JavaScript and React.js Interview Questions and Answers

---

1. Is JavaScript single-threaded or multi-threaded? How can you achieve concurrency?

- JavaScript is **single-threaded**, meaning it executes one task at a time in the event loop.
- Concurrency is achieved using **asynchronous programming** (e.g., callbacks, Promises, async/await).

Example:

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

setTimeout(() => console.log("Async task"), 1000);

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

---

2. Are setTimeout and setInterval asynchronous or synchronous? How do they work?

- Both setTimeout and setInterval are **asynchronous**. They schedule tasks to run after a delay without blocking the main thread.
- These methods utilize the **event loop** and the **callback queue**.

Feature	`setTimeout`	`setInterval`
Purpose	Executes a function **once** after a specified delay.	Repeats execution of a function at specified intervals.
Usage	For one-time delayed tasks.	For repetitive tasks (e.g., polling).
Arguments	Function, delay, and optional arguments.	Function, interval, and optional arguments.
Returns	Returns a unique timeout ID.	Returns a unique interval ID.
Clearing	Use `clearTimeout(id)` to stop it.	Use `clearInterval(id)` to stop it.

How Do They Work?

1. The JavaScript engine sends the function to the **Web API** (e.g., Timer API).
2. The Web API waits for the specified delay or interval.
3. When the delay/interval expires, the callback function is sent to the **callback queue**.
4. The **event loop** moves the callback to the **call stack** if it's empty and executes it.
5. Both are **asynchronous** because their callbacks are handled by Web APIs.
6. setTimeout is for delayed one-time tasks, while setInterval is for repeated tasks.
7. Always clear timers using clearTimeout or clearInterval to avoid memory leaks.

---

3. What is hoisting in JavaScript? How to stop hoisting?

- Hoisting is JavaScript's default behavior of moving declarations to the top of the scope.
- You can stop hoisting by using let or const instead of var.

Example:

```
console.log(x); // undefined due to hoisting

var x = 5;
```

How to Stop Hoisting?

1. **Use let and const Instead of var:** Variables declared with let and const are hoisted, but they remain in a **"temporal dead zone" (TDZ)** until their declaration is reached in the code. This prevents access before initialization.

```
javascript

CopyEdit

console.log(a); // ReferenceError: Cannot access 'a' before initialization

let a = 10;
```

2. **Avoid Using Function Expressions Before Initialization:** Unlike function declarations, function expressions are **not hoisted** in the same way.

```
javascript

CopyEdit

console.log(greet); // Output: undefined

var greet = function () {

  console.log("Hello!");

};
```

3. **Write Code in a Strict Manner ("use strict"):** Strict mode prevents the use of undeclared variables, helping avoid issues related to hoisting.

```
javascript

CopyEdit

"use strict";

console.log(a); // ReferenceError: a is not defined

var a = 10;
```

Key Points About Hoisting:

1. **Function Declarations** are hoisted completely.
2. **Variable Declarations (var)** are hoisted but initialized to undefined.
3. **let and const Variables** are hoisted but remain in a **temporal dead zone (TDZ)**, throwing a ReferenceError if accessed before initialization.
4. **Function Expressions and Arrow Functions** are not hoisted like function declarations. Only their variable declaration is hoisted.

---

4. Difference between let, var, and const? Why were let and const introduced in ES6?

- **var**: Function-scoped, can be redeclared.
- **let**: Block-scoped, cannot be redeclared.
- **const**: Block-scoped, must be initialized and cannot be reassigned.
- let and const were introduced to fix var's scoping issues.

Feature	`var`	`let`	`const`
Scope	Function-scoped	Block-scoped	Block-scoped
Hoisting	Hoisted and initialized as `undefined`.	Hoisted but stays in the Temporal Dead Zone (TDZ).	Hoisted but stays in the Temporal Dead Zone (TDZ).
Re-declaration	Allowed in the same scope.	Not allowed in the same scope.	Not allowed in the same scope.
Re-assignment	Allowed.	Allowed.	Not allowed.
Initialization	Optional.	Optional.	Mandatory (must be initialized at the time of declaration).
Use Case	For older projects, not recommended for new ones.	For variables whose values can change.	For constants whose values should not change.

Why Were let and const Introduced in ES6?

1. **Fix Scoping Issues with var:**
    - var is function-scoped, which can lead to unintended behavior in block-scoped structures like loops or conditional statements. let and const provide block-scoping, which ensures variables are only accessible within the block where they are defined.

When to Use let and const?

- Use const by default for all variables.
- Use let only when you know the variable's value will change.
- Avoid using var in modern JavaScript unless you're maintaining legacy code.

---

5. Array methods in JavaScript

- Common methods: map(), filter(), reduce(), forEach(), find(), slice(), splice().

