HTML

Question1: What does HTML stand for and what is it used for?

Answer: HTML stands for HyperText Markup Language. It is used for creating the structure of a webpage.

Question2: What does "HyperText" mean?

Answer: HyperText refers to text that contains links to other texts, allowing users to navigate between pages or sections.

Question3: What does "Markup Language" mean?

Answer: Markup Language is a way to use tags to tell the browser how to display text and other elements on a webpage.

Question4: What is the difference between tags and elements?

Answer: Tags are the basic building blocks of HTML, used to define and structure the content on a webpage. They are written within angle brackets, like `<tagname>`. Tags usually come in pairs: an opening tag (`<tagname>`) and a closing tag (`</tagname>`), though some are self-closing.

Ex: `<!DOCTYPE>, <html>, <head>, <title>, <meta>, <body>, <h1> to <h6>, <p>, <img>, <a>, <ul>, <ol>, <li>, <table>, <tr>, <td>, <div>, <span>, <form>, <input>.`

Elements: These are the complete structures formed by tags, including the opening tag, content, and closing tag. An element represents a part of the webpage's content.

Ex: `<h1>This is a heading</h1>`

Question5: What is `<!DOCTYPE html>`?

Answer: `<!DOCTYPE html>` declares the document type and HTML version, helps the browser to render the page correctly.

Question6: What is the `<html>` tag?

Answer: The <html> tag represents the root of the HTML document and encloses all the content on the webpage.

Question7: What is the <head> tag?

Answer: The <head> tag contains meta-information about the document, such as the title, links to stylesheets, and scripts.

Question8: What is the <meta> tag?

Answer: The <meta> tag provides information about metadata, such as the author, description, keywords, and viewport settings. It helps with SEO and does not display on the browser.

Question9: What is the <body> tag?

Answer: The <body> tag contains the main content of the webpage, such as text, images, links, and other elements.

Question10: What are block-level elements and inline elements?

Answer:

- **Block-Level Elements:** These elements start on a new line and take up the full width available. Examples include <div>, <p>, <h1>, <ul>, and <table>.
- **Inline Elements:** These elements do not start on a new line and only take up as much width as necessary. Examples include <span>, <a>, <img>, <strong>, and <input>.

Question11: What is the difference between <div> and <span>?

Answer:

- **<div>:** A block-level element used to group and structure larger sections of content. It starts on a new line and takes up the full width available.
- **<span>:** An inline element used to style or group small portions of text or other inline elements. It does not start on a new line and only takes up as much width as necessary.

Question12: What are inline-block elements?

Answer: Inline-block elements are a combination of block-level and inline-level behaviors. They allow you to set height and width like block-level elements, but they do not start on a new line and only take up as much width as necessary, like inline elements.

Question11: What is the current version of HTML, and what are some new features introduced in HTML5?

Answer: The current version of HTML is HTML5. Some new features introduced in HTML5 include:

1. New semantic elements: `<header>`, `<footer>`, `<article>`, `<section>`, `<nav>`, `<aside>`, `<figure>`, `<figcaption>`.
2. Enhanced form controls: New input types like `email`, `date`, `range`, `color`.
3. `<audio>` and `<video>` tags for embedding media.
4. `<canvas>` for drawing graphics with JavaScript.
5. SVG integration for high-quality graphics.
6. Local storage: `localStorage` and `sessionStorage`.

Question12: What are semantic elements and what is their use?

Answer: Semantic elements are HTML tags that clearly describe their meaning and purpose. They improve accessibility, enhance SEO, and make the code more readable. Examples include `<header>`, `<footer>`, `<article>`, `<section>`, `<nav>`, and `<aside>`.

Question13: What are attributes, and what are some examples of their use?

Answer: Attributes are extra information added to HTML tags to define their properties or behavior. Examples include:

- **`<a>` tag:** `href` (URL of the link), `target` (where to open the link).

- **<img> tag:** src (URL of the image), alt (alternative text).
- **<input> tag:** type (type of input), placeholder (hint text).
- **<div> tag:** id (unique identifier), class (styling class).

Question14: What is the purpose of the name attribute in HTML?

Answer: The name attribute identifies form elements for server-side processing and data handling. It helps associate the data with corresponding fields when the form is submitted.

Question15: What is the use of the target attribute in a <a> tag? Provide use cases.

Answer: The target attribute specifies where to open the linked document:

1. **_blank**: Opens the link in a new tab or window.
2. **_self**: Opens the link in the same frame (default).
3. **_parent**: Opens the link in the parent frame.

Question16: What is the difference between <strong> and <b> tags?

- **Answer:** <strong> indicates that text is of strong importance and has semantic meaning, while <b> simply makes the text bold without adding any semantic value.

Question17: What are self-closing (or empty) elements in HTML?

Answer: Self-closing elements are HTML tags that do not require closing tags and do not contain content. Examples include:

1. **<img>**: Embeds an image.
2. **<input>**: Creates an input field.
3. **<br>**: Inserts a line break.
4. **<hr>**: Creates a horizontal line.
5. **<meta>**: Provides metadata.
6. **<link>**: Links to external resources.

Question18: What are the types of lists in HTML?

Answer:

1. **Ordered List (`<ol>`):**
    - **Definition:** Lists items in a specific order with numbers or letters.
    - **Use:** When the sequence of items matters.
2. **Unordered List (`<ul>`):**
    - **Definition:** Lists items with bullets (dots).
    - **Use:** When the order of items does not matter.
3. **Definition List (`<dl>`):**
    - **Definition:** Lists terms with their descriptions.
    - **Use:** For glossaries or term-definition pairs.

Question19: What are the ways to embed JavaScript in an HTML page?

Answer:

1. **Inline JavaScript:** Adding JS directly within an element's attribute.
2. **Internal JavaScript:** Using the `<script>` tag inside `<head>` or `<body>`.
3. **External JavaScript:** Linking an external JS file with the `<script>` tag.

Question20: What are the advantages of adding JavaScript at the end of the `<body>`?

Answer: Improved page load time and reduced render blocking.

Question21: What are the disadvantages of adding JavaScript in the `<head>` section?

Answer: Slower page load time and render blocking.

JavaScript Introduction:

JavaScript is a scripting language as well as a high level programming language that is used for creating dynamic web content. It can run both client side(browser) and server side(Node.js).

JavaScript is a synchronous and single threaded language because when JavaScript code runs, code executes line by line.

Ch-1

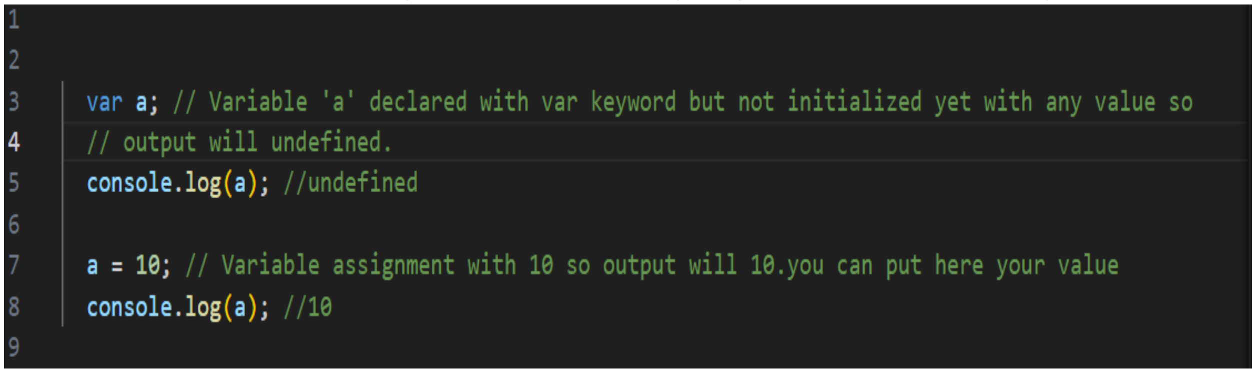
Explain variables in JavaScript:

Variables are containers for storing data. there are three types of variables based on their declaration keywords

(a) var Variables
(b) let Variables
(c) const Variables

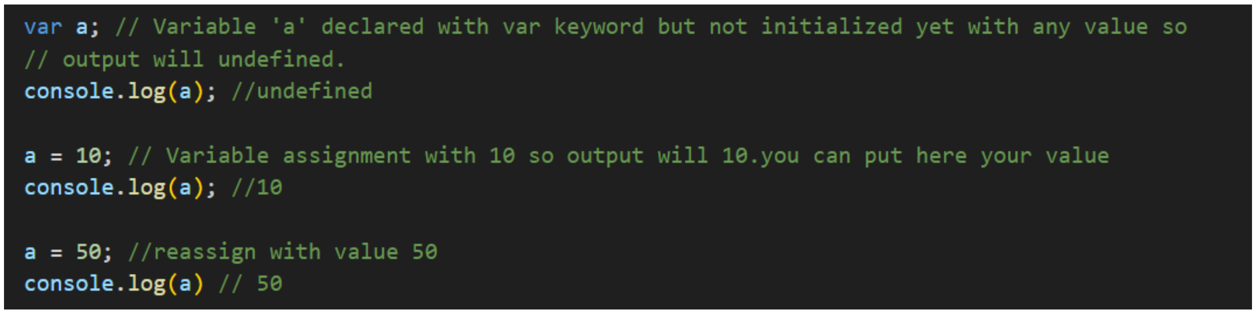
let and const are introduced in 2015(ES6).

Let's move towards declaring variable and assigning the value with var keyword:



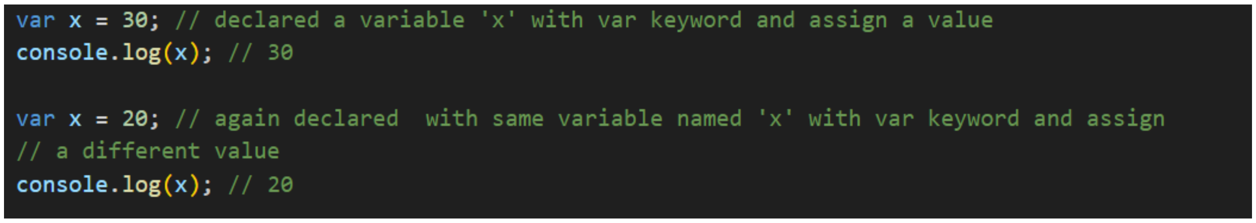
```
var a; // Variable 'a' declared with var keyword but not initialized yet with any value so
// output will undefined.
console.log(a); //undefined

a = 10; // Variable assignment with 10 so output will 10.you can put here your value
console.log(a); //10
```

Re-assigning a value to variable with var keyword:



```
var a; // Variable 'a' declared with var keyword but not initialized yet with any value so
// output will undefined.
console.log(a); //undefined

a = 10; // Variable assignment with 10 so output will 10.you can put here your value
console.log(a); //10

a = 50; //reassign with value 50
console.log(a) // 50
```

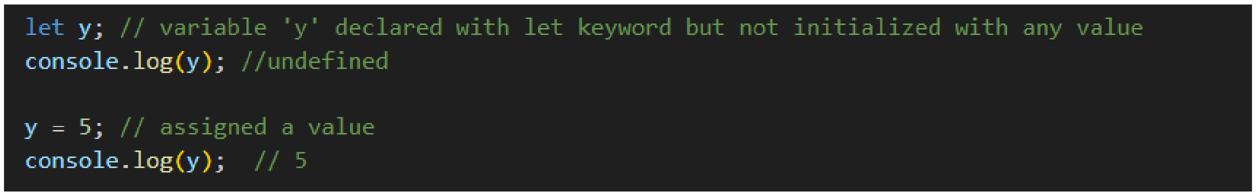
Re-declaring a variable with var keyword:



```
var x = 30; // declared a variable 'x' with var keyword and assign a value
console.log(x); // 30

var x = 20; // again declared  with same variable named 'x' with var keyword and assign
// a different value
console.log(x); // 20
```

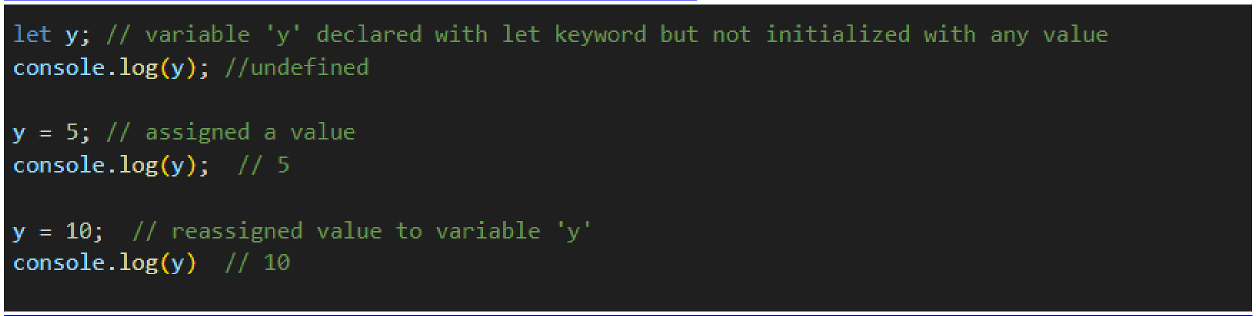
See above code, we can easily redeclared a variable and reassigned value to variable with var keyword.

