in implementation details.

Q12.What is a function in Javascript, what are the different types of functions, and why it is useful?

Answer -
A function in JavaScript is a block of code that is designed to perform a specific task. Functions are essential for code modularity and reusability. They allow you to encapsulate functionality and organize your code into well-defined blocks, making it easier to read, understand, and maintain.

Types of Functions in JavaScript
JavaScript offers various types of functions, each with its own characteristics and usage scenarios -

1. Functions Declaration
2. Function Expressions
3. Arrow Function
4. IIFE (Immediately Invoked Function Expression)
5. Generator Functions
6. Recursive Functions

Benefits of Using Functions

- **Code Reusability:** Functions allow you to reuse code multiple times throughout your program, avoiding duplication and promoting maintainability.
- **Improved Readability:** Functions break down complex tasks into smaller, more manageable units, making your code easier to read and understand.
- **Modular Design:** Functions promote modular design, allowing you to organize your code into well-defined modules, enhancing maintainability and reusability.
- **Encapsulation:** Functions encapsulate code and data, hiding implementation details and promoting data protection.
- **Parameterization:** Functions allow you to pass arguments to customize their behavior, making them more versatile and adaptable.
- **Abstraction:** Functions provide a level of abstraction, allowing you to focus on the high-level functionality without getting bogged down in implementation details.

Q13.What is BOM, Give an example of BOM

Answer -
The Browser Object Model (BOM) is a collection of objects that provide JavaScript with access to the browser's functionality. It allows JavaScript to interact with the browser's features, such as windows, navigation, history, location, user input, and more. The BOM is an essential tool for creating interactive web pages and applications.

Some of the key components of the BOM are -

1. Window
2. Location
3. History
4. Navigator
5. Screen
6. Document

Example of BOM Usage -

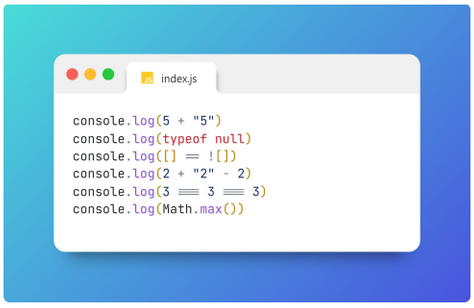
Q14.What is nullish Coalescing operator

Answer -
The nullish coalescing operator (??) is a logical operator introduced in ECMAScript 2020 (ES2020) that provides a more concise and readable way to handle undefined and null values. It returns the first operand if it's not null or undefined, and otherwise returns the second operand.

Example -

From the above example, the finalName will be 'John' because the name is null, and the nullish coalescing operator defaults to defaultName.

Q15. Guess the Output of the following code



Answer -

Q16. Write a JavaScript function to find the first non-repeated character in a given string.

index.js

```
// example --
firstNonRepeatedChar("javascript"); // Should return "j"
firstNonRepeatedChar("programming"); // Should return "p"
```

Answer

example.js

```
function firstNonRepeatedChar(str) {
  // Create an object to store character frequencies
  const charFrequency = {};

  // Iterate through the string to populate the charFrequency object
  for (const char of str) {
    charFrequency[char] = (charFrequency[char] || 0) + 1;
  }

  // Iterate through the string again to find the first non-repeated character
  for (const char of str) {
    if (charFrequency[char] === 1) {
      return char;
    }
  }

  // If no non-repeated character is found, return null or an appropriate value
  return null;
}

// Example usage:
const inputString = "javascript";
const result = firstNonRepeatedChar(inputString);

console.log(`The first non-repeated character in "${inputString}" is: ${result}`);

// The first non-repeated character in "javascript" is: j
```

Q17. Write a JavaScript function to reverse a given string without using the built-in reverse method.

index.js

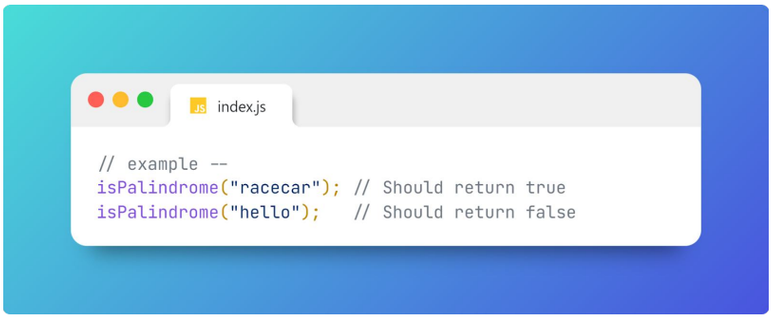
```
// example --
reverseString("javascript") // tpircsavaj
reverseString("programming") // gnimmargorp
```

Answer -

example.js

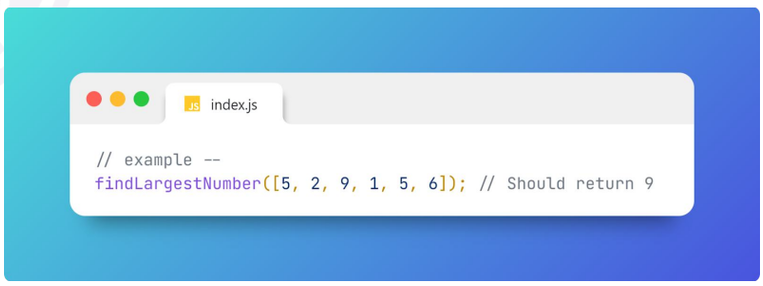
```
function reverseString(str) {
  let reversed = '';

  // Iterate through the string in reverse order
  for (let i = str.length - 1; i >= 0; i--) {
    reversed += str[i];
  }

  return reversed;
}

// Example usage:
const inputString = "Hello, World!";
const reversedString = reverseString(inputString);

console.log(`Original String: ${inputString}`);
console.log(`Reversed String: ${reversedString}`);

/**
 Original String: Hello, World!
Reversed String: !dlroW ,olleH
 **/
```

Q18. Write a function to check if a given string is a palindrome Palindrome - Reads the same forwards and backward