Example:

const arr = [1, 2, 3];

const doubled = arr.map(num => num * 2); // [2, 4, 6]

1. Adding and Removing Elements

Method	Description	Example
`push()`	Adds one or more elements to the **end** of an array.	`arr.push(4, 5);` → `[1, 2, 3, 4, 5]`
`pop()`	Removes and returns the **last** element from an array.	`arr.pop();` → `[1, 2]` and returns `3`.
`unshift()`	Adds one or more elements to the **beginning** of an array.	`arr.unshift(0);` → `[0, 1, 2, 3]`
`shift()`	Removes and returns the **first** element from an array.	`arr.shift();` → `[2, 3]` and returns `1`.
`splice()`	Adds, removes, or replaces elements in an array at a specific index.	`arr.splice(1, 1, 10);` → `[1, 10, 3]`

2. Accessing or Iterating

Method	Description	Example
`forEach()`	Executes a function for each array element.	`arr.forEach(el => console.log(el));`
`map()`	Returns a new array with the results of calling a function on every element.	`arr.map(x => x * 2);` → `[2, 4, 6]`
`filter()`	Returns a new array with elements that satisfy a given condition.	`arr.filter(x => x > 2);` → `[3]`
`find()`	Returns the **first** element that satisfies a given condition.	`arr.find(x => x > 2);` → `3`
`findIndex()`	Returns the **index** of the first element that satisfies a condition.	`arr.findIndex(x => x > 2);` → `2`
`some()`	Checks if **at least one** element satisfies a condition. Returns `true` or `false`.	`arr.some(x => x > 2);` → `true`
`every()`	Checks if **all** elements satisfy a condition. Returns `true` or `false`.	`arr.every(x => x > 0);` → `true`

3. Transforming Arrays

Method	Description	Example
`reduce()`	Applies a function to accumulate a single value (e.g., sum, product) from array elements.	`arr.reduce((acc, curr) => acc + curr, 0);` → `6`
`concat()`	Combines two or more arrays into a new array.	`arr.concat([4, 5]);` → `[1, 2, 3, 4, 5]`
`flat()`	Flattens nested arrays into a single array.	`[1, [2, [3]]].flat(2);` → `[1, 2, 3]`
`flatMap()`	Maps and flattens the result into a new array.	`[1, 2].flatMap(x => [x, x * 2]);` → `[1, 2, 2, 4]`

4. Searching and Sorting

Method	Description	Example
`indexOf()`	Returns the **first index** of a specified element, or `-1` if not found.	`arr.indexOf(2);` → `1`
`lastIndexOf()`	Returns the **last index** of a specified element, or `-1` if not found.	`arr.lastIndexOf(3);` → `2`
`includes()`	Checks if the array contains a specified element. Returns `true` or `false`.	`arr.includes(2);` → `true`
`sort()`	Sorts the elements of an array (mutates the array).	`arr.sort((a, b) => a - b);` → `[1, 2, 3]`
`reverse()`	Reverses the order of array elements (mutates the array).	`arr.reverse();` → `[3, 2, 1]`

5. Extracting Subarrays

Method	Description	Example
`slice()`	Returns a shallow copy of a portion of an array (does not modify the original).	`arr.slice(1, 3);` → `[2, 3]`
`splice()`	Removes/replaces elements from a specific index and optionally adds new ones.	`arr.splice(1, 2);` → Removes `[2, 3]`

---

6. Joining and Converting

Method	Description	Example
`join()`	Joins all elements of an array into a string.	`arr.join('-');` → `"1-2-3"`
`toString()`	Converts the array to a string.	`arr.toString();` → `"1,2,3"`

---

6. Destructuring in JavaScript: Does it work index-wise or key-wise?

- Destructuring works **key-wise** for objects and **index-wise** for arrays.

Example:

```
const obj = { a: 1, b: 2 };

const { a, b } = obj;

const arr = [1, 2];

const [first, second] = arr;
```

Destructuring is a feature in JavaScript that allows you to unpack values from arrays or properties from objects into distinct variables. It simplifies extracting data from complex structures like arrays and objects.

How Destructuring Works?

- **For Arrays**: Destructuring works **index-wise** because arrays are ordered collections.
- **For Objects**: Destructuring works **key-wise** because objects are collections of key-value pairs.

Aspect	Array Destructuring (Index-Wise)	Object Destructuring (Key-Wise)
Basis	Order of elements (index) matters.	Order does not matter; keys are used.
Default Values	Applied when index value is `undefined`.	Applied when key is not present or `undefined`.
Usability	For ordered data (e.g., lists).	For unordered data (e.g., objects).

---

7. Promises in JavaScript

- Promises handle asynchronous operations.
- **Stages:**
    1. Pending
    2. Fulfilled
    3. Rejected

Example:

```
const promise = new Promise((resolve, reject) => {
  resolve("Success");
});
```

A **Promise** is an object in JavaScript that represents the eventual completion (or failure) of an asynchronous operation. It allows you to write asynchronous code in a more readable and structured way compared to using callbacks.

Why Use Promises?

Promises simplify handling asynchronous tasks like API calls, file reading, or timeouts by chaining operations and handling errors effectively, avoiding "callback hell."