Let's move towards declaring variable and assigning the value with let keyword:



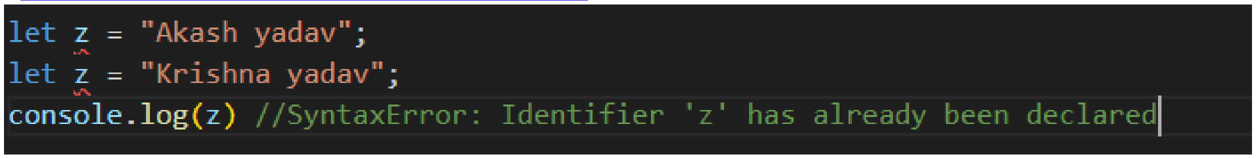
```
let y; // variable 'y' declared with let keyword but not initialized with any value
console.log(y); //undefined

y = 5; // assigned a value
console.log(y);  // 5
```

Re-assigning a value to variable with let keyword:



```
let y; // variable 'y' declared with let keyword but not initialized with any value
console.log(y); //undefined

y = 5; // assigned a value
console.log(y);  // 5

y = 10;  // reassigned value to variable 'y'
console.log(y)  // 10
```

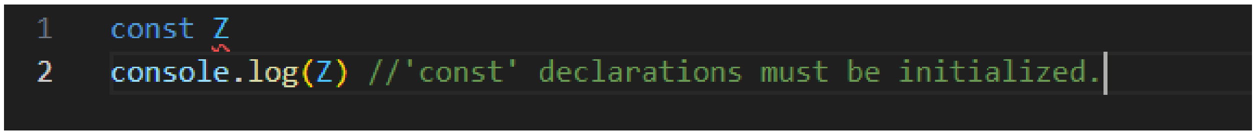
Re-declaring a variable with let keyword:



```
let z = "Akash yadav";
let z = "Krishna yadav";
console.log(z) //SyntaxError: Identifier 'z' has already been declared
```

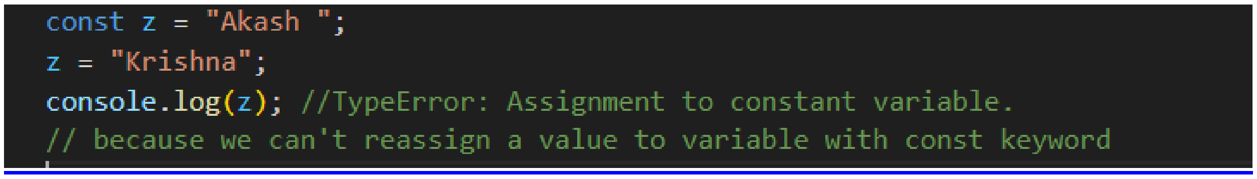
See above code,we can reassign a value to variable 'y' with let keyword but can not re-declare variable with let keyword.

Let's move towards declaring variable and assigning the value with const keyword:



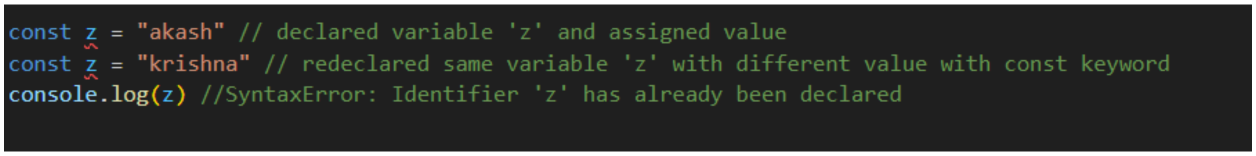
```
const Z
console.log(Z) //'const' declarations must be initialized.
```

Re-assigning a value to variable with const keyword:



```
const z = "Akash ";
z = "Krishna";
console.log(z); //TypeError: Assignment to constant variable.
// because we can't reassign a value to variable with const keyword
```

Re-declaring a variable with const keyword:



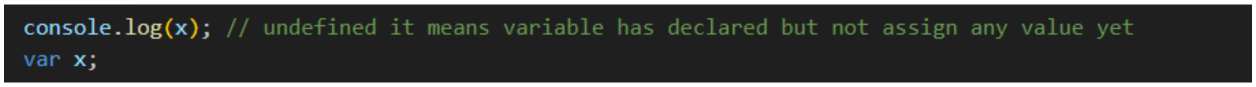
```
const z = "akash" // declared variable 'z' and assigned value
const z = "krishna" // redeclared same variable 'z' with different value with const keyword
console.log(z) //SyntaxError: Identifier 'z' has already been declared
```

See above code,we neither re-declare a variable nor re-assign the value with const keyword.

Summary:
Variables with var keyword : re-assign + re-declare.
Variables with let keyword : re-assign but not re-declare.
Variables with const keyword : neither reassign nor re-declare

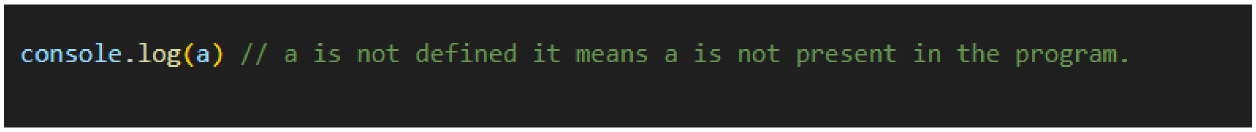
Ch-2

- **Difference between 'undefined' and 'not defined' ?**
  undefined:



```
console.log(x); // undefined it means variable has declared but not assign any value yet
var x;
```

Not defined:



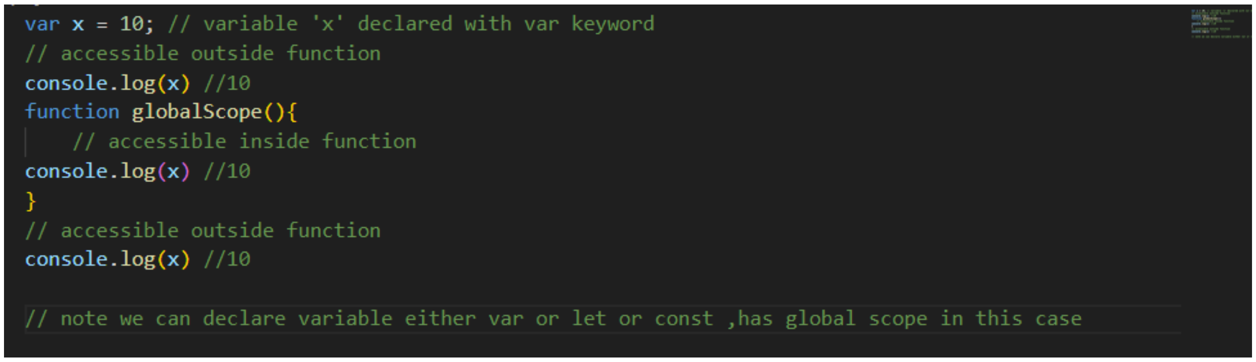
```
console.log(a) // a is not defined it means a is not present in the program.
```

Ch-3

Scope in javaScript: **Scope refers to where the variables are accessible.**

(1)Global Scope:

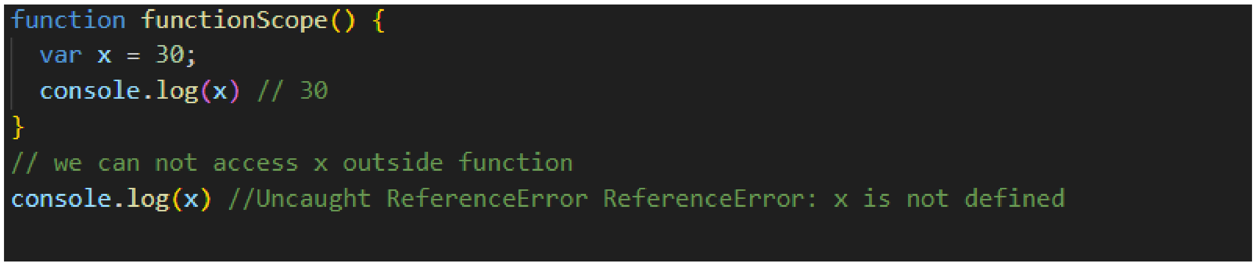
variables declared (either with var or let or const) outside of any function or block have global scope. And they are accessible from anywhere in the program after initialization.



```
var x = 10; // variable 'x' declared with var keyword
// accessible outside function
console.log(x) //10
function globalScope(){
    // accessible inside function
console.log(x) //10
}
// accessible outside function
console.log(x) //10

// note we can declare variable either var or let or const ,has global scope in this case
```

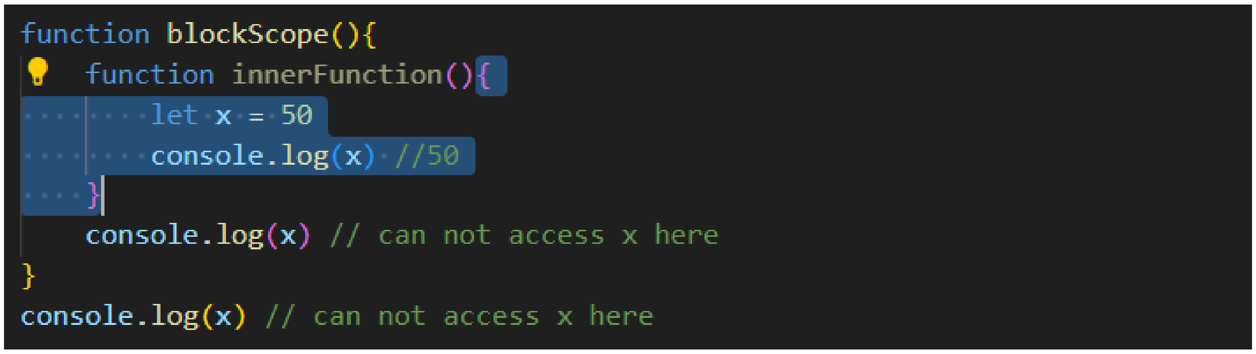
(2)Function Scope:

variables declared inside any function have function scope and are only accessible within that specific function.



```
function functionScope() {
  var x = 30;
  console.log(x) // 30
}
// we can not access x outside function
console.log(x) //Uncaught ReferenceError ReferenceError: x is not defined
```

(3)Block Scope: variables declared inside any block have block scope. And are only accessible within that specific block.



```
function blockScope(){
    function innerFunction(){
        let x = 50
        console.log(x) //50
    }
    console.log(x) // can not access x here
}
console.log(x) // can not access x here
```

*

Difference between var , let and const .
var - function scope , re-declare + re-assign and hoisted.
Let - block scope , reassign but not redeclare , hoisted.
const -block scope , neither redclare nor reassign , hoisted

Primitive and Non-Primitive data types in JavaScript:

1. Primitive Data Types:

- Number: Represents numeric values (integer and floating-point).
- String: Represents text or sequences of characters.
- Boolean: Represents `true` or `false`.
- Undefined: Variable declared but not assigned a value.
- Null: Represents an intentional absence of a value.
- Symbol: Unique and immutable value, often used as object keys.
- BigInt: Represents large integers beyond `Number.MAX_SAFE_INTEGER`.

Key Characteristics:

- Immutable: Cannot be altered after creation.
- Stored by Value: Directly holds data in memory.

2. Non-Primitive (Reference) Data Types:

- Object: Key-value pairs.
- Array: Ordered list of values.
- Function: A block of code that performs a task.
- Date: Represents dates and times.
- RegExp: Represents regular expressions.

Key Characteristics:

- Mutable: Can be changed after creation.
- Stored by Reference: Points to a memory location where data is stored.

Explain the difference between `==` and `===`.

- `==` compares values while `===` compares both values and types strictly.

What are the falsy values in JavaScript?

- The falsy values are: `false`, `0`, `""` (empty string), `null`, `undefined`, and `NaN`.

What is the difference between `null` and `undefined`?

- `undefined` means a variable has been declared but not assigned a value, while `null` is an assigned value representing no value or emptiness.

How do you convert a string to a number in JavaScript?

- Use `Number(string)`

What is the purpose of `JSON.stringify()` and `JSON.parse()`?

- `JSON.stringify()` converts a JavaScript object to a JSON string, while `JSON.parse()` parses a JSON string back into a JavaScript object.

How do you create a deep copy of an object?