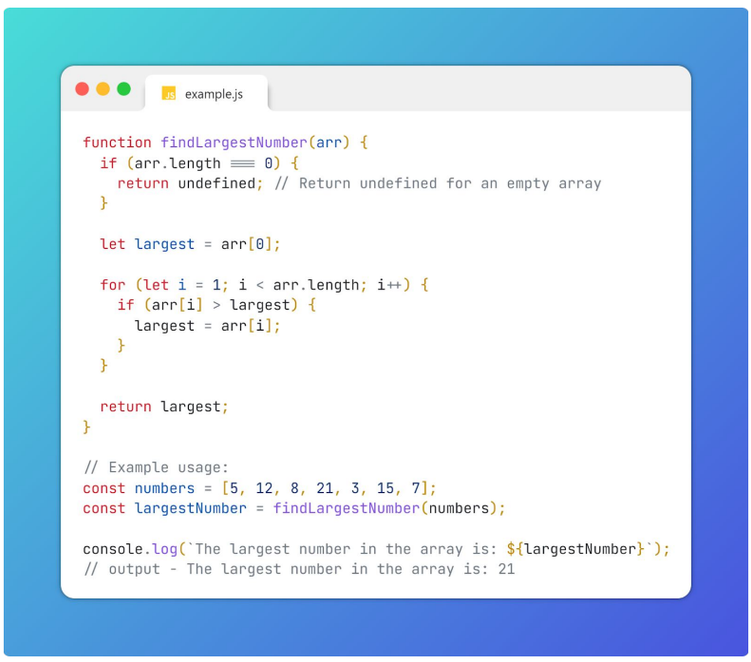


Answer

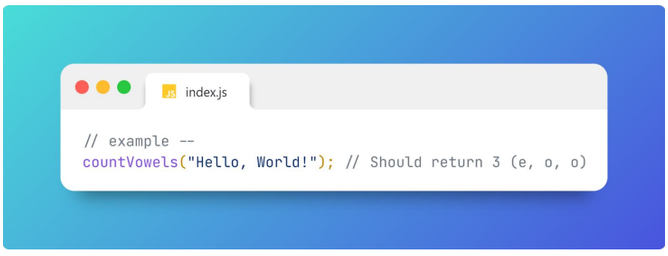
Q19. Find the largest number from a given array of numbers without using the Math.max() built-in function.



Answer



Q20. Write a function that counts the number of vowels in a given string



Answer

MEDIUM

Q1.How does JavaScript execute in the web browser?

Answer –
JavaScript execution in the web browser involves a series of steps that transform the JavaScript code into machine-readable instructions that the browser can interpret and execute. This process involves several key components:

1. Parsing
2. AST(Abstract Syntax Tree)
3. Code Generation
4. Code Execution
5. Event Handling
6. Memory Management
7. Error Handling
8. Debugging Tools

Q2.What is a Polyfill in JavaScript, and why are they used explain with an example.

Answer –
In JavaScript, a polyfill is a piece of code that provides functionality for newer browser features that are not yet natively supported by older browsers. Polyfills allow developers to write code using newer features without worrying about browser compatibility issues. This ensures that their applications can work consistently across different browsers and provide a seamless user experience.

Why Polyfills are Used –

1. Browser Compatibility
2. Feature Availability
3. Code Reusability
4. Progressive Enhancement

Example of a Polyfill –
Consider the Array.prototype.includes() method, which was introduced in ES2017 (ECMAScript 2017). Older browsers may not have this method natively available. A polyfill for Array.prototype.includes() can be implemented as follows -

```
if (!Array.prototype.includes) {
  Array.prototype.includes =
    function(searchElement, fromIndex) {
    fromIndex = fromIndex || 0;
    const length = this.length;
    for (let i = fromIndex; i < length; i++) {
      if (this[i] === searchElement) return true;
    }
    return false;
  };
}
```

Q3.How do you declare a variable using var, let, and const, and what are the differences between them?

Answer -

var -

keyword is the oldest and most traditional way to declare variables in JavaScript. It is used to create variables with function scope, meaning the variable is accessible throughout the function where it is declared.

Example

```
var name = "John Doe";
console.log(name); // Output: John Doe
```

let -

keyword was introduced in ES6 (ECMAScript 2015) and provides block scope for variables. This means the variable is only accessible within the block where it is declared, such as an if statement, a for loop, or a function body.

Example

```
if (true) {
  let age = 30;
  console.log(age); // Output: 30
}
console.log(age);
// ReferenceError: age is not defined
```

const -

keyword, also introduced in ES6, declares variables with constant values. Once declared, the value of a const variable cannot be changed. This helps prevent accidental reassignments and promotes data integrity.

Example

```
const pi = 3.14159;
pi = 2.71828;
// TypeError: Assignment to constant variable
```

Key differences -

Feature	var	let	const
Scope	Function Scope	Block Scope	Block Scope
Reassignment	Allowed	Allowed	Not allowed
Hoisting	Yes	Yes	No
Initialization	optional	Optional	Required

Q4. Explain the concept of callback functions in JavaScript and provide an example.

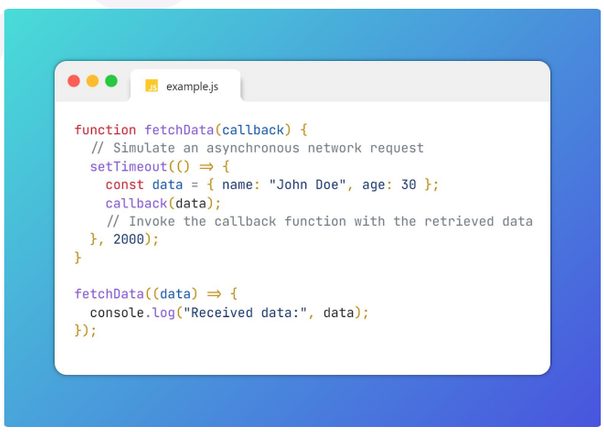
Answer -

Callback functions are a fundamental concept in JavaScript programming. They are functions that are passed as arguments to other functions, allowing for asynchronous operations and event-driven programming. Callback functions enable JavaScript to handle tasks that take time to complete, such as making network requests or reading files, without blocking the main thread of execution.