Structure of a Promise

A Promise has three main stages (or states):

1. **Pending**: The initial state; the promise is neither fulfilled nor rejected.
2. **Fulfilled**: The operation completed successfully, and the promise now has a resolved value.
3. **Rejected**: The operation failed, and the promise now has a reason (error message).

How Promises Work

A Promise is created using the Promise constructor and takes a function (called the executor) with two parameters:

- resolve: Call this function to mark the promise as fulfilled.
- reject: Call this function to mark the promise as rejected.

What Does a Promise Return?

A promise always returns an object with the following methods:

1. **.then(onFulfilled)**: Runs when the promise is fulfilled.
2. **.catch(onRejected)**: Runs when the promise is rejected.
3. **.finally(callback)**: Runs when the promise is settled, regardless of its outcome (fulfilled or rejected).

---

8. Difference between async and await

- async: Declares a function as asynchronous.
- await: Pauses execution until a Promise resolves.

Example:

```
async function fetchData() {
  const response = await fetch(url);
}
```

Feature	`async`	`await`
Definition	A keyword used to define an asynchronous function that always returns a Promise.	A keyword used inside an `async` function to pause execution until a Promise is resolved.
Purpose	Declares a function as asynchronous, allowing the use of `await` inside it.	Waits for a Promise to resolve or reject before proceeding.
Usage Context	Used before the function declaration.	Used only inside `async` functions.
Return Value	Always returns a Promise, even if the function returns a non-Promise value.	Returns the resolved value of the Promise.
Execution Behavior	Doesn't block the main thread.	Pauses the execution of the async function until the Promise is settled.
Error Handling	Errors can be caught using `.catch()` or `try-catch`.	Errors can be caught using `try-catch`.

---

9. What is Fetch API in JavaScript?

- Fetch API is a modern way to make HTTP requests.

Example:

```
fetch('https://api.example.com')
  .then(response => response.json())
  .then(data => console.log(data));
```

How Fetch API Works

1. **Initiates an HTTP Request**:
    - When you call fetch(), it sends an HTTP request to the given URL.
2. **Returns a Promise**:
    - The fetch() function returns a Promise that resolves to a **Response** object.
3. **Process the Response**:
    - You can use methods like .json(), .text(), or .blob() on the Response object to extract the data in the desired format.
4. **Error Handling**:
    - Errors like network failures are caught in the .catch() block.

- For HTTP errors (like 404 or 500), you must manually check the response.ok property.

Error Handling in Fetch

The Fetch API does not reject the Promise on HTTP errors (like 404 or 500). You need to check the response status manually.

Features of Fetch API

1. **Streaming Responses**:
   - You can process data chunks as they are received.
2. **CORS (Cross-Origin Resource Sharing)**:
   - Fetch handles cross-origin requests securely with proper headers.
3. **Aborting Requests**:
   - Fetch supports aborting requests using the AbortController.

---

10. Handling API errors (500 status)

- Use try-catch with error handling.

Example:

```
try {
  const response = await fetch(url);
  if (!response.ok) throw new Error("Server error");
} catch (error) {
  console.error(error);
}
```

An HTTP 500 error indicates an internal server issue. To handle it effectively, I would take a two-pronged approach:

On the **backend**, I'd ensure proper error handling with middleware to catch exceptions and log them using tools like Winston or Sentry. The API should return a meaningful, user-friendly error message instead of exposing raw server details.

On the **frontend**, I'd check the status code of the error response. If it's a 500 error, I'd display a user-friendly message like, 'Something went wrong on our end. Please try again later.' Additionally, I'd provide options like retrying the action or contacting support. Here's an example using Axios:

---

11. What is debounce?

- Debouncing limits how often a function can run.

Example:

```
function debounce(func, delay) {
  let timer;
  return (...args) => {
    clearTimeout(timer);
    timer = setTimeout(() => func(...args), delay);
  };
}
```

Debounce is a programming concept used to **limit the number of times a function is executed** in a short period, especially in response to high-frequency events such as user input, scrolling, resizing, or key presses.

In simple terms, **debouncing delays the execution of a function until after a specified period of inactivity**. It ensures that the function is only executed once the event has stopped firing.

Why Use Debounce?

Debouncing is commonly used to:

1. Improve **performance** by reducing unnecessary function calls.
2. Avoid **overloading** the browser or server.
3. Handle **real-time events** (like user input or resizing) efficiently.

How Debounce Works

When a debounced function is called, it:

1. **Cancels any previous pending calls** to the function.
2. Waits for a specified delay (e.g., 300ms).
3. Executes the function only if no new event is triggered during the delay.

Debounce vs Throttle

- **Debounce:** Executes the function **after the events stop firing** for a specified delay.

- **Throttle:** Executes the function **at regular intervals** while the event keeps firing.

Example:

- Use **debounce** for a search box (wait until typing stops).
- Use **throttle** for scroll or resize events (execute at intervals).

---

12. ES6 functions

- **Arrow functions**: Shorter syntax for functions.
- **Default parameters**: Set default values for function arguments.

  1. Arrow Functions (=>)

  Arrow functions provide a more concise syntax for writing functions and automatically bind this to the surrounding context.