- One common way to create a deep copy is using `JSON.parse(JSON.stringify(object))`

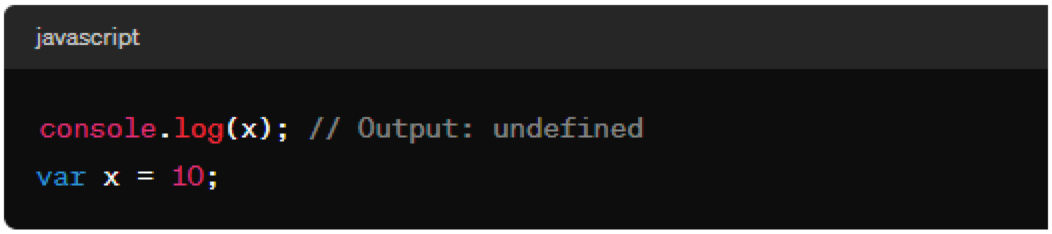
Ch-4 Hoisting

Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their containing scope during the compilation phase, regardless of where they are

declared in the code. This means that we can use a variable or call a function before it's declared in the code, and JavaScript will still recognize it.

Hoisting with `var`:

- **Variables declared with `var keyword` are hoisted to the top of their containing scope, but only the declaration is hoisted, not the initialization.**



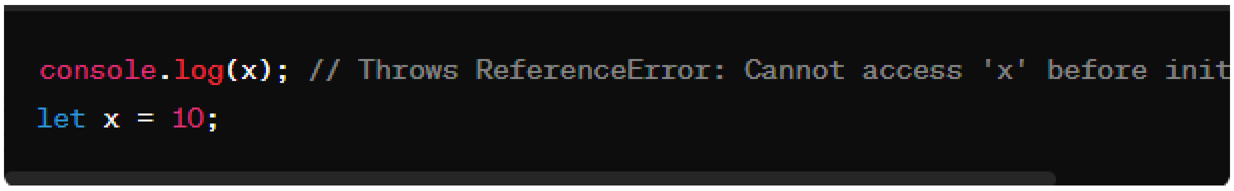
javascript

```javascript
console.log(x); // Output: undefined
var x = 10;
```

the value of `x` is `undefined` because only the declaration, not the initialization (`x = 10`), is hoisted.

Hoisting with `let` and `const`:

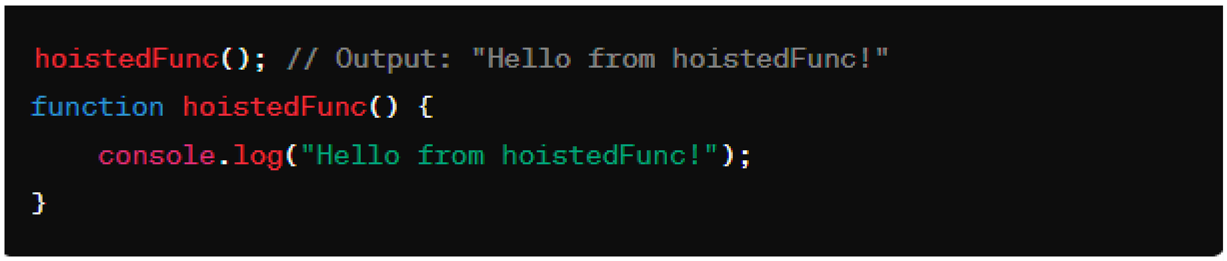
- **Variables declared with `let` and `const` are hoisted to the top of their containing block scope, but they are not initialized. They remain in the "temporal dead zone" until they are declared.**



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

Hoisting with Function Declarations:

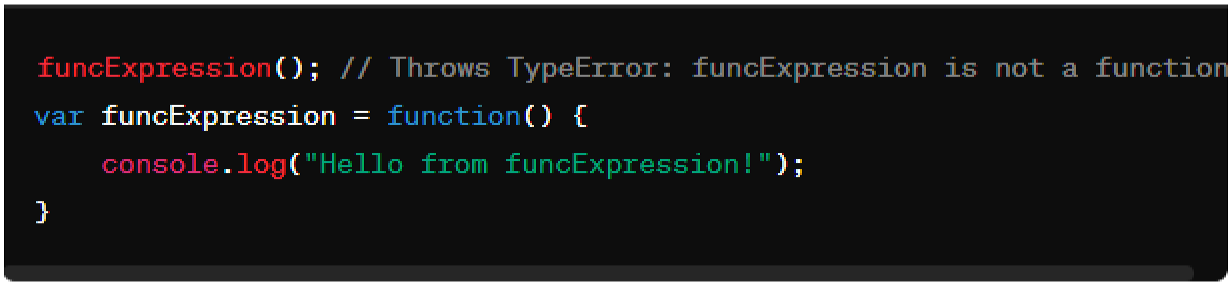
- **Function declarations are fully hoisted, including both the declaration and the function body.**



```javascript
hoistedFunc(); // Output: "Hello from hoistedFunc!"
function hoistedFunc() {
    console.log("Hello from hoistedFunc!");
}
```

Hoisting with Function Expressions:

- **Function expressions are not hoisted in the same way as function declarations. Only the variable declaration is hoisted, not the function assignment.**



```
funcExpression(); // Throws TypeError: funcExpression is not a functio
var funcExpression = function() {
    console.log("Hello from funcExpression!");
}
```

Imp:

- In modern JavaScript, prefer `const` for constants that shouldn't change and `let` for variables that may be reassigned. Avoid using `var` because it has function scope and is prone to hoisting issues, making it less predictable and error-prone. `const` and `let` introduce block scope, reducing bugs and improving code readability, aligning with contemporary coding practices.

```
for (var i = 1; i <= 5; i++) {
  setTimeout(function() {
    console.log(i);
  }, i * 1000);
}
```

output->

```
6
6
6
6
6
```

```
for (let i = 1; i <= 5; i++) {
  setTimeout(function() {
    console.log(i);
  }, i * 1000);
}
```

```
1
2
```

3
4
5

Ch-5

Function : **A function is a block of code . it is a reusable piece of code that is called(revoke) from anywhere in the program.**

What is need of function: **suppose we have to perform a specific task multiple times in our program, so the first way is to write that code again and again and the second way is to make a block of code and use it again and again.**

Syntax-

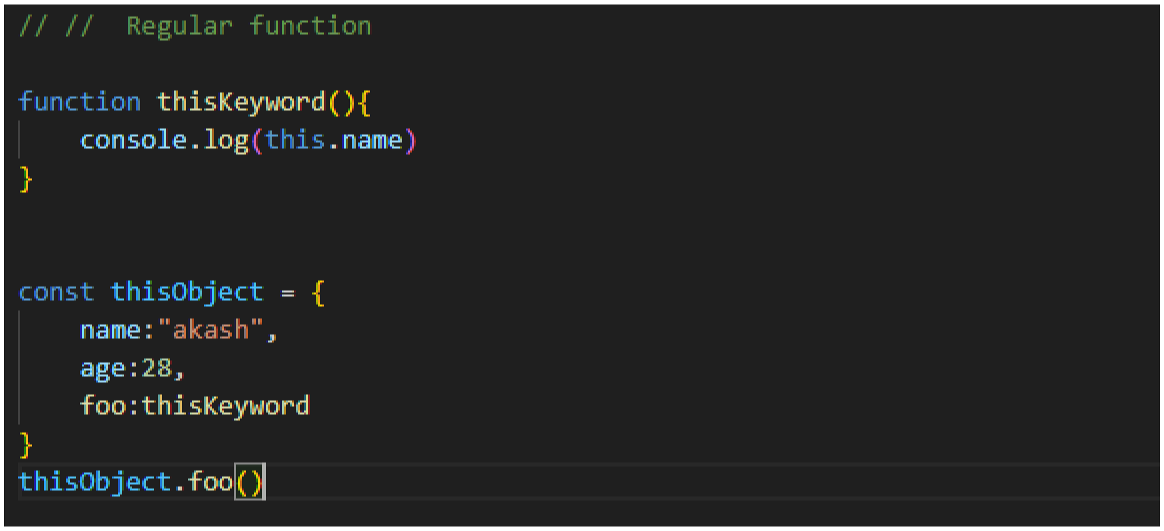
```
function  functionName(parameters){
  // statements
}
```

Arrow Function: **Arrow function makes function expressions easier to write. () => { }**
Syntax-

```
const  arrowFunctionName =(parameters)=>{

}
```

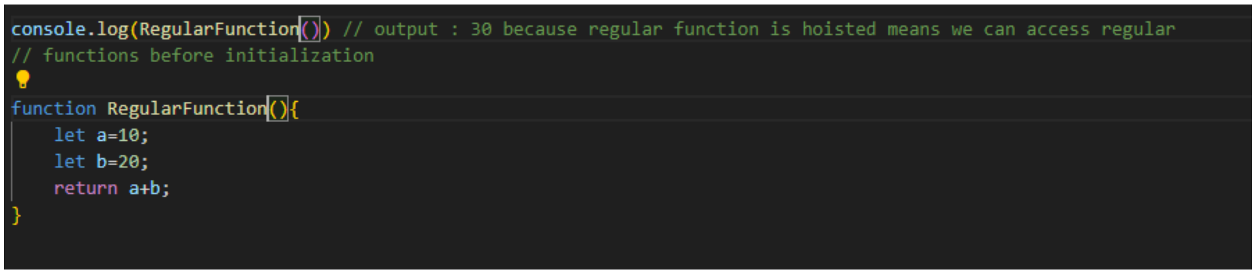
Difference between regular function and arrow function

Regular function in the context of "this" keyword-



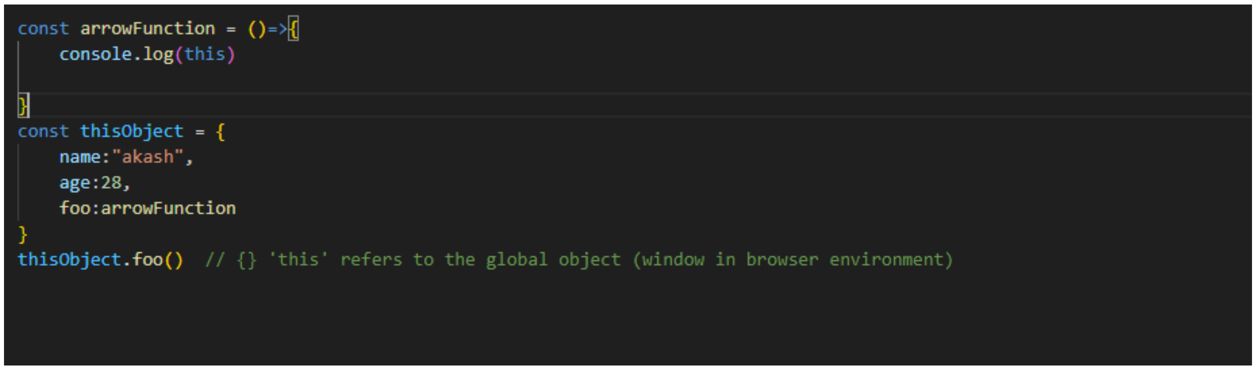
```
// //  Regular function

function thisKeyword(){
    console.log(this.name)
}


const thisObject = {
    name:"akash",
    age:28,
    foo:thisKeyword
}
thisObject.foo()
```

Hoisting behavior of Regular Function: regular function is hoisted



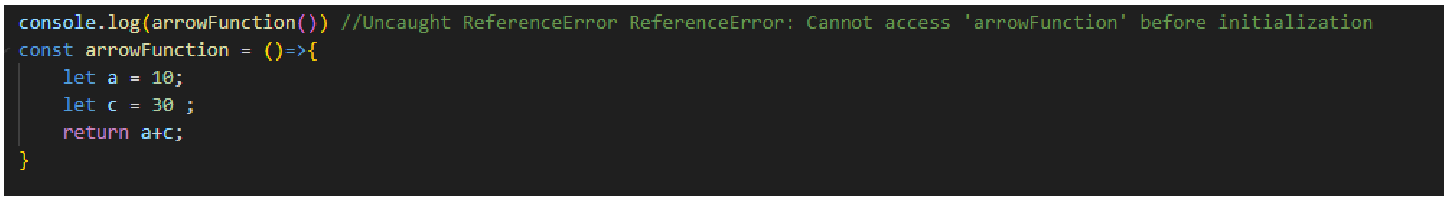
```
console.log(RegularFunction()) // output : 30 because regular function is hoisted means we can access regular
// functions before initialization

function RegularFunction(){
    let a=10;
    let b=20;
    return a+b;
}
```

