52 Reactjs Interview questions & Answers

1. What is React?

- React is an opensource component based JavaScript library which is used to develop interactive user interfaces.

2. What are the features of React ?

- Jsx
- Virtual dom
- one way data binding
- Uses reusable components to develop the views
- Supports server side rendering

3. What is JSX ?

- JSX means javascript xml. It allows the user to write the code similar to html in their javascript files.
- This jsx will be transpiled into javascript that interacts with the browser when the application is built.

4. What is DOM ?

- DOM means document object model. It is like a tree like structure that represents the elements of a webpage.

5. What is Virtual Dom ?

- When ever any underlying data changes or whenever user enters something in textbox then entire UI is rerendered in a virtual dom representation.
- Now this virtual dom is compared with the original dom and creates a changeset which will be applied on the real dom.
- So instead of updating the entire realdom, it will be updated with only the things that have actually been changed.

Rendering Efficiency:

- **Virtual DOM**: Updates to the Virtual DOM are typically faster because they occur in memory and are not directly reflected in the browser.
- **Actual DOM**: Manipulating the actual DOM is slower due to its complexity and the potential for reflows and repaints.

6. What is state in Reactjs?

- State is an object which holds the data related to a component that may change over the lifetime of a component.
- When the state changes, the component re-renders.
- Eg: for functional component and class component

```
 import React, { useState } from "react";

function User() {
  const [message, setMessage] = useState("Welcome React");

  return (
    <div>
      <h1>{message}</h1>
    </div>
  );
}
```

```
import React from 'react';
class User extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      message: "Welcome to React world",
    };
  }

  render() {
    return (
      <div>
        <h1>{this.state.message}</h1>
      </div>
    );
  }
}
```

---

7. What is the purpose of callback function as an argument of `setState()` ?

- If we want to execute some logic once state is updated and component is rerendered then we can add it in callback function.

---

8. What are props ?

- Props are inputs to the component.
- They are used to send data from parent component to child component.
- Props are immutable, so they cannot be modified directly within the child component.
- **Example:**

```
// ParentComponent.js
import React from 'react';
import ChildComponent from './ChildComponent';

function ParentComponent() {
  const name = "John";

  return (
    <div>
      <h1>Parent Component</h1>
      <ChildComponent name={name} />
    </div>
  );
}

export default ParentComponent;
```

```
// ChildComponent.js
import React from 'react';

function ChildComponent(props) {
  return (
    <div>
      <h2>Child Component</h2>
      <p>Hello, {props.name}!</p>
    </div>
  );
}

export default ChildComponent;
```

---

9. What are the differences between State and Props in react ?

- Both props and state are used to manage the data of a component.
- State is used to hold the data of a component whereas props are used to send data from one component to another component.
- State is mutable but props are immutable.
- Any change in state causes rerender of component and its children.

---

10. What is props drilling ?

- Props drilling is the process of sending the data from one component to the component thats needs the data from several interconnected components

---

11. What are the disadvantages of props drilling and How we can avoid props drilling ?

- **Code complexity:**

Prop drilling can make code difficult to read and maintain, especially in large applications with many components. This is because props need to be passed down through multiple levels of components, and it can be difficult to keep track of which components are using which props.

- **Reduced maintainability:**

  Prop drilling can also make code less maintainable. This is because if a prop needs to be changed, the change needs to be propagated through all of the components that use it. This can be a time-consuming and error-prone process.

- **Increased risk of errors:**

  Prop drilling can also increase the risk of errors. This is because it can be difficult to keep track of which components are using which props, and it can be easy to forget to pass a prop down to a component that needs it. This can lead to errors in the application.

- **Performance overhead:**

  Prop drilling can also have a performance overhead. This is because every time a prop is passed down to a component, the component needs to re-render. This can be a significant performance overhead in large applications with many components.

Makes application slower.

We can avoid props drilling using context api or Redux or by using any state management libraries.

---

12. What is useReducer hook ?

- It is an alternative to useState hook which is used when state of the component is complex and requires more than one state variable.

---

13. What is useMemo ?

- useMemo is useful for performance optimization in react.
- It is used to cache the result of a function between re-renders.
- **Example :**
  - In our application we have a data vizualization component where we need to display charts based on performing complex calculations on some large data sets. By using useMemo we can cache the computed result, which ensures that the component does not recalculate on every re-renders.
  - This saves computational resources and provides smoother user experience.

```
import React, { useMemo } from 'react';

const DataVisualization = ({ data }) => {
  const processedData = useMemo(() => {
    // Perform expensive computations on data
    // ...
    return processedData;
  }, [data]);

  // Render the visualization using the processed data
  // ...

  return <div>{/* Visualization component */}</div>;
};
```

In this example, the `processedData` is memoized using `useMemo` to avoid recomputing it on every render. The expensive computations are performed only when the `data` prop changes.

14. What is useCallback ?

- useCallback caches the function defination between the re-renders
- It takes two arguments: the callback function and an array of dependencies. The callback function is only recreated if one of the dependencies has changed.

Good Ref: https://deadsimplechat.com/blog/usecallback-guide-use-cases-and-examples/#the-difference-between-usecallback-and-declaring-a-function-directly

---

15. What are the differences between useMemo and useCallback ?

- Both useMemo and useCallback are useful for performance optimization.
- useMemo will cache the result of the function between re-renders whereas useCallback will cache the function itself between re-renders.

---

16. Which lifecycle hooks in class component are replaced with useEffect in functional components ?

1. **componentDidMount():** equivalent to useEffect with empty array.

```
useEffect(()=>{
   console.log("Called on initial mount only once")
},[])
```

2. **componentDidUpdate():** equivalent to useEffect with array of dependencies

```
useEffect(()⇒{
  console.log("Called on every dependency update")
},[props.isFeature,props.content])
```

This will be called whenever dependency value changes (here Eg: isFeature or content).

3. **componentDidUnmount():** equivalent to useEffect with return statement.

```
useEffect(()⇒{
  return ()⇒{
    console.log(`Any cleanup activities
    or unsubscribing etc here`)
  }
})
```

---

17. What is component life cycle of React class component ?

- React life cycle consists of 3 phases.
  - mounting
  - updating
  - unmounting
- **Mounting:**
  - In this phase the component is generally mounted into the dom.
  - It is an initialization phase where we can do some operations like getting data from api, subscribing to events etc.

  1. **Constructor:**
     - It is a place to set the initial state and other initial values.

2. **getDerivedStateFromProps:**
    - This is called right before rendering the elements into the dom.
    - Its a natural place to set the state object based on the initial props.
    - It takes state as an argument and returns an object with changes to the state.

    ```
    getDerivedStateFromProps(props,state){
      return { favColor: props.favColor }
    }
    ```

3. **render():**
    - It contains all the html elements and is method that actually outputs the html to the dom.
4. **ComponentDidMount():**
    - This is called once component is mounted into the dom.
    - Eg: fetch api calls, subscribing to events etc.

- **Updating phase:**
    - This is when the component is updated. The component will be updated when ever there is change in state or props.

1. **getDerivedStateFromProps:** same as above
2. **ShouldComponentUpdate:**
    - This will return boolean value that specifies whether react should continue with the rendering or not. default is true.

    ```
    shouldComponentUpdate(){
        return true/false
    }
    ```

3. **Render:** same as above

4. **getSnapshotBeforeUpdate:**
    - It will have access to the props and state before update. means that even after the update you can check what are the values were before update.

    ```
    getSnapshotBeforeUpdate(prevProps,prevState){
       console.log(prevProps,prevState)
    }
    ```

5. **ComponentDidUpdate:**
    - Called after the component is updated in the dom.

- **Unmounting phase:**
    - In this phase the component will be removed from the dom. here we can do unsubscribe to some events or destroying the existing dialogs etc.

    1. **ComponentWillUnmount:**
        - This is called when component is about to be removed from the dom.

---

18. What are keys in React ?

- Keys are used to uniquely identify the elements in the list.
- react will use this to indentify, which elements in the list have been added, removed or updated.

```
function MyComponent() {
  const items = [
    { id: 1, name: "apple" },
    { id: 2, name: "banana" },
```

```
    { id: 3, name: "orange" }
  ];

  return (
    <ul>
      {items.map((item) => (
        <li key={item.id}>{item.name}</li>
      ))}
    </ul>
  );
}
```

---

19. What are fragments in react ?

- React fragments allows us to wrap or group multiple elements without adding extra nodes to the dom.

---

20. What are Pure components in React ?

- A component which renders the same output for the same props and state is called as pure component.
- It internally implements **shouldComponentUpdate** lifecycle method and performs a shallow comparision on the props and state of the component. If there is no difference, the component is not re-rendered.
- **Advantage:**
  - It optimizes the performance by reducing unnecessary re-renders.

Example for class component:

```
import React, { PureComponent } from 'react';

class MyPureComponent extends PureComponent {
  render() {
    return (
      <div>
        <h1>Pure Component Example</h1>
        <p>Props: {this.props.text}</p>
      </div>
    );
  }
}

export default MyPureComponent;
```

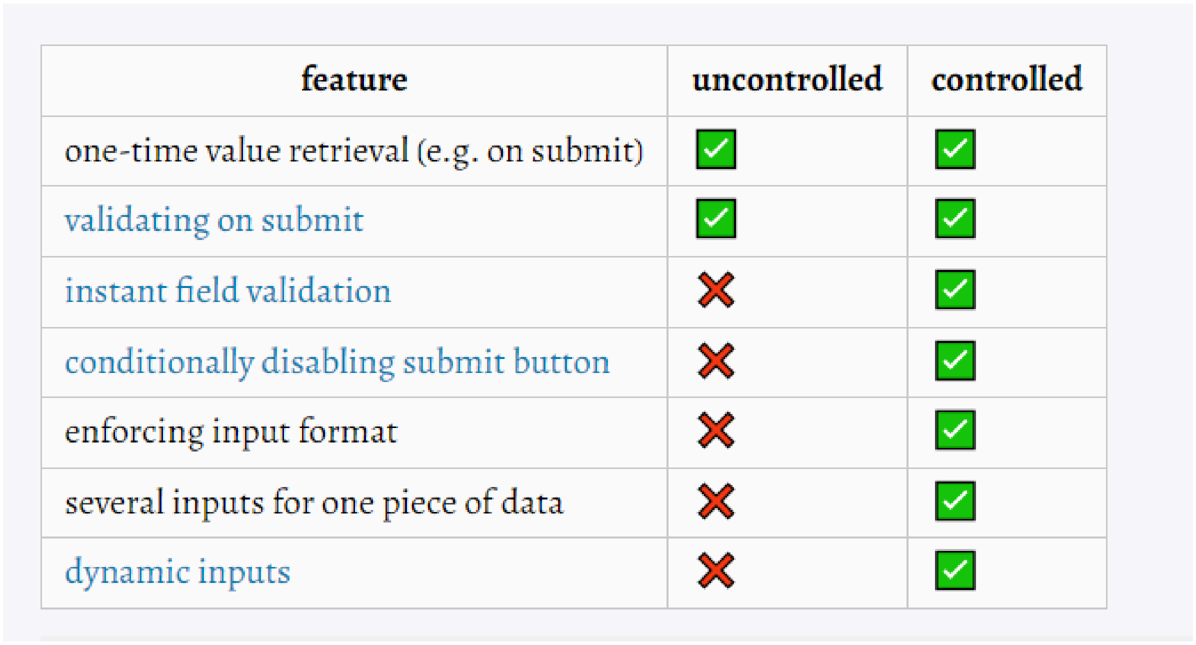
Reference: https://react.dev/reference/react/PureComponent

```
import { PureComponent, useState } from 'react';

class Greeting extends PureComponent {
  render() {
    console.log("Greeting was rendered at",
    new Date().toLocaleTimeString());
    return <h3>Hello{this.props.name && ', '}
    {this.props.name}!</h3>;
  }
}

export default function MyApp() {
  const [name, setName] = useState('');
  const [address, setAddress] = useState('');
  return (
    <>
      <label>
        Name{': '}
        <input value={name} onChange={e => {
        setName(e.target.value)}
        } />
      </label>
      <label>
        Address{': '}
        <input value={address} onChange={e => {
         setAddress(e.target.value)}
         } />
      </label>
      <Greeting name={name} />
    </>
  );
}
```

---

21. What are the differences between controlled and uncontrolled components ?

- **Controlled components:**
  - In controlled components, the form data is handled by the react component
  - We use event handlers to update the state.
  - React state is the source of truth.
- **Uncontrolled components:**
  - In uncontrolled components, the form data is handled by the dom.
  - We use Ref's to update the state.
  - Dom is the source of truth.



feature	uncontrolled	controlled
one-time value retrieval (e.g. on submit)	✅	✅
validating on submit	✅	✅
instant field validation	❌	✅
conditionally disabling submit button	❌	✅
enforcing input format	❌	✅
several inputs for one piece of data	❌	✅
dynamic inputs	❌	✅

Ref: https://goshacmd.com/controlled-vs-uncontrolled-inputs-react/

---

22. Whatare Ref's in React?

- ref's are the way to access the dom elements created in the render method.
- they are helpful when we want to update the component whith out using state and props and prevents triggering rerender.