  Example:

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

// Arrow function
const add = (a, b) => a + b;
```

  Key Features of Arrow Functions:

- **Shorter Syntax:** Single-line functions can omit the return keyword and curly braces.
- **Lexical this:** Arrow functions do not bind their own this. Instead, they inherit this from the surrounding context, making them useful in callbacks.

  Example of Lexical this:

```
class Counter {
  count = 0;

  increment() {
    setTimeout(() => {
```

```
    this.count++;
    console.log(this.count); // `this` refers to the Counter instance
  }, 1000);
 }
}

const counter = new Counter();
counter.increment();
```

---

2. Default Parameters

Functions in ES6 can now have default parameter values, which are used if no argument is provided or if the argument is undefined.

Example:

```
const greet = (name = "Guest") => {
 console.log(`Hello, ${name}!`);
};

greet(); // Hello, Guest!
greet("John"); // Hello, John!
```

---

3. Rest Parameters

The rest parameter syntax (...args) allows a function to accept an indefinite number of arguments as an array.

Example:

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

console.log(sum(1, 2, 3, 4)); // 10
```

---

4. Spread Operator in Function Calls

The spread operator (...) can be used to spread an array or iterable into individual arguments for a function.

Example:

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

console.log(Math.max(...numbers)); // 3
```

---

5. Enhanced Object Literals

ES6 allows for shorthand syntax when defining methods in objects.

Example:

```
const obj = {
  name: "John",
  greet() {
    console.log(`Hello, ${this.name}!`);
  },
};


obj.greet(); // Hello, John!
```

---

6. Template Literals in Functions

Template literals (``` ` ```) make it easier to create strings with variables and expressions.

Example:

```
const greet = (name, age) => `Hello, my name is ${name} and I am ${age} years old.`;


console.log(greet("John", 25)); // Hello, my name is John and I am 25 years old.
```

---

7. Block Scope with let and const in Functions

ES6 introduced let and const for block-scoped variables, replacing var. This reduces issues with variable scoping in functions.

Example:

```
function example() {
  if (true) {
    let x = 10; // Block-scoped
    const y = 20; // Block-scoped
    console.log(x, y); // 10, 20
  }
  // console.log(x, y); // Error: x and y are not defined
}
example();
```

---

8. Function Name Property

Functions in ES6 have a name property, which gives the name of the function.

Example:

```
const func = function exampleFunc() {};
console.log(func.name); // "exampleFunc"
```

---

9. Tail Call Optimization

ES6 introduced tail call optimization, which allows functions to call other functions as their last action without adding a new frame to the call stack. This improves performance for recursive functions.

Example:

```
function factorial(n, acc = 1) {
  if (n === 0) return acc;
  return factorial(n - 1, n * acc); // Tail call
}

console.log(factorial(5)); // 120
```

---

10. Destructuring in Function Parameters

You can use destructuring to extract values directly from objects or arrays in function parameters.

Example:

```
const displayUser = ({ name, age }) => {
  console.log(`Name: ${name}, Age: ${age}`);
};

const user = { name: "John", age: 25 };
displayUser(user); // Name: John, Age: 25
```

---

11. Generators (function*)

ES6 introduced generator functions, which can pause and resume execution using yield.

Example:

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

const gen = generatorFunc();
console.log(gen.next().value); // 1
console.log(gen.next().value); // 2
console.log(gen.next().value); // 3
```

ES6 added several features that made functions more concise, powerful, and easier to use. Key features include **arrow functions**, **default parameters**, **rest/spread operators**, **destructuring**, and **generators**, which improve readability, flexibility, and performance. These features have become essential in modern JavaScript development.

---

13. Higher-order functions

- Functions that take or return another function.

Example:

const higherOrder = fn => x => fn(x);

Aspect	Higher-Order Function	Pure Function
Definition	A function that takes another function as an argument or returns a function.	A function that produces the same output for the same input and does not cause side effects.
Purpose	Used for functional programming patterns like callbacks, mapping, filtering, and reducing.	Used to ensure predictability and avoid side effects in the program.
Example	`map()`, `filter()`, `reduce()`, `setTimeout()`	Functions like `add(a, b) { return a + b; }`
Takes Functions as Input	Yes, higher-order functions take other functions as arguments.	No, pure functions typically take primitive or object values as input.
Returns a Function	Can return a function as output (closures).	Does not return a function, just a value.
Side Effects	Can have side effects depending on implementation.	Never has side effects (e.g., doesn't modify variables outside its scope).
Immutability	May modify data if improperly used.	Always works on immutable data and never modifies the input.
Usage	Useful for abstraction, code reusability, and functional programming.	Useful for maintaining data integrity and avoiding unpredictable behavior.

---

14. Difference between == and ===?

- ==: Checks value equality.
- ===: Checks value and type equality.

---

15. What is React.js? Why use React.js?

- React.js is a library for building UIs with reusable components and efficient rendering using the virtual DOM.

Features of React.js

- **Component-Based Architecture:**
  - React applications are made up of reusable components, making it easier to manage large applications.

