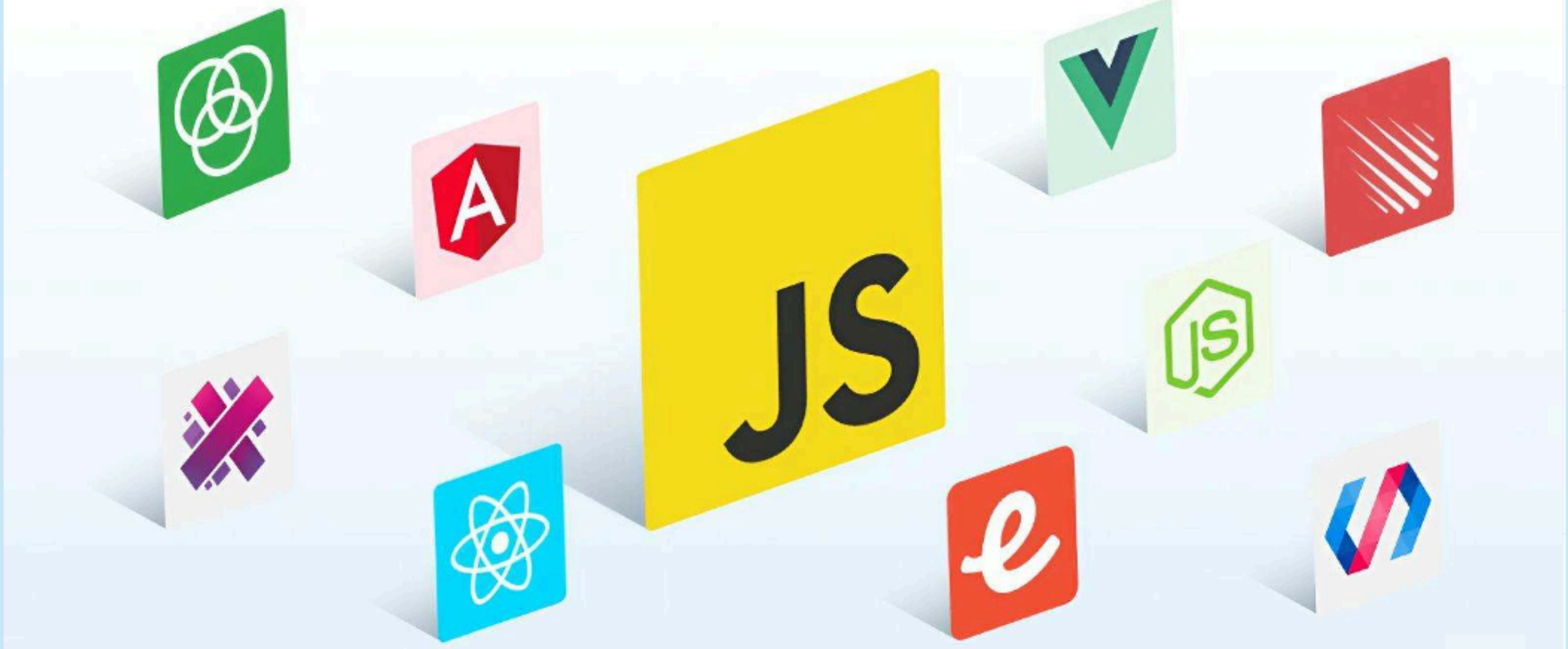
JAVASCRIPT

TECHNICAL INTERVIEW



@KhushiChoudhary

EASY

Q.1

How do you detect primitive or non-primitive value types in Javascript?

In JavaScript, values are generally categorized as either primitive or non-primitive (also known as reference types). Primitive values include:

1. **Number:** Represents numeric values.
2. **String:** Represents textual data.
3. **Boolean:** Represents true or false.
4. **Undefined:** Represents an uninitialized variable or absence of a value.
5. **Null:** Represents the intentional absence of any object value.
6. **Symbol:** Represents a unique identifier.

Non-primitive values are objects, which include arrays, functions, and custom objects.

We can detect primitive or non primitive in Javascript in the following ways:

1. Using the `typeof` operator:

- This operator returns a string indicating the type of a value.
- Primitive types will return their corresponding strings (e.g., "number", "string", "boolean").
- Non-primitive types will typically return "object" or "function".