Common useCases:

- Managing input focus and text selection
- Media control/Playback
- Communicating between react components that are not directly related through the component tree.

Examples:

1. Managing input focus

```
function App() {
  const inputRef = useRef();
  const focusOnInput = () ⇒ {
    inputRef.current.focus();
  };

  return (
    <div>
      <input type='text' ref={inputRef} />
      <button onClick={focusOnInput}>Click Me</button>
    </div>
  );
}
```

2.Managing Audio playback:

```
function App() {
  const audioRef = useRef();

  const playAudio = () ⇒ {
    audioRef.current.play();
  };

  const pauseAudio = () ⇒ {
    audioRef.current.pause();
  };

  return (
    <div>
      <audio
        ref={audioRef}
        type='audio/mp3'
        src='https://s3-us-west-2.amazonaws.com/keyboard-32-12.mp3'
      ></audio>
      <button onClick={playAudio}>Play Audio</button>
      <button onClick={pauseAudio}>Pause Audio</button>
    </div>
  );
}
```

Reference: https://www.memberstack.com/blog/react-refs

---

23. What is meant by forward ref ?

- In React, forwardref is a technique which is used to send the ref from parent component to one of its children. This is helpful when we want to access the child component dom node from the parent component.

Example:

1. **Creating the Forward Ref Component:** Define a component that forwards the ref to a child component.

```
import React, { forwardRef } from 'react';

const ChildComponent = forwardRef((props, ref) ⇒ {
  return <input ref={ref} />;
});

export default ChildComponent;
```

2. **Using the Forward Ref Component:** Use this component and pass a ref to it.

```
import React, { useRef } from 'react';
import ChildComponent from './ChildComponent';

function ParentComponent() {
  const inputRef = useRef(null);
  return (
    <div>
      <ChildComponent ref={inputRef} />
      <button onClick={() ⇒ inputRef.current.focus()}>
      Focus Input
      </button>
    </div>
  );
}

export default ParentComponent;
```

In this example, `ChildComponent` is a functional component that forwards the ref it receives to the `input` element it renders. Then, in the `ParentComponent`, a ref is created using `useRef`, passed to `ChildComponent`, and used to focus the input when a button is clicked.

Reference: https://codedamn.com/news/reactjs/what-are-forward-refs-in-react-js

---

24. What are Error boundaries ?

- Error boundaries are one of the feature in react which allows the developers to catch the errors during the rendering of component tree, log those erros and handle those errors without crashing the entire application by displaying a fallback ui.

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
  /* Update state so the next render will
  show the fallback UI. */
    return { hasError: true };
  }
  componentDidCatch(error, info) {
    // Example "componentStack":
    //   in ComponentThatThrows (created by App)
    //   in ErrorBoundary (created by App)
    //   in div (created by App)
    //   in App
    logErrorToMyService(error, info.componentStack);
  }

  render() {
    if (this.state.hasError) {
      // You can render any custom fallback UI
      return this.props.fallback;
    }

    return this.props.children;
  }
}
Then you can wrap a part of your component tree with it:

<ErrorBoundary fallback={<p>Something went wrong</p>}>
  <Profile />
</ErrorBoundary>
```

25. What are Higher order components in react ?

- Higher order component is a function which takes the component as an argument and returns a new component that wraps the original component.

For example if we wanted to add a some style to multiple components in our application, Instead of creating a `style` object locally each time, we can create a HOC that adds the `style` objects to the component that we pass to it.

```
function withStyles(Component) {
  return props => {
    const style = { padding: '0.2rem', margin: '1rem' }
    return <Component style={style} {...props} />
  }
}

const Button = () = <button>Click me!</button>
const Text = () => <p>Hello World!</p>

const StyledButton = withStyles(Button)
const StyledText = withStyles(Text)
```

26. What is Lazy loading in React ?

- It is a technique used to improve the performance of a webapplication by splitting the code into smaller chunks and loading them only when its required intead of loading on initial load.

```
import React, { lazy, Suspense } from 'react';
const LazyComponent = lazy(() => import('./LazyComponent'));

function App() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <LazyComponent />
      </Suspense>
    </div>
  );
}
```

```
export default App;
```

---

27. What is suspense in React ?

The lazy loaded components are wrapped by **Suspense.** The Suspense component receives a fallback prop which is displayed until the lazy loaded component in rendered.

```
import React, { lazy, Suspense } from 'react';
const LazyComponent = lazy(() => import('./LazyComponent'));

function App() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <LazyComponent />
      </Suspense>
    </div>
  );
}

export default App;
```

---

28. What are custom hooks ?

- Custom hooks helps us to extract and reuse the stateful logic between components.
- Eg:
    - Fetch data

- To find user is in online or offline.

https://react.dev/learn/reusing-logic-with-custom-hooks

```
import { useState, useEffect } from 'react';

// Custom hook to fetch data from an API
function useFetch(url) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await fetch(url);
        if (!response.ok) {
          throw new Error('Network response was not ok');
        }
        const result = await response.json();
        setData(result);
      } catch (error) {
        setError(error);
      } finally {
        setLoading(false);
      }
    };

    fetchData();

    // Cleanup function
    return () => {
      // Cleanup logic if needed
    };
  }, [url]); // Dependency array to watch for changes
  return { data, loading, error };
}

// Example usage of the custom hook
function MyComponent() {
  const { data, loading, error } = useFetch('https://api.com/data');

  if (loading) {
    return <div>Loading...</div>;
  }

  if (error) {
    return <div>Error: {error.message}</div>;
  }

  return (
    <div>
      {data && (
        <ul>
          {data.map(item => (
            <li key={item.id}>{item.name}</li>
          ))}
        </ul>
      )}
    </div>
  );
}

export default MyComponent;
```

29. What is context api in react ?

- Context api is a feature in react by using which we can share the data across the application with out having to pass the data manually through every level of component tree.

- It solves the problem of props drilling in react.

We need to follow 3 main steps to implement context api in our application.

- Create context
- Create Provider and wrap this provider around root level component and pass the data.
- Create consumer and utilize data using useContext.

Examples Where we can use context api:

- Shopping cart: Managing cart data in ecommerce application and share between product listings and checkout pages.
- Authentication: sharing user information across the application.
- Data fetching : Sharing fetched data across multiple components that need to display this data. (for displaying search results when user makes a search request).

---

30. Give an example of context api usage ?

Example: Once user loggedin, I wanted to share the user information between the components.

- **Step - 1:** We need to create a context using **createContext** method in userContext.js file.

```
UserContext.js

import React,{createContext} from "react";

const UserContext = createContext();
const UserProvider = UserContext.Provider;
const UserConsumer = UserContext.Consumer;
```

```
export {UserProvider, UserConsumer};
```

- **Step - 2:** We need to wrap the root level component with provider and pass the user Information.

```
App component:

import React from "react";
import {ComponentA} from "./ComponentA.js";
import {UserProvider} from "./UserContext.js";

const App = () ⇒ {
  const userinfo = {
    id:"1",
    name:"saikrishna"
  }

  return (
    <div>
      <UserProvider value={userinfo}>
        <ComponentA/>
      </UserProvider>
    </div>
  );
};
```

- **Step - 3:** In componentA, We can get the data using useContext and utilize the user Information.

```
ComponentA:

import React,{useContext} from "react";
import {UserConsumer} from "./UserContext.js";
```

```
export const ComponentA = () ⇒ {
  const {id,name} = useContext(UserConsumer);

  return <div>Hello {id}{name}</div>
}
```

31. What is Strict mode in react ?

It identifies the commonly occured bugs in the development time itself.

- Checks if there are any depricated apis usage.
- Checks for deprecated lifecycle methods.
- Checks if Duplicate keys in list.
- Warns about Possible memory leaks. etc.

```
===============================================================
// Enabling strict mode for entire App.
===============================================================

import { StrictMode } from 'react';
import { createRoot } from 'react-dom/client';

const root = createRoot(document.getElementById('root'));
root.render(
  <StrictMode>
    <App />
  </StrictMode>
);

===============================================================
// Any part of your app
===============================================================
import { StrictMode } from 'react';

function App() {
  return (
    <>
      <Header />
      <StrictMode>
        <main>
          <Sidebar />
          <Content />
        </main>
      </StrictMode>
      <Footer />
    </>
  );
}
```

Good Ref: https://dev.to/codeofrelevancy/what-is-strict-mode-in-react-3p5b

---

32. What are the different ways to pass data from child component to parent component in react ?

There are 4 common ways to send data from child component to parent component. They are.,

1. Callback Functions
2. Context API
3. React Hooks (useRef)
4. Redux

---

33. How do you optimize your react application ?

- **Code Splitting:** Break down large bundles into smaller chunks to reduce initial load times
- **Lazy Loading:** Load non-essential components\asynchronously to prioritize critical content.
- **Caching and Memoization:** Cache data locally or use memoization libraries to avoid redundant API calls and computations.
- **Memoization:** Memoize expensive computations and avoid unnecessary re-renders using tools like React.memo and useMemo.
- **Optimized Rendering:** Use shouldComponentUpdate, PureComponent, or React.memo to prevent unnecessary re-renders of components.
- **Virtualization:** Implement virtual lists and grids to render only the visible elements, improving rendering performance for large datasets.
- **Server-Side Rendering (SSR):** Pre-render content on the server to improve initial page load times and enhance SEO.
- **Bundle Analysis:** Identify and remove unused dependencies, optimize images, and minify code to reduce bundle size.
- **Performance Monitoring:** Continuously monitor app performance using tools like Lighthouse, Web Vitals, and browser DevTools.
- **Optimize rendering with keys:** Ensure each list item in a mapped array has a unique and stable key prop to optimize rendering performance. Keys help React identify which items have changed, been added, or removed, minimizing unnecessary DOM updates.
- **CDN Integration:** Serve static assets and resources from Content Delivery Networks (CDNs) to reduce latency and improve reliability.

---

34. What are Portals in react ?

- Portals are the way to render the child components outside of the parent’s dom hierarchy.

- We mainly need portals when a React parent component has a hidden value of overflow property(overflow: hidden) or z-index style, and we need a child component to openly come out of the current tree hierarchy.
- It is commonly used in Modals, tooltips, loaders, notifications toasters.
- This helps in improving the performance of application.