Key Characteristics of Callback Function

1. Asynchronous Execution
2. Event Handling
3. Data Passing

Example of callback functions -



Q5. Describe the purpose of the for...in and for...of loops in JavaScript and their differences

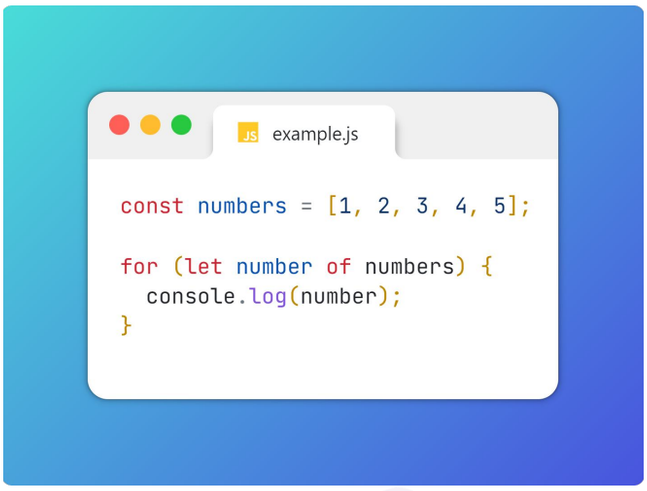
Answer -

for...in

The for...in loop iterates over the enumerable properties of an object. An enumerable property is a property that can be listed or iterated over using a loop. These properties can be directly accessed using dot notation or bracket notation.

Example

for...of
The for...of loop iterates over the values of an iterable object. An iterable object is an object that has a built-in iterator method that allows it to be iterated over. Examples of iterable objects include arrays, strings, and Maps.
Example -



Q6.How is function hoisting different from variable hoisting in JavaScript?

Answer -
Function Hoisting
Function hoisting is the phenomenon where function declarations are moved to the top of their scope, making them accessible before their actual declaration in the code. This means that you can call a function before it is defined in the code, and it will still execute correctly.

Variable Hoisting
Variable hoisting, on the other hand, involves the declaration of variables, but not their initialization. When a variable is hoisted, its declaration is moved to the top of its scope, but its value remains undefined until it is initialized in the code.

Feature	Function Hoisting	Variable Hoisting
Immutability	Immutable	Mutable
Declaration	Entire function definition is hoisted	Only variable declaration is hoisted
Initialization	Function initialization is not affected	Variable initialization occurs in the original line
Scope	Function Scope	Function or global scope
Usage	Allows calling functions before their declaration	Allows accessing variable before their initialization

Q7. Explain event propagation in JavaScript

Answer

Event propagation is a crucial concept in JavaScript that governs how events travel through the Document Object Model (DOM) tree. When an event occurs, such as clicking a button or pressing a key, it initially triggers on the target element, the element directly receiving the user interaction. From there, the event embarks on a journey through the DOM tree, ascending through parent elements until it reaches the document root. This mechanism, known as event propagation, enables event handling at various levels of the DOM tree.

There are two distinct types of Event Propagation -

1. **Bubbling (Default Behavior)** - In bubbling, the event originates at the target element and bubbles up the DOM tree, passing through parent elements in a hierarchical order. Each element along the propagation path has the opportunity to handle the event. This behavior is the default mechanism for most events.
2. **Capturing** - Capturing, on the other hand, follows an opposite flow. The event commences at the document root and cascades down the DOM tree, traversing through child elements in reverse order. Similar to bubbling, each element in the propagation path can intercept and handle the event before it reaches the target element. Capturing is less commonly used but can be beneficial for specific scenarios.

Q8.Explain the concepts of immutable and mutability in the context of arrays, How can you achieve immutability in a JavaScript array illustrate with an example

Answer

In JavaScript, the concepts of immutability and mutability play a crucial role in data handling and program design. Immutability refers to the inability to modify the state of a data structure once it is created. In contrast, mutability allows for the alteration of a data structure's state after its creation.

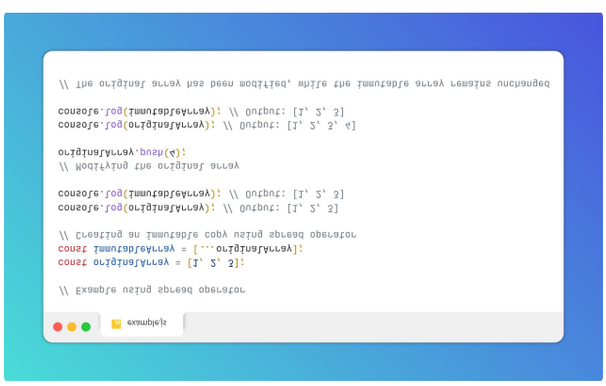
Immutability -

When discussing arrays, immutability implies that once an array is created, its contents cannot be directly changed. This means that elements cannot be added, removed, or modified within the original array. Instead, new arrays must be created to represent any changes to the original data.

Mutability -

Mutability, on the other hand, permits the direct manipulation of an array's contents. Elements can be added, removed, or modified, and these changes are reflected in the original array.

Immutability can be achieved in javascript array with the example below -



Q9. What is event delegation and Why is it useful in web development

Answer

Event delegation is a fundamental concept in web development that involves handling events at a higher level in the DOM tree rather than attaching event listeners to each individual element. This technique is particularly useful for enhancing event handling efficiency and responsiveness, especially when dealing with large numbers of elements or dynamically generated content.

It is useful in web development due to following reason -

1. Reduced Event Listener Overhead
2. Simplified Event Handling
3. Effiecient Handling of Dynamic Content
4. Improve Performance

Q10. Describe nested destructuring array and object in JavaScript and provide an example.

Answer

Nested destructuring in JavaScript allows you to extract data from deeply nested arrays and objects in a concise and structured manner. It enables you to directly target and assign values from nested levels of data structures, simplifying data access and reducing the need for multiple destructuring operations

Desturing of Nested Arrays -

Nested destructuring of arrays involves extracting elements from nested arrays within an array. To achieve this, you can use multiple pairs of square brackets within the destructuring assignment.