EASY

Javascript

```
// Using the typeof operator:


let num = 10;
let str = "Hello";
let bool = true;
let obj = {};
let func = function() {};

console.log(typeof num);   // Output: "number"
console.log(typeof str);   // Output: "string"
console.log(typeof bool);  // Output: "boolean"
console.log(typeof obj);   // Output: "object"
console.log(typeof func);  // Output: "function"
```

Important note:

`typeof null` returns "object" even though it's a primitive value.

@KhushiChoudhary

EASY

2. Using the `Object()` constructor:

- This constructor creates a new object wrapper for a value.
- If a value is primitive, it will be equal to its object-wrapped version.
- If a value is non-primitive, it won't be equal to its object-wrapped version.

Javascript

```
// Using the Object() constructor:


console.log(num === Object(num));   // Output: true (primitive)
console.log(obj === Object(obj));   // Output: false (non-primitive)
```

EASY

Q.2

Explain the key features introduced in Javascript ES6

In ES6, JavaScript introduced these key features:

1. **Arrow Functions:**
   - Concise syntax for anonymous functions with lexical scoping.
2. **Template Literals:**
   - Enables multiline strings and variable inclusion for improved readability.
3. **Destructuring Assignment:**
   - Simplifies extraction of values from arrays or objects.
4. **Enhanced Object Literals:**
   - Introduces shorthand notation for defining object methods and dynamic property names.
5. **Promises:**
   - Streamlines asynchronous programming with a cleaner, structured approach.

EASY

Q.3

What are the differences between var, const & let in JavaScript?

Attribute	var	let	const
Scope	Functional scope	Block scope	Block scope
Update/ Re-declaration	Can be updated and re-declared within the scope	Can be updated but cannot be re-declared within the scope	Cannot be updated or re-declared within the scope
Declaration without Initialization	Can be declared without being initialized	Can be declared without being initialized	Cannot be declared without being initialized
Access without Initialization	Accessible without initialization (default: undefined)	Inaccessible without initialization (throws 'ReferenceError')	Inaccessible without initialization (throws 'ReferenceError')
Hoisting	Hoisted and initialized with a 'default' value	Hoisted but not initialized (error if accessed before declaration/ initialization)	Hoisted but not initialized (error if accessed before declaration/ initialization)

MEDIUM

Q.4

What are arrow functions in Javascript?

Arrow functions are a concise way to write anonymous function expressions in JavaScript. They were introduced in ECMAScript 6 (ES6) and are especially useful for short, single-expression functions.

Here's the basic syntax for an arrow function:

```Javascript
const add = (a, b) => {
  return a + b;
};
```

In this example, the arrow function `add` takes two parameters (`a` and `b`) and returns their sum. The ⇒ syntax is used to define the function, and the body of the function is enclosed in curly braces `{}`. If there's only one expression in the function body, you can omit the curly braces and the `return` keyword:

```Javascript
const add = (a, b) => {
  return a + b;
};
```

MEDIUM

Here is an example showing how both traditional function expression and arrow function to illustrate the difference in handle the this keyword.

Traditional Function Expression:

```Javascript
// Define an object
let obj1 = {
  value: 42,
  valueOfThis: function() {
    return this.value; // 'this' refers to the object calling the function (obj1)
  }
};

// Call the method
console.log(obj1.valueOfThis()); // Output: 42
```

In this example, `obj1.valueOfThis()` returns the `value` property of `obj1`, as `this` inside the function refers to the object `obj1`.

MEDIUM

Arrow Function:

```Javascript
// Define another object
let obj2 = {
  value: 84,
  valueOfThis: () => {
    return this.value; // 'this' does not refer to obj2; it inherits from the parent scope (window in this case)
  }
};

// Call the method
console.log(obj2.valueOfThis()); // Output: undefined or an error (depending on the environment)
```

In the arrow function within `obj2`, this does not refer to `obj2`. Instead, it inherits its value from the parent scope, which is the global object (`window` in a browser environment). Consequently, `obj2.valueOfThis()` returns `undefined` or may even throw an error, as `this.value` is not defined in the global scope.

MEDIUM

Q.5 What is hoisting in Javascript?

In JavaScript, hoisting is a phenomenon where variable and function declarations are conceptually moved to the top of their respective scopes, even if they're written later in the code. This behaviour applies to both global and local scopes.