Arrow function in the context of “this” keyword-

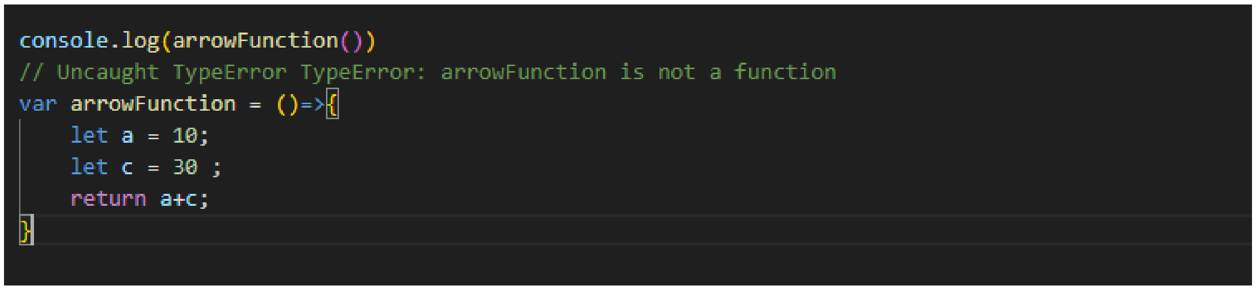


```
const arrowFunction = ()=>{
    console.log(this)

}
const thisObject = {
    name:"akash",
    age:28,
    foo:arrowFunction
}
thisObject.foo()  // {} 'this' refers to the global object (window in browser environment)
```

Hoisting behavior of Arrow Function: arrow function is not hoisted



```
console.log(arrowFunction()) //Uncaught ReferenceError ReferenceError: Cannot access 'arrowFunction' before initialization
const arrowFunction = ()=>{
    let a = 10;
    let c = 30 ;
    return a+c;
}
```



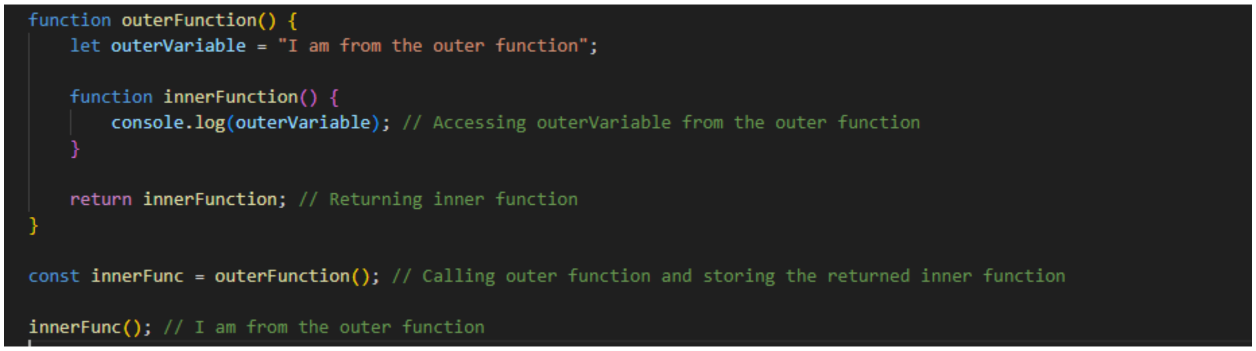
```
console.log(arrowFunction())
// Uncaught TypeError TypeError: arrowFunction is not a function
var arrowFunction = ()=>{
    let a = 10;
    let c = 30 ;
    return a+c;
}
```

Summary 👍

- **Regular functions have dynamic "this" binding, while arrow functions inherit "this" lexically.**
- **Regular functions are hoisted along with their function declarations, allowing them to be called before they are declared. Arrow functions are not hoisted and cannot be used before they are declared.**

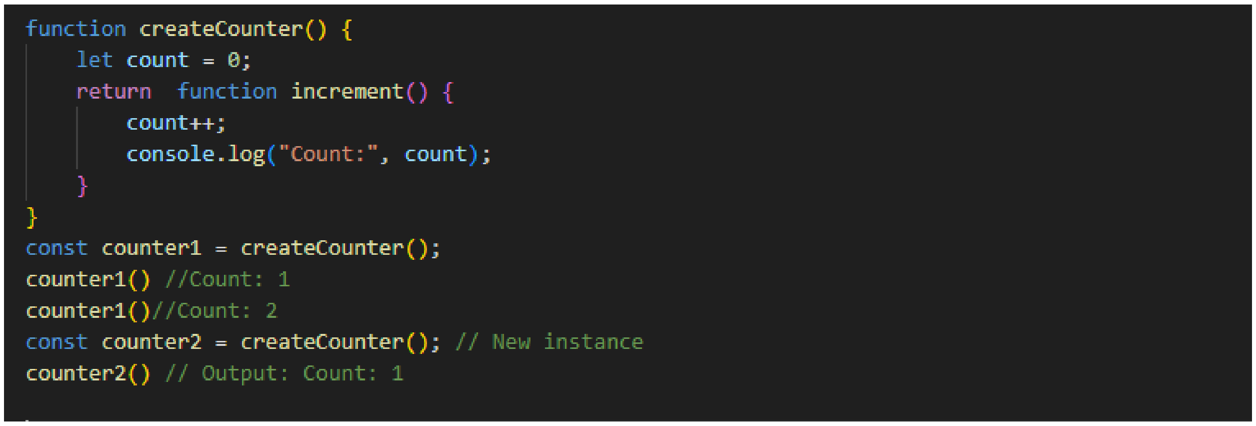
Ch-6 Closure

Closure - "A closure is a function that has access to variables from its outer function even after the outer function has finished execution."
Used for managing scope, encapsulating data, and creating powerful and reusable functions.
ex-2



```
function outerFunction() {
    let outerVariable = "I am from the outer function";

    function innerFunction() {
        console.log(outerVariable); // Accessing outerVariable from the outer function
    }

    return innerFunction; // Returning inner function
}

const innerFunc = outerFunction(); // Calling outer function and storing the returned inner function

innerFunc(); // I am from the outer function
```

ex-2



```
function createCounter() {
    let count = 0;
    return  function increment() {
        count++;
        console.log("Count:", count);
    }
}
const counter1 = createCounter();
counter1() //Count: 1
counter1()//Count: 2
const counter2 = createCounter(); // New instance
counter2() // Output: Count: 1
```

Chap-7 this,call,apply,bind

“this” keyword

“this” keyword refers to the current execution context or the object to which a function or method belongs.
The behavior of “this” can vary in different contexts.

1. Global Context:

- In the global context, outside of any function, `this` refers to the global object. In a browser environment, this is often the `window` object.

javascript Copy code

```javascript
console.log(this); // Refers to the global object (e.g., window in a br
```

2. Function Context:

- Inside a regular function (not an arrow function), `this` refers to the global object (in non-strict mode) or `undefined` (in strict mode).

javascript Copy code

```javascript
function myFunction() {
  console.log(this);
}

myFunction(); // Refers to the global object (window in a browser)
```

3. Method Context:

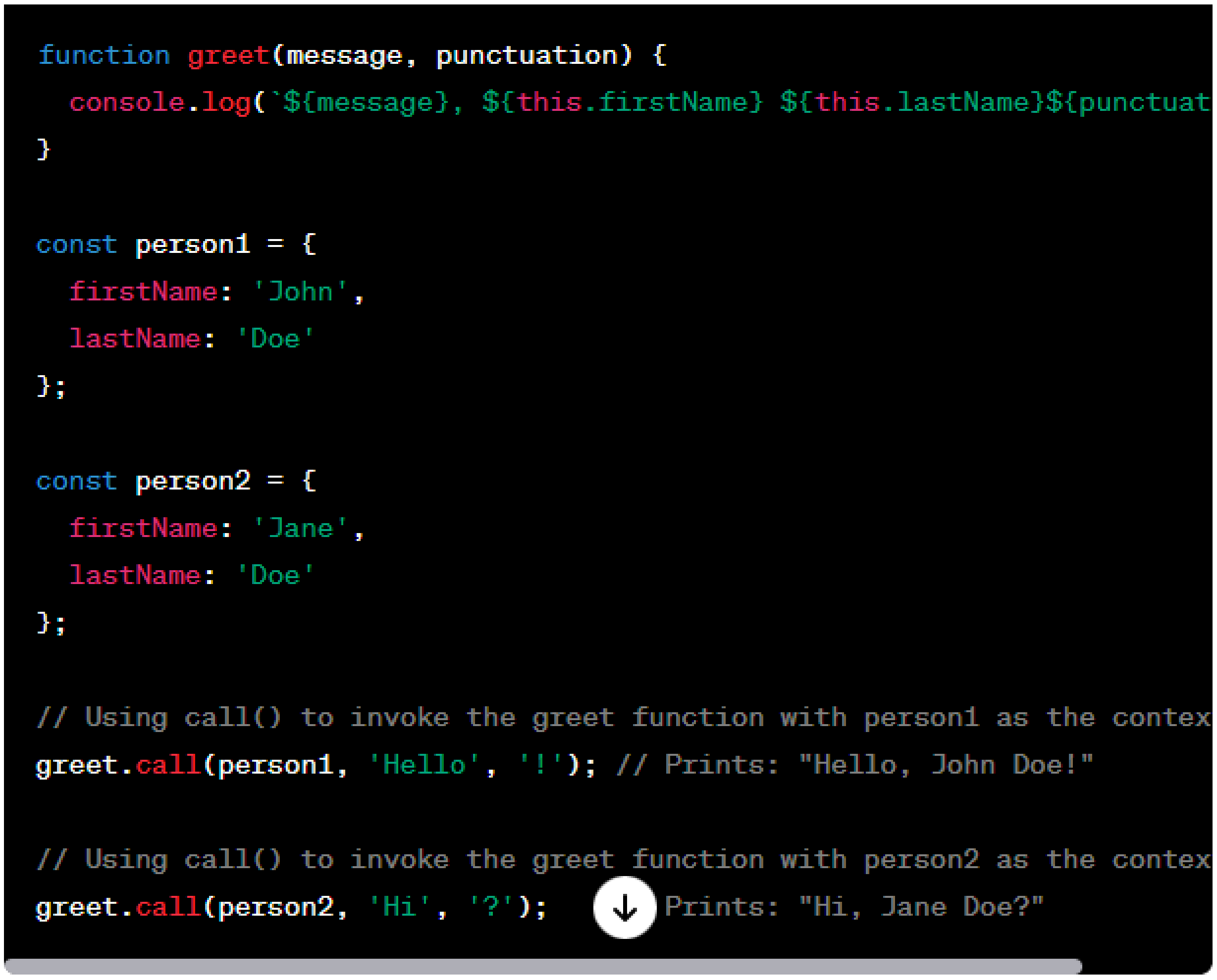
- Inside a method of an object, `this` refers to the object that owns the method.

javascript Copy code

```javascript
const myObject = {
  myMethod() {
    console.log(this);
  }
};

myObject.myMethod(); // Refers to myObject
```

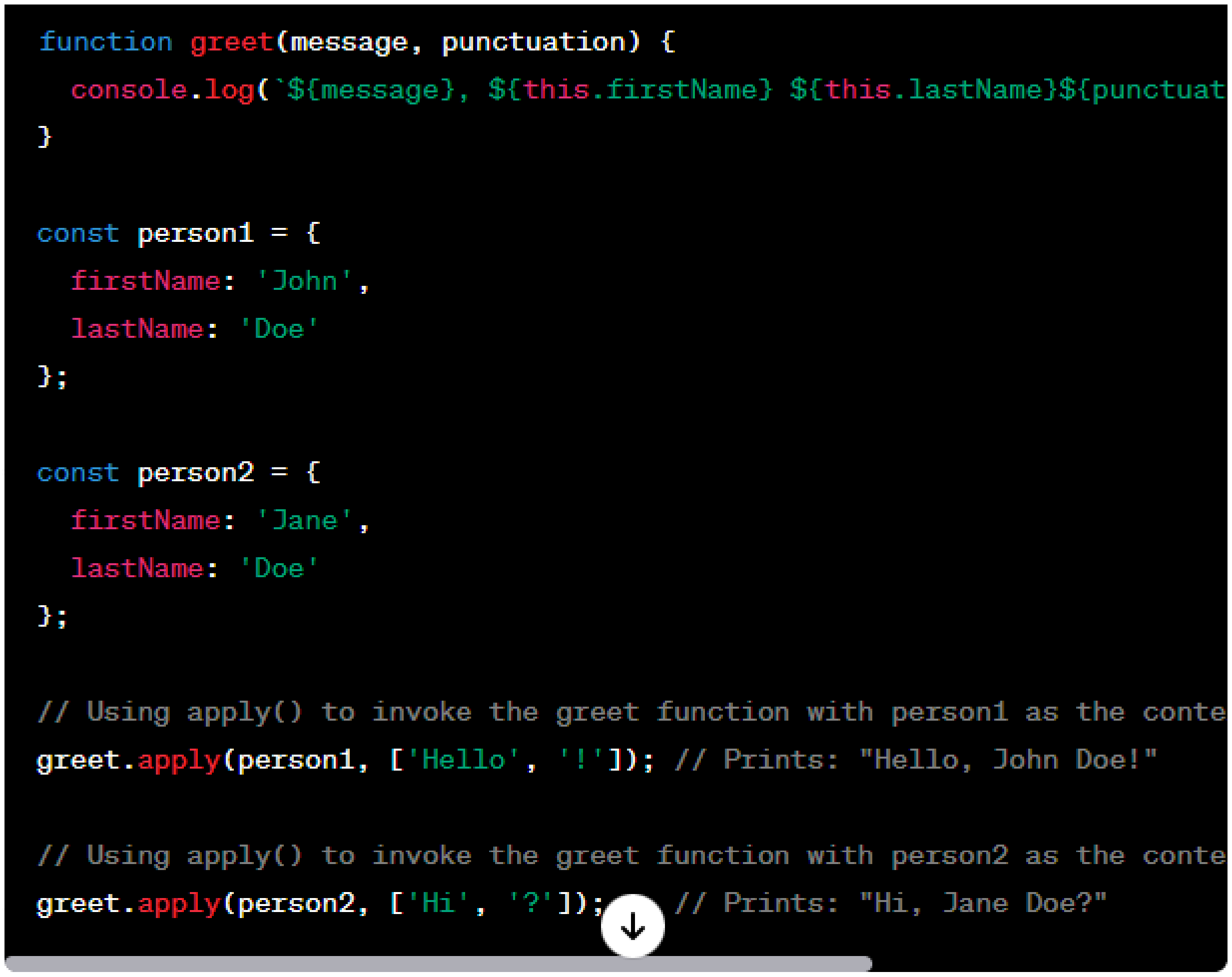
Call Method

call () method used to call a function with a specified “this” value and arguments provided as parameter.
 When you want to invoke a function immediately and specify the `this` value and individual arguments.