    - Components can have their own state and logic, and they combine to form complex UIs.
- **Virtual DOM:**
    - React uses a lightweight copy of the real DOM (called the virtual DOM) to update the UI efficiently.
    - Only the necessary changes are applied to the real DOM, improving performance.
- **One-Way Data Binding:**
    - Data flows in a single direction, making the code more predictable and easier to debug.
- **Declarative Syntax:**
    - Developers can describe "what" the UI should look like, and React takes care of rendering the UI based on the state.
- **JSX (JavaScript XML):**
    - A syntax extension that combines JavaScript and HTML-like markup to describe the UI.
- **React Ecosystem:**
    - React integrates well with libraries like Redux, React Router, and others for state management, routing, and more.

Use React.js?

1. Efficiency and Performance

- The **virtual DOM** minimizes expensive DOM manipulations, making React apps faster than traditional JavaScript frameworks.
- It efficiently updates and renders components when data changes.

2. Reusability and Maintainability

- React's **component-based architecture** allows developers to create reusable UI components, reducing code duplication and improving maintainability.

3. Strong Community and Ecosystem

- React has a large community of developers and extensive resources, making it easy to find solutions to problems.
- A wide range of third-party libraries, tools, and extensions are available.

4. Cross-Platform Development

- React can be used for web development (React.js) and mobile app development (React Native).

5. Declarative UI

- With React, developers focus on what the UI should look like at any given time, rather than manually updating the DOM.

6. SEO-Friendly

- React supports server-side rendering (SSR) with frameworks like **Next.js**, improving search engine optimization for web applications.

7. Easy Integration

- React can be integrated with existing projects or other frameworks without rewriting the entire codebase.

Where is React.js Used?

- **Web Applications**: SPAs like dashboards, e-commerce websites, and social media platforms.
- **Mobile Applications**: Using React Native for cross-platform mobile app development.
- **Dynamic Content Websites**: Blogs, news sites, and portfolios with dynamic content.
- **Enterprise Applications**: Tools like CRMs, project management software, and analytics dashboards.

---

16. What is the render function?

- A required method in class components to define UI elements.

In **React.js**, the render() function is a **lifecycle method** used in class components. Its primary purpose is to define what the **UI (User Interface)** should look like by returning **React elements**, which describe how the DOM should be structured and updated.

React uses this function to "render" the component on the screen. It is called automatically whenever the **state** or **props** of the component changes.

Key Characteristics of the render() Function

1. **Part of Class Components:**
    - The render() method is used exclusively in **class components**. Functional components do not have a render() method; they return JSX directly.
    - Example of a class component:

javascript

CopyEdit

class MyComponent extends React.Component {

```
  render() {
    return <h1>Hello, World!</h1>;
  }
}
```

2. **Must Return JSX or React Elements:**
    - The render() function returns JSX (JavaScript XML), which is syntactic sugar for React's React.createElement().
3. **Pure Function:**
    - The render() method must be a **pure function**, meaning it does not modify state or interact with external systems. It should only return UI based on the current props and state.
4. **React Updates the DOM:**
    - React uses the **Virtual DOM** to track changes and updates only the parts of the UI that have changed when the render() function is called.

How the render() Function Works

- React automatically calls the render() function whenever:
    1. The **state** of the component is updated using setState().
    2. The **props** passed to the component change.

When is render() Called?

1. **Initially:**
    - The render() function is called when the component is mounted for the first time.
2. **On Updates:**
    - Whenever the component's state or props changes, the render() method is called again to re-render the UI.
3. **Parent Re-renders:**
    - If a parent component re-renders, the child's render() function may also be called.

Rules of the render() Function

1. **Do Not Modify State or Props:**

- The render() method should not update the state or interact with the DOM.

2. **Return a Single Element:**

- You must wrap all returned elements in a single parent element (e.g., or <React.Fragment>).

---

17. What is JSX?

- JavaScript XML syntax for defining UI components.

  JSX (JavaScript XML) is a **syntax extension** for JavaScript used in React. It allows developers to write **HTML-like code** inside JavaScript. This makes it easier to define the structure of the user interface in a more readable and intuitive way.

  JSX is not a requirement for React but is widely used because it simplifies coding and improves developer experience.

Why Use JSX?

1. **Improved Readability**:
   - It combines the power of JavaScript with the simplicity of HTML-like syntax.
   - Example:

const element = <h1>Hello, World!</h1>;

2. **Declarative Syntax**:
   - It helps developers describe what the UI should look like directly in the code.
3. **React's Virtual DOM Integration**:
   - JSX is transformed into **React.createElement()** calls, which React uses to create and update the virtual DOM efficiently.
4. **Cleaner Code**:
   - Without JSX, the code would look more verbose. Compare:

Advantages of JSX

1. **Simplifies Development**:
   - Writing the UI structure directly in JavaScript improves developer productivity.
2. **Enhanced Readability**:

    - HTML-like syntax makes it easier for developers (even those new to React) to understand the code.