Here are some examples to illustrate hoisting:

Example 1: Variable Hoisting

```Javascript
console.log(myMessage);  // Outputs "undefined", not an error

var myMessage = "Greetings!";
```

While `myMessage` appears declared after its use, it's hoisted to the top of the scope, allowing its reference (but not its initial value) before the actual declaration line.

Example 2: Function Hoisting

```Javascript
sayHello();  // Outputs "Hello, world!"
 function sayHello() {
  console.log("Hello, world!");
}
```

@KhushiChoudhary

MEDIUM

Even though sayHello is defined after its call, JavaScript acts as if it were declared at the beginning of the scope, enabling its execution.

Example 3: Hoisting within Local Scopes

```Javascript
function performTask() {
  result = 100;  // Hoisted within the function
  console.log(result);  // Outputs 100
  var result;
}

performTask();
```

Hoisting also occurs within local scopes, like functions. Here, `result` is hoisted to the top of the `performTask` function, allowing its use before its explicit declaration.

Key Points:

- Only declarations are hoisted, not initializations. The example with `console.log(x)`; demonstrates this, as x is declared but not initialised before its use, resulting in `undefined`.
- Strict mode enforces declaration: Using `"use strict"`; at the beginning of your code prevents using variables before they're declared, helping avoid potential hoisting-related issues.

@KhushiChoudhary

EASY

EASY

Q.6

What is Strict Mode in Javascript?

Strict Mode is a feature that allows you to place a program, or a function, in a “strict” operating context. This way it prevents certain actions from being taken and throws more exceptions. The literal expression "use strict"instructs the browser to use the javascript code in the Strict mode.

Strict mode helps in writing "secure" JavaScript by notifying "bad syntax" into real errors.

The strict mode is declared by adding "use strict"; to the beginning of a script or a function. If declared at the beginning of a script, it has global scope.

Example:

```Javascript
'use strict';
x = 15; // ReferenceError: x is not defined
function strict_function() {
  'use strict';
 x = 'Test message';
  console.log(x);
}
strict_function(); // ReferenceError: x is not defined
```

EASY

Q.7 What is NaN?

The NaN property in JavaScript represents a value that is "Not-a-Number," indicating an illegal or undefined numeric value. When checking the type of NaN using the `typeof` operator, it returns "Number."

To determine if a value is NaN, the `isNaN()` function is employed. It converts the given value to a Number type and then checks if it equals NaN.

Example:

```
isNaN("Hello");    // Returns true, as "Hello" cannot be converted to a valid number
isNaN(NaN);    // Returns true, as NaN is, by definition, Not-a Number
isNaN("123ABC");    // Returns true, as "123ABC" cannot be converted to a valid number
isNaN(undefined);    // Returns true, as undefined cannot be converted to a valid number
isNaN(456);    // Returns false, as 456 is a valid numeric value
isNaN(true);    // Returns false, as true is converted to 1, a valid number
isNaN(false);    // Returns false, as false is converted to 0, a valid number
isNaN(null);    // Returns false, as null is converted to 0, a valid number
```

@KhushiChoudhary

EASY

Q.8

Is javascript a statically typed or a dynamically typed language?

JavaScript is a dynamically typed language. In a dynamically typed language, variable types are determined at runtime, allowing a variable to hold values of any type without explicit type declarations. This flexibility can make coding more convenient but may also lead to runtime errors if types are not handled appropriately.

JavaScript, being dynamically typed, allows variables to change types during execution and accommodates a wide range of data types without explicit type annotations.

MEDIUM

Q.9 **What is NaN?**

- Functions that treat other functions as values, either by:

1. Taking one or more functions as arguments
2. Returning a function as a result

Common Examples of Built-in HOFs:

1. `map()`:

- Applies a function to each element of an array and creates a new array with the results.
- *Example:*
  const numbers = [1, 2, 3, 4, 5];
  const doubledNumbers = numbers.map(number ⇒ number * 2); // [2, 4, 6, 8, 10]

2. `filter()`:

- Creates a new array containing only elements that pass a test implemented by a provided function.
- *Example:*
  const numbers = [1, 2, 3, 4, 5];
  const evenNumbers = numbers.filter(number ⇒ number % 2 === 0); // [2, 4]

MEDIUM

3. reduce():