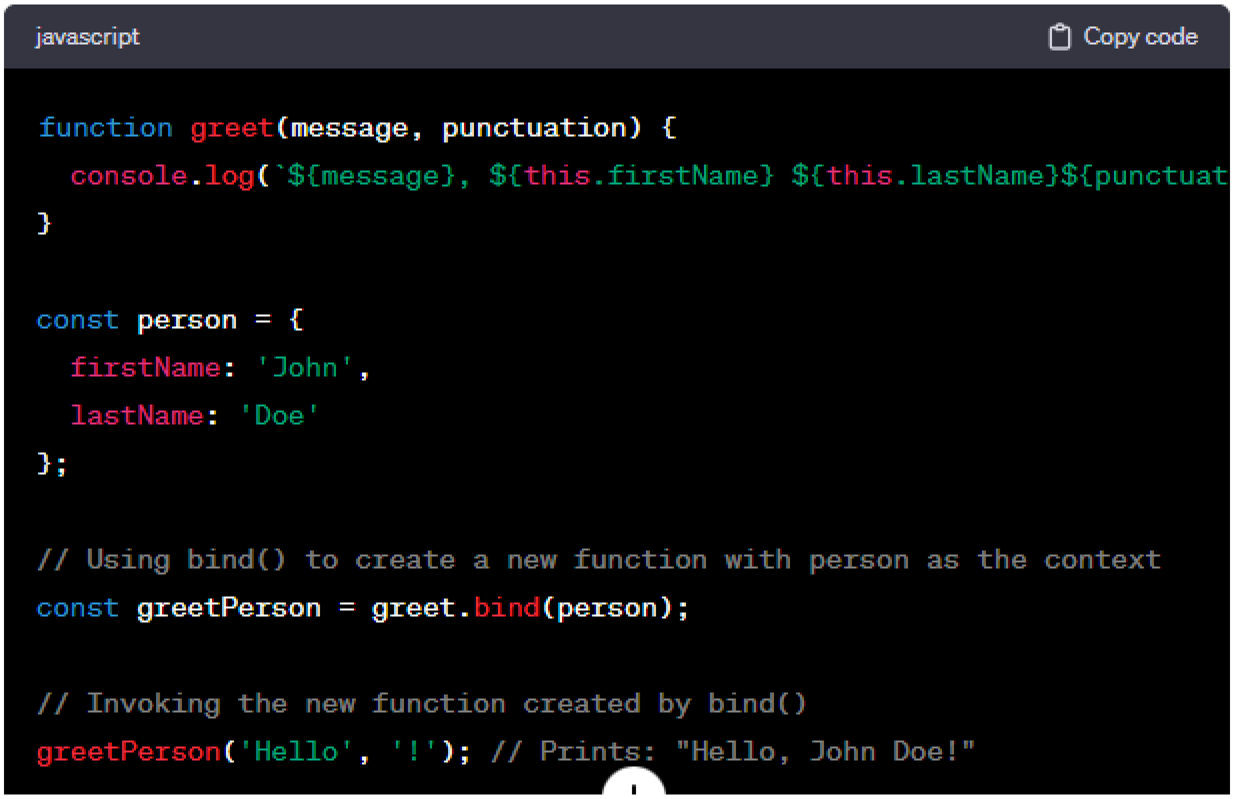
```
function greet(message, punctuation) {
  console.log(`${message}, ${this.firstName} ${this.lastName}${punctuat
}

const person1 = {
  firstName: 'John',
  lastName: 'Doe'
};

const person2 = {
  firstName: 'Jane',
  lastName: 'Doe'
};

// Using call() to invoke the greet function with person1 as the contex
greet.call(person1, 'Hello', '!'); // Prints: "Hello, John Doe!"

// Using call() to invoke the greet function with person2 as the contex
greet.call(person2, 'Hi', '?');      Prints: "Hi, Jane Doe?"
```

apply() method -   is used to call a function with a specified `this` value and an array of arguments.
 When you want to invoke a function immediately and specify the `this` value and arguments, but the arguments are stored in an array or an array-like object.



```
function greet(message, punctuation) {
  console.log(`${message}, ${this.firstName} ${this.lastName}${punctuat
}

const person1 = {
  firstName: 'John',
  lastName: 'Doe'
};

const person2 = {
  firstName: 'Jane',
  lastName: 'Doe'
};

// Using apply() to invoke the greet function with person1 as the conte
greet.apply(person1, ['Hello', '!']); // Prints: "Hello, John Doe!"

// Using apply() to invoke the greet function with person2 as the conte
greet.apply(person2, ['Hi', '?']);    // Prints: "Hi, Jane Doe?"
```

`bind()` does not immediately invoke the function but returns a new function with the bound `this` value.
 When you want to create a partially applied function



javascript Copy code

```javascript
function greet(message, punctuation) {
  console.log(`${message}, ${this.firstName} ${this.lastName}${punctuat
}

const person = {
  firstName: 'John',
  lastName: 'Doe'
};

// Using bind() to create a new function with person as the context
const greetPerson = greet.bind(person);

// Invoking the new function created by bind()
greetPerson('Hello', '!'); // Prints: "Hello, John Doe!"
```

Chap-8 Asynchronous programming

What is asynchronous programming in JavaScript?

- **Answer: Asynchronous programming allows tasks to execute independently, without blocking main thread**

Explain the event loop in JavaScript.

- **Answer: The event loop manages the execution of tasks, enabling non-blocking behavior by handling the callback queue and call stack.**

  Microtask - Promise:(primary)

  Macrotask - setTimeout:

How does JavaScript handle asynchronous operations?

- **Answer: JavaScript uses mechanisms like callbacks, promises, and async/await to handle asynchronous operations.**

What is a callback function?

- **Answer: A callback function is a function that is passed as an argument to another function**
- **Explain the concept of the callback queue.**
- **Answer: where callbacks are placed to await execution**

What is the purpose of the `setTimeout` function?

`setTimeout` **is a function in JavaScript that allows us to schedule the execution of a function or the evaluation of a code snippet after a specified delay, measured in milliseconds.**

- `callback`**: The function to be executed after the specified delay.**
- `delay`**: The time (in milliseconds) the system should wait before executing the code.**

Good Things:

Asynchronous Execution:

- `setTimeout` **enables asynchronous execution, allowing certain code to run after a specific delay without blocking the main thread.**

UI Updates:

- **It's commonly used for UI-related tasks, such as showing alerts, modals, or updating elements with a delay.**

Animation Sequencing:

- `setTimeout` **is useful for creating animation sequences by scheduling updates to occur at specific intervals.**

Delaying Code Execution:

- **Useful when you need to delay the execution of a particular function or block of code.**

KAMIYA

Callback Hell:

- **Overuse of** `setTimeout` **can lead to callback hell, making the code harder to read and maintain. Asynchronous operations might be better handled with Promises or async/await.**

Inaccurate Timing:

- **The actual delay might not be precise due to the single-threaded nature of JavaScript and other tasks in the event loop.**

Scope Issues:

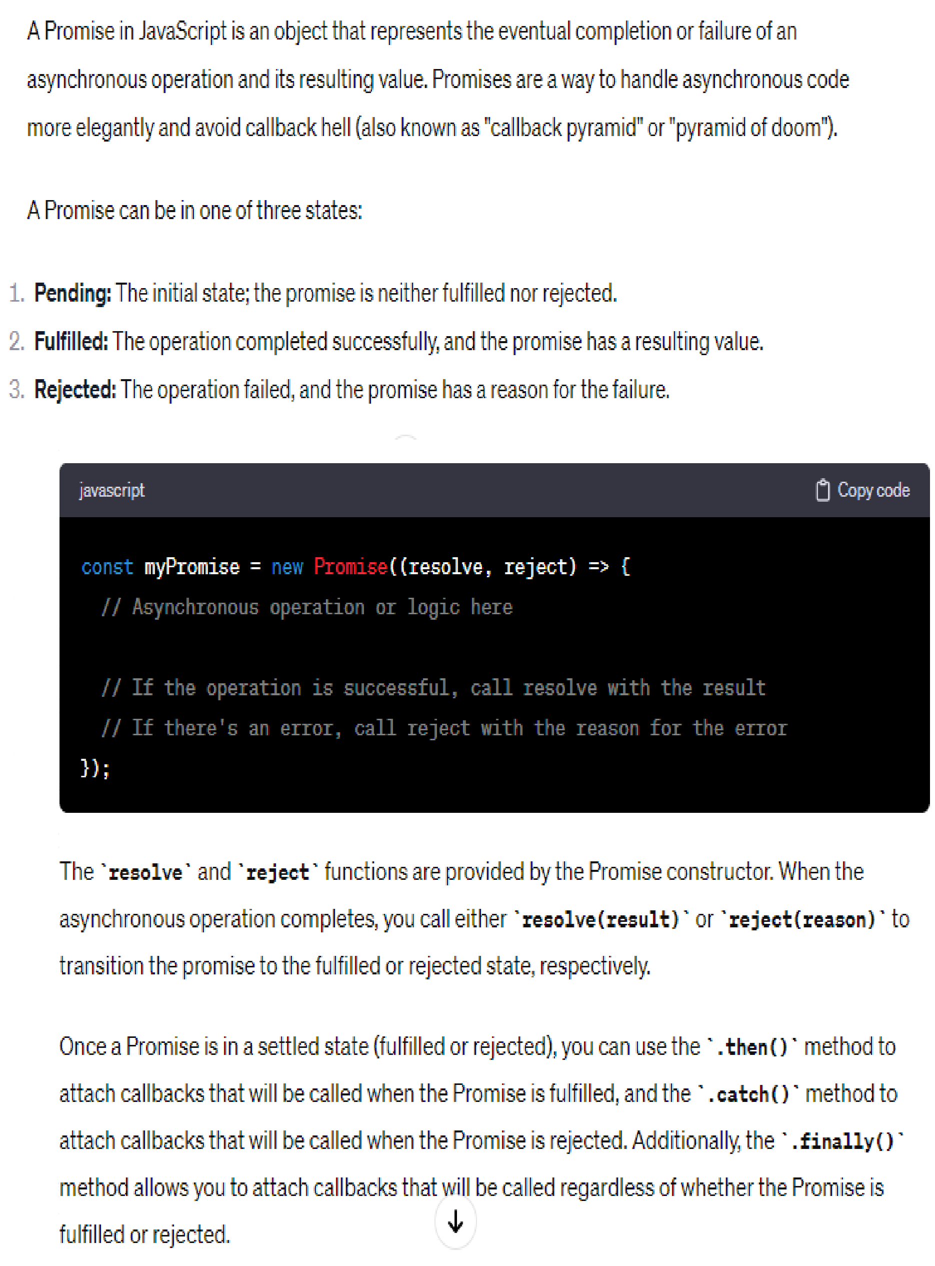
- **Be mindful of the scope of variables inside the callback function, especially if it's a separate function.**

Memory Leaks:

- **Not clearing or handling the returned timeout ID can lead to memory leaks if the associated code is no longer needed.**

    ▪

- **Explain the role of the `Promise` in asynchronous programming.**



A Promise in JavaScript is an object that represents the eventual completion or failure of an asynchronous operation and its resulting value. Promises are a way to handle asynchronous code more elegantly and avoid callback hell (also known as "callback pyramid" or "pyramid of doom").