Example -

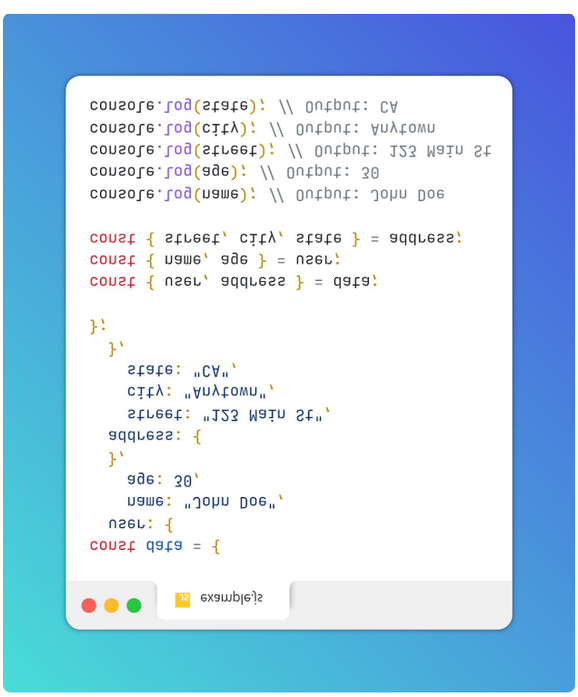
example.js

```
const data = [
  [1, 2, 3],
  [4, 5, 6],
];

const [firstNestedArray, secondNestedArray] = data;
const [firstElement, secondElement, thirdElement] = firstNestedArray;
const [fourthElement, fifthElement] = secondNestedArray;

console.log(firstElement); // Output: 1
console.log(secondElement); // Output: 2
console.log(thirdElement); // Output: 3
console.log(fourthElement); // Output: 4
console.log(fifthElement); // Output: 5
```

Destructuring Nested Objects

Nested destructuring of objects involves extracting properties from nested objects within an object. Similar to arrays, you can use multiple pairs of curly braces within the destructuring assignment.

Example -



Q11. Explain prototype-based inheritance in JavaScript and how it differs from classical inheritance.

Answer -

Prototype-Based Inheritance

Prototype-based inheritance is a fundamental mechanism of object-oriented programming in JavaScript. It allows objects to inherit properties and methods from other objects, forming a chain of prototypes. In this model, every object has a prototype, which is another object that serves as the blueprint for its properties and methods.

Key Characteristics of Prototype-Based Inheritance:

1. **Object-Centric Approach** - focuses on objects rather than classes. Objects are the primary building blocks, and inheritance occurs by linking objects together through prototypes.
2. **Dynamic Inheritance** - nheritance happens at runtime, and objects can inherit from any other object in the prototype chain, providing flexibility and adaptability.
3. **Prototypal Delegation** - Objects delegate method calls to their prototypes, allowing them to access properties and methods from the prototype chain.

Classical Inheritance

Classical inheritance, also known as class-based inheritance, is a more traditional approach to object-oriented programming. It involves defining classes as blueprints for creating objects, and inheritance occurs through a hierarchical class structure.

Key Characteristics of Classical Inheritance

1. **Class-centric Approach** - Classical inheritance emphasizes classes as the primary building blocks, and inheritance is defined through parent-child class relationships
2. **Static Inheritance** - Inheritance is determined at compile time, and classes inherit from a single predefined parent class, limiting flexibility.
3. **Method overriding** - Classes can override inherited methods, allowing them to provide their own implementation for inherited methods.

Difference between Prototype Based and Classical Inheritance -

Feature	Prototype-Based Inheritance	Classical Inheritance
Inheritance mechanism	Object-centric, links object through prototypes	Class-centric, defines inheritance through class relationships
Inheritance Timing	Dynamic, occurs at runtime	Static, determined at compile time
Inheritance Flexibility	Can inherit from any object	Limited to a single parent class
Method Overriding	Supports method delegation and overriding	Supports method overriding

Q12. Explain Implicit and Explicit Types of Coercion in javaScript

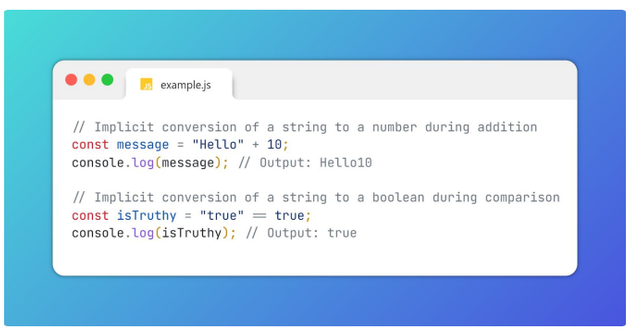
Answer -

Type coercion, also known as type conversion, is the process of automatically converting values from one data type to another. This can happen implicitly, when the language automatically converts a value to the appropriate type for a particular operation, or explicitly when the developer manually converts a value using a built-in function or constructor.

Implicit Type Coercion

Implicit type coercion occurs automatically when the language interprets code and determines that a type conversion is necessary. This often happens during operations involving different data types, such as arithmetic operations or comparisons.

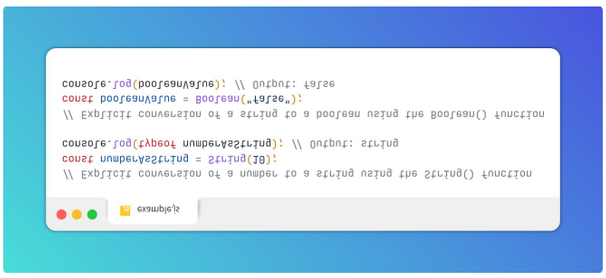
Example



Explicit Type Coercion

Explicit type coercion involves manually converting a value from one data type to another using a built-in function or constructor. This provides greater control over the type conversion process and can be useful for ensuring type compatibility in specific situations.

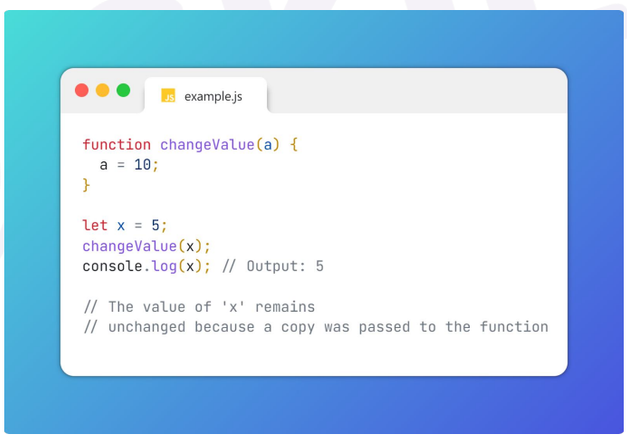
Example



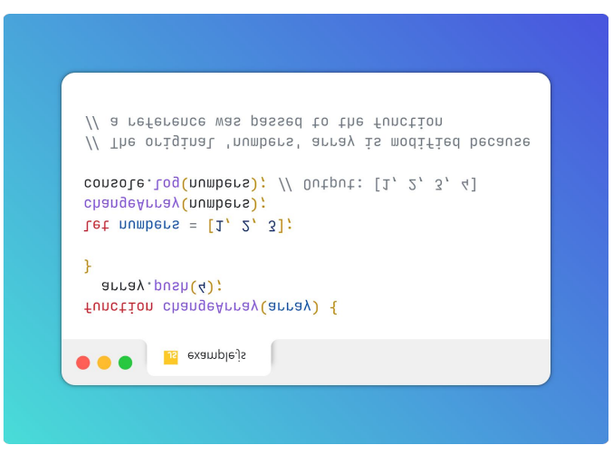
Q13.Explain passed-by-value and passed-by-reference