```
import React from 'react';
import ReactDOM from 'react-dom';

class MyPortal extends React.Component {
  render() {
    return ReactDOM.createPortal(
      this.props.children,
      document.getElementById('portal-root')
    );
  }
}

class App extends React.Component {
  render() {
    return (
      <div>
        <h1>My App</h1>
        <MyPortal>
          <p>
          This is rendered in a different part of the DOM!
          </p>
        </MyPortal>
      </div>
    );
  }
}

ReactDOM.render(<App />, document.getElementById('root'));
```

35. What are synthetic events in react ?

- Synthetic events are the wrapper around native browser events.
- By using this react will remove browser inconsistencies and provides a unified api interface for event handling
- They optimise the performance by event pooling and reusing event objects.

36. What are the difference between Package.json and Package.lock.json

- **Package.json:** is a metadata file that contains information about your project, such as the name, version, description, author, and most importantly, the list of dependencies required for your project to run. This file is used by npm (Node Package Manager) to install, manage, and update dependencies for your project.
- **Package.lock.json:** is a file that npm generates after installing packages for your project. This file contains a detailed description of the dependencies installed in your project, including their versions and the dependencies of their dependencies. This file is used by npm to ensure that the same version of each package is installed every time, regardless of the platform, environment, or the order of installation.
- Package.json is used to define the list of required dependencies for your project, while package-lock.json is used to ensure that the exact same versions of those dependencies are installed every time, preventing version conflicts and guaranteeing a consistent environment for your project.

37. What are the differences between client side and server side rendering

- **Rendering location:** In csr, rendering occurs on the client side after receiving raw data from the server where as in ssr, rendering occurs on server side side and server returns the fully rendered html page to the browser.
- **Initial Load time:** csr has slow initial load time as browser needs to interpret the data and render the page. where as ssr has faster initial load times as server send pre-rendered html page to the browser.
- **Subsequent interactions:** subsequent interactions in csr involves dynamic updates with out requiring full page reload whereas, ssr requires full page reload as server generates new html for every interactions.
- **Seo:** Ssr is seo friendly when compared to csr as fully rendered html content is provided to the search engine crawlers whereas csr needs to parse javascript heavy content.

---

38. What is react-router ?

React-router is a javascript library which is used to implement routing in react applications.
It allows us to navigate between views or pages with out refreshing the page.

- **Router:** It wraps the entire application and provides the routing context for the application. It contains 2 types of routers,
  Browser router and Hash router.
- **Route:** It contains mapping between urlpath and the component. When the url matches, respective component will be rendered.
- **Link:** is used to create the navigation links which user can click to navigate to different routes.
- **switch:** is used to render the first matching route among its children. It ensures only one route is rendered.

```
import React from 'react';
import { BrowserRouter as Router, Route, Switch, Link } from 'react-router-dc

function Home() {
  return <h2>Home</h2>;
}

function About() {
  return <h2>About</h2>;
}

function Contact() {
  return <h2>Contact</h2>;
}

function App() {
  return (
    <Router>
      <div>
        <nav>
          <ul>
            <li>
              <Link to="/">
              Home
              </Link>
            </li>
            <li>
              <Link to="/about">
              About
              </Link>
            </li>
            <li>
              <Link to="/contact">
              Contact
              </Link>
            </li>
          </ul>
        </nav>

        <Switch>
          <Route exact path="/"
          component={Home}
          />
          <Route path="/about"
          component={About}
          />
          <Route path="/contact"
          component={Contact}
          />
        </Switch>
      </div>
    </Router>
  );
}

export default App;
```

---

39. What are protected routes in react ?

- By using the protected routes, we can define which users/user roles have access to specific routes or components.

---

40. What is the difference between useEffect and useLayoutEffect ?

- useeffect is asynchronous where as uselayouteffect is synchronous

- uselayouteffect runs after browser calculation of dom measurements and before browser repaints the screen whereas useeffect runs parallely.
- uselayouteffect really needed when we need to do something based on layout of the dom, if we want to measure dom elements and do something etc.

Ref : https://www.youtube.com/watch?v=wU57kvYOxT4

https://mtg-dev.tech/blog/real-world-example-to-use-uselayouteffect

---

41. Practical question: How to send data from child to parent using callback functions ?

- Define a function in the parent component that takes data as an argument.
- Pass this function as a prop to the child component.
- When an event occurs in the child component (like a button click), call this function with the data to be passed to the parent.

Parent Component:

```
import React, { useState } from 'react';
import ChildComponent from './ChildComponent';

function ParentComponent() {
  const [dataFromChild, setDataFromChild] = useState('');

  const handleDataFromChild = (data) => {
    setDataFromChild(data);
  };

  return (
    <div>
      <ChildComponent onData={handleDataFromChild} />
      <p>Data from child: {dataFromChild}</p>
    </div>
  );
}

export default ParentComponent;
```

Child Component:

```
import React from 'react';

function ChildComponent({ onData }) {
  const sendDataToParent = () => {
    const data = 'Hello from child';
    onData(data);
  };

  return (
    <button onClick={sendDataToParent}>
    Send Data to Parent
    </button>
  );
}

export default ChildComponent;
```

---

42. Practical question: How to send the data from child component to parent using useRef ?

- It is used to store the data without re-rendering the components.
- It will not trigger any event by itself whenever the data is updated.

Parent component:

```
import React from "react";
import TextEditor from "./TextEditor";

export default function Parent() {
  const valueRef = React.useRef("");

  return (
    <>
      <TextEditor valueRef={valueRef} />
      <button onClick={() => {
      console.log(valueRef.current)}
      }
      >Get</button>
    </>
  );
}
```

Child component:

```
export default function TextEditor({ valueRef }) {
  return <textarea
  onChange={(e) => (valueRef.current = e.target.value)}>
  </textarea>;
}
```

43. Practical question: Create a increment decrement counter using useReducer hook in react ?

```
import React, { useReducer } from 'react';
```

```
// Initial state
const initialState = {
  count: 0
};

// Reducer function
const reducer = (state, action) ⇒ {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      return state;
  }
};

// Component
const Counter = () ⇒ {
  const [state, dispatch] = useReducer(reducer, initialState);

  return (
    <div>
      <p>Count: {state.count}</p>
      <button onClick={() ⇒ dispatch(
      { type: 'increment' }
      )}>Increment</button>
      <button onClick={() ⇒ dispatch(
      { type: 'decrement' }
      )}>Decrement</button>
    </div>
  );
};

export default Counter;
```

44. Practical question: create a custom hook for increment/decrement counter ?

- **Create a custom hook for counter:**

```
useCounter.js:

export const useCounter = () => {
  const [counter, setCounter] = useState(0);

  const increment = () => {
    setCounter(counter + 1);
  };

  const decrement = () => {
    setCounter(counter - 1);
  };

  return {
    counter,
    increment,
    decrement,
  };
};
```

- **Utilize useCounter in our component:**

```
component.js:
import { useCounter } from './useCounter.js';

const App = () => {
  const { counter, increment, decrement } = useCounter();

  return (
    <div>
      <button onClick={decrement}>Decrease</button>
      <div>Count: {counter}</div>
      <button onClick={increment}>Increase</button>
    </div>
  );
};
```

---

45. Machine coding question: Dynamic checkbox counter

Display 4 checkboxes with different names and a button named selectall

User can select each checkbox

Select all button click will check all checkboxes

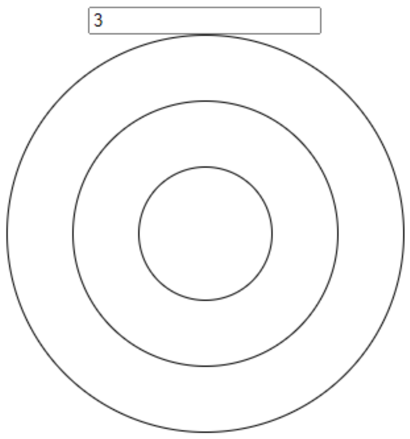
Button should be disabled once all checkboxes are selected.

Display selected checkboxes count and names in ui.

```
import React, { useState } from 'react';
import { render } from 'react-dom';

const Checkbox = ({ label, checked, onChange }) => {
  return (
    <div>
      <label>
        <input type="checkbox"
        checked={checked}
        onChange={onChange} />
        {label}
      </label>
    </div>
  );
```

```
};
const App = () => {
const [checkboxes, setCheckboxes] = useState([
   { id: 1, label: 'Checkbox 1', checked: false },
   { id: 2, label: 'Checkbox 2', checked: false },
   { id: 3, label: 'Checkbox 3', checked: false },
   { id: 4, label: 'Checkbox 4', checked: false },
  ]);

const [selectAllDisabled,setSelectAllDisabled]= useState(
false
);

  const handleCheckboxChange = (id) => {
    const updatedCheckboxes = checkboxes.map((checkbox) =>
      checkbox.id === id
        ? { ...checkbox, checked: !checkbox.checked }
        : checkbox
    );
    setCheckboxes(updatedCheckboxes);
    const allChecked = updatedCheckboxes.every(
    (checkbox) => checkbox.checked
    );
    setSelectAllDisabled(allChecked);
  };

  const handleSelectAll = () => {
    const updatedCheckboxes = checkboxes.map(
    (checkbox) => ({
      ...checkbox,
      checked: !selectAllDisabled,
    }));
    setCheckboxes(updatedCheckboxes);
    setSelectAllDisabled(!selectAllDisabled);
  };
  const selectedCheckboxes = checkboxes.filter(
  (checkbox) => checkbox.checked
  );
  const selectedCount = selectedCheckboxes.length;

  return (
    <div>
      {checkboxes.map((checkbox) => (
        <Checkbox
          key={checkbox.id}
          label={checkbox.label}
          checked={checkbox.checked}
          onChange={
          () => handleCheckboxChange(checkbox.id)
          }
        />
      ))}
      <button
      onClick={handleSelectAll}
      disabled={selectAllDisabled}>
        {selectAllDisabled ? 'Deselect All' : 'Select All'}
      </button>
      <p>Selected: {selectedCount}</p>
      <ul>
        {selectedCheckboxes.map((checkbox) => (
          <li key={checkbox.id}>{checkbox.label}</li>
        ))}
      </ul>
    </div>
  );
};

render(<App />, document.getElementById('root'));
```

46. Create a nested circles based on user input like below image.



Solution:

Create App component and circle component. We will use recursion concept to achieve this.

```
App.js:

import React, { useState } from "react";
import { Circle } from "./Circle.js";
import "./styles.css";

export default function App() {
  const [num, setNum] = useState(0);

  const handleInput = (e) => {
    setNum(e.target.value);
  };

  return (
    <div className="nested-circles-container">
```

```
      <input
        onChange={(e) => handleInput(e)}
        placeholder="Enter number of circles"
        type="number"
      />
      <Circle numbCircles={num}></Circle>
    </div>
  );
}

App.css/Style.css:

.nested-circles-container {
  display: flex;
  justify-content: center;
  align-items: center;
  flex-direction: column;
}
```

```
Circle.js:

import React from "react";
import "./Circle.css";

export const Circle = ({ numbCircles }) => {
  let size = numbCircles * 100;
  return (
    <div
      className="circle-new"
      style={{
        width: `${size}px`,
        height: `${size}px`,
      }}
    >
```

```
      {numbCircles > 1 &&
       <Circle numbCircles={numbCircles - 1}>
       </Circle>
       }
     </div>
   );
  };

  Circle.css:

  .circle-new {
    border-radius: 100%;
    border: 1px solid black;
    display: flex;
    justify-content: center;
    align-items: center;
  }
```

---

47. What are the various design patterns in react ?

- Compound pattern
- HOC Pattern
- Render props pattern
- Container/Presentational pattern

---

48. What is compound pattern ?

- By using this compound pattern we will create multiple components that work together to perform a single task.

- We can achieve this using context api.
- This promotes separation of concerns between parent and child components, promotes reusability of logic and maintainability.

---

49. What is render props pettern ?

- With the Render Props pattern, we pass components as props to other components. The components that are passed as props can in turn receive props from that component.
- Render props make it easy to reuse logic across multiple components.

https://javascriptpatterns.vercel.app/patterns/react-patterns/render-props

---

50. What is Container/Presentational pattern ?

It enforces separation of concerns by separating the view from the application logic.

- We can use the Container/Presentational pattern to separate the logic of a component from the view. To achieve this, we need to have a:
  - **Presentational** Component, that cares about **how** data is shown to the user.
  - **Container** Component, that cares about **what** data is shown to the user.

For example, if we wanted to show listings on the landing page, we could use a container component to fetch the data for the recent listings, and use a presentational component to actually render this data.

https://javascriptpatterns.vercel.app/patterns/react-patterns/conpres

---

51. What is React.memo ?

- React.memo is a Higher order component provided by react that memoizes the functional components.
- It caches the result of component's rendering and rerenders the component only when the props have changed.