A Promise can be in one of three states:

1. **Pending:** The initial state; the promise is neither fulfilled nor rejected.
2. **Fulfilled:** The operation completed successfully, and the promise has a resulting value.
3. **Rejected:** The operation failed, and the promise has a reason for the failure.

```javascript
const myPromise = new Promise((resolve, reject) => {
  // Asynchronous operation or logic here

  // If the operation is successful, call resolve with the result
  // If there's an error, call reject with the reason for the error
});
```

The `resolve` and `reject` functions are provided by the Promise constructor. When the asynchronous operation completes, you call either `resolve(result)` or `reject(reason)` to transition the promise to the fulfilled or rejected state, respectively.

Once a Promise is in a settled state (fulfilled or rejected), you can use the `.then()` method to attach callbacks that will be called when the Promise is fulfilled, and the `.catch()` method to attach callbacks that will be called when the Promise is rejected. Additionally, the `.finally()` method allows you to attach callbacks that will be called regardless of whether the Promise is fulfilled or rejected.

```
const fetchData = () => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const success = true; // Simulating a successful request
      if (success) {
        resolve('Data fetched successfully');
      } else {
        reject('Error fetching data');
      }
    }, 2000);
  });
};

// Using the Promise
fetchData()
  .then(result => {
    console.log(result); // Data fetched successfully
  })
  .catch(error => {
    console.error(error); // Error fetching data
  });
```

How does the `async/await` syntax improve asynchronous code readability?

- `async/await` used for handling asynchronous operations in a more synchronous-looking manner. It was introduced in ECMAScript 2017 (ES8) and is built on top of promises.
- The `async` keyword is used to define an asynchronous function,

`await` keyword is used to pause its execution until a Promise is settled.

Benefits:

Readability:

- `async/await` syntax makes asynchronous code look more like synchronous code, improving readability and reducing the "callback hell."

Error Handling:

- Easier error handling using `try...catch` blocks, which resembles synchronous error handling.

Sequential Code:

- Allows writing asynchronous code in a more sequential and natural order, making it easier to understand the flow.

Promise-Based:

- It's built on top of promises, providing compatibility with existing promise-based code.

Potential Pitfalls:

Uncaught Promise Rejections:

- Without proper error handling, uncaught promise rejections can occur. Always include a `catch` block or use `try...catch` to handle errors.

Async Function Returns a Promise:

- An `async` function always returns a promise, which might affect how the function is used or create unexpected behavior if not handled correctly.

Sequential vs. Parallel Execution:

- `await` waits for the resolution of the current promise before moving to the next line. If multiple promises can be executed concurrently, this sequential nature might not be optimal.

Blocking Nature:

- While `async/await` can make code appear synchronous, it doesn't change the fact that JavaScript is single-threaded. Long-running synchronous tasks could still block the event loop.

Best Practices:

Proper Error Handling:

- Always handle errors by using `try...catch` or `.catch()` to prevent unhandled promise rejections.

Avoid Blocking:

- Be mindful of potentially blocking operations. If a function is time-consuming, consider breaking it into smaller asynchronous tasks or using other concurrency techniques.

Understand Promise Chain:

- Be aware that `await` only waits for the completion of the current promise in the chain. If there are multiple asynchronous operations that can be parallelized, consider using `Promise.all` or other concurrency patterns.
-

What is the purpose of the `then` method in promises?

- Answer: The `then` method is used to handle the fulfillment of a promise and takes two optional callbacks for success and failure.

Explain the difference between micro tasks and macro tasks.

- Answer: Microtasks (e.g., promises) have higher priority than macrotask (e.g., setTimeout), and they are executed before the next macrotask.

How do you create a promise in JavaScript?

- Answer: Use the `Promise` constructor, providing an executor function with `resolve` and `reject` parameters.

What is the purpose of the `catch` method in promises?

- Answer: The `catch` method is used to handle promise rejections and is an alternative to placing a rejection callback in the `then` method.

Explain the difference between `Promise.all` and `Promise.race`.

- **Answer: `Promise.all` resolves when all promises in an iterable resolve; `Promise.race` resolves or rejects as soon as one promise in an iterable resolves or rejects.**

How do you handle errors in asynchronous code using `async/await`?

- **Answer: Use a try/catch block around the `await` expression to handle errors in `async/await` functions.**

What is the purpose of the `finally` block in promises?

- **Answer: The `finally` block contains code that is executed regardless of whether the promise is fulfilled or rejected.**

How does the `fetch` API work, and how does it handle promises?

- **Answer: The `fetch` API is used for making network requests. It returns a promise that resolves to the `Response` object representing the result of the request.**

What is the purpose of the `yield` keyword in JavaScript generators?

- **Answer: The `yield` keyword is used in generators to pause and produce a value, allowing the generator to produce a sequence of values over time.**

How can you avoid callback hell in JavaScript?

- **Answer: Avoid callback hell by using named functions, modularization, and adopting promises or `async/await`.**

How does JavaScript handle long-running tasks without blocking the main thread?

- **Answer: JavaScript handles long-running tasks by using asynchronous operations such as promises, web workers, and events to prevent blocking the main thread**
-
-

What are Web Workers, and how do they contribute to asynchronous programming?

- **Answer: Web Workers are JavaScript scripts that run in the background, separate from the main thread. They enable parallel execution, contributing to asynchronous programming by performing tasks without blocking the main thread.**

How do you handle multiple asynchronous operations concurrently in JavaScript?

- **Answer: Use `Promise.all` to handle multiple asynchronous operations concurrently, waiting for all promises to settle.**

How does the `async` attribute in the `<script>` tag affect script execution?

- Answer: The `async` attribute allows scripts to be executed asynchronously, preventing them from blocking the HTML parsing and rendering.

What is the role of the `setImmediate` function in Node.js?

- Answer: The `setImmediate` function in Node.js is used to execute a callback after the current event loop cycle, allowing I/O events to be processed first.

Explain the concept of the JavaScript task queue.

- Answer: The task queue is where tasks (macrotasks) are queued for execution. Tasks in the task queue are processed after the call stack is empty and microtasks are executed.

What is the purpose of the `localStorage` API in the context of asynchronous programming?

- Answer: The `localStorage` API is synchronous and doesn't directly contribute to asynchronous programming. It is often used for simple, synchronous storage and retrieval of key-value pairs.

How do you handle memory leaks in asynchronous JavaScript code?

- Answer: To handle memory leaks, make sure to clean up event listeners, close connections, and release resources when they are no longer needed. Avoid circular references and be mindful of long-running operations.

Explain the difference between the `resolve` function and the `reject` function in a promise.

- Answer: The `resolve` function is used to fulfill a promise with a resolved value, while the `reject` function is used to reject a promise with a specified reason.

How does the `async` keyword affect the return value of a function in JavaScript?

- Answer: The `async` keyword makes a function return a promise. If the function returns a value, the promise will be resolved with that value. If the function throws an exception, the promise will be rejected with the thrown value.
-

Difference between fetch and axios 👍

Automatic JSON Handling:

- Axios: Automatically transforms JSON data (both in requests and responses), meaning no need to call `.json()`.
- Fetch: Requires manually calling `.json()` on the response object to parse JSON data.

Error Handling:

- **Axios: Treats HTTP status codes other than 2xx as errors and handles them in the `.catch()` block.**
- **Fetch: Only rejects a promise for network-level errors (like connection issues), not for HTTP status errors (like 404 or 500).**

Request Abortion (Cancellation):

- **Axios: Supports request cancellation using `CancelToken`.**
- **Fetch: Has support for request aborting using `AbortController`, but requires more setup compared to Axios.**

Request and Response Interceptors:

- **Axios: Provides built-in request and response interceptors for modifying requests or handling responses globally.**
- **Fetch: Does not have interceptors natively; requires custom implementation.**

Chap-8 API, HTTPS

What is an API?

- **Answer: API stands for Application Programming Interface. It is a set of protocols, routines, and tools for building software and applications. APIs define how different software components should interact, making it easier for developers to integrate various services and functionalities into their applications.**

Explain the difference between SOAP and REST.

- **Answer: SOAP (Simple Object Access Protocol) and REST (Representational State Transfer) are both web service communication protocols. SOAP is a protocol with strict standards for message format and typically uses XML, while REST is an architectural style that relies on** stateless communication and can use various data formats, including JSON. REST is generally considered more lightweight and easier to implement.

What is the purpose of HTTP methods such as GET, POST, PUT, and DELETE in the context of APIs?

- Answer: HTTP methods, also known as HTTP verbs, are fundamental to the Hypertext Transfer Protocol (HTTP) and are used to indicate the desired action to be performed on the resource identified by the request URL. In the context of APIs (Application Programming Interfaces), these methods are used to interact with resources on a server. Here's an explanation of the commonly used HTTP methods:

GET: The GET method is used to retrieve data from the server. It requests a representation of the specified resource. GET requests should only retrieve data and should not have any other effect on the server. They are idempotent, meaning that making the same request multiple times should have the same effect as making it once.
Example: When you visit a webpage in your browser, your browser sends a GET request to the server to retrieve the HTML, CSS, and JavaScript files needed to render the webpage.
POST: The POST method is used to submit data to be processed to the server. It creates a new resource on the server or performs some action based on the provided data. POST requests are not idempotent, meaning that making the same request multiple times may result in different outcomes.
Example: When you submit a form on a website, such as a login form or a contact form, the form data is typically sent to the server using a POST request.
PUT: The PUT method is used to update an existing resource or to create a new resource if it does not already exist. It replaces the entire representation of the resource with the request payload. PUT requests are idempotent, meaning that making the same request multiple times should have the same effect as making it once.
Example: When you update your profile information on a social media website, such as changing your bio or profile picture, the updated data is typically sent to the server using a PUT request.
DELETE: The DELETE method is used to delete the specified resource from the server. It removes the resource identified by the request URL. DELETE requests are idempotent, meaning that making the same request multiple times should have the same effect as making it once.
Example: When you delete a post or comment on a social media website, the

request to delete the content is typically sent to the server using a DELETE request.

-

What is the significance of status codes in HTTP responses?

- Answer: HTTP status codes indicate the result of an HTTP request. Common codes include 200 (OK), 201 (Created), 400 (Bad Request), 401 (Unauthorized), 404 (Not Found), and 500 (Internal Server Error). Status codes provide information about the success or failure of a request.

What is the role of headers in an HTTP request and response?

- Answer: HTTP headers carry additional information about the request or response. They can include metadata, authentication credentials, content type, caching directives, and more. Headers provide crucial details for the proper processing of HTTP messages.

Describe the process of handling authentication in an API.

- Answer: Authentication in APIs verifies the identity of the client. Common methods include API keys, OAuth, and JSON Web Tokens (JWT). Clients include authentication information in the request, and servers validate it before processing the request.

What is CORS, and why is it important in the context of web APIs?

- Answer: CORS (Cross-Origin Resource Sharing) is a security feature implemented by web browsers. It controls which web domains are allowed to access resources on a given domain. It is essential to prevent unauthorized access to resources and protect against cross-site request forgery.

What are webhooks, and how do they differ from traditional APIs?

- Answer: Webhooks are a way for one system to provide real-time information to another by sending HTTP requests when events occur. Unlike traditional APIs where the client initiates requests, webhooks enable servers to push data to clients when specific events occur.

How would you secure sensitive information transmitted via an API?