Answer -
Pass-by-value and pass-by-reference are fundamental concepts in programming that determine how data is passed between functions. These mechanisms have a significant impact on how data is manipulated and shared within a program.

Pass-by-value
Pass-by-value involves creating a copy of the original data and passing that copy to the function. Any changes made to the copy within the function do not affect the original data. This is the default behavior in JavaScript for primitive data types like numbers, strings, and booleans.
Example



Pass-by-reference
Pass-by-reference involves passing the reference or memory address of the original data to the function. Any changes made to the data within the function directly affect the original data. This is the behavior for non-primitive data types in JavaScript, such as objects and arrays.
Example



Q14.What is Currying in JavaScript, Illustrate with an example

Answer -
Currying is a technique in functional programming that transforms a function that takes multiple arguments into a sequence of functions that each take a single argument. This technique allows for partial application of functions, which can make code more concise and reusable.
In JavaScript, currying can be achieved using nested functions or the bind() method. The idea is to break down a multi-argument function into a series of single-argument functions, each taking one argument and returning another function.

Example

Q15.What is IIFE, illustrate it with an example

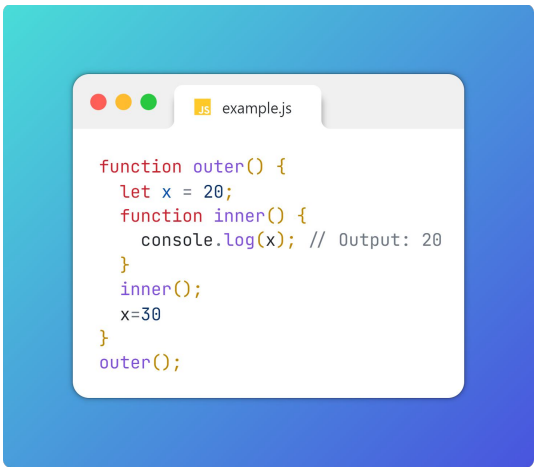
Answer -
An IIFE, also known as a self-executing anonymous function, is a JavaScript function that is invoked immediately after it is defined. This technique is often used to create private scope for variables and to prevent global namespace pollution.
Example

From the above example, the function expression function() { console.log("Hello from IIFE!"); } is wrapped in parentheses, and then called immediately by appending another set of parentheses. This causes the function to be executed as soon as it is defined.

Q16. Code Snippet 1 - Guess the output of the code given below -

Answer -
Explaination -
In this code, there is a variable x declared in the global scope with a value of 10. Inside the bar function, there is another variable x declared with a value of 20. When foo is called from within bar, it looks for the value of x in its immediate scope and doesn't find it, so it looks in the outer scope (global scope) and finds the value of x, which is 10. Therefore, the output will be 10

Q17. Code Snippet 2 - Guess the output of the code given below -



Answer -

Explanation -

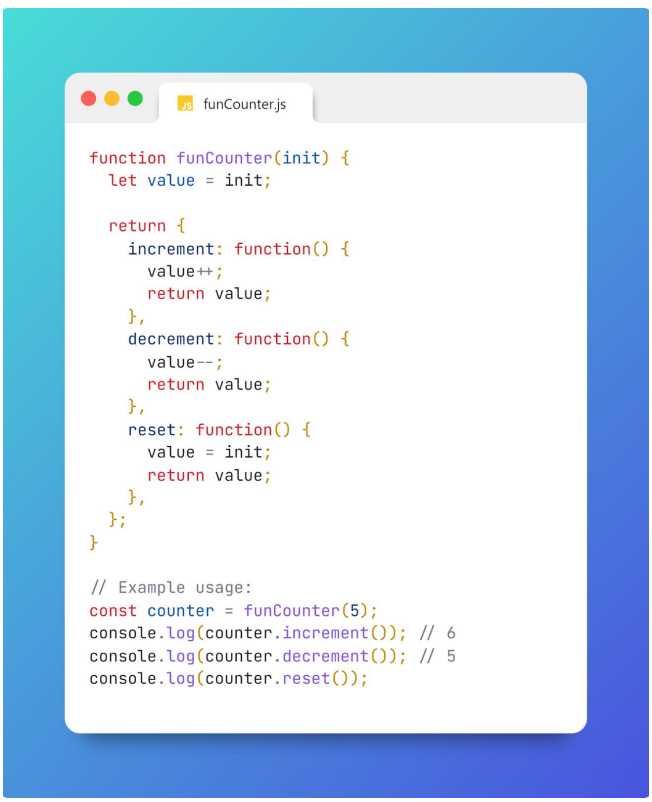
In this code, there are two levels of scope. The outer function outer has a variable x with a value of 20, and the inner function inner tries to log the value of x. Since x is not defined in the local scope of inner, it looks for the value of x in the outer scope (the scope of outer) and finds the value of x, which is 20. Therefore, the output will be 20.