3. **Power of JavaScript**:
    - You can leverage all JavaScript functionalities inside JSX, like loops, conditionals, and expressions.
4. **Debugging Made Easier**:
    - Since JSX is converted to JavaScript, errors are clear and easy to debug.

JSX is a powerful feature of React that allows developers to write cleaner, more readable code while integrating HTML and JavaScript seamlessly. While optional, it has become a standard part of React development due to its simplicity and flexibility.

---

18. What is useEffect Hook?

- Used for side effects in functional components (e.g., API calls).

  The **useEffect** hook is a **built-in React Hook** that allows you to perform **side effects** in functional components. Side effects include actions like:

- Fetching data from an API.
- Directly interacting with the DOM.
- Setting up subscriptions or event listeners.
- Cleaning up resources (e.g., timers, subscriptions).

  It is the functional component equivalent of lifecycle methods like componentDidMount, componentDidUpdate, and componentWillUnmount in class components.

- **Effect Function:** The first argument is a callback function that contains the side effect logic.
- **Cleanup Function (Optional):** The return value of the effect function is another function used for cleanup.
- **Dependencies Array:** The second argument is an array of dependencies that determine when the effect should run.

  When to Use useEffect

1. **Data Fetching:**
    - Fetch data from an API when a component mounts or when a specific value changes.

2. **Subscriptions:**
    - Add and remove event listeners or set up WebSocket connections.
3. **DOM Manipulation:**
    - Perform DOM operations like scrolling, animations, or focus handling.
4. **Timers:**
    - Set up intervals or timeouts.
5. **State Synchronization:**
    - Update local state based on props or perform logging for debugging.

Calling an API using useEffect in React

When you want to fetch data from an API in a React functional component, you can use the useEffect hook. This ensures the API call happens at the right time in the component's lifecycle, such as when it mounts or when a dependency changes.

Steps to Call an API Using useEffect

1. **Setup the State:**
    - Use useState to store the fetched data or any error.
2. **Define the API Call Inside useEffect:**
    - Make the API call using fetch or libraries like Axios.
3. **Add a Dependency Array:**
    - Use an empty dependency array ([]) to fetch data on component mount, or include specific dependencies to re-fetch when those dependencies change.
4. **Handle Errors and Loading States:**
    - Use state variables to track loading or errors.
5. **Run the API Call Once:**
    - Use an empty dependency array ([]) for a one-time fetch on component mount.
6. **Handle Loading and Error States:**
    - Always show feedback for loading or errors.
7. **Prevent Memory Leaks:**
    - Use cleanup (return in useEffect) to cancel in-progress API calls if the component unmounts.
8. **Use Async Function Inside useEffect:**
    - useEffect itself cannot handle an async function directly; define it inside the effect.

---

19. Why use state? Syntax of state?

- **State** manages dynamic data.

Example:

const [count, setCount] = useState(0);

In React, **state** is a built-in object that is used to manage and track dynamic data in a component. When the state changes, React automatically re-renders the component to reflect those changes in the UI. It makes React apps interactive and dynamic.

Reasons to Use State:

1. **Dynamic UI Updates:**
    - State enables components to update dynamically based on user interactions or other changes.
2. **Component-Specific Data Management:**
    - State is isolated to the specific component where it is declared, allowing each component to manage its data independently.
3. **Reactivity:**
    - React listens for state changes and re-renders components when the state is updated, ensuring the UI stays in sync with the data.
4. **Declarative Approach:**
    - By using state, you describe "what" the UI should look like at any given point, and React handles the "how."

Feature	State	Props
Definition	Data that is mutable and managed locally in a component.	Data passed from a parent component.
Mutability	Can be updated using `setState` or `useState`.	Immutable; cannot be modified directly.
Usage	Used to manage component-specific data.	Used to pass data from parent to child.
Scope	Local to the component.	Available in child components.

Why Use State?

1. To manage dynamic data that changes over time.
2. To keep UI and data synchronized automatically.
3. To handle user interactions and responses.

By managing **state**, React ensures that your UI always reflects the latest data, making your application dynamic and interactive!

---

20. Lifecycle methods in React

- Class components: componentDidMount, componentDidUpdate, componentWillUnmount.
- Functional components: useEffect.

In React, **lifecycle methods** are special functions or hooks that are called at specific stages in a component's lifecycle. They allow developers to execute code when a component is created, updated, or destroyed. Lifecycle methods are primarily used in **class components**, while **hooks like useEffect** serve a similar purpose in functional components.

React Component Lifecycle

The lifecycle of a React component can be divided into three main phases:

1. **Mounting**: When the component is being created and inserted into the DOM.
2. **Updating**: When the component is re-rendered due to changes in state or props.
3. **Unmounting**: When the component is being removed from the DOM.