- Answer: Sensitive information should be transmitted over HTTPS to encrypt data in transit. Additionally, using authentication mechanisms like OAuth or JWTs can add an extra layer of security. Encryption and proper data validation are crucial for protecting sensitive information

Chap-8 ES6 features

Top ECMAScript – ES6 Features Every Javascript Developer Should Know

ES6, also known as ECMAScript 2015, introduced a multitude of new features and syntax enhancements to JavaScript, revolutionizing the way developers write code. Here's a comprehensive list of ES6 features:

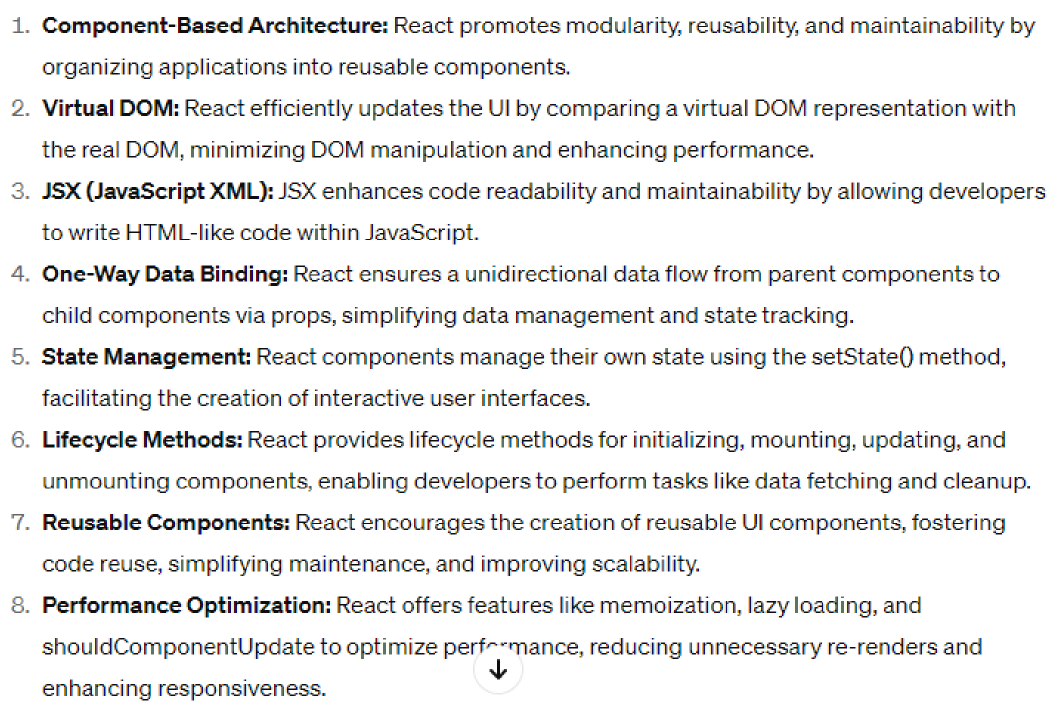
1- Arrow Functions: Provides a concise syntax for writing function expressions, with implicit return and lexical `this` binding.
2- Template Literals: Allow for string interpolation and multiline strings using backticks (`).
3- Destructuring Assignment: Provides a convenient syntax to extract values from arrays or objects into variables.
4- Default Parameters: Allows function parameters to have default values if no value or undefined is passed.
5- Rest Parameters: Allows functions to accept an indefinite number of arguments as an array.
6- Spread Syntax: Allows an iterable (e.g., array) to be expanded into individual elements.
7- Promises: Introduces a built-in mechanism for handling asynchronous operations, improving code readability and maintainability.
8- Symbol: Introduces a new primitive data type for creating unique identifiers, useful for defining object properties.
9- Iterators and Iterables: Provides a way to iterate over data structures like arrays and objects using the `for...of` loop.

In conclusion, the features introduced in ECMAScript 2015 (ES6) represent a significant leap forward in JavaScript development

What is React ?

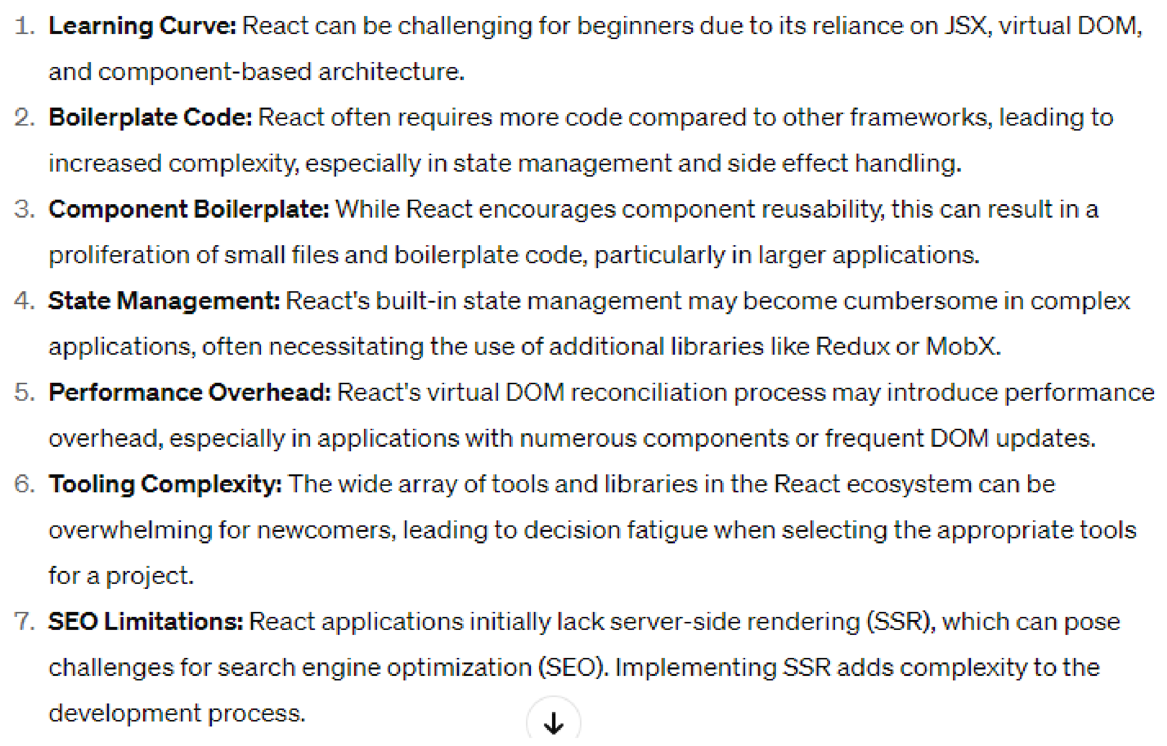
React is a JavaScript library for building user interfaces, developed and maintained by Facebook. It is widely used for creating interactive and dynamic web applications. React follows a component-based architecture, allowing developers to create reusable UI components that manage their own state and can be composed together to build complex user interfaces.

Key features of React:



1. **Component-Based Architecture:** React promotes modularity, reusability, and maintainability by organizing applications into reusable components.
2. **Virtual DOM:** React efficiently updates the UI by comparing a virtual DOM representation with the real DOM, minimizing DOM manipulation and enhancing performance.
3. **JSX (JavaScript XML):** JSX enhances code readability and maintainability by allowing developers to write HTML-like code within JavaScript.
4. **One-Way Data Binding:** React ensures a unidirectional data flow from parent components to child components via props, simplifying data management and state tracking.
5. **State Management:** React components manage their own state using the setState() method, facilitating the creation of interactive user interfaces.
6. **Lifecycle Methods:** React provides lifecycle methods for initializing, mounting, updating, and unmounting components, enabling developers to perform tasks like data fetching and cleanup.
7. **Reusable Components:** React encourages the creation of reusable UI components, fostering code reuse, simplifying maintenance, and improving scalability.
8. **Performance Optimization:** React offers features like memoization, lazy loading, and shouldComponentUpdate to optimize performance, reducing unnecessary re-renders and enhancing responsiveness.

Limitations or drawbacks of React:



1. **Learning Curve:** React can be challenging for beginners due to its reliance on JSX, virtual DOM, and component-based architecture.
2. **Boilerplate Code:** React often requires more code compared to other frameworks, leading to increased complexity, especially in state management and side effect handling.
3. **Component Boilerplate:** While React encourages component reusability, this can result in a proliferation of small files and boilerplate code, particularly in larger applications.
4. **State Management:** React's built-in state management may become cumbersome in complex applications, often necessitating the use of additional libraries like Redux or MobX.
5. **Performance Overhead:** React's virtual DOM reconciliation process may introduce performance overhead, especially in applications with numerous components or frequent DOM updates.
6. **Tooling Complexity:** The wide array of tools and libraries in the React ecosystem can be overwhelming for newcomers, leading to decision fatigue when selecting the appropriate tools for a project.
7. **SEO Limitations:** React applications initially lack server-side rendering (SSR), which can pose challenges for search engine optimization (SEO). Implementing SSR adds complexity to the development process.

How Virtual DOM works:

React me, Virtual DOM (Document Object Model) ek lightweight representation hai actual DOM tree ka. Jab aap apne React components me changes karte hain, React pehle Virtual DOM ko update karta hai, actual DOM ke saath seedha interact karne ke bajaye. Yeh kaise kaam karta hai:

Shuruaati Rendering: Jab aap apna React application shuruaat me render karte hain, React ek Virtual DOM tree banata hai jo actual DOM ki structure ko reflect karta hai. Har component aapke React application me ek node ke taur par Virtual DOM me hota hai.
Virtual DOM ko Update karna: Jab kisi component ka state ya props change hota hai, React us component ko re-render karta hai aur corresponding nodes ko Virtual DOM me update karta hai.

Diffing: Virtual DOM ko update karne ke baad, React ek process ko perform karta hai jo "diffing" ya "reconciliation" ke naam se jaana jaata hai. React purane aur naye Virtual DOM ke beech me farq ko pehchanta hai.
DOM Operations ko minimize karna: React purane aur naye Virtual DOM ke beech ke farq ko pehchane ke baad, yeh determine karta hai ki actual DOM ko update karne ke liye sabse efficient tareeka kya hoga. Har ek change ke liye seedha DOM manipulation na karke, React calculate karta hai ki kya minimum DOM operations chahiye honge actual DOM ko updated Virtual DOM ke saath sync karne ke liye.
Updates ko Batch karna: React ek sath kayi DOM updates ko ek single update operation me batch karta hai, performance ko optimize karke DOM manipulations aur reflows ki sankhya ko kam kar deta hai.
Actual DOM ko Update karna: Ant mein, React calculated DOM operations ko apply karta hai taaki actual DOM ko update kiya ja sake, ensuring ki user interface changes ko reflect karta hai.

Virtual DOM ka upyog karke, React direct DOM ko manipulate kar ke performance overhead ko kam karta hai aur user interface ko update karne ka ek adhik efficient tareeka pradaan karta hai. Yah approach React ko web applications me fast aur responsive user experience dene ke liye prasiddh banata hai.

Component:

Component is a reusable and modular building block for building user interfaces. Components are the fundamental units of React applications, and they encapsulate both the structure and behavior of a part of the UI.

There are two main types of components in React:

Functional Components:

- Functional components are defined using JavaScript functions.
- They are also known as stateless components or presentational components.

- Functional components receive props (properties) as input and return JSX (JavaScript XML) to describe what should be rendered on the screen.
- With the introduction of React hooks (like useState, useEffect, etc.), functional components can now also manage state and perform side effects, making them more powerful.

Class Components:

- Class components are defined using ES6 classes that extend the React.Component class.
- They are also known as stateful components or container components.
- Class components have a render method that returns JSX, describing the UI.
- Class components can also have their own state, managed using this.state, and lifecycle methods like componentDidMount, componentDidUpdate, etc.

Higher-Order Component (HOC)

A Higher-Order Component (HOC) is a pattern used in React.js for reusing component logic. It's a function that takes a component and returns a new component with extended functionality. HOCs are commonly used for cross-cutting concerns like logging, authentication, authorization, or data fetching.

Here are some reasons why and when we might want to use Higher-Order Components:

Code Reusability: HOCs promote code reuse by encapsulating common logic into a reusable function. Instead of duplicating the same logic across multiple components, we can create an HOC and apply it to any component that needs that logic.
Cross-Cutting Concerns: HOCs are useful for implementing cross-cutting concerns such as authentication, logging, or data fetching. For example, we can create an authentication HOC that restricts access to certain components unless the user is logged in.

Separation of Concerns: HOCs help in separating concerns by isolating specific functionality into separate components. This makes our codebase easier to understand, maintain, and test.
Composition: HOCs enable composability by allowing us to compose multiple higher-order components together to create complex behavior. This makes it easy to mix and match different functionalities in our components.
Decorator Pattern: HOCs follow the decorator pattern, which is a design pattern that allows behavior to be added to individual objects dynamically. This pattern promotes code decoupling and extensibility.

However, it's important to use HOCs judiciously as they can introduce complexity and make the component hierarchy less transparent. Overuse of HOCs can also lead to a higher level of indirection and make the code harder to reason about.

Component's lifecycle

The lifecycle methods categorized by mounting, updating, and unmounting:

Mounting:

`componentWillMount()` (deprecated)
`componentDidMount()`

Updating:

`shouldComponentUpdate(nextProps, nextState)`
`componentWillUpdate(nextProps, nextState)` (deprecated)
`componentDidUpdate(prevProps, prevState)`

Unmounting:

10. `componentWillUnmount()`

These methods provide hooks into different stages of a component's lifecycle, allowing us to perform actions such as initialization, updating based on changes in props or state, and cleanup before a component is removed from the DOM.

In functional components, we can achieve similar functionality using React Hooks. React Hooks were introduced in React 16.8 to allow functional components to use state and lifecycle methods. Here's how we can achieve the equivalent of lifecycle methods in functional components:

Mounting:

> `useEffect(() => { /* code to run after first render */ }, [])` - This hook is equivalent to `componentDidMount()` in class components. The callback function inside `useEffect` runs after the component is mounted to the DOM. The empty dependency array `[]` ensures that it only runs once after the initial render.

Updating:

2. `useEffect(() => { /* code to run on prop or state change */ }, [props, state])` - This hook is equivalent to `componentDidUpdate()` in class components. The callback function inside `useEffect` runs whenever the specified dependencies (props or state) change.