- Applies a function against an accumulator and each element in an array (from left to right) to reduce it to a single value.
- Example:
  const numbers = [1, 2, 3, 4];
  const sum = numbers.reduce((accumulator, number) ⇒ accumulator + number, 0); // 10

Creating Custom HOFs:

You can define your own HOFs to encapsulate common patterns and operations:

```Javascript
function createMultiplier(factor) {
  return number => number * factor;
}
const triple = createMultiplier(3);
const tripledNumbers = numbers.map(triple); // [3, 6, 9, 12, 15]
```

EASY

Q.10

What is difference between Null and Undefined

Feature	Null	Undefined
Type	Object	Undefined
Definition	Assignment value indicating no object	Variable declared but not yet assigned a value
Nature	Primitive value representing null/empty	Primitive value used when a variable is unassigned
Representation	Absence of a value for a variable	Indicates the absence of the variable itself
Conversion in Operations	Converted to zero (0)	Converted to NaN during primitive operations

MEDIUM

Q.11 What is DOM?

DOM stands for Document Object Model, serving as a programming interface for web documents.

1. **Tree Structure:** It represents the document as a tree, with the document object at the top and elements, attributes, and text forming the branches.
2. **Objects:** Every document component (element, attribute, text) is an object in the DOM, allowing dynamic manipulation through programming languages like JavaScript.
3. **Dynamic Interaction:** Enables real-time updates and interactions on web pages by modifying content and structure in response to user actions.
4. **Programming Interface:** Provides a standardized way to interact with a web document, accessible and modifiable using scripts.
5. **Cross-platform and Language-Agnostic:** Not bound to a specific language and works across various web browsers, ensuring a consistent approach to document manipulation.
6. **Browser Implementation:** While browsers have their own DOM implementations, they follow standards set by the World Wide Web Consortium (W3C), ensuring uniformity in document representation and manipulation.

MEDIUM

Q.12 What is BOM?

BOM (Browser Object Model) is a programming interface extending beyond DOM, providing control over browser-related features.

1. **Window Object:** Core BOM element representing the browser window, with properties and methods for browser control.
2. **Navigator, Location, History, Screen Objects:** Components handling browser information, URL navigation, session history, and screen details.
3. **Document Object:** Accessible through BOM, allowing interaction with the structure of web pages.
4. **Timers:** Functions like setTimeout and setInterval for scheduling code execution.
5. **Client Object:** Represents user device information, aiding in responsive web design.
6. **Event Object:** Manages events triggered by user actions or browser events.

MEDIUM

Q.13

Explain about this keyword in Javascript with an example.

In JavaScript, the `this` keyword is a special variable that is automatically defined in the scope of every function. Its value depends on how the function is invoked. The `this` keyword is used to refer to the object that is the current context of the function or, more simply, the object that the function is a method of.

Here are some common scenarios that affect the value of `this`:

Global Context:

When `this` is used outside of any function or method, it refers to the global object (in a browser environment, it usually refers to `window`).

```Javascript
console.log(this); // refers to the global object
(e.g., window in a browser)
```

Method Invocation:

When a function is a method of an object, this refers to that object.
When a function is a method of an object, this refers to that object.

MEDIUM

Javascript

```
const myObject = {
  myMethod: function() {
    console.log(this); // refers to myObject
  }
};

myObject.myMethod();
```

Constructor Function:

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

Javascript

```
function MyClass() {
  this.property = 'some value';
}

const myInstance = new MyClass();
console.log(myInstance.property); // 'some value'
```

MEDIUM

Q.14

What is scope in Javascript?

In JavaScript, the term "scope" refers to the context in which variables and functions are declared and accessed. It defines the visibility and accessibility of these variables and functions within the code. Understanding scope is crucial for managing the lifecycle and behaviour of variables and functions in a program.

There are two main types of scope in JavaScript: global scope and local scope.

Global Scope:

- Variables declared outside of any function or block have global scope.
- Global variables are accessible throughout the entire code, including within functions.

```Javascript
var globalVar = "I am global";

function exampleFunction() {
    console.log(globalVar); // Accessible inside the function
}

exampleFunction();

console.log(globalVar); // Accessible outside the function
```

MEDIUM

Local Scope:

- Variables declared inside a function or block have local scope.
- Local variables are only accessible within the function or block where they are declared.

```Javascript
function exampleFunction() {
    var localVar = "I am local";
    console.log(localVar); // Accessible inside the function
}

exampleFunction();
// console.log(localVar); // This would result in an error because localVar is not accessible outside the function
```

Scope Chain:

The scope chain refers to the hierarchy of scopes in a program. When a variable or function is referenced, JavaScript looks for it in the current scope and then traverses up the scope chain until it finds the variable or reaches the global scope.