Q18.Write a function "funCounter". It should accept an initial integer "init". It should return an object with three function

The three functions are as follows

- **"increment()" increases the current value by 1 and then returns it**
- **"decrement()" reduces the current value by 1 and then returns it**
- **"reset()" sets the current value to "init" and then returns it.**

Answer

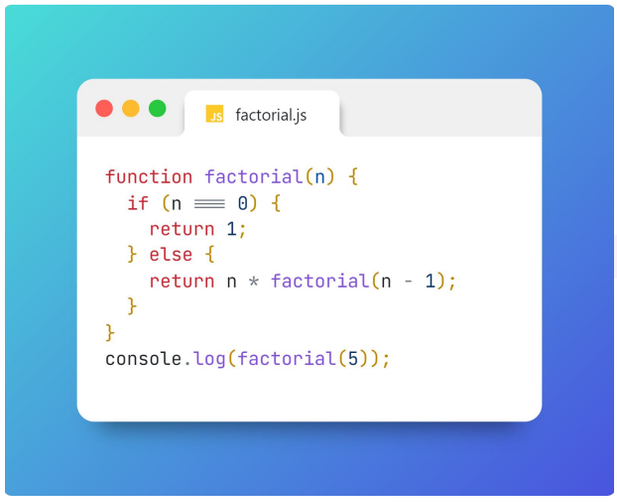


Q19.Identify the type of error in the JavaScript code and provide a solution to fix it.

Answer -

Type of Error - RangeError: Maximum call stack size exceeded

Added a check to handle negative input values and return an appropriate result.



Q20. Write a function that returns the count of the argument passed to it.

Answer -

21. What is Variable shadowing

Answer:

Variable shadowing in JavaScript occurs when a variable declared within a specific scope (like a function or block) has the same name as a variable declared in an outer scope. This inner variable then "shadows" the outer one, making it inaccessible within that scope. It's essential to be aware of shadowing to avoid unexpected behavior in your code. Using descriptive variable names, minimizing variable scope, and choosing appropriate variable declaration keywords (let, const, var) can help manage and prevent issues related to variable shadowing.

22. Define Memoize function

Answer:
The memoize function takes another function as input and returns a new function that caches the results of previous function calls. Here's a basic implementation.

```
function memoize(fn) {
  const cache = new Map();

  return function(...args) {
    const key = JSON.stringify(args);

    if (cache.has(key)) {
      return cache.get(key);
    }

    const result = fn(...args);
    cache.set(key, result);
    return result;
  };
}
```

23. What is typescript

Answer:
TypeScript is a superset of JavaScript that adds optional static typing, interfaces, classes, and other features to JavaScript. It's developed by Microsoft and is transpiled into plain JavaScript. TypeScript helps catch errors during development, provides better code organization, and supports modern JavaScript features. It's popular for building large-scale applications due to its enhanced type safety and tooling support.

24. What is eval and enum in javascript

Answer:

- **eval:** eval is a global function in JavaScript that evaluates JavaScript code represented as a string. It can execute dynamic code, but it's generally considered risky and should be used with caution due to security vulnerabilities and performance implications.
- **enum:** enum is a reserved keyword in JavaScript, but it's not yet supported as a language feature. However, TypeScript, a superset of JavaScript, supports enums. Enums allow developers to define a set of named constants, making code more readable and maintainable by providing meaningful names to values.

25. How to create private variable in js

Answer:
Using Closures:

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

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

Using Module Pattern:

```
const counter = (function() {
  let count = 0;

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

Both approaches encapsulate variables within a function scope, making them inaccessible from outside the scope. This promotes data privacy and prevents unintended modifications.

HARD

Q1.Explain the Just-In-Time (JIT) compilation in the V8 engine.

Answer:
Just-In-Time (JIT) compilation in the V8 engine is a technique used to improve the runtime performance of JavaScript code. Instead of interpreting the entire script line by line, the V8 engine analyzes and compiles the script into machine code just before executing it. This allows for more efficient execution since the compiled code can be optimized for the specific runtime environment.

Q2.Can you explain how the Chrome V8 engine handles memory management and garbage collection?

Answer:
The Chrome V8 engine employs automatic memory management through a garbage collector. V8 uses a generational garbage collection algorithm, which divides objects into two main generations: the young generation and the old generation. Newly created objects are initially placed in the young generation, and if they survive a few garbage collection cycles, they are promoted to the old generation. The V8 garbage collector utilizes techniques such as incremental marking and generational collection to efficiently identify and reclaim memory occupied by unreferenced objects. This approach helps minimize pauses in application execution by spreading the garbage collection process across multiple smaller steps.

Q3.Explain the concept of variable hoisting in JavaScript, illustrated with a suitable example

Answer:
Variable hoisting in JavaScript refers to the behavior where variable declarations are moved to the top of their containing scope during the compilation phase. However, only the declarations are hoisted, not the initializations. For example:

```
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, so the first console.log(x) prints undefined. The actual assignment (var x = 5;) remains in place, and the second console.log(x) prints the expected value of 5.

Q4.Explain the temporal dead zone (TDZ) in JavaScript and its relationship with variable hoisting.

Answer:
The temporal dead zone (TDZ) in JavaScript is the period between entering a scope and the initialization of a variable. During this period, accessing the variable results in a ReferenceError. TDZ is closely related to variable hoisting because, although the variable declaration is hoisted to the top of its scope, the initialization remains in its original position. Trying to access the variable before its initialization results in the TDZ. For example:

```
console.log(x); // ReferenceError: x is not defined
let x = 5;
console.log(x); // Output: 5
```

In this case, the first console.log(x) triggers a **ReferenceError** because the variable x is in the TDZ until the let x = 5; statement is encountered.

Q5. Explain what is Closure and explain how can you use closure to encapsulate data and create private variables in JavaScript.

Answer:
A closure in JavaScript is the combination of a function and the lexical environment within which that function was declared. Closures allow a function to access variables from its outer (enclosing) scope even after the outer function has finished executing. They are often used to create private variables and encapsulate data.