```
const MyComponent = React.memo((props) => {
  console.log('Rendering MyComponent');
  return (
    <div>
      <h1>Hello, {props.name}!</h1>
      <p>{props.message}</p>
    </div>
  );
});
```

---

52. Differences between dependencies and devdependencies ?

- **dependencies:** These are the packages that your project needs to run in production. These are essential for the application to function correctly.
- **devDependencies:** These packages are only needed during the development phase.
    - Eg: Jest, react-developer-tools etc.

Reactjs Scenario based Questions:

1. How to send the data from parent component to child component in react ?

Ans. To send data from a parent component to a child component in React, We will use **props** (short for "properties"). Here's a step-by-step explanation with examples:

a. Pass Data via Props in the Parent Component

- Include the child component in the parent's JSX.
- Attach data to it as a custom attribute (e.g., `dataName={value}`).

```
// ParentComponent.jsx

import ChildComponent from './ChildComponent';

function ParentComponent() {
  const message = "Hello from Parent!";// Data to send
  const user = { name: "Alice", age: 30 };

  return (
    <div>
      {/* Pass data as props */}
      <ChildComponent text={message} userData={user} isAdmin={false}/>
    </div>);
}
```

b. Access Props in the Child Component

- The child receives data via its `props` parameter (or via destructuring).

Method A: Using `props` Directly

```
// ChildComponent.jsx

function ChildComponent(props) {
  return (
    <div>
      <h1>{props.text}</h1>
      <p>Name: {props.userData.name}, Age: {props.userData.age}</p>
      {props.isAdmin && <button>Admin Panel</button>}
    </div>);
}
```

Method B: Destructuring Props

```
// ChildComponent.jsx (cleaner with destructuring)

function ChildComponent({ text, userData, isAdmin }) {
  return (
    <div>
      <h1>{text}</h1>
      <p>Name: {userData.name}, Age: {userData.age}</p>
      {isAdmin && <button>Admin Panel</button>}
    </div>);
}
```

---

2. How to call parent component method from child component in reactjs ?

To call a parent component's method from a child component in React, follow these steps:

1. Pass the Parent's Method as a Prop to the Child

- In the parent component, pass the method as a prop (e.g., `onAction`).
- In the child component, invoke the prop when needed (e.g., on a button click).

Example with Functional Components:

```jsx
// ParentComponent.jsx
import React from 'react';
import ChildComponent from './ChildComponent';

function ParentComponent() {
  const handleChildAction = (data) => {
    console.log('Method called from child!', data);
  };

  return (
    <div>
      <ChildComponent onAction={handleChildAction} />
    </div>);
}
```

```jsx
// ChildComponent.jsx
import React from 'react';

function ChildComponent({ onAction }) {
  const handleClick = () => {
// Call parent method with data
      onAction('Hello from child');
  };

  return <button onClick={handleClick}>
  Trigger Parent Method
  </button>;
}
```

Example with Class Components:

```jsx
// ParentComponent.jsx
import React from 'react';
import ChildComponent from './ChildComponent';
```

```
class ParentComponent extends React.Component {
  handleChildAction = (data) => {
    console.log('Method called from child!', data);
  };

  render() {
    return <ChildComponent onAction={this.handleChildAction} />;
  }
}
```

```
// ChildComponent.jsx
import React from 'react';

class ChildComponent extends React.Component {
  render() {
    return (
      <button onClick={() =>
      this.props.onAction('Hello from child')}
      >
        Trigger Parent Method
      </button>);
  }
}
```

3.How to bind array/array of objects to dropdown in react ?

Step 1: Define Array or Array of Objects in your component

```
import React, { useState } from 'react';

const DropdownExample = () ⇒ {
  // Simple Array
  const fruits = ['Apple', 'Banana', 'Orange'];

  // Array of objects
  const countries = [
    { id: 1, name: 'India' },
    { id: 2, name: 'USA' },
    { id: 3, name: 'Canada' },
  ];

  // State hooks for selected values
  const [selectedFruit, setSelectedFruit] = useState('');
  const [selectedCountryId, setSelectedCountryId] = useState('');

  return (
    <div>
      {/* Dropdown using simple array */}
      <select value={selectedFruit} onChange={(e) ⇒ setSelectedFruit(e.target.
value)}>
        <option value="" disabled>Select Fruit</option>
        {fruits.map((fruit, index) ⇒ (
          <option key={index} value={fruit}>
            {fruit}
          </option>
        ))}
      </select>

      <p>Selected Fruit: {selectedFruit}</p>

      {/* Dropdown using array of objects */}
      <select value={selectedCountryId} onChange={(e) ⇒ setSelectedCountryI
d(e.target.value)}>
        <option value="" disabled>Select Country</option>
        {countries.map((country) => (
          <option key={country.id} value={country.id}>
            {country.name}
          </option>
        ))}
      </select>

      <p>Selected Country ID: {selectedCountryId}</p>
    </div>
  );
};

export default DropdownExample;
```

📌 Explanation:

- **Mapping (`array.map`)**: Used to loop through the array and render options.
- **Keys (`key`)**: Essential for React to track each rendered item.
- **State (`useState`)**: Maintains selected values.
- **onChange handler**: Updates the selected values in the state.

🖥️ Output Example:

- Selecting **Banana** sets `selectedFruit` to `'Banana'`.
- Selecting **India** sets `selectedCountryId` to `1`.

---

4. Create a lazy loaded component in react ?

1. Create your Lazy Loaded Component (e.g., `MyComponent.jsx`)

```jsx
// MyComponent.jsx
import React from 'react';

const MyComponent = () => {
  return <div>This is a lazy loaded component!</div>;
};

export default MyComponent;
```

2. Lazy load this component in your main application file (e.g., `App.jsx`)

```jsx
// App.jsx
import React, { lazy, Suspense } from 'react';

const LazyMyComponent = lazy(() => import('./MyComponent'));

const App = () => {
  return (
    <div>
      <h1>Main App Component</h1>

      <Suspense fallback={<div>Loading component...</div>}>
        <LazyMyComponent />
      </Suspense>
    </div>
  );
};

export default App;
```

Explanation:

- `lazy`: This function allows you to dynamically import a component.
- `Suspense`: Wraps lazy-loaded components and provides a fallback UI (e.g., a spinner or loader) while the component loads.

---

5. How to display data entered by the user in another textbox ?

Step-by-step code:

```
import React, { useState } from 'react';

const DisplayInputExample = () => {
  const [inputValue, setInputValue] = useState('');

  const handleChange = (event) => {
    setInputValue(event.target.value);
  };

  return (
    <div>
      <inputtype="text"
        placeholder="Enter something..."
        value={inputValue}
        onChange={handleChange}
      />

      <br /><br />

      <inputtype="text"
        placeholder="Displayed value..."
      value={inputValue}
      readOnly
    />
   </div>
  );
};

export default DisplayInputExample;
```

Explanation:

- `useState` hook manages the entered value.
- The first `input` captures user input via `onChange`.
- The second `input` displays the entered text, set as `readOnly` to prevent manual editing.

This allows you to see the entered text mirrored immediately in another textbox.

---

6. How to conditionally render an element or text in react ?

You can conditionally render elements or text in React using these common approaches:

Using the ternary (`? :`) operator:

```
function MyComponent({ isLoggedIn }) {
 return (
  <div>
   {isLoggedIn ? <p>Welcome back!</p> : <p>Please log in.</p>}
```

```
    </div>
  );
}
```

Using logical AND (`&&`) operator:

```
function MyComponent({ showMessage }) {
  return (
    <div>
      {showMessage && <p>This message appears if true.</p>}
    </div>
  );
}
```

Conditional rendering via variables:

```
function MyComponent({ isAdmin }) {
  let content;

  if (isAdmin) {
    content = <p>Admin Dashboard</p>;
  } else {
    content = <p>User Dashboard</p>;
  }

  return <div>{content}</div>;
}
```

Inline conditional rendering (short-circuiting):

```
function MyComponent({ notifications }) {
  return (
    <div>
     {notifications.length > 0 && <p>You have notifications!</p>}
    </div>
  );
}
```

Choose the approach based on clarity and complexity of your conditions.

---

7. How to perform debouncing in react ?

Step-by-step example:

```
import React, { useState, useEffect, useRef } from 'react';

function DebounceExample() {
  const [inputValue, setInputValue] = useState('');
  const [debouncedValue, setDebouncedValue] = useState('');

  const timerRef = useRef(null);

  const handleInputChange = (event) => {
    setInputValue(event.target.value);
  };

  useEffect(() => {
    // Clear the previous timer if inputValue changes
    if (timerRef.current) clearTimeout(timerRef.current);

    // Set new timer for debounce
    timerRef.current = setTimeout(() ⇒ {
      setDebouncedValue(inputValue);
    }, 500); // 500ms debounce interval

    // Cleanup function
    return () ⇒ clearTimeout(timerRef.current);
  }, [inputValue]);

  return (
    <div>
      <inputtype="text"
        value={inputValue}
        placeholder="Type something..."
        onChange={handleInputChange}
      />

      <p>Debounced Input: {debouncedValue}</p>
    </div>
  );
}

export default DebounceExample;
```

Explanation:

- `useState`: Manages the immediate (`inputValue`) and debounced values (`debouncedValue`).
- `useRef`: Stores the timer reference to clear previous timers effectively.
- `useEffect`: Triggers on each change to `inputValue`, resets the debounce timer, and updates `debouncedValue` only after the specified delay (500ms).

This technique ensures actions are performed **only after the user stops typing for a short period**, optimizing performance and preventing unnecessary function calls.

8. Create a component to fetch data from api in reactjs ?

```
import React, { useState, useEffect } from "react";

const DataFetcher = ({ apiUrl }) ⇒ {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() ⇒ {
    setLoading(true);
    setError(null);

    fetch(apiUrl)
      .then((response) ⇒ {
        if (!response.ok) {
          throw new Error("Network response was not ok");
        }
        return response.json();
      })
      .then((json) ⇒ {
        setData(json);
        setLoading(false);
      })
      .catch((err) ⇒ {
        setError(err.message || "Something went wrong");
        setLoading(false);
      });
  }, [apiUrl]);
  if (loading) return <p>Loading...</p>;
  if (error) return <p>Error: {error}</p>;

  return (
    <div>
      <pre>{JSON.stringify(data, null, 2)}</pre>
    </div>
  );
};

export default DataFetcher;
```

Usage:

```
<DataFetcher apiUrl="https://jsonplaceholder.typicode.com/posts/1" />
```

Explanation:

- Uses `useEffect` to fetch data when the component mounts or when `apiUrl` changes.
- Shows loading text while fetching.
- Handles and displays errors.
- Displays fetched data in a formatted JSON block.

9. Given two dropdowns, select 2nd dropdown options based on value selected in one dropdown in reactjs ? (Load states based on country selected)

```
import React, { useState } from "react";

const countries = [
  { id: "in", name: "India" },
  { id: "us", name: "USA" },
];

const states = {
  in: [
    { id: "mh", name: "Maharashtra" },
    { id: "tg", name: "Telangana" },
  ],
  us: [
    { id: "ca", name: "California" },
    { id: "ny", name: "New York" },
  ],
};

const CountryStateDropdown = () ⇒ {
  const [selectedCountry, setSelectedCountry] = useState("");
  const [selectedState, setSelectedState] = useState("");

  const handleCountryChange = (e) ⇒ {
    setSelectedCountry(e.target.value);
    setSelectedState(""); // reset state on country change
  };

  const handleStateChange = (e) ⇒ {
    setSelectedState(e.target.value);
  };

  return (
    <div>
      <label>
```

```
        Country:{" "}
        <select value={selectedCountry} onChange={handleCountryChange}>
          <option value="">-- Select Country --</option>
          {countries.map((country) => (
            <option key={country.id} value={country.id}>
              {country.name}
            </option>
          ))}
        </select>
      </label>

      <br />

      <label>
        State:{" "}
        <selectvalue={selectedState}
          onChange={handleStateChange}
          disabled={!selectedCountry}
        >
          <option value="">-- Select State --</option>
          {selectedCountry &&
            states[selectedCountry]?.map((state) => (
              <option key={state.id} value={state.id}>
                {state.name}
              </option>
            ))}
        </select>
      </label>
    </div>
  );
};

export default CountryStateDropdown;
```

How it works:

- When you select a country, it updates `selectedCountry`.
- The second dropdown populates with states corresponding to the selected country.
- If no country is selected, the states dropdown is disabled.
- Selecting a new country resets the selected state.

10. How to display dynamic html data in reactjs ?

To display dynamic HTML content (like a string containing HTML tags) safely in React, you use **`dangerouslySetInnerHTML`**.