Unmounting:

3. `useEffect(() => { return () => { /* cleanup code */ }; }, [])` - This hook is equivalent to `componentWillUnmount()` in class components. The returned function inside `useEffect` runs when the component is unmounted from the DOM. It's used for cleanup tasks such as unsubscribing from subscriptions or canceling network requests.

Use cases for cleanup

Subscriptions to External Data Sources:

- Use Case: we subscribe to a WebSocket for real-time updates.
- Cleanup: When the component unmounts, we need to unsubscribe from the WebSocket to prevent memory leaks and unnecessary network traffic.

Timers and Intervals:

- **Use Case: we use `setTimeout` or `setInterval` for periodic tasks like polling an API.**
- **Cleanup: When the component unmounts, we need to clear the timer or interval to stop unnecessary executions and free up resources.**

Asynchronous Tasks:

- **Use Case: we fetch data from an API using `fetch` or a similar asynchronous function.**
- **Cleanup: When the component unmounts, we need to cancel the fetch request or close any connections to prevent the request from completing after the component is no longer in use.**

DOM Manipulation:

- **Use Case: we attach event listeners directly to DOM elements.**
- **Cleanup: When the component unmounts, we need to remove these event listeners to avoid memory leaks and ensure that the DOM is in a consistent state.**

Media Playback:

- **Use Case: we play audio or video files using HTML `<audio>` or `<video>` elements.**
- **Cleanup: When the component unmounts, we need to pause or stop the playback and release any resources associated with it to prevent performance issues and memory leaks.**

Third-party Library Integration:

- **Use Case: we integrate a third-party library that performs DOM manipulation or sets up event listeners.**
- **Cleanup: When the component unmounts, we need to clean up any changes made by the library or remove any event listeners it has added to avoid memory leaks and ensure that the application behaves correctly.**

In each of these use cases, cleanup functions are necessary to ensure that resources are properly released and side effects are cleaned up when the component is no longer in use. This helps maintain the performance and reliability of the application.

Hook

A hook is a function that allows us to use React features in functional components. React provides several built-in hooks that enable us to add state, lifecycle methods, and other features to functional components without needing to convert them to class components. Hooks were introduced in React 16.8 as a way to simplify component logic and reuse code.

1) useState:

useState allows us to add state management to functional components.

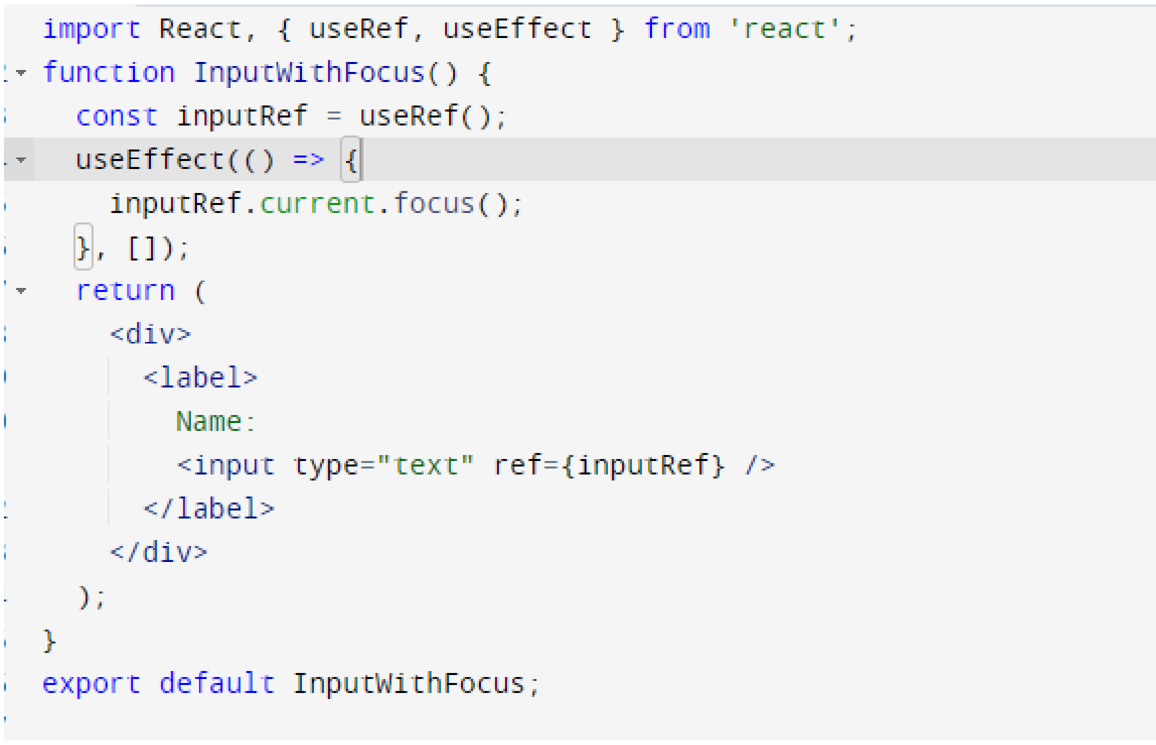
```
const [state, setState] = useState(initialState);
```

`state`: This variable holds the current value of the state.

`setState`: This function is used to update the state. When we call `setState` with a new value, React re-renders the component with the updated state value.

2) useEffect - see above paragraph
3) useRef:

   The `useRef` hook allows us to create a mutable reference that persists across re-renders of a component. It's commonly used for accessing and interacting with DOM elements or for keeping a reference to values that should persist without causing re-renders


```
import React, { useRef, useEffect } from 'react';
function InputWithFocus() {
  const inputRef = useRef();
  useEffect(() => {
    inputRef.current.focus();
  }, []);
  return (
    <div>
      <label>
        Name:
        <input type="text" ref={inputRef} />
      </label>
    </div>
  );
}
export default InputWithFocus;
```

props:

"props" is short for "properties". Props are a way of passing data from parent components to child components in a React application.

Difference between State and Props:

- **Props: Props are passed from parent to child components and are immutable. They are used to configure a component based on the data passed to it.**
- **State: State is managed within a component and is mutable. It represents the internal state of a component and can change over time due to user interactions or other factors.**

How to Avoid Props Drilling:

Props drilling refers to the situation where you need to pass props through multiple layers of components, which can make your code harder to maintain and understand. To avoid props drilling, we can use techniques like:

- **Context API: React's Context API allows us to share data across components without manually passing props through every level of the component tree.**
- **State Management Libraries: Libraries like Redux or MobX can be used for managing global state in our application, reducing the need to pass props down through multiple levels of components.**

When to Use Context API and When to Use Redux:

- **Context API: Use Context API when we have a small to medium-sized application with a few shared state values or when we need to share state between components that are not directly related.**
- **Redux: Use Redux when we have a large application with complex state management needs, such as handling asynchronous data fetching, managing multiple slices of state, or implementing time-travel debugging.**

Redux Toolkit

Slice: A slice is a portion of Redux state with its related reducer logic.
Reducers: Functions that specify state changes in response to actions.
combineReducers: Combines multiple reducers into one root reducer.
Store: Holds the entire state tree of the Redux application.
useDispatch: Hook to dispatch actions from React components.
useSelector: Hook to extract data from the Redux store into React components.

Difference between memo and useMemo

`memo`:

- `memo` functional components ke re-renders ko optimize karta hai, taki agar props mein koi change nahi hai to component ko re-render nahi kiya jaye.
- Yeh props ke changes ko dekhta hai aur agar koi change nahi hai to component ko cache karke rakhta hai, taki dobara se render na karna pade.

`useMemo`:

- `useMemo` ek value ko memoize karke rakhta hai, taki agar dependencies mein koi change nahi hai to us value ko phir se compute nahi kiya jaye.
- Yeh dependencies ko dekhta hai aur agar koi change nahi hai to pehle se memoized value ko return karta hai, naye computation ke bina.

React application ki performance ko badhane ke liye kai tarah ke techniques hote hain. Niche kuch mukhya techniques di gayi hain:

Code Splitting:

- Code splitting ka matalab hai aap apne code ko chhote chhote bundles mein divide karte hain, jisse user ko initial load time mein sirf woh code download karna padta hai jo current page ke liye zaroori hai.
- Iske liye aap dynamic import ka istemal kar sakte hain, jaise ki `import()` function. Isse aap asynchronous modules load kar sakte hain.
- Code splitting se aap apne application ka performance improve kar sakte hain kyun ki initial load time kam hoti hai.

Lazy Loading:

- Lazy loading ka matalab hai aap components, images, aur data ko tab tak load nahi karte jab tak ki unki zarurat na ho.
- Iske liye aap React Suspense aur React.lazy() ka istemal kar sakte hain. Suspense ek wrapper component hai jo lazy loading ke liye istemal hota hai.

- **Isse aap initial load time aur memory consumption ko kam kar sakte hain, khaaskar large applications mein.**

Memoization:

- **Memoization ka matalab hai kisi function ya computation ke result ko cache karke rakhna, taki agar wahi computation dubara kiya jaaye to uska result phir se compute na karna pade.**
- **React mein `memo` hook ka istemal karke aap functional components ko memoize kar sakte hain, jisse re-rendering ko optimize kiya ja sake.**
- **`useMemo` aur `useCallback` hooks bhi memoization ke liye istemal hote hain, jisse performance ko improve kiya ja sakta hai.**

Virtualization:

- **Virtualization ka matalab hai large lists ya tables ko render karte waqt sirf visible area mein hi elements ko render karna.**
- **Isse aap memory consumption aur rendering time ko kam kar sakte hain. Popular libraries jaise ki React Virtualized aur React Window iske liye istemal kiye jaate hain.**

Optimized Bundling:

- **Webpack ya anya bundler ka istemal karke aap apne assets ko minimize aur optimize kar sakte hain. Isse bundle size kam hoti hai aur page load time ghata.**
- **Iske liye aap code splitting, tree shaking, aur code minification ka istemal kar sakte hain.**

Server-side Rendering (SSR):

- **SSR ka matalab hai apne React application ka initial render server-side par karna.**
- **Isse aap page load time kam kar sakte hain aur SEO ke liye bhi beneficial hota hai, kyun ki search engines aasani se content ko index kar sakte hain.**

State Management Optimization:

- **Redux ya Context API ka sahi tareeke se istemal karke aap state management ko optimize kar sakte hain.**
- **Unnecessary re-renders ko avoid karke aur state ko efficient tareeke se manage karke aap application ka performance improve kar sakte hain.**

Performance Monitoring:

- **Application ke performance ko monitor karte rahen aur performance issues ko detect karke solve karte rahen.**
- **Chrome DevTools, Lighthouse, aur anya performance monitoring tools ka istemal karke aap application ke performance ko analyze kar sakte hain aur sudhar sakte hain.**

Controlled Component:

- **Controlled Component mein, component ka state bahar se manage hota hai. Yani ki, component ke state aur uski value, parent component ya kisi aur external source dwara control kiya jata hai.**
- **Ismein, state aur uski value component ke props ke madhyam se pass ki jati hai. Jab bhi state badalti hai, parent component dwara state ko update kiya jata hai, jo phirse props ke roop mein controlled component mein aata hai.**
- **Controlled components ka upyog karke, parent component se data ko ek centralise aur niyantrit tarike se prabandhit kiya ja sakta hai.**

Uncontrolled Component :

- **Uncontrolled Component mein, component apne aap apne state ko niyantrit karta hai. Iska arth hai ki, component ka state aur uski value component ke internal state mein handle hoti hai, aur ismein kisi external source ya parent component ki zarurat nahi hoti.**
- **Generally, uncontrolled components mein, DOM elements ke references ka upyog kiya jata hai taaki component apne state ko track kar sake aur usmein parivartan kar sake.**
- **Uncontrolled components ka upyog karke, complex forms ya user input fields ko handle karna aasaan ho jata hai, khaaskar jab dynamic user input ka nirdeshan kiya ja raha ho.**

unique ID

React me unique ID ka use tab hota hai jab hum kisi list ya collection ko render karte hain. Har ek item ko unique ID dena important hai taki React ko pata rahe ki konsa item kis element ko represent karta hai.

Isse React DOM ko efficiently update karne mein madad milti hai jab hum kuch elements ko add, remove ya reorder karte hain. Agar har ek item ka unique ID hota hai, to React unhe accurately identify kar sakta hai.

Iske bina, React ko update karne mein dikkat ho sakti hai aur unexpected behavior bhi ho sakta hai. For example, agar kisi list ke multiple elements ka same key hota hai, to React ko unhe alag nahi kar pata aur wo unhe sahi se update nahi kar pata.

In short, React me unique IDs ka istemal rendering process ko stable aur efficient banata hai, accessibility ko improve karta hai, aur testing ko bhi facilitate karta hai. Unique IDs ka use karke React elements ko identify karna asan ho jata hai aur errors ko prevent karne mein madad milti hai.