MEDIUM

Javascript

```
var globalVar = 42;

function mainFunction(){
  var localVar1 = 777;

  var innerFunction1 = function(){
 console.log(localVar1); // Accesses localVar1 inside innerFunction1, outputs 777
  }

  var innerFunction2 = function(){
 console.log(globalVar); // Accesses globalVar inside innerFunction2, outputs 42
  }

  innerFunction1();
  innerFunction2();
}

mainFunction();
```

@KhushiChoudhary

MEDIUM

Q.15 What is closure in Javascript?

In JavaScript, a closure is a function along with its lexical scope, which allows it to access variables from its outer (enclosing) scope even after that scope has finished executing. A closure allows a function to remember and access variables from the environment in which it was created, even if the function is executed in a different scope.

Here's an example to illustrate closures in JavaScript:

MEDIUM

Javascript

```
function outerFunction() {
  // Outer function scope
  let outerVariable = 10;

  function innerFunction() {
    // Inner function scope
    let innerVariable = 5;
    // Accessing both inner and outer variables
    console.log("Inner Variable:", innerVariable);
    console.log("Outer Variable:", outerVariable);
  }

// Returning the inner function, creating a closure
  return innerFunction;
}
 // Calling outerFunction returns innerFunction,
which is now a closure
let closureFunction = outerFunction();

// Executing the closure function
closureFunction();
```

@KhushiChoudhary

MEDIUM

`outerFunction` defines an outer variable (`outerVariable`) and an inner function (`innerFunction`).

`innerFunction` has access to the variables of its outer function (`outerVariable`).

`outerFunction` returns `innerFunction`, creating a closure.

The returned `closureFunction` retains access to the `outerVariable` even after `outerFunction` has finished executing.

Calling `closureFunction()` logs both the inner and outer variables to the console.

@KhushiChoudhary

HARD

Q.16

Explain call(), apply() and bind() methods in Javascript.

In JavaScript, the call, apply, and bind methods are used to manipulate how a function is invoked and set the value of this within the function.

`call` method:

The `call` method is used to invoke a function with a specified `this` value and arguments provided individually.

Javascript

```
function sayHello(greeting) {
    console.log(greeting + ' ' + this.name);
}

const person = { name: 'John' };

sayHello.call(person, 'Hello'); // Outputs: Hello John
```

Here, `call` is used to invoke the `sayHello` function with `person` as the `this` value, and 'Hello' as an argument.

HARD

`apply` method:

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

```Javascript
function sayHello(greeting) {
    console.log(greeting + ' ' + this.name);
}

const person = { name: 'John' };

sayHello.apply(person, ['Hello']); // Outputs: Hello John
```

In this example, `apply` is used to achieve the same result as `call`, but the arguments are provided as an array.

`bind` method:

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

HARD

Javascript

```
function sayHello(greeting) {
    console.log(greeting + ' ' + this.name);
}

const person = { name: 'John' };

const sayHelloToJohn = sayHello.bind(person);

sayHelloToJohn('Hello'); // Outputs: Hello John
```

Here, `bind` is used to create a new function (`sayHelloToJohn`) where `this` is permanently set to `person`. When calling `sayHelloToJohn`, it's as if you're calling `sayHello` with `person` as `this`.

These methods are especially useful when dealing with functions that are part of objects or classes, and you want to explicitly set the context (`this`) for their execution.

MEDIUM

Q.17

Explain call(), apply() and bind() methods in Javascript.

In JavaScript, the call, apply, and bind methods are used to manipulate how a function is invoked and set the value of this within the function.

`call` method:

The `call` method is used to invoke a function with a specified `this` value and arguments provided individually.

Javascript

```
function sayHello(greeting) {
    console.log(greeting + ' ' + this.name);
}

const person = { name: 'John' };

sayHello.call(person, 'Hello'); // Outputs: Hello
John
```

Here, `call` is used to invoke the `sayHello` function with `person` as the `this` value, and 'Hello' as an argument.

MEDIUM

In this example, the add function is pure because it only depends on its input parameters (a and b) to produce a result and doesn't modify any external state.