```
function DynamicHtml({ htmlString }) {
  return (
    <div dangerouslySetInnerHTML={{ __html: htmlString }} />
  );
}

// Usage
const someHtml = "<p>This is <strong>dynamic</strong> HTML content.</p
>";

<DynamicHtml htmlString={someHtml} />
```

Important Notes:

- React escapes HTML by default to prevent XSS attacks.
- `dangerouslySetInnerHTML` **bypasses React's escaping**, so **only use it with trusted content**.
- Always sanitize user-generated HTML before rendering with `dangerouslySetInnerHTML`.

11. How to add data into useState array in functional component in react ?

To add data into a `useState` array in a React functional component, you update the state by creating a new array with the existing items plus the new item.

```
import React, { useState } from "react";

function MyComponent() {
  const [items, setItems] = useState([]);

  const addItem = (newItem) => {
    setItems(prevItems => [...prevItems, newItem]);
  };

  return (
    <><button onClick={() => addItem("New item")}>Add Item</button>
      <ul>
        {items.map((item, idx) => (
          <li key={idx}>{item}</li>
        ))}
      </ul>
    </>
  );
}
```

Explanation:

- Use the updater function form of `setItems` to get the latest state.
- Create a new array with the spread operator `[...prevItems, newItem]` to add the new item.
- This ensures immutability and triggers a re-render.

12. Change focus/enable/disable textbox in child component based on parent component button click ?

To **change focus** or **enable/disable a textbox in a child component** based on a button click in the parent component in React, you can use **props** and **refs**.

Here's a minimal example showing both:

a. Parent controls enable/disable and focus on child's textbox

```
import React, { useState, useRef, useEffect } from "react";

function Child({ disabled, shouldFocus }) {
  const inputRef = useRef(null);

  useEffect(() ⇒ {
    if (shouldFocus && inputRef.current) {
      inputRef.current.focus();
    }
  }, [shouldFocus]);

  return (
    <inputref={inputRef}
      type="text"
      disabled={disabled}
      placeholder={disabled ? "Disabled" : "Enabled"}
    />
  );
}

function Parent() {
  const [disabled, setDisabled] = useState(true);
  const [focusRequest, setFocusRequest] = useState(false);
  const toggleDisabled = () ⇒ {
    setDisabled(prev ⇒ !prev);
  };

  const focusInput = () ⇒ {
    setFocusRequest(true);
    // reset the focus request after focusing to allow future focus
    setTimeout(() ⇒ setFocusRequest(false), 0);
  };

  return (
    <><button onClick={toggleDisabled}>
        {disabled ? "Enable" : "Disable"} Textbox
      </button>
      <button onClick={focusInput} disabled={disabled}>
        Focus Textbox
      </button>
      <Child disabled={disabled} shouldFocus={focusRequest} />
    </>
  );
}

export default Parent;
```

How it works:

- `disabled` state in Parent controls whether the child input is enabled or disabled.
- `shouldFocus` prop signals the child to focus the textbox.
- The child uses `useEffect` to focus when `shouldFocus` changes to `true`.
- Parent resets `focusRequest` immediately so subsequent focus events can trigger again.

13. How to display dropdown value selected by user in another textbox ?

```
import React, { useState } from "react";

function DropdownToTextbox() {
  const [selectedValue, setSelectedValue] = useState("");

  const handleChange = (e) => {
    setSelectedValue(e.target.value);
  };

  return (
    <div>
      <select value={selectedValue} onChange={handleChange}>
        <option value="">Select an option</option>
        <option value="Apple">Apple</option>
        <option value="Banana">Banana</option>
        <option value="Orange">Orange</option>
      </select>

      <input type="text" value={selectedValue} readOnly />
    </div>
  );
}

export default DropdownToTextbox;
```

Explanation:

- The `select` element controls the `selectedValue` state.
- When the user selects a value, `handleChange` updates the state.

- The textbox's `value` is set to the current `selectedValue`, showing the selected dropdown option.
- `readOnly` is used on the textbox so the user cannot edit it manually.

---

14. How to display number of characters remaining functionality for textarea using react useRef?

React example showing how to display the number of characters remaining for a `<textarea>` using `useRef`:

```
import React, { useRef, useState } from "react";

function TextareaWithCharCount() {
  const maxLength = 100;
  const textareaRef = useRef(null);
  const [remaining, setRemaining] = useState(maxLength);

  const handleChange = () => {
    const length = textareaRef.current.value.length;
    setRemaining(maxLength - length);
  };

  return (
    <div>
      <textarearef={textareaRef}
        maxLength={maxLength}
        onChange={handleChange}
        rows={4}
        cols={40}
      />
      <div>{remaining} characters remaining</div>
    </div>
  );
}

export default TextareaWithCharCount;
```

How it works:

- `textareaRef` points to the textarea DOM node.
- On each `onChange` event, read `textareaRef.current.value.length` to get current length.
- Calculate remaining chars (`maxLength - length`) and update state.
- Display remaining count below the textarea.

---

15. Create a search textbox filter in reactjs

Here's a simple ReactJS example of a search textbox filter that filters a list of items as you type:

```
import React, { useState } from "react";

const SearchFilter = () ⇒ {
  const items = [
    "Apple",
    "Banana",
    "Orange",
    "Mango",
    "Pineapple",
    "Grapes",
    "Strawberry",
  ];

  const [searchTerm, setSearchTerm] = useState("");

  // Filter items based on search term (case insensitive)
  const filteredItems = items.filter(item ⇒
    item.toLowerCase().includes(searchTerm.toLowerCase())
  );

  return (
    <div>
      <inputtype="text"
        placeholder="Search fruits..."
        value={searchTerm}
        onChange={e ⇒ setSearchTerm(e.target.value)}
      />
      <ul>
        {filteredItems.length > 0 ? (
          filteredItems.map((item, idx) ⇒ <li key={idx}>{item}</li>)
        ) : (
          <li>No results found</li>
        )}
      </ul>
    </div>
  );
};

export default SearchFilter;
```

How it works:

- You type in the input box.
- `searchTerm` state updates on each keystroke.
- The list `items` is filtered to include only those matching the `searchTerm`.
- The filtered list displays below the input.

16. Display radio button data selected by user in another textbox

Here’s a simple React example showing how to display the selected radio button value inside a textbox:

```
import React, { useState } from "react";

function RadioToTextbox() {
  const [selectedOption, setSelectedOption] = useState("");

  const handleChange = (e) ⇒ {
    setSelectedOption(e.target.value);
  };

  return (
    <div>
      <h3>Select an option:</h3>
      <label>
        <inputtype="radio"
          value="Option 1"
          checked={selectedOption === "Option 1"}
          onChange={handleChange}
        />
        Option 1
      </label>
      <br />
      <label>
        <inputtype="radio"
          value="Option 2"
          checked={selectedOption === "Option 2"}
          onChange={handleChange}
        />
        Option 2
      </label>
      <br />
      <br />

      <label>
        Selected option:
        <input type="text" value={selectedOption} readOnly />
      </label>
    </div>
  );
}

export default RadioToTextbox;
```

17. Create an error boundary component in react ?

```

import React from 'react';

class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(/* error */) {
    return { hasError: true };
  }

  componentDidCatch(error, info) {
    if (this.props.onError) {
      this.props.onError(error, info);
    } else {
      console.error(error, info);
    }
  }

  render() {
    if (this.state.hasError) {
      return this.props.fallback || <div>Something went wrong.</div>;
    }
    return this.props.children;
  }
}

export default ErrorBoundary;
```

Usage example:

```
import ErrorBoundary from './ErrorBoundary';
import MyComponent from './MyComponent';

function App() {
  return (
    <ErrorBoundaryfallback={<div>Oops—an error occurred.</div>}
      onError={(error, info) ⇒ {
        // send to logging service
        console.log('Logging error:', error, info);
      }}
    >
      <MyComponent />
    </ErrorBoundary>
  );
}
```

- ErrorBoundary must be a class component.

- It resets to fallback UI when a child throws.
- You can pass a `fallback` prop (any JSX) and an `onError` callback.
- Wrap any tree under it to catch runtime errors in rendering/lifecycle.

---

18. Create a counter component using useState ?

```
import React, { useState } from 'react';

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

  return (
    <div>
      <h2>Count: {count}</h2>
      <button onClick={() => setCount(count + 1)}>
        Increment
      </button>
      <button onClick={() => setCount(count - 1)} disabled={count === 0}>
        Decrement
      </button>
      <button onClick={() => setCount(0)}>
        Reset
      </button>
    </div>
  );
}

export default Counter;
```

19. Create a counter component using useReducer ?

1. Counter Component Implementation

```
import React, { useReducer } from 'react';

// Reducer functionfunction counterReducer(state, action) {
  switch (action.type) {
    case 'INCREMENT':
      return { count: state.count + 1 };
    case 'DECREMENT':
      return { count: state.count - 1 };
    case 'RESET':
      return { count: 0 };
    case 'SET_VALUE':
      return { count: action.payload };
    default:
      return state;
  }
}

// Initial stateconst initialState = { count: 0 };

function Counter() {
  const [state, dispatch] = useReducer(counterReducer, initialState);

  return (
    <div style={{ textAlign: 'center', marginTop: '50px' }}>
      <h1>Counter: {state.count}</h1>

      <div>
        <button onClick={() ⇒ dispatch({ type: 'DECREMENT' })}>
          Decrement -
        </button>

        <button
          onClick={() ⇒ dispatch({ type: 'RESET' })}style={{ margin: '0 10px' }}>
          Reset
        </button>

        <button onClick={() ⇒ dispatch({ type: 'INCREMENT' })}>
          Increment +
        </button>
      </div>

      <div style={{ marginTop: '20px' }}>
        <input
          type="number"value={state.count}onChange={(e) ⇒
            dispatch({
              type: 'SET_VALUE',
              payload: Number(e.target.value) || 0
            })
          }style={{ padding: '5px', width: '60px' }}/>
      </div>
    </div>);
}

export default Counter;
```

2. Key Features

1. **Reducer Function**:
    - Handles state transitions based on action types
    - Pure function that returns new state
    - Handles increment, decrement, reset, and direct value setting
2. **Action Types**:

    - `INCREMENT`: Increases count by 1
    - `DECREMENT`: Decreases count by 1
    - `RESET`: Returns count to 0
    - `SET_VALUE`: Sets count to a specific value from input
3. **useReducer Hook**:
    - `const [state, dispatch] = useReducer(counterReducer, initialState);`
    - Provides current state and dispatch function to send actions
4. **UI Elements**:
    - Buttons for increment/decrement/reset
    - Input field for direct value modification
    - Real-time display of current count

3. How to Use:

1. Create a new file `Counter.js` with the above code
2. Import and use it in your main App component:

```
// App.jsimport Counter from './Counter';

function App() {
  return (
    <div className="App">
      <Counter />
    </div>);
}
```

---

20. How to call a method on every rerender of a component ?

To call a method **on every re-render** of a React component, you use the `useEffect` hook **without a dependency array**.

✅ Syntax:

```
useEffect(() ⇒ {
  // This runs after every render (initial + re-renders)
  yourMethod();
});
```

🔧 Example:

```
import React, { useState, useEffect } from 'react';

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

  const logMessage = () ⇒ {
    console.log('Component rendered. Current count:', count);
  };

  useEffect(() ⇒ {
    logMessage();
  }); // ⚠️ No dependency array → runs after every render

  return (
    <div>
      <h2>Count: {count}</h2>
      <button onClick={() ⇒ setCount(count + 1)}>Increment</button>
    </div>
  );
}
```