Lifecycle Methods in Class Components

1. Mounting (Component Creation Phase)

- **Methods:**
  - constructor():
    - Called before the component is mounted.
    - Used to initialize state or bind event handlers.
    - Example:

jsx

CopyEdit

```
constructor(props) {
  super(props);
  this.state = { count: 0 };
}
```

- static getDerivedStateFromProps(props, state):
  - Used to update state based on props before rendering.
  - Rarely used; called right before render().
- render():
  - Required method to return the JSX that describes the UI.
  - Pure function (does not modify state or props).
- componentDidMount():
  - Called after the component is mounted to the DOM.
  - Used for API calls, setting up subscriptions, or DOM manipulation.
  - Example:

jsx

CopyEdit

```
componentDidMount() {
  console.log("Component mounted!");
}
```

2. Updating (Component Re-rendering Phase)

- **Methods:**
  - static getDerivedStateFromProps(props, state):
    - Called when props or state changes, just like in mounting.
  - shouldComponentUpdate(nextProps, nextState):
    - Determines whether the component should re-render.
    - Returns a boolean (true to re-render, false to skip).
    - Example:

jsx

CopyEdit

```
shouldComponentUpdate(nextProps, nextState) {
  return nextState.count !== this.state.count;
}
```

  - render():
    - Same as in the mounting phase, used to update the UI.

- getSnapshotBeforeUpdate(prevProps, prevState):
  - Captures a snapshot of the DOM before it updates.
  - Returns a value that is passed to componentDidUpdate.
- componentDidUpdate(prevProps, prevState, snapshot):
  - Called after the component is updated in the DOM.
  - Used for DOM manipulation, API calls, or logging.
  - Example:

```
jsx
CopyEdit
componentDidUpdate(prevProps, prevState) {
  if (prevState.count !== this.state.count) {
    console.log("Count updated!");
  }
}
```

3. Unmounting (Component Removal Phase)

- **Methods:**
  - componentWillUnmount():
    - Called just before the component is removed from the DOM.
    - Used for cleanup (e.g., clearing timers, unsubscribing from events).
    - Example:

```
jsx
CopyEdit
componentWillUnmount() {
  console.log("Component will unmount!");
}
```

Lifecycle in Functional Components (Using useEffect)

In functional components, the **useEffect hook** is used to mimic lifecycle methods.

1. Mounting Phase

- Use useEffect with an empty dependency array to simulate componentDidMount.

jsx

CopyEdit

```
useEffect(() => {
  console.log("Component mounted!");
}, []); // Empty array means it runs once after the component mounts
```

2. Updating Phase

- Use useEffect with specific dependencies to simulate updates.

jsx

CopyEdit

```
useEffect(() => {
  console.log("State or props updated!");
}, [dependency]); // Runs whenever `dependency` changes
```

3. Unmounting Phase

- Return a cleanup function from useEffect to simulate componentWillUnmount.

jsx

CopyEdit

```
useEffect(() => {
  const timer = setInterval(() => console.log("Running..."), 1000);

  return () => {
    clearInterval(timer); // Cleanup on unmount
    console.log("Component unmounted!");
  };
}, []);
```

Phase	Lifecycle Method	Purpose
Mounting	`constructor()`	Initialize state and bind methods.
	`getDerivedStateFromProps()`	Sync state with props before rendering.
	`render()`	Render the component's UI.
	`componentDidMount()`	Perform side effects like API calls or event listeners.
Updating	`getDerivedStateFromProps()`	Sync state with updated props.
	`shouldComponentUpdate()`	Optimize performance by skipping unnecessary re-renders.
	`render()`	Render the updated UI.
	`getSnapshotBeforeUpdate()`	Capture DOM state before update.
	`componentDidUpdate()`	Perform side effects after the update.
Unmounting	`componentWillUnmount()`	Clean up timers, subscriptions, or event listeners.

---

21. Props and immutability

- Props are read-only and cannot be changed by the child component.

  Props (short for "properties") are a way to pass data from a **parent component** to a **child component** in React. They are read-only and allow you to customize or configure child components by passing information down from their parent.

  Key Points About Props:

1. **Immutable**:
   Props cannot be modified by the child component that receives them. They are read-only.
2. **Unidirectional Data Flow**:
   Data flows in one direction: from the parent to the child component.
3. **Used to Customize Components**:
   Props let you reuse components by passing different values to them, making them dynamic.
4. **Accessed in Functional Components**:
   Props are accessed as arguments in functional components:

   jsx

   CopyEdit

   const MyComponent = (props) => {

```
  return <h1>{props.title}</h1>;
};
```

Can We Change the Value of Props?

No, props are immutable, meaning you cannot directly modify their values inside the child component. However, you can change the value of props indirectly by updating the **state** in the parent component, which re-renders the child component with the updated props.

Why Can't We Modify Props?

- Props represent data that is controlled by the **parent component**.
- Allowing modifications would break the concept of **unidirectional data flow** in React, making the app harder to debug and maintain.

---

22. Changing data from child to parent

- Pass a callback function as a prop.

Steps to Change Data From Child to Parent Component

1. **Define a Callback Function in the Parent Component**:
   This function will handle the data coming from the child.
2. **Pass the Callback Function as a Prop to the Child Component**:
   The child will use this function to send data to the parent.
3. **Invoke the Callback Function in the Child Component**:
   Pass the data you want to send back to the parent as an argument.