Contrast this with an impure function that relies on external state or has side effects:

Javascript

```
// Impure function (has side effects)
let total = 0;

function addToTotal(value) {
  total += value;
}

// Example usage of the impure function
addToTotal(5);
console.log(total); // Output: 5
```

In this case, `addToTotal` is impure because it modifies the external variable `total` and has a side effect that can affect other parts of the program.

MEDIUM

Q.18 What are prototypes in Javascript?

- **Every object in JavaScript has a prototype**, which acts as a blueprint for shared properties and methods.
- **When you try to access a property or method on an object, JavaScript first checks the object itself.**
- **If it's not found, it looks up the prototype chain**, following a linked list of prototypes until it finds what it's looking for, or reaches the end (null).

Example:

```Javascript
function Person(name) {
  this.name = name;
}

// Add a method to the prototype, shared by all Person objects:
Person.prototype.greet = function() {
  console.log("Hello, my name is " + this.name);
};
```

MEDIUM

Javascript

```
// Create two Person objects:
const person1 = new Person("Alice");
const person2 = new Person("Bob");

// Both objects can access the greet method from the
prototype:
person1.greet();  // Output: "Hello, my name is
Alice"
person2.greet();  // Output: "Hello, my name is Bob"
```

The `Person` function acts as a constructor to create objects with a `name` property.

The `greet` method is added to the `Person.prototype`, meaning it's shared by all instances created from `Person`.

When person1.greet() is called, JavaScript finds the `greet` method on the prototype, so it can be used even though it wasn't defined directly on `person1`.

@KhushiChoudhary

HARD

Q.19

What are callback functions in Javascript and what is callback hell?

In JavaScript, a callback is a function that is passed as an argument to another function and is executed after the completion of some asynchronous operation or at a specified time. Callbacks are commonly used in scenarios like handling asynchronous tasks, event handling, and other situations where the order of execution is not guaranteed.

Javascript

```
function customGreeting(name) {
  console.log("Welcome, " + name + "! How can we assist you today?");
}

function outerFunction(callback) {
  let name = prompt("Please enter your name.");
  callback(name);
}

outerFunction(customGreeting);
```

In this example, the `customGreeting` function is the callback function passed to `outerFunction`

HARD

Callback hell (or "pyramid of doom") is a situation in which multiple nested callbacks make the code difficult to read and maintain. This often occurs when dealing with asynchronous operations, such as making multiple API calls or handling multiple events.

Here's an example of callback hell:

```Javascript
getUser(function(user) {
  getProfile(user.id, function(profile) {
    getPosts(user.id, function(posts) {
      displayUserProfile(user, profile, posts, function() {
        // More nested callbacks...
      });
    });
  });
});
```

In this example, we have nested callbacks for getting a user, fetching their profile, retrieving their posts, and finally displaying the user profile. As more asynchronous operations are added, the code becomes more difficult to read and maintain.

To address callback hell, developers often use techniques like Promises or async/await in modern JavaScript to make code more readable and manageable.

@KhushiChoudhary

MEDIUM

Q.20

What is Temporal Dead Zone in Javascript?

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

Let's illustrate this behaviour with an example:

```Javascript
function exampleMethod() {
  console.log(value1); // Outputs: undefined
  console.log(value2); // Throws a ReferenceError
  var value1 = 1;
  let value2 = 2;
}
```

In this example, attempting to access `value2` before its declaration causes a ReferenceError due to the temporal dead zone, while accessing `value1` results in an output of `undefined`.

HARD

Q.21 What are promises in Javascript?

JavaScript Promises offer a streamlined approach to managing asynchronous operations, mitigating the callback hell problem encountered with events and traditional callback functions. Before Promises, working with callbacks often led to code that was hard to manage due to nested structures. Promises serve as a cleaner solution for handling asynchronous tasks in JavaScript.

Here's the syntax for creating a Promise:

```Javascript
let promise = new Promise(function(resolve, reject)
{
    // Perform asynchronous operations
});
```

The Promise constructor takes a single callback function as its argument, which, in turn, accepts two parameters: `resolve` and `reject`. The operations inside this callback determine whether the Promise is fulfilled by calling `resolve` or rejected by calling `reject`.