```
export default ReRenderExample;
```

📌 Explanation:

- `useEffect()` without a dependency array runs **after every render**.
- Each time the component state or props change, it re-renders → triggering the `useEffect`.

⚠️ Caution:

Avoid expensive logic inside this `useEffect` since it runs frequently. Use dependencies wisely for performance.

---

21. Create a pure component in reactjs ?

🔷 What is a Pure Component?

A **Pure Component** in React is a component that **doesn't re-render** unless its `props` or `state` **actually change**.

It implements `shouldComponentUpdate()` with a **shallow comparison** of `props` and `state` out of the box.

🔷 When to use?

Use it when:

- You want to **optimize performance**.
- You know the component re-renders unnecessarily.
- Props/state are **immutable** and don't change deeply.

✅ Example Using `React.PureComponent`

```
import React from 'react';
```

```
class Counter extends React.PureComponent {
  render() {
    console.log('Counter rendered');
    return <h1>Count: {this.props.count}</h1>;
  }
}

export default Counter;
```

✅ Usage:

```
import React, { useState } from 'react';
import Counter from './Counter';

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

  const handleClick = () => {
    // Updating with the same value won't re-render <Counter />
    setCount(prev => prev);
  };

  return (
    <div>
      <Counter count={count} />
      <button onClick={handleClick}>Update Count</button>
    </div>
  );
}

export default App;
```

✅ Functional Component Alternative

In **functional components**, you use `React.memo()` for the same purpose:

```
import React from 'react';

const Counter = React.memo(({ count }) => {
  console.log('Counter rendered');
  return <h1>Count: {count}</h1>;
});

export default Counter;
```

---

22. Create a controlled and uncontrolled component in react ?

In React, **controlled** and **uncontrolled** components refer to how form data is handled inside the component. Let's explore both with code examples and clear explanations.

✅ 1. Controlled Component

A **controlled component** is one where **React state controls the input value**. You use `useState` or any state management to store and update the input value.

Example:

```
import React, { useState } from 'react';

function ControlledInput() {
  const [name, setName] = useState('');

  const handleChange = (e) => {
    setName(e.target.value);
  };

  return (
    <div>
      <h3>Controlled Component</h3>
      <input type="text" value={name} onChange={handleChange} />
      <p>Entered Name: {name}</p>
    </div>
  );
}
```

✅ Key Points:

- Input value is stored in React state.
- `onChange` updates the state.
- React is always in control of the input's value.

❎ 2. Uncontrolled Component

An **uncontrolled component** is one where **the DOM controls the input value**, and you access it using a `ref`.

🔧 Example:

```
import React, { useRef } from 'react';

function UncontrolledInput() {
  const inputRef = useRef(null);

  const handleSubmit = () => {
    alert('Entered Name: ' + inputRef.current.value);
  };

  return (
    <div>
      <h3>Uncontrolled Component</h3>
      <input type="text" ref={inputRef} />
      <button onClick={handleSubmit}>Show Value</button>
    </div>
  );
}
```

✅ Key Points:

- No React state used to track input value.
- You directly interact with the DOM via `ref`.
- Useful for simple scenarios or performance-sensitive operations.

📌 Summary

Feature	Controlled Component	Uncontrolled Component
Input value managed by	React state	DOM (via `ref`)
Use case	Complex forms, validation	Simple inputs, quick forms
Access value	`useState` + `value` prop	`useRef().current.value`
React control	Full control	Minimal control

---

23. Create a custom hook using reactjs ?

A **custom hook** in React is a JavaScript function whose name starts with `use` and **lets you reuse logic across components**. Instead of repeating code (e.g., `useState`, `useEffect` setups) in multiple components, you can extract that logic into a custom hook.

Use Case: `useWindowWidth` (Get the current window width)

Step 1: Create the Custom Hook

```
// useWindowWidth.js
import { useState, useEffect } from 'react';

function useWindowWidth() {
  const [width, setWidth] = useState(window.innerWidth);

  useEffect(() => {
    const handleResize = () => setWidth(window.innerWidth);

    window.addEventListener('resize', handleResize);

    // Clean up on unmount
    return () => window.removeEventListener('resize', handleResize);
  }, []);

  return width;
}

export default useWindowWidth;
```

🔧 Step 2: Use it in a Component

```
import React from 'react';
import useWindowWidth from './useWindowWidth';

function WindowWidthComponent() {
  const width = useWindowWidth();

  return (
    <div>
      <h2>Custom Hook Example</h2>
      <p>Current window width: {width}px</p>
    </div>
  );
}
```

✅ Explanation

- `useWindowWidth` is a **custom hook** that:
    - Uses `useState` to track the window width.
    - Uses `useEffect` to listen to the resize event.
    - Cleans up the event listener on unmount.
- This hook can now be **reused in any component** without duplicating logic.

💡 When to Use Custom Hooks

- Fetching data (`useFetch`)
- Managing timers (`useTimeout`, `useInterval`)
- Reusable state behavior (e.g., form handlers, toggle logic)
- Subscribing to global events (scroll, resize, visibility)

---

24. Give an emaple of optimization using useMemo ?

✅ Example of Optimization using `useMemo` in React

Let's say you have a component that performs a heavy calculation every time it renders. This is inefficient **if the inputs to the calculation haven't changed**.

Code Example

```
import React, { useState, useMemo } from 'react';
```

```
function ExpensiveComponent() {
  const [count, setCount] = useState(0);
  const [input, setInput] = useState('');

  // Heavy calculation
  const expensiveValue = useMemo(() ⇒ {
    console.log('Calculating expensive value...');
    let total = 0;
    for (let i = 0; i < 100000000; i++) {
      total += i;
    }
    return total;
  }, []); // Only runs once, on initial render

  return (
    <div>
      <h2>Expensive Value: {expensiveValue}</h2>
      <inputtype="text"
        value={input}
        onChange={(e) ⇒ setInput(e.target.value)}
        placeholder="Type something"
      />
      <button onClick={() ⇒ setCount(count + 1)}>Count: {count}</button>
    </div>
  );
}

export default ExpensiveComponent;
```

🔍 Explanation

- `useMemo(() ⇒ {/* calculation */}, [dependencies])` caches the result of a computation.
- In the example, the expensive loop is **only run once** because the dependency array is empty `[]`.

- Without `useMemo`, this loop would execute **on every re-render**, even if the result doesn't need to change.

💡 When to Use `useMemo`

Use it to **optimize performance** when:

- You have a **heavy computation**.
- The computation depends on certain variables.
- You want to **avoid recalculating** when those variables haven't changed.

---

25. How to Share data between components using context api ?

The **Context API** allows you to create a global state or shared values that can be accessed by any component in the tree, **without passing props manually** at every level.

✅ Use Case Example

We want to share a `user` object between `App`, `Navbar`, and `UserProfile`.

Step-by-Step Implementation

1. Create a Context

```
// UserContext.js
import { createContext } from 'react';

export const UserContext = createContext(null);
```

---

2. Create a Provider Component

```
// UserProvider.js
import React, { useState } from 'react';
import { UserContext } from './UserContext';

const UserProvider = ({ children }) ⇒ {
  const [user, setUser] = useState({ name: "Saikrishna", email: "sai@email.co
m" });

  return (
    <UserContext.Provider value={{ user, setUser }}>
      {children}
    </UserContext.Provider>
  );
};

export default UserProvider;
```

3. Wrap your app with the Provider

```
// App.js
import React from 'react';
import UserProvider from './UserProvider';
import Navbar from './Navbar';
import UserProfile from './UserProfile';

function App() {
  return (
    <UserProvider>
      <Navbar />
      <UserProfile />
    </UserProvider>
  );
}
```

```
export default App;
```

4. Consume the Context in Any Component

```
// Navbar.js
import React, { useContext } from 'react';
import { UserContext } from './UserContext';

const Navbar = () => {
  const { user } = useContext(UserContext);
  return <div>Welcome, {user.name}!</div>;
};

export default Navbar;
```

```
// UserProfile.js
import React, { useContext } from 'react';
import { UserContext } from './UserContext';

const UserProfile = () => {
  const { user, setUser } = useContext(UserContext);

  return (
    <div>
      <h2>User Profile</h2>
      <p>Email: {user.email}</p>
      <button onClick={() => setUser({ ...user, name: 'Krishna' })}>
        Change Name
      </button>
    </div>
  );
};

export default UserProfile;
```

Summary:

Create a context using `createContext()`

Wrap your app with a `Provider`

Use `useContext()` to consume the shared data in any component

---

26. Which lifecycle hooks in class component are replaced with useEffect in functional components ?

In React, the `useEffect` hook in **functional components** replaces several **lifecycle methods** from **class components**. Here's a clear breakdown:

🔄 Class Component Lifecycle Methods vs useEffect

Class Component Lifecycle Method	Equivalent with `useEffect`	Purpose
`componentDidMount()`	`useEffect(() => { ... }, [])`	Runs **once after initial render** (mount)
`componentDidUpdate()`	`useEffect(() => { ... }, [dependency])`	Runs **after re-render** when **dependency changes**
`componentWillUnmount()`	`useEffect(() => { return () => { ... } }, [])`	Runs cleanup code **on unmount**
`componentDidMount` + `componentDidUpdate`	`useEffect(() => { ... })` (no dependency array)	Runs **after every render** (not recommended often)

✅ Examples for Better Understanding

1. `componentDidMount`

```
// Class
componentDidMount() {
  console.log('Component mounted');
}

// Functional
useEffect(() => {
  console.log('Component mounted');
}, []);
```

2. `componentDidUpdate`

```
// Class
componentDidUpdate(prevProps, prevState) {
  if (prevProps.count !== this.props.count) {
    console.log('Count updated');
  }
}

// Functional
useEffect(() => {
  console.log('Count updated');
}, [count]);
```

3. `componentWillUnmount`

```
// Class
componentWillUnmount() {
  console.log('Cleanup before unmount');
}

// Functional
useEffect(() => {
  return () => {
    console.log('Cleanup before unmount');
  };
}, []);
```

💡 Summary

The `useEffect` hook in functional components handles:

- Mounting
- Updating
- Unmounting

All in one unified API — depending on how you use the **dependency array** and **cleanup function**.

---

27. Create a popup using portal ?

✅ 1. Create the `Popup` component using portal

```
// Popup.js
import React from 'react';
import ReactDOM from 'react-dom';

const Popup = ({ children, onClose }) => {
  return ReactDOM.createPortal(
    <div style={overlayStyle}>
      <div style={popupStyle}>
        <button onClick={onClose} style={closeBtn}>X</button>
        {children}
      </div>
    </div>,
    document.getElementById('popup-root')
  );
};

const overlayStyle = {
  position: 'fixed',
  top: 0, left: 0, right: 0, bottom: 0,
  backgroundColor: 'rgba(0,0,0,0.5)',
  display: 'flex', alignItems: 'center', justifyContent: 'center',
};

const popupStyle = {
  background: '#fff',
  padding: '20px',
  borderRadius: '10px',
  minWidth: '300px',
  position: 'relative'
};

const closeBtn = {
  position: 'absolute',
  top: '10px',
  right: '10px',
  border: 'none',
  background: 'transparent',
  fontSize: '18px',
  cursor: 'pointer'
};

export default Popup;
```

✅ 2. Use the `Popup` in your App

```
// App.js
import React, { useState } from 'react';
import Popup from './Popup';

const App = () ⇒ {
  const [showPopup, setShowPopup] = useState(false);

  return (
    <div>
      <h1>React Portal Popup</h1>
      <button onClick={() ⇒ setShowPopup(true)}>Open Popup</button>

      {showPopup && (
        <Popup onClose={() ⇒ setShowPopup(false)}>
          <h2>This is a Portal Popup!</h2>
          <p>You can close it by clicking the X button.</p>
        </Popup>
      )}
    </div>
  );
};

export default App;
```

✅ 3. Add a `div` in `public/index.html` for the portal

```
<!-- index.html →
<body>
  <div id="root"></div>
  <div id="popup-root"></div> <!-- required for ReactDOM.createPortal →
</body>
```

28. How to call a method immediately after state is updated or
after component is rerendered in reactjs

✅ 1. Call a method immediately after state is updated

Use `useEffect` and pass the specific state variable as a dependency.