---

23. What is event propagation?

- Flow of events through capturing and bubbling phases.

Event propagation refers to the order in which events are captured and handled in the DOM (Document Object Model) when an event (like a click) occurs on an element. It describes how events flow through the DOM hierarchy and consists of three phases:

Three Phases of Event Propagation

1. **Capturing Phase (Event Capturing)**:
    - The event starts at the root of the DOM tree and travels down toward the target element.
    - Also called "capture phase."
    - Event listeners registered with { capture: true } are triggered during this phase.
2. **Target Phase**:
    - The event reaches the target element where it was triggered.
    - Event listeners on the target element execute at this point.
3. **Bubbling Phase (Event Bubbling)**:
    - The event bubbles back up from the target element to the root of the DOM tree.
    - This is the default behavior in most browsers.
    - Event listeners without the capture option are triggered in this phase.

Event Propagation

1. **Event Bubbling**:
    - The event starts at the target element and bubbles up through its ancestors.
    - Default behavior unless explicitly stopped using e.stopPropagation().
2. **Event Capturing**:
    - The event is captured as it moves from the root to the target element.
    - Requires the event listener to be registered with { capture: true }.
3. **stopPropagation()**:
    - Stops further propagation of the event in both the bubbling and capturing phases.
4. **preventDefault()**:
    - Prevents the browser's default behavior for the event (e.g., preventing a form submission or a link navigation).

---

24. Difference between higher-order and pure functions

- **Higher-order functions**: Operate on functions.
- **Pure functions**: No side effects, deterministic.

Aspect	Higher-Order Function	Pure Function
Definition	A function that takes another function as an argument or returns a function.	A function that produces the same output for the same input and does not cause side effects.
Purpose	Used for functional programming patterns like callbacks, mapping, filtering, and reducing.	Used to ensure predictability and avoid side effects in the program.
Example	`map()`, `filter()`, `reduce()`, `setTimeout()`	Functions like `add(a, b) { return a + b; }`
Takes Functions as Input	Yes, higher-order functions take other functions as arguments.	No, pure functions typically take primitive or object values as input.
Returns a Function	Can return a function as output (closures).	Does not return a function, just a value.
Side Effects	Can have side effects depending on implementation.	Never has side effects (e.g., doesn't modify variables outside its scope).
Immutability	May modify data if improperly used.	Always works on immutable data and never modifies the input.
Usage	Useful for abstraction, code reusability, and functional programming.	Useful for maintaining data integrity and avoiding unpredictable behavior.

---

25. Controlled vs Uncontrolled Components

- Controlled: Managed by React state.
- Uncontrolled: Managed by DOM.

Controlled Components

A **controlled component** is a form element (e.g., input, textarea) whose value is controlled by React state. The value of the input is determined by the state, and any changes to the input update the state through event handling.

Key Characteristics:

- React controls the value of the input via the state.
- Always has a value prop that connects the input to the component's state.
- Changes are handled via onChange event handlers.

Uncontrolled Components

An **uncontrolled component** is a form element whose value is handled by the DOM itself. React does not control the value, and you use a ref to access and retrieve the value from the DOM when needed.

Key Characteristics:

- The value is not stored in the React state.
- A ref is used to access the current value of the input when required.
- React acts more as an observer rather than actively managing the value.

Aspect	Controlled Components	Uncontrolled Components
Value Management	React state controls the input value.	DOM controls the input value.
Access to Value	Value is retrieved directly from the `state`.	Value is accessed using a `ref`.
Event Handling	Requires `onChange` to update state for every input change.	Does not require event handlers for state updates.
Validation	Easier to validate input in real-time (e.g., during typing).	Validation happens after retrieving the value from the DOM.
Use Case	Useful when you need real-time validation or control over input.	Useful for simpler forms or when integrating with non-React libraries.
Code Complexity	Slightly more complex due to state management.	Simpler for basic forms as it doesn't require state.

---

26. What is a key in React?

- Unique identifier for list items to optimize rendering.

A **key** is a special attribute used in React to uniquely identify elements in a list. It helps React efficiently manage and update the DOM when the list of elements changes.

Why is a Key Important?

Keys enable React to:

1. **Identify Elements Uniquely**: Keys provide a way to identify which items have changed, been added, or removed.
2. **Improve Performance**: Keys help React minimize re-renders by efficiently reordering elements instead of recreating them.
3. **Ensure Consistency**: React uses keys to ensure that the correct element is updated during reconciliation.

Where to Use Keys?

Keys are typically used when rendering lists of elements, such as in map() calls.

1. Keys are used to **identify elements in a list** and ensure efficient rendering during updates.
2. React uses keys during the **reconciliation process** to determine which elements need to be updated, removed, or added.
3. Keys must be **unique** among siblings in a list.
4. Use **unique IDs** wherever possible and avoid using array indices unless you are working with static lists.
5. Keys **do not get passed to the component** and are only used internally by React for rendering.