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

  return function() {
    count++;
    console.log(count);
  };
}

const counter = createCounter();
counter(); // Output: 1
counter(); // Output: 2
```

In this example, **createCounter** returns a function that has access to the count variable even though **createCounter** has already been executed. This encapsulates the **count** variable, creating a private variable accessible only through the closure.

Q6. Describe the event propagation phases in JavaScript and how they relate to event handling.

Answer:
Event propagation in JavaScript involves three phases: the capturing phase, the target phase, and the bubbling phase. During the capturing phase, the event traverses from the outermost ancestor to the target element. The target phase involves processing the event on the target element. Finally, in the bubbling phase, the event travels back up from the target to the outermost ancestor.
Event handling in JavaScript takes advantage of these phases through event listeners. You can register event listeners for capturing (useCapture: true), target, or bubbling phases. This allows for fine-grained control over how events are handled. For example:

```
element.addEventListener('click', handleClick, true); // Capturing phase
element.addEventListener('click', handleClick); // Bubbling phase (default)
```

In this scenario, handleClick will be invoked during the capturing phase and the bubbling phase, respectively. The third argument (true) indicates the use of the capturing phase.

Q7.How do you handle and prevent DOM manipulation-related security vulnerabilities, such as Cross-Site Scripting (XSS)?

Answer:
To handle and prevent DOM manipulation-related security vulnerabilities, such as Cross-Site Scripting (XSS), you should follow best practices for secure coding. Here are some strategies:

- Use proper input validation
- Sanitize user input, and avoid using innerHTML with user-generated content.
- Employ Content Security Policy (CSP) headers to restrict sources of executable scripts.

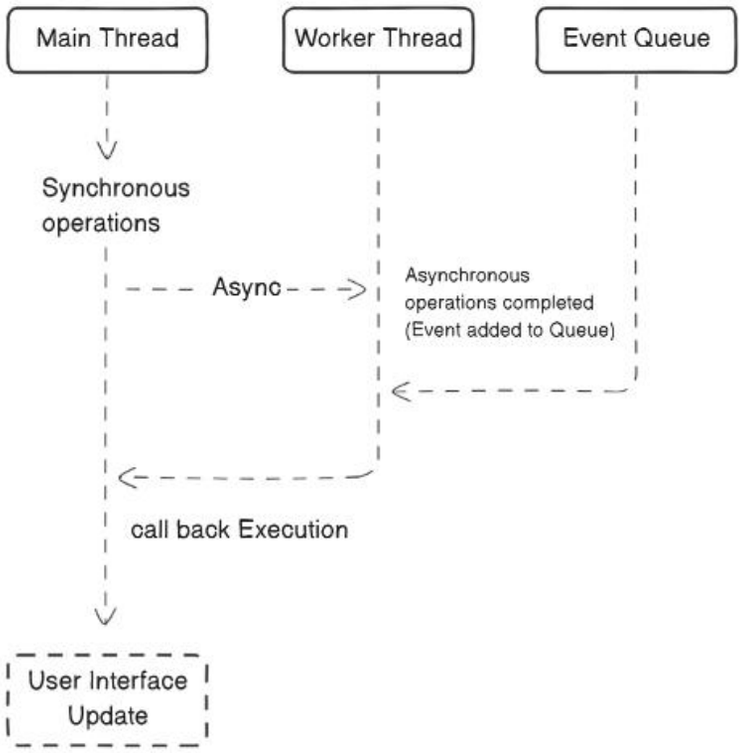
Q8.Describe the event loop in JavaScript and its role in managing asynchronous operations.

Answer:
The event loop is a fundamental mechanism in JavaScript that manages the flow of asynchronous operations and ensures that the user interface remains responsive. It acts as a central processing unit, receiving events, executing callbacks, and updating the DOM in a coordinated manner.

Key components of the event loop

1. Call stack
2. Event Queue
3. Timer Queue

Role in Managing Asynchronous Operations
The event loop plays a crucial role in managing asynchronous operations, which are tasks that take time to complete without blocking the main thread. When an asynchronous operation is initiated, it is typically handled by a separate worker thread, and the main thread continues to execute other tasks. Once the asynchronous operation is finished, an event is added to the event queue, notifying the main thread that the operation has completed and the callback function should be executed. This allows the user interface to remain responsive while asynchronous operations are in progress.



Q9.Explain the concept of a higher-order function and give an example.

Answer:
A higher-order function in JavaScript is a function that takes one or more functions as arguments or returns a function as its result. Essentially, it treats functions as first-class citizens, allowing them to be manipulated and used in a flexible manner.

```
function operateOnNumbers(a, b, operation) {
  return operation(a, b);
}

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

function multiply(x, y) {
  return x * y;
}

console.log(operateOnNumbers(3, 4, add));       // Output: 7
console.log(operateOnNumbers(3, 4, multiply));  // Output: 12
```

In this example, **operateOnNumbers** is a higher-order function that takes two numbers and a function (operation) as arguments. It then applies the provided function to the given numbers.

Q10.Explain call(), apply(), and bind() methods in JavaScript

Answer:
These methods are used to set the context (this value) of a function.

- **call():** Invokes a function with a specified value and individual arguments.
- **apply():** Invokes a function with a specified value and an array-like object of arguments.
- **bind():** Creates a new function with a specified value and initial arguments.

```
function greet(message) {
  console.log(`${message}, ${this.name}`);
}

const person = { name: 'John' };

// Example using call()
greet.call(person, 'Hello'); // Outputs: Hello, John

// Example using apply()
greet.apply(person, ['Hola']); // Outputs: Hola, John

// Example using bind()
const greetPerson = greet.bind(person);
greetPerson('Hi'); // Outputs: Hi, John
```

Q11.What is Memoization in JavaScript

Answer:
Memoization in JavaScript is an optimization technique that involves caching the results of expensive function calls to avoid redundant computations. By storing previously calculated results in a cache, subsequent calls with the same inputs can be retrieved from the cache, saving computation time. This is particularly useful for functions with repetitive or recursive calls, improving performance by eliminating unnecessary recalculations.

Q12.What is the Generator function?

Answer:
A Generator function in JavaScript is a special type of function that can be paused and resumed, allowing for the creation of iterators with a more concise syntax. It uses the function* declaration and the yield keyword to control the flow of execution. Generators are particularly useful for handling asynchronous operations and producing a sequence of values on demand.

```
function* numberGenerator() {
  yield 1;
  yield 2;
  yield 3;
}

const iterator = numberGenerator();

console.log(iterator.next().value); // Output: 1
console.log(iterator.next().value); // Output: 2
console.log(iterator.next().value); // Output: 3
```

Q13.What is a pure function, give an example

Answer:
A pure function in JavaScript is a function that, given the same input, always produces the same output and has no side effects. It doesn't modify the external state or rely on external factors, making it predictable and easy to reason about. Pure functions are valuable for functional programming principles.

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

const result1 = add(3, 5); // Output: 8
const result2 = add(3, 5); // Output: 8
```

Q14.What is the difference between shallow and deep copy

Answer:
In JavaScript, a shallow copy duplicates the top-level structure of an object or array, creating a new object, but it doesn't recursively copy nested objects or arrays. This means that nested references are shared between the original and the copy. On the other hand, a deep copy creates a completely independent copy of both the top-level structure and all nested elements, ensuring that changes in the copy do not affect the original.

```
// Shallow copy example
const originalArray = [1, [2, 3]];
const shallowCopy = [...originalArray];
shallowCopy[1][0] = 'modified';
console.log(originalArray[1][0]); // Output: 'modified'

// Deep copy example
const deepCopy = JSON.parse(JSON.stringify(originalArray));
deepCopy[1][0] = 'not modified';
console.log(originalArray[1][0]); // Output: 'modified'
```

In the shallow copy example, modifying a nested element in the copy affects the original. In the deep copy example, changes in the copy do not impact the original, showcasing the difference between the two.