```
jsx
CopyEdit
import React, { useState, useEffect } from 'react';

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

  // Runs after 'count' is updated and the component is re-rendered
  useEffect(() => {
    console.log('Count updated:', count);
    yourMethod();
  }, [count]); // 👈 Dependency

  const yourMethod = () => {
    console.log("Method called after state update");
  };

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(prev => prev + 1)}>Increment</button>
    </div>
  );
}
```

✅ 2. Call a method after *every* re-render

Pass **no dependency array** to `useEffect`.

```
jsx
CopyEdit
useEffect(() ⇒ {
  console.log("Runs after every render");
  yourMethod();
});
```

⚠️ Note:

- React state updates are **asynchronous**, so calling a function **immediately after** `setState` inside the same block **won't reflect the updated state**.
- Always rely on `useEffect` for post-update logic.

---

29. How to Force a component to rerender with out using useState in react ?

To force a React component to re-render **without using `useState`**, you can use the `useReducer` or `useRef` + `forceUpdate` trick.

✅ Option 1: `useReducer` (Cleanest Way)

```
import React, { useReducer } from "react";

const forceUpdateReducer = (x) ⇒ x + 1;
```

```
function MyComponent() {
  const [, forceUpdate] = useReducer(forceUpdateReducer, 0);

  return (
    <div>
      <p>Component content</p>
      <button onClick={forceUpdate}>Force Rerender</button>
    </div>
  );
}
```

✅ Option 2: Using `useRef` + Manual `forceUpdate`

```
import React, { useRef } from "react";

function MyComponent() {
  const renderCount = useRef(0);
  const [, forceRerender] = React.useReducer((x) => x + 1, 0);

  const handleClick = () => {
    renderCount.current++;
    forceRerender();
  };

  return (
    <div>
      <p>Render Count: {renderCount.current}</p>
      <button onClick={handleClick}>Force Rerender</button>
    </div>
  );
}
```

✅ Option 3: Using `this.forceUpdate()` in Class Components

```
class MyComponent extends React.Component {
  force = () => {
    this.forceUpdate();
  };

  render() {
    return (
      <div>
        <p>Class Component</p>
        <button onClick={this.force}>Force Rerender</button>
      </div>
    );
  }
}
```

⚠️ Avoid `forceUpdate` tricks in production unless necessary.

Prefer `useState` / `useReducer` for state-driven re-renders. Let me know your use case and I can recommend the cleanest solution.

---

30. How to rerender a component on value change in react ?

To re-render a React component when a value changes, the most common and recommended way is to use **`useState`** or **`useEffect`** **with props/state**.

✅ 1. Using `useState` (most common)

The component re-renders **automatically** when state changes.

```
import React, { useState } from 'react';
```

```
function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Count is: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}
```

Every time `setCount` is called, React re-renders the component with the updated state.

✅ 2. Using `useEffect` to respond to value changes (props or state)

```
import React, { useEffect, useState } from 'react';

function Watcher({ value }) {
  useEffect(() => {
    console.log('Value changed:', value);
    // This will run every time `value` changes
  }, [value]);

  return <div>Value is: {value}</div>;
}
```

If `value` is updated in a parent component, `Watcher` will re-render automatically.

✅ 3. Using props — re-render happens automatically if props change

```
function Parent() {
  const [name, setName] = useState("John");

  return (
    <><Child name={name} />
      <button onClick={() => setName("Doe")}>Change Name</button>
    </>
  );
}

function Child({ name }) {
  return <div>Hello, {name}</div>;
}
```

When `setName` updates the `name`, `Child` component re-renders with the new prop.

🧠 Summary:

Trigger	Causes Re-render?			
`useState`	✅ Yes			
`useReducer`	✅ Yes			
Changing props	✅ Yes			
`useRef`	❌ No			
`useEffect`	⚠️ Used to react to changes, not trigger re-render itself			

---

31. Give an example of optimization using usecallbacks in react ?

✅ Scenario

Suppose you have a **parent component** that renders a **child component**. The child receives a function as a prop. If the parent re-renders, the function is re-created, causing the child to re-render unnecessarily.

`useCallback` helps **memoize** the function so that its reference doesn't change unless its dependencies change.

🔧 Example

```
import React, { useState, useCallback } from 'react';

const Child = React.memo(({ handleClick }) ⇒ {
  console.log('Child rendered');
  return <button onClick={handleClick}>Click Me</button>;
});

const Parent = () ⇒ {
  const [count, setCount] = useState(0);
  const [otherState, setOtherState] = useState(false);

  // Without useCallback, this function will be recreated on every render
  const handleClick = useCallback(() ⇒ {
    setCount((prev) ⇒ prev + 1);
  }, []);

  return (
    <div>
      <h1>Count: {count}</h1>
      <Child handleClick={handleClick} />
      <button onClick={() ⇒ setOtherState(!otherState)}>Toggle Other State</b
utton>
    </div>
  );
};
```

```
export default Parent;
```

🧠 Explanation

- **Without `useCallback`**: The `handleClick` function gets re-created every time `Parent` renders. So `Child` sees a new `handleClick` prop and re-renders.
- **With `useCallback`**: The function is memoized, so `Child` won't re-render unless the function actually changes (which only happens if its dependencies change).

✅ Benefit

Using `useCallback` avoids **unnecessary re-renders of child components**, especially when:

- The child is wrapped in `React.memo()`.
- The function doesn't need to change on every render.
- Performance matters due to many children or heavy renders.

---

32. How to call a method when component is rendered for the first time in react ?

To **call a method when a component is rendered for the first time** in React (i.e., on component mount), you should use the `useEffect` hook with an **empty dependency array** `[]`.

✅ Syntax:

```
useEffect(() => {
  // code here runs only once when the component mounts
}, []);
```

✅ Example:

```
import React, { useEffect } from 'react';

function MyComponent() {
  useEffect(() => {
    console.log('Component mounted');
    fetchData(); // method call
  }, []); // empty array ensures it runs only once

  const fetchData = () => {
    // Simulate API call
    console.log('Fetching data...');
  };

  return <div>Hello World</div>;
}
```

✅ Explanation:

- `useEffect` runs **after the component is mounted**.
- Passing `[]` as the second argument means it **won't re-run** unless the component is unmounted and re-mounted.
- This is equivalent to `componentDidMount()` in class components.

✅ Class Component Equivalent:

```
class MyComponent extends React.Component {
  componentDidMount() {
    console.log('Component mounted');
    this.fetchData();
  }

  fetchData = () => {
    console.log('Fetching data...');
  };

  render() {
    return <div>Hello World</div>;
  }
}
```

---

33. How to change styles based on condition in react ?

✅ 1. Using Inline Styles with Ternary Operator

You can directly apply conditional styles using a ternary operator:

```
function StatusBox({ isActive }) {
  return (
    <divstyle={{
        backgroundColor: isActive ? 'green' : 'gray',
        color: 'white',
        padding: '10px',
      }}
    >
      {isActive ? 'Active' : 'Inactive'}
    </div>
  );
}
```

✅ 2. Conditional Class Names

You can apply different CSS classes conditionally:

```
function Button({ isPrimary }) {
  return (
    <button className={isPrimary ? 'btn-primary' : 'btn-secondary'}>
      Click Me
    </button>
  );
}
```

CSS:

```
.btn-primary {
  background-color: blue;
  color: white;
}

.btn-secondary {
  background-color: gray;
  color: black;
}
```

---

34. How do you access the dom element ?

In React, you can **access DOM elements** using **Refs**.

✅ Method: Using `useRef` (Functional Components)

`useRef` allows you to directly access a DOM node created by JSX.

Example:

```
import React, { useRef, useEffect } from "react";

function InputFocus() {
  const inputRef = useRef(null);

  useEffect(() => {
    // Access the DOM element and focus it
    inputRef.current.focus();
  }, []);

  return <input ref={inputRef} placeholder="Focus on mount" />;
}
```

✅ Method: `createRef` (Class Components)

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.myDivRef = React.createRef();
  }

  componentDidMount() {
    // Access DOM element
    this.myDivRef.current.style.backgroundColor = 'yellow';
  }

  render() {
    return <div ref={this.myDivRef}>Hello</div>;
  }
}
```

🔎 When to use it?

- To focus an input field
- To measure DOM dimensions
- To scroll an element into view
- To interact with third-party libraries (charts, modals, etc.)

⚠️ Best Practice

Avoid excessive use of DOM access in React. Prefer **React state and props** for controlling behavior and appearance.

---

35. Perform type checking using prop-types in reactjs ?

In ReactJS, you can use the `prop-types` package to perform **type checking** on component props. It helps ensure that components are used with the correct data types and provides warnings in the console during development if types don't match.

✅ Steps to Use `prop-types`:

1. **Install `prop-types`:**

```
npm install prop-types
```

1. **Import and use it in your component:**

🔷 Example: Functional Component

```
import React from 'react';
import PropTypes from 'prop-types';
```

```
const UserProfile = ({ name, age, isAdmin }) => {
  return (
    <div>
      <h2>{name}</h2>
      <p>Age: {age}</p>
      {isAdmin && <p>Admin Access</p>}
    </div>
  );
};

// Type checking with PropTypes
UserProfile.propTypes = {
  name: PropTypes.string.isRequired,
  age: PropTypes.number,
  isAdmin: PropTypes.bool
};

export default UserProfile;
```

🔷 Example: Class Component

```
import React from 'react';
import PropTypes from 'prop-types';

class Product extends React.Component {
  render() {
    return (
      <div>
        <h3>{this.props.title}</h3>
        <p>Price: ${this.props.price}</p>
      </div>
    );
  }
}
```

```
Product.propTypes = {
  title: PropTypes.string.isRequired,
  price: PropTypes.number.isRequired
};

export default Product;
```

🔶 Common Prop Types

Type	Usage
`PropTypes.string`	String
`PropTypes.number`	Number
`PropTypes.bool`	Boolean
`PropTypes.array`	Array
`PropTypes.object`	Object
`PropTypes.func`	Function
`PropTypes.node`	Anything that can be rendered
`PropTypes.element`	React element
`PropTypes.symbol`	ES6 Symbol

🔶 Advanced Usage

```
PropTypes.arrayOf(PropTypes.string)
PropTypes.objectOf(PropTypes.number)
PropTypes.oneOf(['small', 'medium', 'large'])
PropTypes.oneOfType([PropTypes.string, PropTypes.number])
PropTypes.shape({
  id: PropTypes.number.isRequired,
  name: PropTypes.string
})
```

36. How to Implement a Theme Switcher Using Context API

Step 1: Create Theme Context

We'll create a context that holds the current theme and a function to toggle it.

```
// ThemeContext.js
import React, { createContext, useState, useContext } from 'react';

const ThemeContext = createContext();

export const ThemeProvider = ({ children }) ⇒ {
  // Theme state: 'light' or 'dark'
  const [theme, setTheme] = useState('light');

  // Toggle between light and dark
  const toggleTheme = () ⇒ {
    setTheme(prev ⇒ (prev === 'light' ? 'dark' : 'light'));
  };

  return (
    <ThemeContext.Provider value={{ theme, toggleTheme }}>
      {children}
    </ThemeContext.Provider>
  );
};

// Custom hook for easier usage
export const useTheme = () ⇒ useContext(ThemeContext);
```

Step 2: Wrap Your App with ThemeProvider

Wrap your main app component with the `ThemeProvider` so the theme context is available throughout the app.

```
// index.js or App.js
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';
import { ThemeProvider } from './ThemeContext';

ReactDOM.createRoot(document.getElementById('root')).render(
  <ThemeProvider>
    <App />
  </ThemeProvider>
);
```

Step 3: Create ThemeSwitcher Component

This component will read the current theme and toggle it on button click.

```
// ThemeSwitcher.js
import React from 'react';
import { useTheme } from './ThemeContext';

const ThemeSwitcher = () => {
  const { theme, toggleTheme } = useTheme();

  return (
    <button onClick={toggleTheme}>
      Switch to {theme === 'light' ? 'Dark' : 'Light'} Mode
    </button>
  );
};
```

```
export default ThemeSwitcher;
```

Step 4: Use Theme in Your App

You can now consume the theme value anywhere to apply styles conditionally.

```
// App.js
import React from 'react';
import { useTheme } from './ThemeContext';
import ThemeSwitcher from './ThemeSwitcher';

const App = () => {
  const { theme } = useTheme();

  const appStyle = {
    height: '100vh',
    display: 'flex',
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: theme === 'light' ? '#fff' : '#222',
    color: theme === 'light' ? '#222' : '#fff',
    flexDirection: 'column',
    gap: '20px',
  };

  return (
    <div style={appStyle}>
      <h1>{theme.charAt(0).toUpperCase() + theme.slice(1)} Mode</h1>
      <ThemeSwitcher />
    </div>
  );
};

export default App;
```

Explanation

- **Context API** creates a global state (`ThemeContext`) that holds the current theme and the toggle function.
- **ThemeProvider** wraps the app and provides access to theme state everywhere.
- **useTheme** custom hook simplifies accessing context values.
- **ThemeSwitcher** component reads current theme and toggles it.
- **App** component uses the theme value to apply styles dynamically.