A Promise can exist in one of four states:

- **fulfilled:** The action related to the promise succeeded.
- **rejected:** The action related to the promise failed.
- **pending:** The promise is still awaiting fulfilment or rejection.
- **settled:** The promise has been either fulfilled or rejected.

MEDIUM

Q.22 Explain rest parameter in Javascript

In JavaScript, the rest parameter is a feature that allows you to represent an indefinite number of arguments as an array. It is denoted by three dots (...) followed by the parameter name. The rest parameter collects all the remaining arguments passed to a function into a single array.

Here's a simple example to illustrate the concept:

```Javascript
function sum(...numbers) {
  return numbers.reduce((total, num) => total + num,
0);
}


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

In this example, the sum function accepts any `number` of arguments. The rest parameter ...`numbers` collects all the arguments into an array called `numbers`. The function then uses the `reduce` method to sum up all the numbers in the array.

It's important to note that the rest parameter must be the last parameter in the function declaration. For example, this is valid:

MEDIUM

```Javascript
function example(firstParam, ...restParams) {
  // code here
}
```

But this is not:

```Javascript
function invalidExample(...restParams, lastParam) {
  // code here
}
```

HARD

Q.23

What are generator functions in Javascript?

In JavaScript, generator functions are a special kind of function that allows you to control the execution flow and pause/resume it at certain points. Generator functions are defined using the `function*` syntax and use the `yield` keyword to produce a sequence of values. When a generator function is called, it returns an iterator called a generator.

Here's a simple example of a generator function:

```Javascript
function* simpleGenerator() {
  yield 1;
  yield 2;
  yield 3;
} // Creating a generator
const generator = simpleGenerator();
// Using the generator to get values
console.log(generator.next()); // { value: 1, done: false }
console.log(generator.next()); // { value: 2, done: false }
console.log(generator.next()); // { value: 3, done: false }
```

HARD

```
console.log(generator.next()); // { value: undefined,
done: true }
```

In this example:

- The `function* simpleGenerator()` syntax defines a generator function.
- The `yield` keyword is used to produce values. Each time `yield` is encountered, the generator pauses its execution, and the yielded value is returned to the caller along with `done: false`. The generator can be resumed later.
- The `generator.next()` method is used to advance the generator's execution. It returns an object with two properties: `value` (the yielded value) and `done` (a boolean indicating whether the generator has finished).

Generators are useful for lazy evaluation, asynchronous programming, and creating iterable sequences.

MEDIUM

Q.24

What is the difference between function declarations and function expressions?

Function Declaration:

- A function declaration is a statement that defines a function and hoists it to the top of the current scope.
hoists it to the top of the current scope.N
- It starts with the `function` keyword, followed by the function name, parameters (enclosed in parentheses), and the function body.
- Example:

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

Function declarations can be called before they are declared in the code because of hoisting.

Function Expression:

- A function expression is an assignment where a function is defined as part of an expression.
- It does not get hoisted in the same way as function declarations.

MEDIUM

- Example:

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

In this example, `add` is a variable that holds an anonymous function.

Function declarations are hoisted, while function expressions are not hoisted in the same way. If you try to call a function expression before its definition, you'll get an error.

Function expressions are often used in cases where you need to assign a function to a variable or pass it as an argument to another function.

HARD

Q.25

What is the difference between setTimeout, setImmediate and process.nextTick?

setTimeout, setImmediate, and process.nextTick are all functions in Node.js that allow you to schedule the execution of a callback function, but they have some differences in terms of when the callback will be executed.

1. setTimeout:

- Schedules the callback to be executed after a specified delay (in milliseconds).
- The callback is added to the event queue, and it will be executed after the specified delay, but the exact timing is not guaranteed.

```Javascript
setTimeout(() => {
  console.log('This will be executed after 1000 milliseconds');
}, 1000);
```

2. setImmediate:

- Schedules the callback to be executed in the next iteration of the event loop.
- It's often used when you want the callback to be executed immediately after the current event loop cycle.

HARD

```Javascript
setImmediate(() => {

  console.log('This will be executed in the next iteration of the event loop');

});
```

3. process.nextTick:

- Executes the callback after the current event loop cycle, but before the event loop continues processing other I/O events.
- It is often used when you want to execute a callback after the current operation but before I/O events.

```Javascript
process.nextTick(() => {

  console.log('This will be executed in the next event loop cycle');

});
```

FOLLOW FOR MORE

@KHUSHICHOUDHARY