Q15. Describe the prototype chaining and how it works, and explain with an example

Answer:
Prototype chaining is a mechanism in JavaScript that allows objects to inherit properties and methods from other objects through a prototype chain. Each object has an associated prototype object, and when a property or method is not found on an object, JavaScript looks for it in the object's prototype chain.

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

// Adding a method to the prototype of Animal
Animal.prototype.makeSound = function () {
  console.log("Some generic sound");
};

// Child constructor function inheriting from Animal
function Dog(name, breed) {
  Animal.call(this, name);
  this.breed = breed;
}

// Setting Dog's prototype to be an instance of Animal
Dog.prototype = Object.create(Animal.prototype);

// Adding a method to the prototype of Dog
Dog.prototype.bark = function () {
  console.log("Woof! Woof!");
};

// Creating an instance of Dog
const myDog = new Dog("Buddy", "Labrador");

// Using methods from both Animal and Dog
myDog.makeSound(); // Outputs: Some generic sound
myDog.bark();      // Outputs: Woof! Woof!
```

In this example:

- Animal is a parent constructor function with a makeSound method added to its prototype.
- Dog is a child constructor function inheriting from Animal using Object.create(Animal.prototype).
- Instances of Dog can access methods from both Dog and Animal due to the prototype chain.

The prototype chain allows for code reuse and promotes a hierarchical structure in object-oriented JavaScript programming.

Q16. Write a function to implement memoization for a given function. Memoization is an optimization technique that stores the results of previously executed function calls, avoiding redundant computations.

Answer:

```
function memoize(func) {
  const cache = {};
  return function (...args) {
    const key = JSON.stringify(args);
    if (!(key in cache)) {
      cache[key] = func(...args);
    }
    return cache[key];
  };
}

// Example usage:
function add(a, b) {
  console.log('Performing expensive calculation...');
  return a + b;
}

const memoizedAdd = memoize(add);

console.log(memoizedAdd(5, 3)); // Performs the calculation and returns 8
console.log(memoizedAdd(5, 3)); // Uses the cached result and returns 8 without recalculating
```

In this example, the memoize function takes another function (func) as an argument and returns a new function. The new function checks whether the result for the given set of arguments is already stored in the cache object. If it is, the cached result is returned. Otherwise, the original function is called, the result is stored in the cache and then returned. This helps avoid redundant computations for the same set of arguments.

Q17.Write a function to flatten a nested array into a single-level array. For example, flatten([1, [2, 3], [4, 5]]) should return [1, 2, 3, 4, 5].

Answer:

```
function flattenArray(arr) {
  return arr.reduce((flat, current) => flat.concat(Array.isArray(current) ? flattenArray(current) : current), []);
}

// Example usage:
const nestedArray = [1, [2, 3], [4, 5]];
const flattenedArray = flattenArray(nestedArray);

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

In this function:

The Array.isArray(current) condition checks if the current element is an array.

- If it is an array, flattenArray is called recursively to flatten it.
- If it is not an array, the current element is concatenated directly to the flattened array (flat).
- The reduce function is used to iterate through the array and accumulate the flattened result.

Q18.Implement a custom debounce function that delays the execution of a function until a certain amount of time has elapsed after the last call to the function. Debouncing is useful for preventing excessive function calls, such as when handling input events.

Answer:

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

  return function (...args) {
    clearTimeout(timeoutId);

    timeoutId = setTimeout(() => {
      func.apply(this, args);
    }, delay);
  };
}

// Example usage:
function handleInput(value) {
  console.log(`Handling input: ${value}`);
}

const debouncedInputHandler = debounce(handleInput, 500);

// Simulating input events
debouncedInputHandler("A");
debouncedInputHandler("AB");
debouncedInputHandler("ABC");

// After 500 milliseconds of no input, the handler will be invoked with the latest value "ABC"
```

Output
Handling ABC

In this example:

- The **debounce** function takes a function **(func)** and a delay in milliseconds as arguments.
- It returns a new function that, when invoked, sets a timeout to execute the original function after the specified delay.
- If the new function is invoked again before the timeout completes, the previous timeout is cleared, and a new one is set.
- This ensures that the original function is only called after the specified delay has elapsed since the last invocation.

Q19. Write a function to implement throttling for a given function. Throttling limits the execution of a function to a specific frequency, ensuring that it doesn't get called too frequently.

```
function throttle(func, delay) {
  let canRun = true;

  return function (...args) {
    if (canRun) {
      func.apply(this, args);
      canRun = false;

      setTimeout(() => {
        canRun = true;
      }, delay);
    }
  };
}

// Example usage:
function logMessage(message) {
  console.log(message);
}

const throttledLog = throttle(logMessage, 1000);

// Simulating frequent calls
throttledLog("First call");  // Logs "First call"
throttledLog("Second call"); // Ignored
throttledLog("Third call");  // Ignored

// After 1000 milliseconds, the next call will be allowed
```

In this example:

- The throttle function takes a function **(func)** and a **delay** in milliseconds as arguments.
- It returns a new function that, when invoked, checks whether it's allowed to run based on the **canRun** flag.
- If allowed, it calls the original function and sets a timeout to enable the next invocation after the specified delay.
- Subsequent calls within the delay period are ignored until the timeout completes, ensuring the function is not executed too frequently.

20. What are event emitters

Answer:
Event emitters are objects that facilitate communication between different parts of a system by emitting events. They work by allowing objects to register callback functions (listeners) to handle specific events. When an event occurs, the emitter triggers all registered listeners for that event. Event emitters are commonly used in event-driven architectures, such as in Node.js applications, to handle asynchronous operations and manage the flow of data and control within the system.