This keeps theme state centralized and allows any component in the tree to read or update the theme easily.

---

37. How to update Document Title on Mount in a React Component

To update the **document title** when a React component mounts, you typically use the `useEffect` hook in a functional component.

Why use `useEffect`?

- React components render many times during their lifecycle.
- We want to update the document title **only once when the component mounts** (or when specific values change).
- `useEffect` with an empty dependency array runs only once after the first render (mount).

Example: Update document title on mount

```
import React, { useEffect } from 'react';

function MyComponent() {
  useEffect(() => {
    document.title = "My New Page Title";
  }, []);  // empty dependency array means run once on mount

  return <div>Hello, this is my component!</div>;
}

export default MyComponent;
```

Explanation:

- `useEffect(() => { ... }, [])` runs the callback function once after the component is mounted.
- Inside the callback, we set `document.title` to the desired string.
- The empty array `[]` ensures this effect does **not** run on subsequent re-renders.

In class components

If you use class components, you do this in `componentDidMount`:

```
class MyComponent extends React.Component {
  componentDidMount() {
    document.title = "My New Page Title";
  }

  render() {
    return <div>Hello from class component</div>;
  }
}
```

Summary:

- Use `useEffect` with empty dependencies in functional components.
- Use `componentDidMount` in class components.

- This updates the browser tab's title when the component mounts.

If you want to update the title dynamically based on props or state, add those dependencies in the array. But for just updating once on mount, keep it empty.

---

38. How to create a Higher-Order Component (HOC) to Log Props in reactjs ?

✅ What is a Higher-Order Component (HOC) in React?

A **Higher-Order Component (HOC)** is a **function** that takes a **component as input** and returns a **new component** with extended or enhanced functionality.

Example use case: You want to log props every time a component renders – without modifying the component directly. A HOC is the perfect solution.

✅ How to Create a HOC to Log Props in React

Here's a step-by-step explanation with code:

🔧 Step 1: Create the HOC

```
// withPropsLogger.js
import React, { useEffect } from 'react';

const withPropsLogger = (WrappedComponent) => {
  return function PropsLoggerWrapper(props) {
    useEffect(() => {
      console.log(`[Props Logger] ${WrappedComponent.name} props:`, props);
    }, [props]); // log every time props change

    return <WrappedComponent {...props} />;
  };
};
```

```
export default withPropsLogger;
```

✅ What this does:

- Takes a `WrappedComponent` as input.
- Logs the props to the console whenever they change.
- Renders the `WrappedComponent` with all received props.

🔧 Step 2: Use the HOC

```
// MyComponent.js
import React from 'react';

const MyComponent = ({ name }) => {
  return <div>Hello, {name}</div>;
};

export default MyComponent;
```

```
// App.js
import React from 'react';
import MyComponent from './MyComponent';
import withPropsLogger from './withPropsLogger';

const MyComponentWithLogging = withPropsLogger(MyComponent);

function App() {
  return <MyComponentWithLogging name="Saikrishna" />;
}

export default App;
```

---

39. How will you Manage Environment-Specific Configurations in Reactjs in your project ?

1. Using `.env` Files with Create React App (CRA)

CRA supports environment variables through `.env` files.

- `.env` (default, for all envs)
- `.env.development`
- `.env.production`
- `.env.test`

Important: All variables must start with `REACT_APP_` to be accessible in your React app.

Example Setup

Create these files in your project root:

.env.development

```
REACT_APP_API_URL=https://dev.api.example.com
REACT_APP_FEATURE_FLAG=true
```

.env.production

```
REACT_APP_API_URL=https://api.example.com
REACT_APP_FEATURE_FLAG=false
```

2. Accessing Variables in React Code

```
function App() {
  const apiUrl = process.env.REACT_APP_API_URL;
  const featureFlag = process.env.REACT_APP_FEATURE_FLAG === 'true';
  return (
    <div>
      <h1>Environment: {process.env.NODE_ENV}</h1>
      <p>API URL: {apiUrl}</p>
      <p>Feature Enabled: {featureFlag ? 'Yes' : 'No'}</p>
    </div>
  );
}

export default App;
```

- `process.env.NODE_ENV` will be `'development'` or `'production'` depending on how you run/build your app.
- `process.env.REACT_APP_API_URL` will be replaced at build time with the matching `.env` file value.

3. Running Your App

- `npm start` will use `.env.development`
- `npm run build` will use `.env.production`

4. Custom Configuration Object (Optional)

If you want more control, create a config file that switches based on `NODE_ENV`:

```
// src/config.js
const devConfig = {
  apiUrl: "https://dev.api.example.com",
  featureFlag: true,
};

const prodConfig = {
  apiUrl: "https://api.example.com",
  featureFlag: false,
};
```

```
const config = process.env.NODE_ENV === "production" ? prodConfig : devC
onfig;

export default config;
```

Use it in your app:

```
import config from './config';

function App() {
  return (
    <div>
      <h1>API URL: {config.apiUrl}</h1>
      <p>Feature Enabled: {config.featureFlag ? 'Yes' : 'No'}</p>
    </div>
  );
}
```

Summary

- Use `.env` files with variables prefixed by `REACT_APP_`.
- Access them in your React code via `process.env.REACT_APP_*`.
- CRA automatically picks the right `.env` based on environment.
- For more flexibility, create your own config file and export based on `NODE_ENV`.

---

40. How to create a Custom Hook for Data Fetching with Loading and Error States

Creating a custom React hook for data fetching with loading and error states is a great way to encapsulate and reuse fetch logic cleanly across your app.

Why a Custom Hook for Data Fetching?

- Reuse the same fetch logic in multiple components
- Keep components cleaner and focused on UI
- Handle loading, error, and data states in one place

How to Create It — Step by Step

1. Define the Hook

Create a function called `useFetch` which takes a URL (or any fetch parameter).

It manages:

- `data` — the fetched result
- `loading` — boolean while fetching
- `error` — error message if fetch fails

2. Use `useState` for state management and `useEffect` to trigger fetch

Example Code

```
import { useState, useEffect } from "react";

function useFetch(url) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    if (!url) return;

    setLoading(true);
    setError(null);

    fetch(url)
```

```
      .then((response) => {
        if (!response.ok) {
          throw new Error("Network response was not ok");
        }
        return response.json();
      })
      .then((json) => {
        setData(json);
        setLoading(false);
      })
      .catch((err) => {
        setError(err.message || "Something went wrong");
        setLoading(false);
      });
  }, [url]);

  return { data, loading, error };
}

export default useFetch;
```

3. Using the Hook in a Component

```
import React from "react";
import useFetch from "./useFetch";

function UserList() {
  const { data, loading, error } = useFetch("https://jsonplaceholder.typicode.co
m/users");

  if (loading) return <p>Loading users...</p>;
  if (error) return <p>Error: {error}</p>;

  return (
    <ul>
```

```
      {data.map(user => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
}

export default UserList;
```

How it Works

- When the component mounts or the URL changes, `useEffect` triggers the fetch.
- While fetching, `loading` is `true`.
- If the fetch succeeds, `data` is set and `loading` is `false`.
- If it fails, `error` is set and `loading` is `false`.
- The component using the hook can respond to these states accordingly.

---

41. Create a component to upload the file in reactjs ?

```
import React, { useState } from "react";

function FileUpload() {
  const [file, setFile] = useState(null);
  const [status, setStatus] = useState("");

  const onFileChange = (e) => {
    setFile(e.target.files[0]);
    setStatus("");
```

```
};

const onUpload = async () ⇒ {
  if (!file) {
    setStatus("Please select a file first");
    return;
  }

  const formData = new FormData();
  formData.append("file", file);

  setStatus("Uploading...");

  try {
    const response = await fetch("https://your-api-endpoint.com/upload", {
      method: "POST",
      body: formData,
    });

    if (!response.ok) {
      throw new Error(`Upload failed with status ${response.status}`);
    }

    const result = await response.json();
    setStatus(`Upload successful: ${result.message || file.name}`);
  } catch (error) {
    setStatus(`Upload error: ${error.message}`);
  }
};

return (
  <div>
    <input type="file" onChange={onFileChange} />
    <button onClick={onUpload}>Upload</button>
    {status && <p>{status}</p>}
  </div>
  );
}

export default FileUpload;
```

Replace `"https://your-api-endpoint.com/upload"` with your actual upload URL.

This sends the selected file as multipart/form-data via POST to your API.

---

42. How to cancel the api call when the timeout occurs using abort controller ?

What is AbortController?

- `AbortController` is a browser API that lets you abort (cancel) a fetch request.
- You create an instance of `AbortController`.
- Pass its `signal` property to the fetch request.
- Call `abort()` on the controller to cancel the fetch.

How to use AbortController with timeout in React?

You combine `AbortController` with `setTimeout` to cancel the fetch if it takes too long.

Example code and explanation

```
import React, { useEffect, useState } from 'react';

function FetchWithTimeout() {
  const [data, setData] = useState(null);
  const [error, setError] = useState(null);
  const [loading, setLoading] = useState(false);

  useEffect(() => {
    const controller = new AbortController();  // Create AbortController
    const signal = controller.signal;

    const timeoutId = setTimeout(() ⇒ {
      controller.abort(); // Cancel the fetch after timeout
    }, 5000); // 5000ms = 5 seconds timeout

    setLoading(true);
    fetch('https://jsonplaceholder.typicode.com/posts/1', { signal })
      .then((response) ⇒ {
        if (!response.ok) {
          throw new Error('Network response was not ok');
        }
        return response.json();
      })
      .then((json) ⇒ {
        setData(json);
        setError(null);
      })
      .catch((err) ⇒ {
        if (err.name === 'AbortError') {
          setError('Fetch request was aborted due to timeout');
        } else {
          setError(err.message);
        }
      })
      .finally(() ⇒ {
        setLoading(false);
        clearTimeout(timeoutId); // Clear the timeout if fetch finished before time
out
      });

    // Cleanup if component unmounts before fetch completes
    return () ⇒ {
      clearTimeout(timeoutId);
      controller.abort();
    };
  }, []);

  return (
    <div>
      {loading && <p>Loading...</p>}
      {error && <p style={{ color: 'red' }}>Error: {error}</p>}
      {data && (
        <div>
          <h3>{data.title}</h3>
          <p>{data.body}</p>
        </div>
      )}
    </div>
  );
}

export default FetchWithTimeout;
```

Explanation:

- We create an `AbortController` instance and get its `signal`.
- Pass the `signal` to `fetch` options.
- We set a `setTimeout` to call `controller.abort()` after 5 seconds.
- If the fetch completes before timeout, `clearTimeout` cancels the timeout.
- If timeout happens first, fetch is aborted, and `.catch` handles the `AbortError`.
- The cleanup function cancels the fetch and clears the timeout if the component unmounts early.

Summary

- Use `AbortController` to cancel fetch requests.
- Combine with `setTimeout` to implement timeout logic.
- Handle `AbortError` to detect when fetch is canceled.